Title

Breaking the ISA Barrier in Modern Computing

Permalink

https://escholarship.org/uc/item/8rp233s4

Author

Venkat, Ashish

Publication Date

2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Breaking the ISA Barrier in Modern Computing

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Ashish Venkat

Committee in charge:

Professor Dean M. Tullsen, Chair
Professor Andrew B. Kahng
Professor Sorin Lerner
Professor Timothy Sherwood
Professor Deian Stefan

2018

The dissertation of Ashish Venkat is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

DEDICATION

To Suchetha.

EPIGRAPH

If we knew what it was we were doing,

it would not be called research, would it?

—Albert Einstein

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I owe immense thanks to my advisor Dean Tullsen, who has been an inspiring teacher, a prolific research mentor, and a father figure who has shaped my life in many profound ways. Dean's tireless enthusiasm of pursuing new and ambitious research ideas and his incredible dedication to science has been a constant source of motivation and inspiration to me as a young researcher. He has always encouraged me to set challenging goals for myself, and has never settled for anything but the best in everything that I do. He has provided me with a plethora of opportunities that have had a significant role to play in my growth as a researcher and an academic. I could never thank him enough for his words of wisdom and reassurance during turbulent times, and for the invaluable life lessons of ethics, empathy, and sensitivity he has given me through an exemplary lifestyle, which I will strive to emulate through the rest of my life.

My many thanks are due to my PhD committee members Tim Sherwood, Sorin Lerner, Andrew Kahng, and Deian Stefan for providing me with valuable feedback from time to time. I'd like to thank Hovav Shacham for his many useful insights and suggestions during my thesis proposal. I'd also like to take this opportunity to thank Jason Mars and Hadi Esmaeilzadeh for believing in me and for their support and encouragement throughout my PhD career.

I'd like to thank many research colleagues who have made this dissertation possible. In particular, I'd like to thank Matt DeVuyst for his mentorship during my first year. I'd also like to acknowledge Arvind Krishnaswamy, Rajan Palanivel, Koichi Yamada, and other colleagues at Intel for the many brainstorming sessions during the initial days of formulating HIPStR. I'd like to thank Doug Burger and Aaron Smith for their mentorship while I was at Microsoft. I'd also like to thank Sriskanda Shamasunder for being a diligent hacker, and for so many fun-filled conversations in the lab amidst all the craziness. I'd like to thank Harsha Basavaraj for taking up the onerous task of the RTL design of the x86 decoder. Finally, I'd like to thank my first mentee, Kazem Taram, for all his enthusiasm, hard work, and dedication that have made my mentorship an enjoyable experience.

I'd like to take this opportunity to thank my lab mates – Hung-Wei Tseng, Manish Gupta, Sam Wasmundt, Andreas Prodromou, Brian Tsui, Jinghao Jia, Yishin Shih, and Zinsser Zhang. I'd also like to thank my friends from college, Freescale, Brocade, and graduate school, and my family members including my brother and sister-in-law, my parents-in-law, and my brother-in-law for always believing in me.

I owe so much to Suchetha for being an incredibly understanding and supportive wife. I thank her for an extraordinarily fun-filled life journey together that has kept my sanity through the many trials and tribulations of graduate school and academic job search. We have been through many life adventures together, and honestly none of this would matter, if not for her. The successful completion of this dissertation is due, in large part, to her unconditional love and unwavering support.

Finally, I'd like to thank my parents for being strong pillars of support throughout my life. They have always encouraged me to explore and pursue opportunities, go above and beyond my potential, think big, and never give up. I'd also like to acknowledge that my mom is my first teacher – she has taught me to be hard working and meticulous in whatever I do. My dad has had a significant influence in my life as an academic, in honing my writing and speaking skills, and in inducing, promoting, and reinforcing my love for Computer Science.

Chapter 3, in part, is a reprint of the material as it appears in proceedings of ASPLOS 2012. DeVuyst, Matthew; Venkat, Ashish; Tullsen, Dean M., Execution Migration in a Heterogeneous-ISA Chip Multiprocessor, 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March, 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in proceedings of ISCA 2014. Venkat, Ashish; Tullsen, Dean M., Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor, 41st ACM International Symposium on Computer Architecture (ISCA), June, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in proceedings of ASPLOS 2016. Venkat, Ashish; Shamasunder, Sriskanda; Shacham, Hovav; Tullsen, Dean M., HIPStR: Heterogeneous-ISA Program State Relocation, 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April, 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in full, is currently being prepared for submission for publication of the material. Venkat, Ashish; Basavaraj, Harsha; Tullsen, Dean M., Composite-ISA Cores: Enabling Multi-ISA Heterogeneity using a Single ISA. The dissertation author was the primary investigator and author of this material.

VITA

2008	Bachelor of Engineering, NIE Mysuru
2008-2009	Freescale Semiconductor, Bengaluru
2009-2010	Brocade Communications, Bengaluru
2011	Software Development Intern Amazon.com, Inc., Seattle, WA
2012	Graduate Technical Intern Intel Corporation, Santa Clara, CA
2014	Master of Science, Computer Science, University of California, San Diego
2014	C. Phil. in Computer Science, University of California, San Diego
2015	Research Intern Microsoft Research, Redmond, WA
2016	Research Intern IBM Research Labs, Haifa, Israel
2011-2017	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego
2011-2018	Research Assistant, Department of Computer Science and Engineering University of California, San Diego
2018	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Ashish Venkat, Harsha Basavaraj, Dean M. Tullsen. “Composite-ISA Cores: Enabling Multi-ISA Heterogeneity using a Single ISA”, *Under Review*, 2018.

Andreas Prodromou, Ashish Venkat, and Dean M. Tullsen. “Deciphering Predictive Schedulers for Heterogeneous-ISA Architectures”, *Under Review*, 2018.

Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. “Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization”, *In Preparation*, 2018.

Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. “Mobilizing the Micro-Ops: Exploiting Context-Sensitive Decoding for Security and Energy Efficiency”, *ISCA*, 2018.

Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean M. Tullsen, and Rajesh Gupta. “Reliability-Aware Data Placement for Heterogeneous Memory Architecture”, *HPCA*, 2018.

Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. “NuHIPStR: Heterogeneous-ISA Program State Relocation”, *ASPLOS*, 2016.

Ashish Venkat, Arvind Krishnaswamy, Yamada Koichi, and Rajan Palanivel. “Binary Translation-Driven Program State Relocation”, *US Patent*, 2015.

Ashish Venkat and Dean M. Tullsen. “Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor”, *ISCA*, 2014.

Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. “Execution Migration in a Heterogeneous-ISA Chip Multiprocessor”, *ASPLOS*, 2012.

ABSTRACT OF THE DISSERTATION

Breaking the ISA Barrier in Modern Computing

by

Ashish Venkat

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Dean M. Tullsen, Chair

In recent years, the computing landscape has witnessed a shift towards hardware specialization in response to the rapid growth and expansion of software, changing market risks, and fundamental technological limitations. However, the largest barrier to full exploitation of heterogeneity has by far been the difficulty of programming for them. There is a pressing need for systems that allow the exploitation of highly heterogeneous platforms without creating additional programmer burden. The goal of this dissertation is to empower the hardware/software interface, specifically the Instruction Set Architecture (ISA) and the runtime system, with diverse capabilities to enable the seamless adoption of heterogeneous hardware, without breaking the traditional models of programming.

Existing heterogeneous designs either constrain CPU cores to feature a single ISA or allow multiple ISAs that assign distinct jobs to distinct cores, or at best statically partition work, resulting in a tight coupling of an application to the underlying ISA. This dissertation challenges the assumption that the single-ISA constraint is necessary, and further enables programs to cross a heretofore forbidden boundary – the ISA. In particular, this dissertation describes a compiler and runtime strategy for swift and seamless process migration across diverse ISAs, and further showcases results from a massive core architecture optimization process that demonstrates the performance and energy efficiency benefits of multi-ISA heterogeneous architectures. In addition to its performance and energy efficiency benefits, this dissertation also explores and demonstrates the security potential of multi-ISA architectures to thwart several evasive variants of the Return-Oriented Programming (ROP) attack. This dissertation further alleviates the complexity concerns of multi-vendor ISA heterogeneity by studying the effect of introducing composite-ISA heterogeneity.

Chapter 1

Introduction

The modern computing landscape is characterized by the rapid evolution of software, changing market risks, rising security threats, and technological limitations. The microprocessor industry, fraught with these challenges, has witnessed consistently diminishing rates of improvement in the execution efficiency of high-end general-purpose CPUs that drive a considerable chunk of the world's computational demands. Consequently, the high cost of the one-size-fits-all computational model has now become increasingly apparent – general-purpose processors perform well on an average case, but they allow no individual application to run as efficiently as it would on specialized hardware. Thus, hardware specialization or heterogeneity has and will continue to play a crucial role in modern processor and system design.

Modern processor architectures employ hardware specialization in two key dimensions. While some architectures employ specialized cores to accelerate the performance of certain domain-specific workloads [san08, fus08, teg10, Qua11, PCC$^+$14, JYP$^+$17], others take advantage of microarchitectural heterogeneity by combining large high-performance cores and small power-efficient cores on the same chip, to create efficient designs that cater to the diverse execution characteristics of general-purpose mixed workloads [teg11, Gre11, HM08, KFJ$^+$03, KTR$^+$04, KTJ06, VCJE$^+$12, cut17]. Multiple commercial offerings exist today, in general-purpose, embedded, and server markets, that exploit both dimensions. However, the largest barrier to full exploitation of heterogeneity has been the difficulty of adapting that heterogeneity to traditional programming and execution models, resulting in several otherwise efficient hardware designs to be discarded as not viable.

This dissertation introduces a new dimension of heterogeneity that exploits a fundamental abstraction of computing – the Instruction Set Architecture (ISA), while preserving the traditional models of programming and execution. Early work on single-ISA heterogeneous multicore processors [Gre11, HM08, KFJ$^+$03, KTR$^+$04, KTJ06] constrained CPU cores to a single ISA in order to maximize efficiency by allowing a thread to dynamically identify, and migrate to, the core to which it is most suited during a particular phase and under the current operating conditions. This dissertation challenges the assumption that the single-ISA boundary is necessary and further demonstrates that limiting an architecture to a single ISA sacrifices a critical dimension of heterogeneity.

By pulling down the boundary wall, this dissertation unlocks several previously unexplored *heterogeneous-ISA architectures* that offer greater gains in terms of performance, energy efficiency, and security. These architectures synergistically combine microarchitectural heterogeneity with ISA heterogeneity to realize more efficient designs that can effectively exploit the inherent ISA affinity of an application. In addition, these architectures have the potential to boost the overall entropy and resilience of the system to provide a formidable defense against state-of-the-art code reuse attacks.

1.1 Breaking the ISA Barrier

This dissertation explores several novel and programmer-transparent hardware architecture design, compiler, and runtime techniques to enable the seamless adoption of heterogeneous-ISA architectures. First, it establishes the viability of multi-ISA heterogeneity by proposing a low-cost cross-ISA process migration infrastructure that is orders of magnitude faster than prior art. Second, it showcases the performance and energy savings potential of multi-ISA heterogeneity via a massive core architecture optimization process. Third, it proposes a security defense that takes advantage of ISA diversification to thwart many evasive variants of the Return-Oriented Programming (ROP) attack [Sha07, RBSS12]. Fourth, it significantly alleviates the complexity concerns of multi-ISA heterogeneity by proposing hardware and software techniques that recreate and in many cases, supersede the gains of multi-ISA heterogeneity using a single composite-ISA derived from a large superset.

1.1.1 Cross-ISA Process Migration

Modern architectures allow us to instantly configure the frequency, voltage, cache size, and other microarchitectural parameters to increase efficiency. Yet our ISA choice is typically constrained by a decision made when our phone, laptop, or server was purchased. And yet, the choice of ISA can have a significant impact on execution efficiency. One of the primary goals of this dissertation is to allow us to now make that choice, not just individually for each program, but every few milliseconds within the execution of a single program. However, process migration across heterogeneous ISAs is a non-trivial problem. This is because the runtime program state of an application is always kept in ISA-specific form, potentially requiring expensive state transformation at the time of migration.

This dissertation proposes novel compiler and runtime mechanisms that allow for seamless and instantaneous cross-ISA process migration at less than 5% degradation in overall performance. The key components of the migration infrastructure being (1) a multi-ISA compilation framework that emits a *symmetrical fat binary* containing multiple ISA-specific text sections, a common stack frame layout, and a common ISA-agnostic data section, and (2) a migration runtime that performs dynamic binary translation until execution reaches a compiler-marked *equivalence point*, at which program state (registers and stack objects) can be safely transformed to a different ISA.

To assist the creation of a *symmetrical fat binary*, we take advantage of a powerful architecture-independent intermediate representation that can act as a bridge between the ISAs, and provide hints for transforming program state at the time of migration. This work shows that careful and consistent multi-ISA compilation can enable faster and more frequent migrations by significantly minimizing the amount of runtime program state to be transformed, while simultaneously enabling most, if not all ISA-specific transformations. Furthermore, due to the relatively high frequency of compiler-marked *equivalence points*, we find that binary translation for the specific use case of cross-ISA migration calls for a different modus operandi – minimize the translation time rather than optimizing the translation itself. Overall, this work crosses a critical threshold by allowing processes to migrate across ISAs potentially every timer interrupt, while advancing prior state-of-the-art by orders of magnitude.

1.1.2 Design of a Heterogeneous-ISA Chip Multiprocessor

By decoupling the execution binary from historical ISA choices, the cross-ISA process migration strategy establishes the viability of multi-ISA heterogeneous architectures that show promising potential in terms of performance and energy efficiency. A critical step in the design of a multi-ISA heterogeneous architecture is choosing a diverse set of ISAs. While ISAs seem to converge over time (RISC ISAs adding complex operations, CISC ISAs translated to RISC micro-ops internally), we find that there remains sufficient diversity in existing modern ISAs to provide useful heterogeneity. This dissertation examines some key aspects that characterize ISA diversity, including code density, decode and instruction complexity, register pressure, native floating-point arithmetic vs emulation, and SIMD processing.

The design of a heterogeneous-ISA chip multiprocessor involves navigating a complex search space, made larger by the additional dimension of freedom. The design space we study in this work encompasses 72 software workloads, 600 single core configurations, and a 128 billion distinct heterogeneous-ISA multicore configurations that harness the diversity offered by three modern ISAs: (1) ARM's ultra-low power Thumb, (2) the traditionally RISC Alpha, and (3) the high-performance CISC x86-64. By co-designing the hardware architectures and the ISAs to provide the best aggregate architecture, we arrive at a more effective and efficient design than one composed of homogeneous cores, or even heterogeneous cores that share a single ISA. Specifically, we show that for a given peak power/area budget constraint, multi-ISA heterogeneous architectures can outperform single-ISA heterogeneous architectures by an average of 21% and save 23% in energy at no loss in performance.

The design space exploration reveals two key insights. First, different applications exhibit a natural affinity for one ISA or another, and that affinity can change as the application progresses into a different execution phase. For example, in a homogeneous-ISA setting, Thumb is not a serious candidate, because it performs so poorly for certain codes; however, as part of a multi-ISA solution, it shines for certain code regions. As a result, ISA-heterogeneity consistently offers superior performance and energy savings, even in scenarios where hardware heterogeneity alone provides diminishing returns. Second, the ISA has a significant influence on microarchitectural design choices that enable efficient transistor investment on the available silicon real estate, calling for a tighter ISA-microarchitecture co-design. In fact, by observing the results of the design space exploration, we provide the CPU architect with a set of tools to enable ISA-microarchitecture co-design and thereby better streamline their search processes.

1.1.3 HIPStR: Security Defense via ISA Diversification

In addition to its potential for greater performance and energy efficiency, this dissertation demonstrates that ISA heterogeneity can be seamlessly leveraged to provide a strong security defense against buffer overflow exploits such as Return-Oriented Programming (ROP) [Sha07, RBSS12]. Buffer overflow vulnerabilities form a major class of security exposures that plague the Internet today. They rank third amongst the common vulnerability types reported by the National Vulnerability Database, finishing just behind cross site and cryptographic vulnerabilities [nvd].

These vulnerabilities have been systematically exploited by code reuse attacks such as ROP to perform arbitrary malicious computation without injecting malicious code. ROP attacks hinge on the attacker being able to chain together short code snippets in the program (called gadgets) that end with a return instruction, by overflowing the stack with a carefully constructed sequence of return addresses, and other malicious data. ROP has been shown to be Turing Complete for multiple ISAs (both RISC and CISC) and for a wide range of applications.

This dissertation introduces HIPStR (Heterogeneous-ISA Program State Relocation), a security defense that has the potential to radically transform the attack landscape of state-of-the-art Return-Oriented Programming. The primary motivation for HIPStR is the fact that ROP attacks thrive on two fundamental properties. First, the knowledge of the underlying ISA is critical to construct a successful exploit. Owing to its unique ability to perform seamless and instantaneous cross-ISA process migration, HIPStR significantly inhibits several code reuse attacks including the notorious JIT-based ROP attacks [SMD$^+$13] by forcing the attacker to chain ROP gadgets across different ISAs. Second, any program including a ROP program requires some amount of program state (in the form of registers and stack objects) to perform computation. To this end, HIPStR employs dynamic binary translation to continuously randomize the register and stack state to an extent that brute force attacks [BBM$^+$14] are rendered practically infeasible on current, or even distant future microprocessors. Overall, HIPStR offers a formidable defense against several variants of ROP attacks, and reduces their overall attack surface to such an extent that it is difficult to construct a four-gadget shellcode exploit, let alone achieve Turing-completeness.

1.1.4 Composite-ISA Architectures

Despite their potential for greater performance, energy efficiency, and security, the deployment of heterogeneous-ISAs on a single chip is non-trivial due to a number of practical concerns. First, integration of multiple vendor-specific commercial ISAs on a single chip is fraught with significant licensing, legal, and verification costs and barriers. Second, process migration in a heterogeneous-ISA CMP necessitates the creation of fat binaries, involves expensive binary translation and state transformation costs due to the difference in encoding schemes and application-binary interfaces (ABI) of fully disjoint ISAs.

This dissertation significantly alleviates these concerns via yet another design space exploration to identify composite-ISA architectures that can recreate the effects of multi-ISA heterogeneity using a single composite-ISA. A composite-ISA is derived by leveraging a large superset ISA that resembles the Intel x86 and offers customization along five different axes of diversity: (1) register depth (8 to 32 programmable registers), (2) register width (32 vs 64-bit), (3) instruction complexity (1:1 vs 1:n micro-op encoding), (4) predication (full vs partial), and (5) specialized support (vector vs scalar). This provides the hardware designer and the compiler with far more control over the choice of ISA, with the ability to make fine-grained choices about the features of importance, maximizing the overall execution efficiency.

Due to the constraint of a single baseline superset ISA, we find that the derived custom ISAs can never incorporate all traits of distinct vendor-specific ISAs (such as the code compression of Thumb). However, the greater flexibility and composability of these designs offer substantial new ISA-affinity advantages. Composite-ISA heterogeneous architectures match and in many cases, supersede the efficiency gains of multi-ISA heterogeneous architectures, and further enhance existing gains due to hardware heterogeneity by an average of 19% in performance and 31% in energy savings. Furthermore, owing to the overlapping nature of the feature sets that make up the composite ISAs, the overall cost of migration drastically drops to just 0.42%. By combining the ISA-affinity advantages of multi-ISA heterogeneity and the simplicity of single-ISA heterogeneity, this research brings the best of both worlds.

1.2 Overview of Dissertation

Chapter 2 gives background information on heterogeneous architectures. It discusses single-ISA heterogeneous architectures in embedded, general-purpose, and server environments. It also briefly discusses some early work that evaluate the benefits, trade-offs, and complexities of multi-ISA heterogeneity and migration techniques for discrete heterogeneous-ISA machines. It further provides a primer on return-oriented programming and existing architectural and runtime support to mitigate code reuse attacks.

Chapter 3 lays out the proposed cross-ISA process migration strategy. In particular, it details our multi-ISA compilation methodology that leverages a powerful architecture-independent intermediate representation provided by LLVM [LA04a] to create a *symmetrical fat binary* that sports multiple ISA-specific text sections, but a single ISA-agnostic data section. It also discusses our runtime strategy that includes ISA-specific state transformation and binary translation for instantaneous cross-ISA process migration.

Chapter 4 outlines the design space navigation process geared at identifying an optimal heterogeneous-ISA multicore processor in terms of performance and/or energy efficiency under specific peak power and area budget constraints. It also discusses several inferences from the design space exploration that could potentially equip CPU architects with a set of tools for tighter ISA-microarchitecture co-design.

Chapter 5 proposes a security defense HIPStR (Heterogeneous-ISA Program State Relocation) that leverages ISA diversification and program state relocation to defend against several variants of the Return-Oriented Programming attack. It describes many binary code transformations for program state relocation in order to boost the entropy of the system, and details a security-aware migration policy that maximizes security with limited impact on performance.

Chapter 6 describes the design of a composite-ISA architecture that has the potential to recreate and in many cases, supersede the gains of multi-ISA heterogeneity, by implementing composite feature sets derived using a single large superset ISA, exploiting greater flexibility in ISA choice. It presents our compiler and runtime strategy that extends the x86 backend to support and exploit the underlying composite ISAs, and enables seamless migration between the composite-ISAs. It also outlines our changes to the Intel x86 decoder both to support the decoding of the superset ISA, but also to be customized and reduced for the subset ISAs, and further studies the effect of these customizations on decoder area and peak power.

Chapter 7 summarizes the contributions of this dissertation.

Chapter 2

Background

This chapter provides background information related to this thesis. Section 2.1 gives a brief overview of modern heterogeneous architectures and their flavors. Section 2.2 discusses contemporary work on the benefits, trade-offs, and system design implications of ISA-heterogeneity. Section 2.3 provides the background on return-oriented programming (ROP) attacks and Section 2.4 discusses existing mitigations against ROP.

2.1 Heterogeneous Architectures

Prior research has shown that heterogeneous chip multiprocessors (CMPs) are capable of higher performance and energy efficiency as compared to homogeneous processors. Kumar, et al. [KFJ$^+$03, KTR$^+$04, KTJ06] introduced single-ISA heterogeneous multicore architectures. These architectures employ cores of different sizes, organizations, and capabilities, allowing an application to dynamically identify and migrate to the most efficient core, thereby maximizing both performance and energy efficiency. In addition, these architectures also employ cores from different process generations that may each operate at a different voltage/frequency domain and/or a different power state. Since its inception, several microarchitectural [LPD$^+$12, PLDM15, LPD$^+$16, LPD$^+$14, SKKK16] and scheduling techniques [VCJE$^+$12, VCE13, VCAH$^+$13, ASC$^+$16, NEE17, MNU$^+$15, MNM$^+$15, NAM$^+$17, SWTB11, AEJE17, DK13] have been proposed in the literature to better harness the gains due to single-ISA heterogeneity.

Multiple commercial offerings in the embedded, GPU, and consumer markets have exploited this technology. ARM's big.LITTLE processor [Gre11] couples a high-performance out-of-order 3-way superscalar Cortex-A15 core with a low-power dual-issue inorder Cortex-A7 core. Many Qualcomm Snapdragon [Qua11] and Samsung Exynos [KKCL13] chipsets employ a variation of the big.LITTLE processor combining a high-performance Cortex-A57/Cortex-A73 with a low-power Cortex-A53 on the same chip to maximize energy efficiency. NVidia's Tegra-3 processor [teg11] employs a variable symmetric multiprocessing companion CPU core built using a low-power silicon process that operates at a lower frequency than the rest of the four cores on chip. Apple's A11 SoC [cut17] features a six-core CPU with two high-performance *Monsoon* cores and four energy-efficient *Mistral* cores.

Yet another class of heterogeneous chip multiprocessors make use of specialized hardware to accelerate the performance of a certain type of workloads. These include the integrated CPU-GPU architectures such as Intel's Sandy Bridge [san08] and AMD's fusion [fus08]. However, these architectures do not allow migration between core types at arbitrary places in the code. Current industry offerings of heterogeneous-ISA CMPs include MPSoCs in the embedded market [Tex], GPUs, and accelerators in the HPC market [teg10]. Though IBM's Cell microprocessor [KDH$^+$05] is a heterogeneous-ISA CMP geared towards general-purpose computing, it suffers from two major concerns that make it unsuitable for general-purpose mixed workloads. First, the Synergistic Processing Elements (SPEs) use a special-purpose ISA that is suitable for only those workloads that exhibit SIMD parallelism. Second, lack of a common address space makes dynamic task migration infeasible.

Interestingly, modern datacenters already employ servers from different generations, and even different vendors, as a result of routine upgrades and competitive vendor markets [MT13, Mor15, Mat16]. This shifting trend away from traditionally homogeneous hardware designs is further evidenced by the latest OpenPower venture of Google and Rackspace [Nic16, Mor15] that capitalizes on finer and cost-effective, albeit heterogeneous design options. While the proclivity to keep the task management runtime design relatively simple has traditionally favored the deployment of architecturally homogeneous commodity servers in datacenters [Höl10, DB13], the literature provides overarching evidence that intelligent QoS-aware management systems [WA12, MT13, DK14, PLD+15, LCG+14, LCG+15, HZL+15] that exploit microarchitectural heterogeneity can significantly improve energy efficiency while meeting the strict QoS requirements of latency-critical datacenter workloads. In fact, these strategies not only take advantage of the "unintentional" heterogeneity due to server upgrades, but call for a microarchitecturally heterogeneous datacenter by design due to its ability to cater to the diverse execution characteristics of the constantly evolving datacenter workloads in a cost-effective manner.

Several researchers have proposed design space exploration methodologies for heterogeneous architectures. Strozek, et al. [SB09] describe a process flow for automatic synthesis and evaluation of heterogeneous CMPs based on runtime profiles of certain embedded applications, given different area and power budgets. Intel's QuickIA [CSH+12] research prototype allows researchers to explore heterogeneous architectures consisting of multiple generations of Intel processors and FPGAs. Open source tools like Fabscalar [CWS+11, CWS+12], OpenPiton [BMF+16, MFN+17], and Alladin [SRWB14, SRWB15] further allow researchers to explore and analyze heterogeneous architectures of varying complexity. The search methodology we employ in this work is similar to the one described by Kumar, et al. [KTJ06]. However, our goal is to not only identify the optimal heterogeneous-ISA multicore designs, but also to lay out the first principles for ISA-microarchitecture co-design in such an architecture.

2.2 The Path to Multi-ISA Heterogeneity

A major contribution of this thesis is a detailed processor architecture design and compiler methodology for heterogeneous-ISA architectures [DVT12, VT14, VSST16, BSR$^+$16]. These architectures allow cores that are already microarchitecturally heterogeneous to further implement diverse instruction sets. By exploiting ISA affinity, where different code regions within an application inherently prefer a particular ISA, they realize substantial performance and efficiency gains over hardware heterogeneity alone.

In contrast, Blem, et al. [BMS13b] claim that modern ISAs such as ARM and x86 have a rather similar impact in terms of performance and energy efficiency. However, that work compares rather similarly register pressure-constrained ISAs (ARM-32 and x86-32), keeps target-independent optimizations on and turns off machine-specific tuning, ignores feature set differences (e.g., Thumb), and makes homogeneous hardware assumptions, unlike the work on heterogeneous-ISA architectures [VT14, BSR$^+$16, BLJ$^+$17, NR16]. Akram and Sawalha [AS17, Akr17] perform extensive validation of the conflicting claims and conclude that the ISA does indeed have a significant impact on performance.

More contemporary studies in the literature show that ISA affinity is beneficial not just in general-purpose environments, but could potentially enable significant energy efficiency in datacenter environments [BLJ$^+$17, NR16, ope16]. Lustig, et al. [LTPM15] describe mechanisms for cross-ISA memory consistency model translation. Wang, et al. [WYZ$^+$17] enable offloading of binary code regions in a heterogeneous-ISA client/server environment. Furthermore, there is considerable amount of work that studies and addresses system implications of ISA-heterogeneity such as differences in page table structure and organization, system call ABI, and POSIX compatibility, via replicated OS kernel support for heterogeneous-ISA and overlapping-ISA architectures [BSA$^+$15, BLJ$^+$17, LBK$^+$10].

The Tui system [SH98] describes a process migration strategy for heterogeneous-ISA machines in the context of wide area computing. The main idea of that work is to transform the runtime program state to an intermediate form and then re-compile it to the required ISA, at the time of migration. Ferrari, et al. [FCG00] describe process introspection, a process state-capture and recovery mechanism initiated by a running process at periodic *poll points*, at which each subroutine in the call stack recursively captures and transforms its own state to suit the ISA the process is being migrated to. We borrow some techniques from both these works; however, our compiler methodology and runtime strategy is geared towards a more diverse set of ISAs in a chip multiprocessor environment, which makes the problem significantly harder and requires additional techniques and optimizations presented in this thesis.

More recently, multiple studies have advocated for an ISA-affinity driven live process/container migration in a heterogeneous-ISA datacenter environment [BLJ$^+$17, NR16], that copies a minimal set of memory pages across the heterogeneous servers during migration and then proactively pushes the rest once execution is resumed. Such a mechanism not only considerably reduces system downtime, but accelerates the convergence to steady state by minimizing the number of remote page faults.

Finally, binary translators have long been used to port/emulate legacy binaries on heterogeneous ISAs [BSGG13]. Chen, et al. [CYH$^+$08] describe techniques to translate ARM binaries for execution on a MIPS-like architecture. While the binary translation technique described in this thesis deals with similar challenges of ISA diversity, that work employs static translation while our work proposes dynamic translation, requiring different approaches in many cases. Managed runtimes and browsers employ dynamic binary translation to perform profile-guided optimization [HS04] of hot code regions, program sheperding, and JIT hardening [VKYP15]. Some examples of dynamic translators used for emulation include Digital's FX!32 [RH97] (which translates x86 applications to Alpha) and HP's Aries [ZT00] (which translates PA-RISC applications to IA-64). These translators use a two-phase translation, where the first phase does emulation and collects runtime profile information, and the second performs optimization. We cannot afford to have a two-phase translation, as binary translation in our case typically runs for far fewer instructions, rather than the entire program—the extra time for profiling and a two-phase translation process cannot be amortized. QEMU [Bel05] is the closest dynamic translator to the one described in our work. However, it is optimized for system emulation and our binary translator is optimized for the migration use case.

2.3 Return-Oriented Programming

Chapter 5 of this thesis demonstrates the security potential of heterogeneous-ISA architectures to defend against Return-Oriented Programming attacks. This section gives the necessary background information on Return-Oriented Programming (ROP).

Buffer overflow vulnerabilities have been systematically exploited by code injection attacks for many decades. These attacks, in their simplest form, inject malicious code into an application and hijack its control flow to result in rogue behavior. To prevent injection and subsequent execution of malicious code, most modern processors and operating systems have now employed Executable Space Protection [PT03b, VdV04] (dubbed as Data Execution Prevention by Windows) which ensures that a memory page is either writable or executable, but not both.

With the advent of Executable Space Protection, classical code injection has been slowly replaced by a more evasive form of attack, called Code Reuse. Typically, these attacks reuse existing code in the memory image of a process to perform malicious computation. An early example of such attacks is the *return-into-libc* attack [SD97], which exploits a buffer overflow on the stack to return into a C library function. Despite being able to subvert the control flow of an application without injecting malicious code, return-into-libc is inherently limited to the C library, and thus incapable of performing arbitrary malicious computation. In recent years, return-into-libc attacks have evolved into a more general and flexible scheme of attacks called Return-oriented Programming [RBSS12, Sha07].

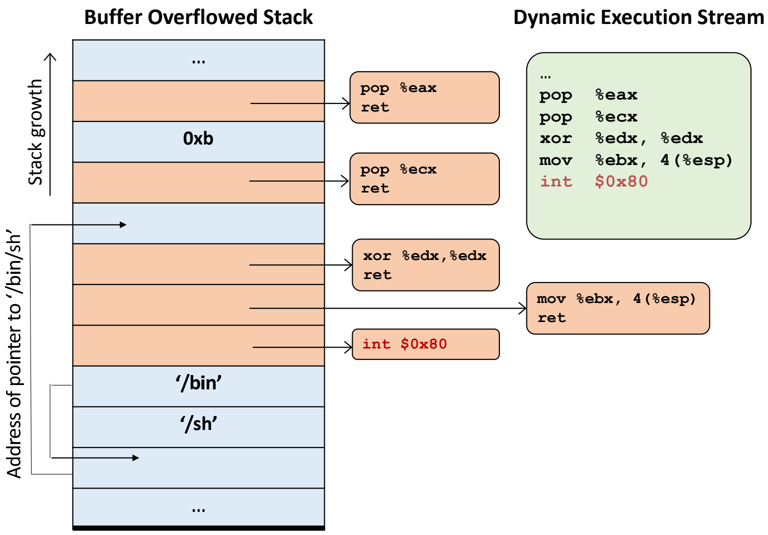

Figure 2.1: Return-oriented Programming

Return-oriented Programming (ROP) typically involves chaining together short code snippets in the program (called gadgets) that end with a return or an indirect jump instruction, by overflowing the stack with a carefully constructed sequence of return addresses, and other data required for malicious computation. Figure 2.1 shows a ROP attack that spawns a command shell. The attack begins with an attacker injecting an exploit payload on to the stack, exploiting a buffer overflow. The payload is crafted to overwrite the return address with the address of a short code snippet within the program, called a *gadget*, that ends in a return instruction. Once the gadget has executed and the instruction pointer has reached the return instruction, the stack pointer points to the address of the next gadget, and the exploit continues.

ROP hinges on the attacker being able to control the stack pointer and use it as the instruction pointer. Several evasive variants of ROP have been described in the literature that use indirect jumps in place of returns (Jump-oriented programming (JOP) [BJFL11, CDD$^+$10, JTL14]), hijack control flow using chains of existing C++ virtual functions [STL$^+$15, CCD$^+$15], corrupt data variables to perform arbitrary malicious computation while staying on legitimate control-flow paths [CW14, CBP$^+$15, HSA$^+$16], and provide Turing-completeness on different instruction set architectures [BRSS08, Kor10]. Moreover, in JIT environments such as browsers and the Adobe Flash, JIT-spraying techniques exploit the just-in-time compilation functionality to generate predictable chunks of exploit code in the text section, using carefully crafted JavaScripts or ActionScripts called GaJITS [RI11]. The advent of automated exploit compilers has further made ROP a formidable attack technique to defend against [SAB11].

2.4 ROP Mitigations

ROP thrives on three fundamental assumptions. First, the attacker should be able to subvert the control flow of the victim to a specific gadget, by exploiting an existing buffer overflow vulnerability. This necessitates the victim system to be void of any hardware or software control flow integrity enforcements. Second, the attacker should have prior knowledge of gadget locations in the process memory image, to overflow the stack with an appropriate sequence of return addresses. Third, the victim program must contain ample gadgets, enough to form, for example, a Turing-complete set. Not surprisingly, mitigation techniques often exploit these assumptions to defend against return-oriented programming.

Several control flow integrity (CFI) techniques have been proposed in the literature. Abadi, et al. [ABEL05] first formalized the idea of CFI. The main idea of that work is to constrain the execution of the program to a predefined control flow graph (CFG) by instrumenting the program to perform ID-checks before every indirect jump. Any jump to an invalid destination instruction (a jump target not defined in the may-point-to set) is flagged as a violation of control flow integrity. They also observe that it is difficult to implement ideal CFI statically without a runtime mechanism to track function calls and indirect jumps. There has been significant follow-up work on CFI at the hardware, runtime, and compiler levels [CPM^{+}98, DSW11, VPMPADK13, CBD^{+}99, OVB^{+}06, SLZD04, Ven01].

More recent work such as CCFIR [ZWC^{+}13], bin-CFI [ZS13], branch regulation [KOAGP12], code pointer integrity [KSP^{+}14], and practical context-sensitive CFI [vdVAG^{+}15] have made significant strides in reducing the attack surface, by employing more fine-grained CFI, in the absence of any source or debug information, and at an acceptable degradation in performance. However, several backdoor attacks [EFG$^+$15, CBP$^+$15, GABP14, GAP$^+$14, CW14, LDDLARS14, DLSM14, STL$^+$15] have been described in the literature to bypass these techniques, thereby exposing the need for a stricter enforcement of CFI.

Numerous hardware and software techniques have been proposed to prevent *stack smashing*, i.e, overflowing the return address on the stack with a spurious jump target, to subvert control flow. StackGuard [CPM$^+$98] is a compiler transformation that places a canary right after the return address on the stack. The validity of the canary is checked before returning control to the caller function, thereby detecting corruption of the return address before the exploit takes control. The major problem with StackGuard is that an adversary who can guess the canary value can overwrite the return address while preserving the integrity of the canary itself. Cowan, et al. [CBD$^+$99] address this drawback using randomly generated and null-terminated canaries. While the attack remains probabilistic, the defense comes at an expense of about 6% more CPU time. Interestingly, GCC implements stack smashing protection as an optional transformation called ProPolice [Eto03]. Most software packages in standard Linux distributions, including Ubuntu, Fedora, and FreeBSD [FT09, Sun13, UT06] have been compiled with ProPolice.

StackShield [Ven01] is yet another compiler solution that uses two separate stacks – a data stack and a control stack. The control stack is exclusively used for maintaining return addresses, and the data stack is used to maintain fixed stack variables, register spills, and temporaries. While this technique reports little to no performance overhead, memory used for the control stack remains vulnerable. Inspired by StackShield, multiple hardware mechanisms [OVB$^+$06, PZL06, XKPI02] have been proposed to exploit the hardware Return Address Stack (RAS) in order to secure the return address on the program stack. These solutions come with significant hardware complexity. First, they should detect a return address compromise with 100% accuracy in spite of speculative execution. Second, they should handle RAS overflow and underflow scenarios with little performance impact. Finally, they should handle exceptional scenarios in the software such as *setjmp and longjmp*. ROPDefender [DSW11] attempts to address the above intricacies by emulating the secure return address stack using binary instrumentation. In similar spirit of detecting anomalies at the RAS level, Pappas, et al. [VPMPADK13] describe a detection technique that leverages the Last Branch Recording (LBR) feature of Intel processors to monitor and detect abnormal control transfer patterns.

Yet another class of ROP defenses randomize the location of gadgets in the process image, making the attack only probabilistic. Several gadget location randomization techniques have been proposed in the literature, at various granularities — module (ASLR [PT03a]), basic block [WMHL12], instruction [HNTC$^+$12], and byte [SKIH12] levels. Furthermore, several binary re-writing and gadget obfuscation mechanisms [KKP03, PPK12, BS08, CAC$^+$08, OBL$^+$10, PLPI13] have been proposed to restrict the number of useful gadgets in a program. These techniques can be applied orthogonally to gadget location randomization, in order to boost the entropy of a system. The performance overhead incurred by these techniques is typically proportional to the level of code obfuscation. The effectiveness of these solutions depends on the amount of entropy (number of randomizable states) they provide and the extent to which they can resist entropy exhausting attacks. In the presence of a memory disclosure vulnerability, these randomization techniques can be bypassed by simple brute-force attacks [SPP$^+$04, BBM$^+$14] that exploit a memory disclosure, in just a matter of a few thousand attempts.

The load-time nature of state-of-the-art randomization techniques makes them highly susceptible to just-in-time code reuse (JIT-ROP) attacks that exploit a single leaked memory disclosure to read code pages in memory, disassemble them, and reconstruct the control flow graph on-the-fly. Snow, et al. [SMD$^+$13] show that JIT-ROP can bypass a combination of fine-grained randomization techniques in a matter of 23 seconds. Several periodic randomization and software diversification techniques [MBSN14, DLS$^+$15, LHBF14, CHB$^+$15, BHR$^+$15, BDL$^+$16, LLNB16, HHD16] have claimed immunity to these types of attacks at varying levels of performance.

Several hardening techniques have been employed to counter JIT-spraying attacks in browsers and other JIT environments. These systems have to invariably bypass Executable Space Protection and use RWX pages in order to generate and execute code just-in-time. The cost of dynamically changing permissions (from WX to RX) for such pages is often extremely high, thereby leaving such systems vulnerable to code injection attacks. In fact, Internet Explorer is the only JIT-based system to implement secure page permissions. Some JIT environments such as the Chrome V8 engine employ a low cost solution called *guard pages* to prevent code injection. The guard pages separate heap pages from JIT pages in the application's address space. Any attempt to overwrite a guard page will be flagged as a security breach.

Both IE11 and Chrome V8 employ several fine-grained randomization techniques to secure JIT-pages. *Page Randomization* provides 16 bits of entropy by randomizing the location of JIT pages. *Constant Blinding* eliminates gaJITs by randomizing the values of constant literals used in JavaScripts and ActionScripts. *Random NOP Insertion*, a technique inspired by G-free [OBL$^+$10], eliminates more gaJITs by randomly scattering NOPs across the code generated by the JIT compiler. Random NOP insertion is also employed by Adobe Flash. Finally, in order to suppress heap spraying attacks [Wev04], both IE11 and Chrome V8 enforce a cap on the number of heap pages that can be allocated. Interestingly, the Java Virtual Machine and Jaeger Monkey of Firefox do not employ any of the above mentioned randomization techniques, to avoid performance penalties.

Chapter 3

Cross-ISA Process Migration

Prior work on single-ISA heterogeneous multicore processors has demonstrated the criticality of process migration in reaping the full benefits of the underlying heterogeneity. First, process migration allows an application to adapt to phase changes by dynamically identifying and migrating execution to the core of its preference, maximizing performance and energy efficiency. Second, it helps to move processes to high-performance or low-power cores in the processor due to changes in the current operating condition and/or power state. Third, it allows migrating processes to cooler parts of the chip in the event of a thermal emergency. Fourth, process migration has traditionally enabled load balancing in environments ranging from general-purpose multicore processors to grid computing.

Cross-ISA process migration in particular is a well known difficult problem [FCG00, SH98, VBSS94]. This is because all runtime state of a program (data and code) is kept in an ISA-specific form, and migration to a different ISA could potentially involve expensive program state transformation. This chapter seeks to address these challenges and consequently establish the viability of a heterogeneous-ISA architecture. In particular, this chapter describes a multi-ISA compilation strategy and a low-overhead and programmer-transparent runtime mechanism that allow applications to seamlessly and instantaneously migrate across heterogeneous-ISA cores.

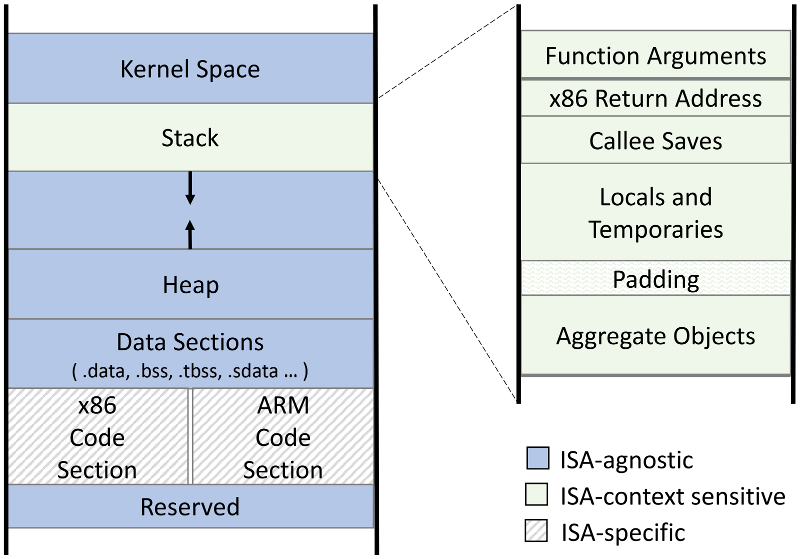

Figure 3.1: Symmetrical Fat Binary

3.1 Symmetrical Fat Binary

The central piece of our cross-ISA process migration strategy is a *symmetrical fat binary* that contains multiple ISA-specific code sections, a common stack frame organization, and a common set of ISA-agnostic data and heap sections. Figure 3.1 illustrates a symmetrical fat binary that is capable of running on a heterogeneous-ISA CMP that implements both ARM and x86 cores. The symmetrical fat binary is created by a multi-ISA compiler that enforces the following set of consistency rules.

Global Data Consistency. In order to keep the data section ISA-agnostic, the multi-ISA compiler enforces common endianness, basic data type size, and alignment rules. This ensures all global data objects are consistently referenced at the same virtual address by both ISAs, and thereby avoids expensive pointer transformations at the time of migration.

Code Section Consistency. Although the fat binary contains multiple code sections, with minor changes to the page table, both code sections can be mapped such that they begin at the same address. This is possible because a core only loads its own code into its private instruction cache. Furthermore, to ensure function pointer consistency, the linker aligns functions in such a way that they are seen at the same address in both ISAs.

Heap Consistency. Libraries that are responsible for dynamic memory allocation must ensure that a consistent view of the heap memory is maintained across both ISAs.

Stack Consistency. To avoid handling pointer inconsistencies on the stack, a common stack frame organization is enforced, as shown in Figure 3.1. In particular, the direction of stack growth, size, alignment, and organization of each stack frame must remain consistent across both ISAs. In doing so, we do not add any additional instructions, since we at most change the relative position of a stack object from the stack/frame pointer. However, each ISA is free to use the calling conventions and register allocation strategies that it finds most beneficial.

3.2 Multi-ISA Compilation

Our compilation strategy is to start with a common intermediate representation, and then perform consistent backend compilation for multiple targets, to generate target-specific code for each ISA along with a common set of target-independent data sections. A by-product of the multi-ISA compilation is a set of transformation rules that are to be applied at the time of process migration, to convert the program state from one ISA-form to another. In the next few paragraphs, we describe our compilation strategy in greater detail.

Common Intermediate Representation. To enforce the above consistency rules, we rely on a well-defined architecture-independent intermediate representation that acts as a bridge between the different ISAs and provides hints to the runtime at the time of migration. In this work, we leverage the LLVM compiler framework [LA04b] and the Clang front-end [Lat08] to generate a common intermediate representation (LLVM bitcode), and perform target-specific backend compilation thereafter. To keep the front-end compilation ISA-agnostic, we make use of the *target-triple* functionality of Clang to specify the data types of a generic target, for all ISAs.

Target-Independent Type Legalization. To minimize the amount of program state to be transformed, we enforce common rules for promotion, truncation, expansion and type conversion. We allow certain exceptions during type legalization that interfere with ISA diversity - e.g., vector widening/scalarizing on x86-64, and long mode/floating point emulation on Thumb. For the most part, target-independent type legalization ensures a consistent view of global data and bitcode-level variables across all ISAs, during every stage of compilation. This is critical for generating a single version of target-independent global data sections.

Intermediate Name Propagation. Once the intermediate representation has been generated, we provide each bitcode-level variable with a unique name. During the subsequent code generation and optimization passes, we ensure that each target-level machine operand (both registers and fixed stack slots) is associated with its corresponding intermediate name, if any. This gives us the ability to distinguish between bitcode-level and target-level variables, which plays a key role at the time of program state transformation.

Hints for State Transformation. At the time of task migration, the runtime transforms the stack in such a way that the program appears to be executing on the migrated-to ISA from the time it was instantiated. To facilitate such a stack transformation, the multi-ISA compiler generates metadata that can be easily incorporated into a symbol-table-like data structure within the executable. The compiler will generate one such table for each ISA. The table itself holds records for each basic block or function call site in the executable, at which program state can be safely transformed, and native execution can be resumed on the migrated-to ISA. Each record of the table contains a mapping from a live register or a stack object, to its corresponding source/intermediate-level variable name.

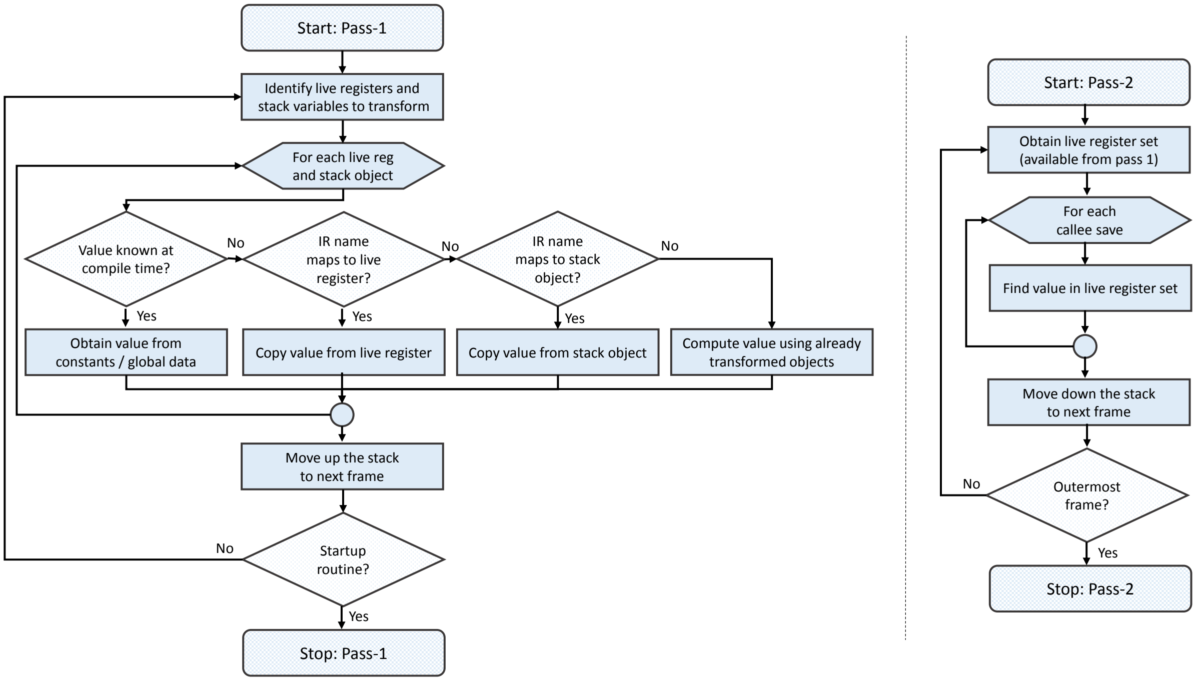
Start: Pass-1
Identify live registers and stack variables to transform
For each live reg and stack object
Value known at compile time?
No
IR name maps to live register?
No
IR name maps to stack object?
No
Yes
Yes
Yes
Obtain value from constants / global data
Copy value from live register
Copy value from stack object
Compute value using already transformed objects
Move up the stack to next frame
No
Startup routine?
Yes
Stop: Pass-1
Start: Pass-2
Obtain live register set (available from pass 1)
For each callee save
Find value in live register set
Move down the stack to next frame
No
Outermost frame?
Yes
Stop: Pass-2

Figure 3.2: Operation of the State Transformer

3.3 State Transformation

To reap full benefits of ISA heterogeneity, it is critical that we dont turn off any target-specific compiler optimization. However, due to a number of architecture-specific transformation passes such as code motion, not all points of execution in a symmetrical fat binary are migration-safe. Therefore, the migration runtime either stalls migration or performs dynamic binary translation until execution reaches an *equivalence point* at which the program state can be safely transformed. The goal of the state transformer is to transform all inconsistent state on the stack (e.g., spilled registers, temporaries, etc) and create the final architectural register state on the migrated-to core. Figure 3.2 illustrates the working of the state transformer.

In the first pass, the state transformer walks up the stack, transforming return addresses, function arguments, and other stack temporaries, simultaneously creating the live register state for each function invocation since the *libc* startup routine. The transformability of a live register or a stack object is decided based on one of the following scenarios.

- Its value is known at compile time, load time, or link time - e.g., constant literals.
- Its value can be found at a specific location - e.g, globals, immutable objects (aggregates, alloca variables and variables whose addresses have been taken).
- Cross referencing compiler-generated metadata could reveal that its intermediate name refers to a live register or stack object on the ISA from which we migrated.
- Its value can be computed using the already transformed live registers and stack slots. This involves a reverse traversal of the def-use chain to find a sequence of instructions that can re-compute the required value.

Owing to the bottom-up nature of the first pass, the state transformer transforms all inconsistent state except callee-saved registers since that information comes from live registers at ancestral function invocations. In the second pass, the state transformer walks down the stack fixing the callee-saved register spill area, by making use of the live register information obtained during the first pass. In addition, it also passes on unchanged live register values across function call sites, ultimately culminating in the construction of the target CPU register state.

3.4 Binary Translation

As a feature that allows instantaneous migration, the runtime performs binary translation on a migrated process until it reaches an equivalence point, at which point the state transformer described in the previous section transforms program state for native execution. This section describes the design of our binary translator for migration across cores implementing the ARM and MIPS ISAs. Our binary translator uses the following scheme of classic just-in-time (JIT) [DS84] dynamic translation:

- Starting from the instruction at the point of migration, each instruction is translated to the ISA of the migrated-to core and placed in a code cache until we encounter a function call site or an indirect/conditional jump instruction (whose target address is not known until execution).

- Next, a stub (a short group of additional instructions) is added to the end of this translated block of instructions. The stub contains a jump to the stack transformer if we've reached a function call site. Otherwise, the stub saves the target address at a known location and jumps to a translator core function called the *translation engine*.
- Control is then transferred to the translated code in the code cache. If the code eventually relinquishes control back to the translation engine, we repeat the above steps from the instruction at the target address until we finally reach a function call site.

3.4.1 Translation Block Chaining

The translation engine, before translating the next block of instructions, checks if the block is already available in the code cache. If it is available, it links the end of the previous block to the beginning of the next block, with a direct branch instruction. This process is known as translation block (TB) chaining and has been extensively applied in emulators and virtual machines.

We extend this idea by allowing translation block chaining from any instruction in the middle of a TB to any instruction in another TB. This allows for a TB to be chained to more than one TB at different instructions (the most common case is a conditional branch). For example, TB *X* can be chained with TB *Y* at *X.i*, with *Z* at *X.j* and with itself at *X.k*, where *X.i,j,k* represent three different instructions in X. The converse is also true: TBs *X* and *Y* can both chain to *Z* at *Z.i* and *Z.j* respectively. In this case, however, *Z.i* and *Z.j* can be the same. Merge point is a classic example for such a scenario, wherein the "if" part can be in TB *X*, "else" part in TB *Y*, and they both chain to TB *Z* at their merge instruction *Z.i*.

We call this *Multiple-Entry Multiple-Exit (MEME) translation block chaining*. The following issues need to be addressed by such a design:

Condition Codes. In a MEME TB, any instruction can have multiple entry points, including those that check condition codes. To improve performance, our binary translator performs lazy condition code evaluation, which defers evaluation of a condition code until it is checked. Any instruction that checks a condition code (CC) evaluates it first, if it has not already been evaluated by an instruction prior to that in the same TB. With MEME chaining, we can never be sure whether a CC has been evaluated or not, due to multiple entry points. Also, we do not know which instruction modified the CC in the first place, to perform lazy evaluation accordingly. To overcome these issues, we update a dirty CC map register at every exit point. The dirty map can be used by instructions to check if a CC has been already evaluated. In addition to this, we also store the opcode of the last CC modifier instruction in a register, so that at the time of lazy evaluation, we would know which instruction modified the CC.

Program Counter Updates. ARM allows instructions to use the program counter as a general-purpose register. For performance reasons, the (virtual) program counter is not updated after executing every block of target instructions that emulates a source instruction. It is instead updated whenever necessary. We further optimize this by adding an offset from the last-calculated PC rather than moving the entire 32 bit address (32 bit move takes at least two MIPS instructions). With MEME chaining, the last-calculated PC can be different for different entry points. To overcome this, we maintain a map of the last-calculated PC at every instruction in a TB. Using this map, the last-calculated PC is updated at the end of every exit point.

Instruction Scheduling. Instructions within a translation block might be reordered. We do not do instruction scheduling optimizations, but we do fill branch delay slots. Branch delay slots are filled with an independent instruction prior to the branch in the same TB. This is not always correct if the branch has multiple entry points. We handle branch delay slots in ARM to MIPS translation as follows:

- All register indirect branches trap into the translation engine.

- All direct branches are evaluated by the translator, which translates instructions at the target address and inserts them inline. If they are already translated, we do MEME chaining with the lazy PC update instruction in the branch delay slot, which has to be executed regardless of the entry point.

- All conditional branches are only dependent on condition codes which are taken care of by lazy CC evaluation. So the instruction just before the branch is not dependent on the branch and hence always goes into the delay slot. When MEME chaining happens at a conditional branch, we move back any instruction in the delay slot and insert a NOP into the delay slot instead. Performance reduction due to this is negligible compared to the significant gain in performance due to TB chaining.

Delay slots are not a problem in MIPS to ARM translation because ARM does not have delay slots.

As a further optimization, we have the ability to preserve the code cache across migrations, so that if the process is migrated to the same core type again, there is a good chance that the code we need is already present in the cache and can be directly used. However, we do not employ this optimization in our presented results.

3.4.2 ISA-Specific Challenges

Despite high-level similarities, ARM and MIPS represent significant diversity. ARM has many features that MIPS lacks: condition codes and an abundance of predicated instructions, load multiple and store multiple instructions, a program counter that is accessible as a general-purpose register, and finally PC-relative load instructions to access data embedded in the text section. MIPS, on the other hand, has double the number of integer registers that are accessible to programmers and allows for 16-bit immediates as opposed to the 8-bit immediate restriction in ARM. Finally, each ISA has a different *application binary interface* (ABI); they use different system call numbers and follow different conventions to make system calls. We discuss several of these challenges in this section.

Register Allocation. *Mapping from ARM to MIPS:* ARM has 16 general-purpose registers visible to the programmer, while MIPS has 31 general-purpose registers (excluding R0). Hence, all 16 registers in ARM are easily mapped to registers in MIPS. In addition, we reserve four MIPS registers for the ARM condition codes (Zero, Negative, Carry, and Overflow) and an extra register for inverse carry. Of the remaining 10 registers, four are used for lazy condition code evaluation as described below, three are used as temporary registers, and the remaining three are reserved for future use.

Mapping from MIPS to ARM: All 31 MIPS general-purpose registers cannot be mapped onto registers in ARM. Hence, we map frequently used MIPS registers (R1–R7, global pointer, stack pointer, frame pointer and link register, collectively called the "mapped" registers) to registers in ARM. One register points to an in-memory register context block where the "unmapped" MIPS registers are placed. In addition to this, we reserve three registers as cache registers that contain the three most frequently used unmapped registers for faster access.

Condition Codes and Predicated Instructions. Most translators use a global data flow analysis technique to perform a lazy evaluation of condition codes. However, since we perform binary translation for a relatively small number of instructions until we reach an equivalence point, a data flow analysis would increase migration overhead significantly. Hence, we use a lazy condition code evaluation scheme similar to the one used in QEMU [Bel05]: for every instruction that updates a condition code, we store the opcode, operands, and result in temporary registers reserved for the lazy evaluation, and compute the condition codes using this information whenever required. In addition to this, a dirty map register is used to support MEME TB chaining as described above.

Once the condition codes necessary for a predicated instruction are evaluated, a branch instruction is used to test the condition, which skips the operation performed by the instruction if the condition is false. The conditional move instruction in MIPS is used to translate conditional move instructions in ARM, but we use a conditional branch around arithmetic instructions for more complex predicated instructions.

Immediate Instructions. MIPS restricts the size of immediates to 16 bits while ARM limits them to 8 bits. This necessitates two ARM instructions to construct a MIPS immediate, store it in a register, and then perform the actual operation. The problem worsens with 32 bit immediate updates like link register or program counter updates, where four ARM instructions are needed to perform the move. To overcome this, we extend the register context block to also include a "cache of immediates", so that one load instruction will suffice for the entire operation, as opposed to the four mutually dependent shift-OR instructions.

System Calls. In this work, we assume a single operating system instance running on both the cores. This guarantees that a system call works in the same way on both the cores. However, the system call numbers and calling conventions are dictated by the ISA's ABI, which requires a remapping step when in binary translation mode. One approach would be to provide system call emulation during translation. As an alternative, it is a minor change to our system to also make system calls equivalence points, like function calls. This eliminates the need for system call emulation, but also increases the frequency of equivalence points, which is also a useful feature. To support migration while executing a system call, we would need to apply our techniques and methodology to the operating system itself.

3.5 Experimental Methodology

We use the SPEC2000 Integer C benchmarks to evaluate our migration strategy. We exclude the *gcc* benchmark because it uses the *alloca* library function to dynamically allocate memory on the stack, creating variable-size stack frames, a feature that is not supported by our infrastructure. All benchmarks are compiled with all optimizations turned on using the multi-ISA

Table 3.1: Architecture detail for ARM and MIPS cores

ARM core			
Frequency	833 MHz	I cache	32 KB, 4 way
Fetch/commit width	2	D cache	32 KB, 4 way
Branch predictor	local	L2 cache	2 MB, 8 way
MIPS core			
Frequency	2 GHz	I cache	64 KB, 4 way
Fetch/commit width	4	D cache	64 KB, 4 way
Branch predictor	tournament	L2 cache	4 MB, 8 way

compilation strategy described above. All experiments are performed on CPU cores modeled after the low-power Cortex-A8 core for ARM, and the high performance R10000 for MIPS. We use the gem5 architectural simulator [BDH$^+$06] to model the CPU cores. The details of each core are given in Table 3.1.

To evaluate the steady-state performance degradation due to multi-ISA compilation, we simulate each program phase (simpoint) of a benchmark compiled for both single-ISA execution and multi-ISA execution.

Migration cost consists of two major components: dynamic binary translation and program state transformation. On every ISA, we take 10 samples of the benchmark's dynamic execution state, each at a 100 million instruction interval, after fast-forwarding execution for the first one billion instructions [SPHC02]. We then simulate heterogeneous-ISA migration scenarios for each sample, by performing dynamic binary translation until an equivalence point is reached, and program state transformation from thereon.

3.6 Results

3.6.1 Steady State Performance

A key goal of this work is to enable fast migration without compromising runtime performance—that is, performance when no migration is occurring. Throughout the toolchain changes to ensure a nearly identical memory image across architectures, care must be taken that performance is not compromised. Among our benchmarks no performance is lost due to changes to make the memory image consistent, including the addition of padding (which is too little to cause more instruction cache misses). Furthermore, the symmetrical fat binary created by our compiler contains multiple code sections, but a core only loads its own code to its private instruction cache. In fact, this can only impact a shared cache. If we change our design to have a shared 16MB LLC, and incorporate the increased working set size, we measure zero performance loss. Overall, we observe no performance degradation at all due to multi-ISA compilation, up to four decimal places of the IPC. Such small degradation of performance comes from the fact that we do not disable any optimization or ISA-specific behavior to enable multi-ISA compilation.

3.6.2 Migration Cost

Migration cost is composed of two major components: binary translation and stack transformation. We characterize the performance of our binary translator based on the following three metrics:

- **Target-to-Source Ratio:** The ratio of the number of dynamic target instructions executed on the migrated-to core to the number of dynamic source instructions executed on the native core. Most static binary translators report a target-to-source ratio between one and two. Dynamic binary translators tend to have a higher target-to-source ratio because they have less scope for optimization.

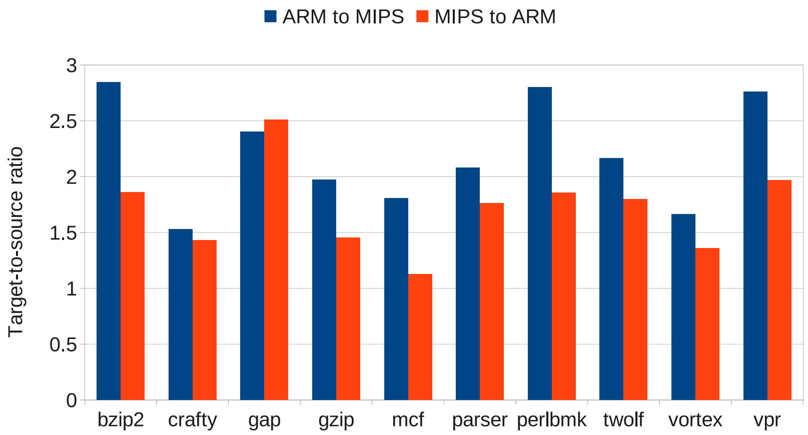

Figure 3.3: The ratio of the number of target instructions executed during binary translation to the number of source instructions during native execution.

- **Total-to-Source Ratio:** The ratio of the number of dynamic instructions inclusive of both the target instructions and the instructions used for translation (as executed on the migrated-to core) to the number of dynamic source instructions executed on the native core. This is the ratio we address with our translation block chaining algorithms.

- **Overhead due to binary translation:** Time taken for binary translation compared against time taken on native core, in microseconds.

Figure 3.3 shows the target-to-source ratio for each benchmark in both directions of migration. The target-to-source ratio for MIPS to ARM translation is generally less than that from ARM to MIPS, due to the more complex instructions in ARM. One exception to this is *gap*, which has a higher ratio for MIPS to ARM translation. This is because *gap* is characterized by tight loops with multiply instructions, which takes additional instructions in ARM to perform stores to “hi” and “lo” memory locations. In both directions, *bzip2*, *perlbmk*, and *vpr* have high target-to-source ratios because of a large number of branches in MIPS code and large number of predicated instructions in ARM code.

Figure 3.4 shows the target-to-source ratio for ARM to MIPS translation without optimization and then with each optimization applied incrementally. The leftmost bar shows the

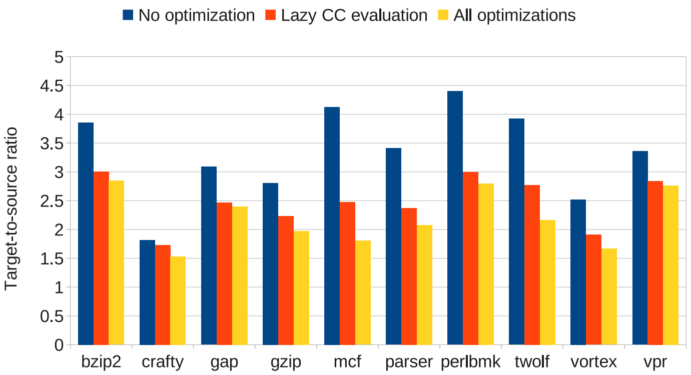

Figure 3.4: Target-to-Source ratio during Binary Translation from ARM to MIPS—without optimization, with lazy condition code evaluation and with full optimization.

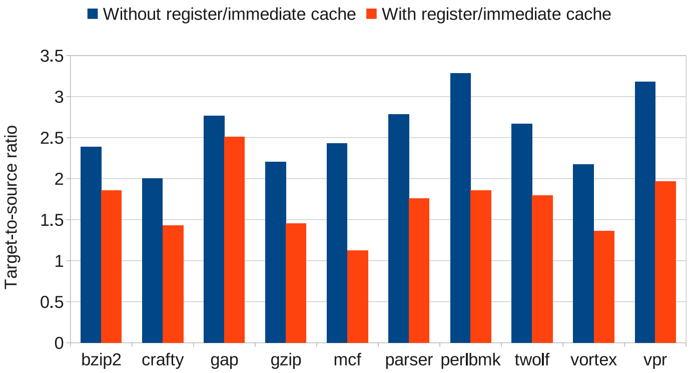

Figure 3.5: Target-to-Source ratio during Binary Translation from ARM to MIPS—with and without Register and Immediate caches

performance of a naïve binary translator without any optimizations. The middle bar shows the performance of the binary translator doing lazy condition code evaluation. This optimization significantly reduces the dynamic instruction count. The rightmost bar shows the binary translator with all optimizations enabled. This includes grouping predicate instructions and certain constant-folding optimizations. Lazy condition code evaluation contributes the most to the overall translation speedup.

Figure 3.5 shows the performance of the MIPS to ARM translator with and without the use of the register cache and immediate cache. Dynamic instruction count is significantly lower with these optimizations. Our register cache is made up of only three temporary registers. We

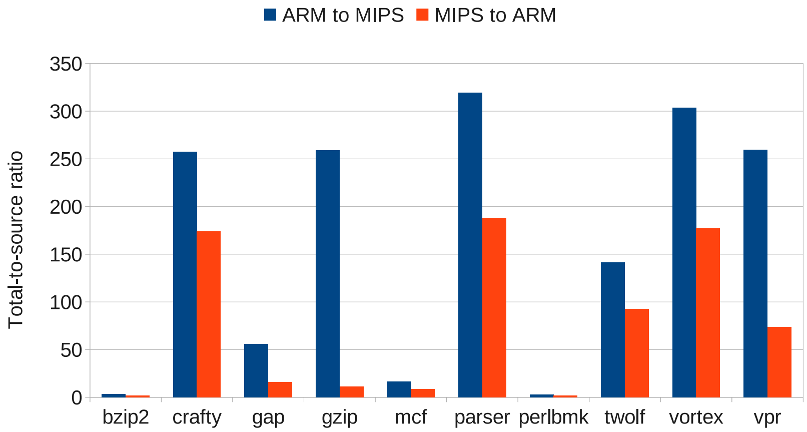

Figure 3.6: The ratio of the number of dynamic instructions executed (including the ones used for translation) during binary translation to the number of source instructions during native execution.

expect that a larger cache with an adaptive register allocation strategy should give even greater speedups.

Figure 3.6 shows the total-to-source ratio for each benchmark in both directions of migration. Again the MIPS to ARM translator has a lower translation cost than ARM to MIPS, because it does not have to make complex decisions like lazy condition code evaluation and lazy PC update during translation. However, there is a high irregularity in the total-to-source ratios of different benchmarks—some are as high as 300 while some are close to one. This is heavily impacted by the number of instructions to the next equivalence point (call site). If it is 84 instructions (the average for *vpr*), the cost of translation is not amortized. If it is hundreds of millions of instructions, the vast majority of execution is in code cache and the translation cost is insignificant. The latter is due, in large part, to the MEME chaining which allows execution to remain in the code cache once a steady state is reached. Table 3.2 shows the percentage of dynamic instructions executed in the code cache during binary translation.

We next compare the performance of our binary translator with native execution in Figure 3.7. Within the migration points we sample, the average execution time for an ARM binary on an ARM core from the time migration is requested until an equivalence point is reached

Table 3.2: Percentage of instructions used from code cache

Benchmark	ARM to MIPS	MIPS to ARM
bzip2	99.9992	99.993
crafty	0.0	0.0
gap	85.9	94.0
gzip	18.1	99.9
mcf	99.96	99.1
parser	15.7	30.7
perlbmk	99.997	99.99
twolf	58.9	65.4
vortex	0.0	0.0
vpr	36.6	67.1

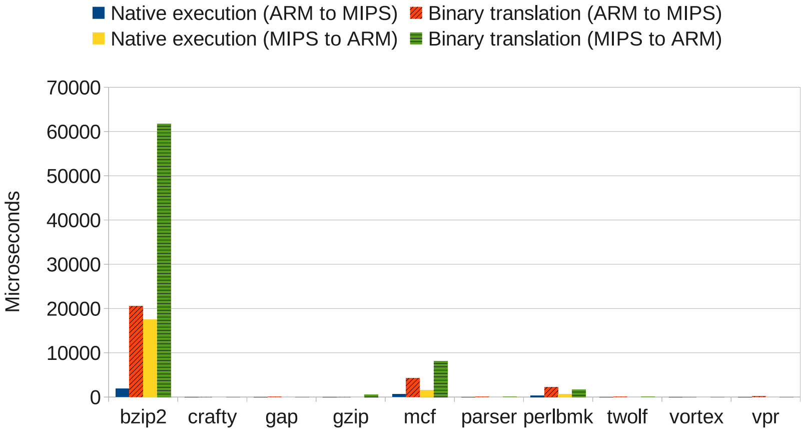

Figure 3.7: Comparison of Binary Translation time with Native Execution time in microseconds

is 284 microseconds, while the average binary translation time of the ARM binary on a MIPS core is 2745 microseconds. For ARM to MIPS, the average execution time on a MIPS core is 1981 microseconds while binary translation time on an ARM core is 7240 microseconds. Thus, binary translation costs us 2461 microseconds while migrating from ARM to MIPS and 5259 microseconds while migrating from MIPS to ARM. All benchmarks except *bzip2*, *perlbmk*, and *mcf* complete binary translation in tens or hundreds of microseconds. These three benchmarks show a high binary translation overhead because they have to translate millions of instructions before reaching an equivalence point.

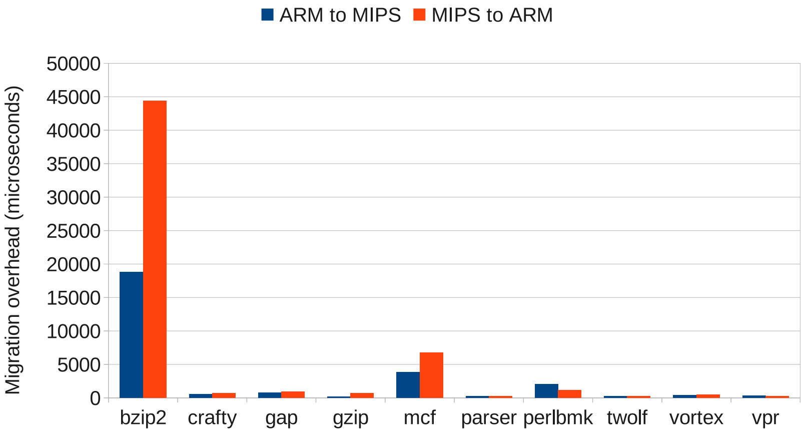

Figure 3.8: Migration Overhead due to Binary Translation and Stack Transformation in microseconds

Finally, we evaluate the performance of our migration strategy by looking at the total migration overhead—time taken by both binary translation and stack transformation. This is shown in Figure 3.8. The average migration overhead for ARM to MIPS migration is 2734 microseconds, while it is 5602 microseconds for MIPS to ARM migration. It should be noted that this average is dominated by a few outliers. If we ignore *bzip2*, *mcf*, and *perlbmk*, the average overhead drops by about a factor of 10.

Figure 3.9 shows performance (relative to native execution without migration) at different migration frequencies (when migrating back and forth between cores at fixed time intervals), assuming average cost values. It breaks down the costs due to compilation for migratability, state transformation, and binary translation. With all costs considered, execution is 95% as fast as native execution when migration occurs, on average, every 87 milliseconds. This would be an extremely high rate of migration for most foreseeable applications. Also recall that much of that 5% lost performance is an artifact of the compiler not being designed from the ground up to emit multi-ISA code.

We believe this level of performance crosses a critical threshold. Unless the code is migrating between cores nearly every timer interrupt, the cost of migration is very small. This means that

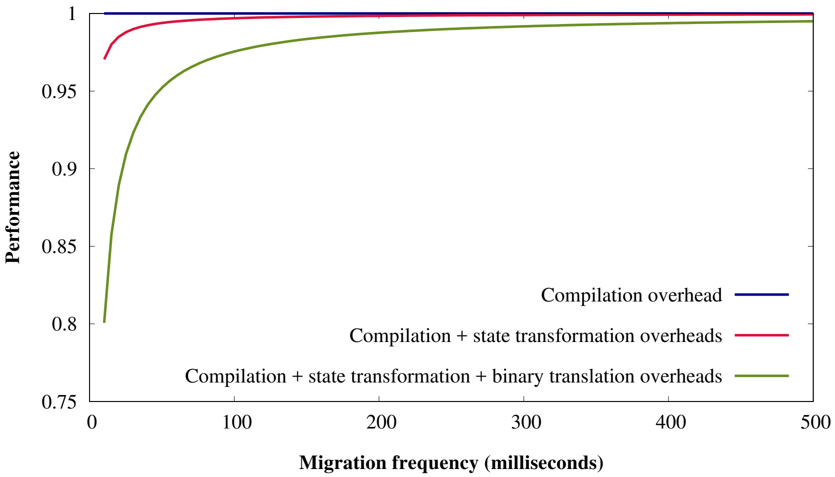

Figure 3.9: Performance vs. migration frequency when migrating back and forth between an ARM core and a MIPS core. Performance includes overheads due to compilation for migratabiltity, state transformation, and binary translation.

the difference in migration cost between a single-ISA heterogeneous CMP and a multi-ISA CMP is negligible for most reasonable assumptions about desired migration frequency. Thus, there is no significant performance barrier to fully exploiting heterogeneity in a multicore architecture, including both microarchitecture heterogeneity and ISA heterogeneity. For comparison, recall that prior work typically measured migration time in hundreds of milliseconds [FCG00], if not worse, and that migration could not occur at an arbitrary point in execution.

3.7 Conclusion

This chapter establishes the viability of heterogeneous-ISA chip multiprocessors by presenting a cross-ISA process migration technique that is orders of magnitude faster than prior art. The chapter describes a multi-ISA compilation technique that leverages LLVM's intermediate representation to create a symmetrical fat binary that ensures memory consistency across all ISAs by using the same endianness, data type size, alignment and padding rules in global data, heap and text sections, and certain portions of the stack. The chapter also describe a runtime mechanism to transform inconsistent state on the stack, and compute the target CPU registers post migration. To support instantaneous migration, the runtime also performs dynamic binary translation till a point in execution, called the *equivalence point* is reached, at which program state can be successfully transformed. The proposed technique incurs an average binary translation cost of 2.75 milliseconds for ARM to MIPS migration and 7.24 milliseconds for MIPS to ARM. Finally, this chapter shows that even under a frequent migration interval of every few hundred milliseconds, the total loss in performance is well under 5%.

Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in proceedings of ASPLOS 2012. DeVuyst, Matthew; Venkat, Ashish; Tullsen, Dean M., Execution Migration in a Heterogeneous-ISA Chip Multiprocessor, 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March, 2012. The dissertation author was the primary investigator and author of this paper.

The dissertation author would also like to thank Matthew DeVuyst for his useful insights and early contributions to the multi-ISA compiler framework based on GCC.

Chapter 4

Design of a Heterogeneous-ISA Chip Multiprocessor

The cross-ISA process migration infrastructure described in Chapter 3 now enables programs to cross a heretofore forbidden boundary – the Instruction Set Architecture. Existing processor architectures either feature a single ISA or cores with multiple ISAs that assign distinct jobs to different cores, or at best statically partition the work. This chapter demonstrates that not only is that assumption unnecessary, but it restricts the potential heterogeneity, sacrificing performance and energy efficiency gains. It motivates the need for ISA diversity, explores the design space of heterogeneous-ISA CMPs characterized by three diverse ISAS (ARM's low-power Thumb, the traditionally RISC Alpha, and the high-performance x86-64) and a multitude of microarchitectural parameters, and further demonstrates the effectiveness of a heterogeneous-ISA architecture in terms of its performance and energy efficiency.

4.1 Harnessing ISA Diversity

To keep both the design-space exploration and the compiler development tractable, we select our target ISAs a priori – considering more ISAs and even considering the possibility of custom ISAs would only increase the potential gains. This section describes our three target ISAs – Thumb, Alpha, and x86-64 – with respect to several axes of diversity. These include code density, dynamic instruction count, register pressure, and support for specialized operations.

Code Density. High code density reduces the number of instruction cache misses, uses less energy and memory bandwidth for instruction fetch, and conserves power by enabling the use of smaller microarchitectural structures. Weaver, et al. [WM09] evaluate a wide range of ISAs for code density. They find that RISC ISAs with fixed-length instructions such as Alpha and SPARC show the lowest code density, while embedded ISAs like Thumb and AVR32 exhibit the highest density owing to a technique called code compression. This technique packs two 16-bit instructions into one 32-bit instruction, which is then unpacked at the decode stage and executed as two instructions. CISC ISAs such as x86-64 and VAX are placed in the middle of the code density spectrum by virtue of variable-length instruction encoding.

Dynamic Instruction Count. While code compression achieves about 32.5% memory savings in Thumb, it increases the dynamic instruction count by 30% [KG05]. This is a direct consequence of using simpler 2-operand instructions to fit Thumb's 16-bit instruction. Thumb instructions also lack the shift-modifier and predication support that ARM instructions enjoy. Alpha employs 3-operand instructions, but is a load-store architecture, meaning that no arithmetic instruction can directly operate on memory. While x86-64 also restricts instructions to the 2-operand format, it implements a number of complex addressing modes that allow instructions to directly operate on memory. x86-64 instructions are decoded into one or more simpler RISC-like μops, thereby increasing the number of dynamic instructions (μops) by about a factor of 1.3 [BMS13a] (as compared to the native x86-64 instruction count).

Register Pressure. Thumb uses a reduced register set, allowing only eight 32-bit programmable registers for integer operations. Thus, all 64-bit integer computation is performed using software emulation. Software emulation is discussed in greater detail in Section 4.3. Alpha, on the other hand, has two banks of thirty-two 64-bit programmable registers, for integer and floating-point computation. x86-64 offers sixteen 64-bit registers for integer operations and sixteen 128-bit registers for floating-point and SIMD operations.

The number of programmable registers is inversely proportional to the amount of register pressure, and thus the number of register spills, for any ISA. Therefore, Thumb suffers from extremely high register pressure, while Alpha enjoys low register pressure. Interestingly, x86-64 enjoys the lowest register pressure among the three ISAs, despite the fact that it has a smaller architectural register file than Alpha. This is a manifestation of the following addressing modes and optimizations: *Absolute memory addressing* allows instructions to directly access memory operands, eliminating the need to allocate registers for temporary storage of loaded values. *Sub-register addressing* allows programmers to address 48 sub-registers to store/operate on smaller data types, which can be further exploited by aggressive sub-register coalescing strategies to reduce the number of register spills. *Program counter relative addressing* enables position-independent code without the overhead (both in performance and allocated registers) of a Global Offset Table. Lastly, *register-to-register spills* allow programmers (compilers) to spill general-purpose registers to XMM registers, thereby minimizing the number of register spills into memory.

Figure 6.2 shows Thumb instructions and x86-64 μops normalized to Alpha instructions, for the SPEC2006 integer benchmarks, compiled using the LLVM/Clang framework [LA04b, Lat08]. The average number of dynamic instructions on Thumb is 43.4% more than that of Alpha. This is due, in large part, to the high register pressure in Thumb. In fact, Thumb makes 91.1% more memory references than Alpha. Other factors include 64-bit emulation and the use of simpler 2-operand instructions.

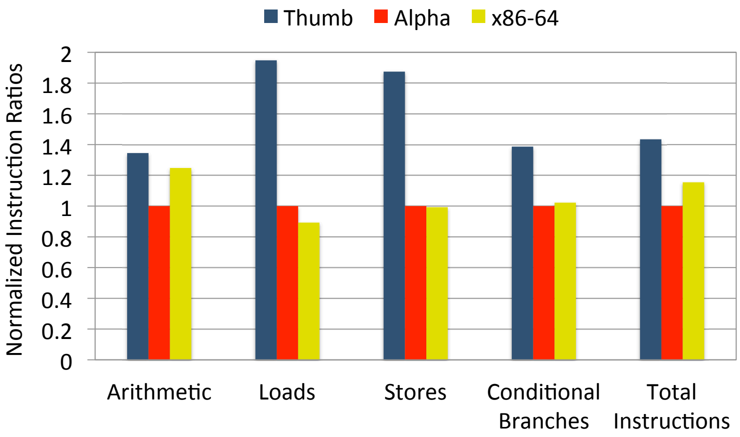

Figure 4.1: Instruction mix (normalized to Alpha) for SPEC2006

x86-64 makes 8.3% fewer memory references than Alpha due to lower register pressure. However, we observe that there is a very small (0.8%) reduction in the number of stores, while the number of loads drops by 10.8%. The compiler generally seeks to spill variables that won't be updated for a long period of time. Therefore, the number of reads from the spill area is much higher than the number of writes.

Interestingly, Alpha makes 10.9% fewer memory references on the high ILP benchmarks *bzip2* and *hmmer*, in which case the compiler does not find enough opportunity to utilize the complex addressing modes and optimizations offered by x86-64. The 2-operand restriction contributes to the 24.7% more arithmetic instructions on x86-64, resulting in an overall increase in the number of dynamic instructions of 15.4%.

Floating-point and SIMD Support. One consequence of code compression is that floating-point instructions are not supported in Thumb. Floating-point operations are emulated in software. While emulation results in slower execution, Thumb cores don't need to include floating-point instruction windows, register files, and functional units, resulting in up to 19.5% reduction in peak power and 30% savings in area. In a heterogeneous-ISA architecture, any program or phase with significant floating-point activity will likely quickly switch to an ISA that executes natively.

x86-64 also provides SIMD support through its SSE/AVX extensions, making vectoriza-

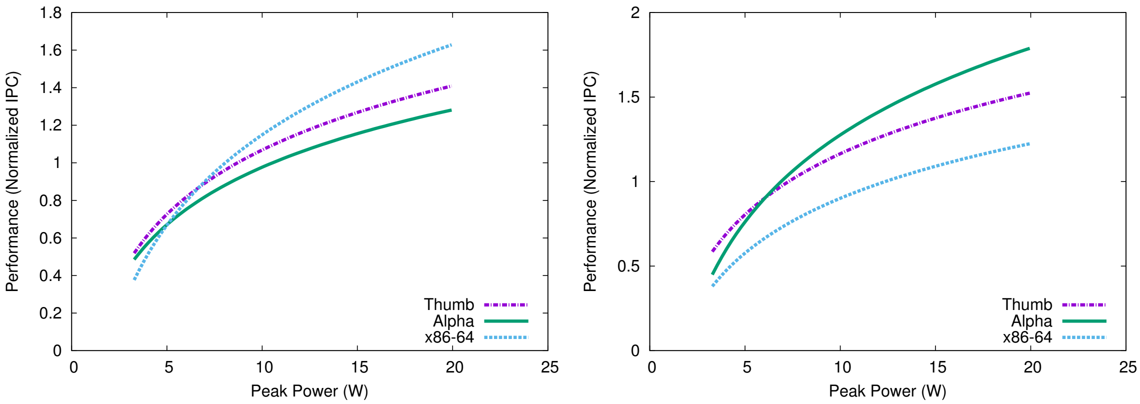

Figure 4.2: Performance comparison under different peak power budgets for two different execution phases of bzip2

tion of loops and basic blocks possible. Alpha's MVI extension allows for only pack, unpack, max, and min operations. Due to the very primitive nature of the MVI extension, we forgo SIMD units in Alpha cores.

To illustrate the benefits of heterogeneity, even on a single application, we examine the performance of *bzip2* during two different phases of its execution (in Figure 4.2), under varying power constraints. We identify program phases using SimPoint [PHVB$^+$03]. The detailed methodology is described in Section 5.5. We see two key results in this graph. First, we see that the most effective ISAs differ between phases of the same application; e.g., at 15 W, Phase 1 prefers x86 and Phase 2 prefers Alpha. Second, we see that even in a single phase, the best ISA varies depending on the design constraints or the operating condition of the processor. For example, in Phase 2, we might prefer Alpha unless we are operating unplugged or perhaps in a low battery state, in which case we'd prefer Thumb because at low power budgets, Thumb provides the highest performance.

Table 4.1: Design space of a Heterogeneous-ISA architecture

Design Parameter	**Design Choices**
ISA	Thumb, Alpha, x86-64
Execution Semantics	In-order, Out-of-order
Issue width	1, 2, 4
Branch Predictor	local, tournament
Reorder Buffer Size	64, 128 entries
Architectural Register File	ISA-specific
Physical Register File (Integer)	96, 160
Physical Register File (FP/SIMD)	64, 96
Integer ALUs	1, 3, 6
Integer Multiply/Divide Units	1, 2
Floating-point ALUs	1, 2, 4
FP Multiply/Divide Units	1, 2
SIMD Units	1, 2, 4
Load/Store Queue Sizes	16, 32 entries
Instruction Cache	32KB 4-way, 64KB 4-way
Private Data Cache	32KB 4-way, 64KB 4-way
Shared Last Level (L2) Cache	4-banked 4MB 4-way, 4-banked 8MB 8-way

4.2 Design Space Exploration

The possible design space of a heterogeneous-ISA CMP is characterized by a diverse set of ISAs and a multitude of microarchitectural parameters. Navigating such a design space is a difficult problem. That difficulty can be reduced and pruned if we understand some of the principles that govern the effective co-design of heterogeneous-ISA, heterogeneous hardware architecture processors. To do this, we execute an exhaustive design space exploration of an architecture with fairly limited, tractable options, and observe the characteristics of the best designs.

The design space we explore in this work includes the three ISAs - Thumb, Alpha, and x86-64, along with a set of micro-architectural parameters that represent a wide range of performance and power control points. The goal of the design-space exploration is to find the optimal 4-core heterogeneous-ISA CMP for varying power and area budgets, and considering all permutations of applications in the workload sharing the cores.

Table 4.2: Pruned design space for faster navigation

Design Parameter	**Design Choices**
ISA	Thumb, Alpha, x86-64
Execution Semantics	In-order, Out-of-order
Branch Predictor	local, tournament
Reorder Buffer-Register File	64-96-64, 128-160-96 entries
Issue Width-Functional Units	1-1-1-1-1-1, 1-3-2-2-2-2, 2-3-2-2-2-2, 4-3-2-2-2-2, 4-6-2-4-2-4
Load/Store Queue Sizes	16, 32 entries
Cache Hierarchy	32K/4-32K/4-4M/4, 32K/4-32K/4-8M/8, 64K/4-64K/4-4M/4, 64K/4-64K/4-8M/8

Table 6.1 enumerates the variables in our design space. The Cartesian product of this design space consists of 750 thousand single core combinations, making it not practically feasible to perform an exhaustive search. To reduce the size of the Cartesian product, we prune the design space by establishing correlations between different variables. While some correlations can be inferred by intuition, others are dictated by specific ISA characteristics: (1) Size of the reorder buffer can be correlated to the physical register file size, as they together establish the window size. (2) The number of functional units varies with the issue width. (3) Thumb cores need not include a floating-point instruction window, retirement units, register files or functional units. (4) Neither alpha nor thumb need include SIMD functional units.

The resulting pruned design space, as shown in Table 4.2, contains 120 in-order cores and 480 out-of-order cores. However, the number of possible 4-core configurations in the pruned design space is still very high (129.6 billion configurations). To further make this problem tractable, we use the following results from prior research on single-ISA heterogeneous architectures – modeling using private LLCs versus a shared *n*-banked LLC in an *n*-core configuration, results in the same performance ordering with respect to all *n*-core configurations [KTJ06], as well as different scheduling/migration strategies [VCJE$^+$12]

Therefore, we model cores using 1MB 4-way or 2MB 8-way private last-level caches,

Table 4.3: Memory Management on Thumb, Alpha and x86-64

ISA	Thumb	Alpha	x86-64
Page Table Hierarchy	2-level	3-level	4-level
Page Size	4KB, 64KB	8KB	4KB, 2MB, 1GB
Page Table Size	16KB first-level, and 1KB second-level	8KB	4KB
TLB Update Mechanism	Hardware page table walker	Low-level firmware (PALcode)	Hardware page table walker

instead of a single shared 4MB or 8MB cache, respectively. Thus, the combined performance of a 4-core configuration with private LLCs can be computed using the sum of the performances of the individual cores. While the design space exploration still involves finding the optimal 4-core configuration out of 129.6 billion different configurations, we can now find the best design with 600 simulations of the single-core permutations, and a software search of the 130 billion sums.

4.3 Programming Environment and Memory Layout

The programming environment for a heterogeneous CMP is dictated by one of the first design choices: separate address space [teg10, teg11] vs unified address space [Gre11, KFJ$^+$03]. We contend that the full benefits of a heterogeneous multicore architecture can be reaped only through dynamic core selection, which requires process migration. Separate address space constraints impose a significant cost to process migration in terms of program state transfer. Therefore, we choose the unified address space model in our design. However, this presents a unique challenge to compilation and process migration, because the memory layout and runtime state of a program is always architecture-specific. The process migration problem has been largely addressed in Chapter 3, in terms of compilation and runtime techniques. This chapter presents a memory management strategy to facilitate a unified address space.

Address Translation. A common address translation mechanism is required to ensure a unified address space in a heterogeneous-ISA environment. From Table 4.3, Alpha emerges as the lowest common denominator due to its 8KB page size. While it is possible to use software virtualization for 8KB page management on Thumb and x86-64, it necessitates the use of multiple page table structures, one for each ISA. Furthermore, software virtualization cannot enable 64-bit virtual address translation on the 32-bit Thumb architecture. Therefore, we use a common page table structure and an MMU based on the 4-level page table walker of x86-64, for all the three ISAs.

Long mode emulation on Thumb. Long mode (64-bit) computation in Thumb is performed using software emulation. Most compilers already support this to perform arithmetic and memory operations on the "long long" data type. The general procedure is to use multiple 32-bit registers to construct 64-bit values and compute on them.

To support memory operations using 64-bit pointers (virtual addresses), we extend the Thumb ISA to include special instructions: LD64 and ST64. These instruct the MMU to look for the higher-order 32-bits of its 64-bit virtual address input in a special register *R8*. However, the memory footprint of most general-purpose workloads seldom exceeds 4GB. In fact, we observe that no SPEC CPU2006 benchmark acquires more than 4GB of memory. Therefore, wherever possible, we use the regular load/store instructions with 32-bit pointers, which are zero-extended by the MMU during address translation.

4.4 Experimental Methodology

In this section, we describe our experimental methodology. Our four-core design space consists of 600 homogeneous processors, 1.5 billion single-ISA heterogeneous processors, and 128.3 billion heterogeneous-ISA chip multiprocessors, that can be each designed out of 600 distinct CPU cores.

Although prior work on ISA characterization [BMS13b, IJJ09, Ter11] chooses to model cores based on actual commercial offerings for each ISA, we seek to remove any non-ISA biases and start each design with a clean slate. Thus, we assume the same basic pipeline design (number of stages and latency), based on the Alpha 21264 [Kes99], across all ISAs (with the exception of instruction decode, which will be the primary stage(s) that depend on the ISA). Additionally, we assume a total-store-order (strictest of all) memory consistency model for all ISAs.

All cores are modeled using 32nm technology and the clock rate is fixed at 1.67GHz. Owing to ISA diversity and the multitude of micro-architectural parameters we consider in this work, the heterogeneous-ISA CMPs are distributed over significant peak power (8.32-80.69 W) and area (32.97-129.87 mm^2) ranges. We use the gem5 [BDH+06] simulator to model CPU core performance, and McPAT [LAS+09a] to model power and area.

Our design methodology selects the optimal multicore configuration over the entire set of workloads (all possible permutations), for different peak power and area budgets. The design space explorations are optimized for two types of workloads: (1) multi-programmed mixed workloads, and (2) single-threaded workloads. The former helps us evaluate the throughput of a conventional CMP, the latter gives us insight into a "Dark Silicon" implementation, where it is expected that only one core (out of a heterogeneous cluster) will be powered up at once [EBA+11, KFJ+03, VSG+10]. In the latter case, a thread will always be assigned it's best core, but in the former case, it will depend on the threads with which it is co-scheduled. We will also examine both the case where threads are placed based on overall execution characteristics (assuming minimal migration), and the case where threads can migrate to other cores at phase changes. That is, in the first case we find the best assignment of applications to cores, in the second, we find the best assignment of phases to cores.

We use the SPEC CPU2006 integer and floating-point C benchmarks to evaluate the proposed architecture. We exclude *h264ref* and *perlbench* from this set because they use ISA-specific programming constructs (e.g., inline assembly), either directly or through library function calls. All benchmarks are compiled at the -O3 optimization level, using the multi-ISA compilation methodology described in Chapter 3. We use SimPoint [PHVB+03] to identify program phases.

Specifically, we obtain multiple simulation points for a program's execution on Alpha, with an interval size of 100 million dynamic instructions. We modify the *atomic* CPU (instruction emulation mode) of gem5 to emit the start and end basic blocks for each simulation point, and their cumulative frequency, which serve as the start and end markers for the corresponding program phase on the other two ISAs, namely thumb and x86-64.

Our workloads include a total of 72 different program phases on the 10 applications we benchmark. When searching for an optimal (e.g. 4-core) design, we consider all permutations of the benchmark set. Note that these experiments seek to find the best designs (best combination of cores) without considering migration cost. To measure the cost of migration, we consider a phase-based scheduling scenario, where migration happens only when phase transitions demand switching to a different core. To identify phase transitions, we rely on SimPoint metadata and profiling information from oracle experiments. This models a system where the compiler is directing migration (or at least migration preferences), or a runtime or hardware system that had been observing execution long enough to accurately detect phase behavior.

4.5 Results

This section seeks to identify the best heterogeneous designs for a given workload. This not only enables us to quantify the potential gains for ISA heterogeneity, but also identify trends and insights from the actual designs that get tagged as optimal. Because the nature of the design exploration is relatively independent of the cost of migration, only results later in this section account for the specific costs of migration, and the extent to which they mitigate the potential gains.

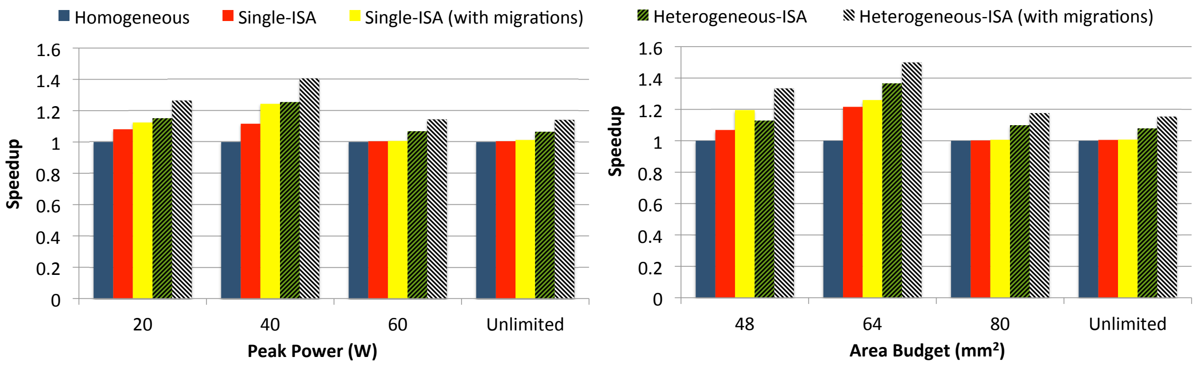

Figure 4.3: Multi-programmed Workload Performance comparison under different peak power and area budgets

4.5.1 Evaluation of the Heterogeneous-ISA Architecture

Processor designs today are as likely to be constrained by power dissipation as they are by area. Thus, in this section, we examine the top-performing designs under both area and power constraints. In addition to finding the best heterogeneous-ISA design, we also find the best homogeneous design (best single configuration for any ISA) and best single-ISA heterogeneous design (best heterogeneous design for which all cores have the same ISA). We consider designs optimized for both multi-programmed workload throughput and single-thread performance.

Multi-programmed workloads. Figure 4.3 compares three architectures: homogeneous, single-ISA heterogeneous, and heterogeneous-ISA CMPs, all optimized for multi-programmed workload performance under different peak power and area constraints. We make several important observations here. First, there are significant gains available from ISA heterogeneity, matching or exceeding the gains from hardware heterogeneity. Second, hardware heterogeneity alone is less effective under tight constraints (all cores have to be small) or liberal constraints (all cores free to be big), because both endpoints tend toward homogeneous designs. Heterogeneous-ISA designs, in contrast, are still effective in those regions, because we can still gain from ISA heterogeneity even when the hardware is homogeneous.

There are two reasons for this advantage. First, different code regions have a natural affinity for one ISA or another, irrespective of the hardware implementation. Second, the ISA

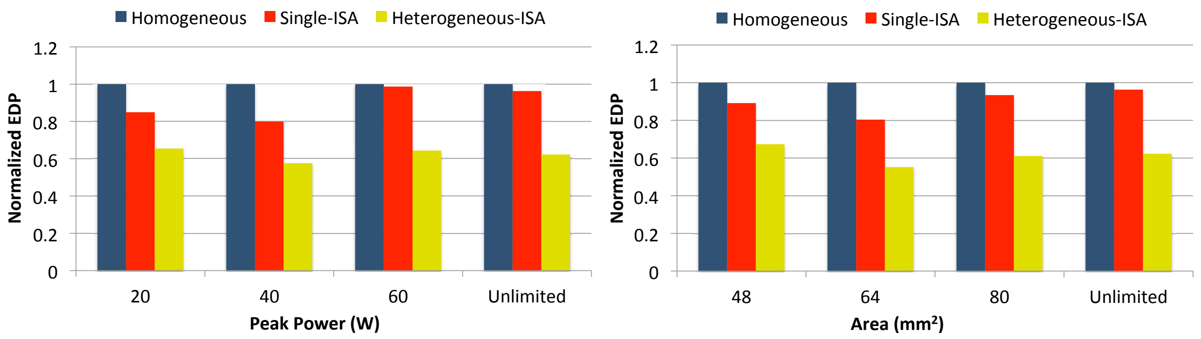

Figure 4.4: Energy-Delay-Product comparison for multi-programmed workloads under different peak power and area budgets

options give the architect more opportunities to create area-effective or power-effective cores.

For instance, at a peak power budget of 20W, the single-ISA heterogeneous CMP manages to employ only 3 out-of-order cores with smaller 32KB L1 caches, while the heterogeneous-ISA CMP sports all out-of-order cores with 64KB L1 caches. This is possible because the area-efficient and power-efficient Thumb cores free up space that is put to good use by the other cores. We find that heterogeneous-ISA CMPs can provide 15.8% better throughput on multi-programmed workloads than the best single-ISA heterogeneous CMPs.

Furthermore, we can achieve a greater speedup if applications are allowed to migrate between the cores at phase boundaries. This is well-documented in the case of single-ISA heterogeneous CMPs [KFJ$^+$03, KTR$^+$04]. On a heterogeneous-ISA CMP, this effect is further enhanced due to ISA affinity. We observe an additional speedup of 11.2% due to migration alone on a heterogeneous-ISA CMP, in contrast to the 4.6% speedup due to migration on a single-ISA heterogeneous CMP. In all subsequent experiments in this chapter, migrations are always enabled.

In order to evaluate energy efficiency, we instead optimize the design space exploration to find energy-efficient cores by identifying the processor configurations that minimize energy-delay product (EDP). Figure 4.4 compares the energy efficiency for the three architectures under different peak power and area budgets. Heterogeneous-ISA CMPs achieve an average energy savings of 21.5% and an average reduction of 27.8% in the EDP over single-ISA heterogeneous

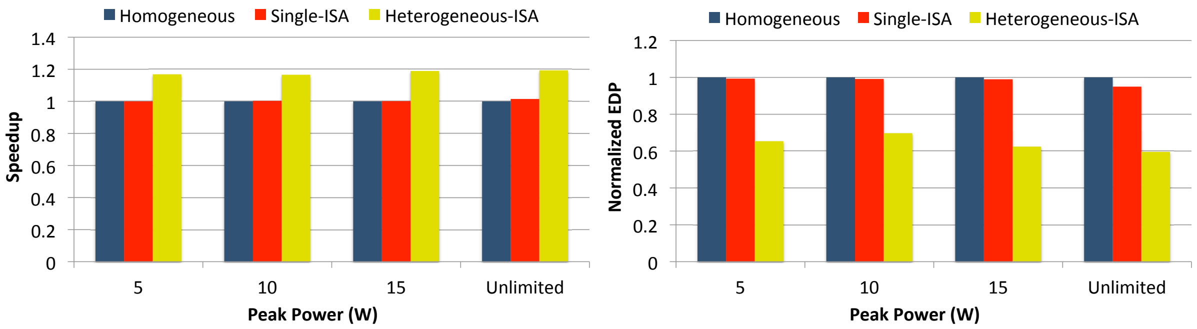

Figure 4.5: Single Thread Performance and EDP evaluation using the dynamic multicore topology

CMPs, with absolutely no loss in performance – that is, we gain performance and decrease energy simultaneously when we employ multi-ISA heterogeneity.

Thus we see that the energy efficiency gains of ISA heterogeneity actually exceed the potential performance gains (for the performance-optimized experiments). Maximizing heterogeneity in this way can be particularly effective in a power-constrained environment. In a homogeneous-ISA general-purpose processor, Thumb is not a serious candidate, because it performs so poorly for certain codes; however, as part of a heterogeneous solution, it shines for certain code regions.

Single-threaded workloads. To evaluate designs optimized for single-threaded workloads, under different peak power budgets, we assume the *dynamic multicore* topology described by Esmaeilzadeh, et al. [EBA+11], in which idle cores are turned off to reduce power consumption. Figure 6.7 shows performance and EDP measurements for the three architectures constructed when searching the design space for multicore architectures that provide optimal performance or energy efficiency over our benchmark set. We apply lower peak power constraints in this scenario, since we are optimizing for the single-powered-core execution scenario.

We observe that in a highly peak power constrained environment, the heterogeneous-ISA CMP still manages to achieve a speedup of 16.9% over a single-ISA heterogeneous CMP, and as the peak power budget becomes slightly higher (at 15 W), it provides a consistent speedup of

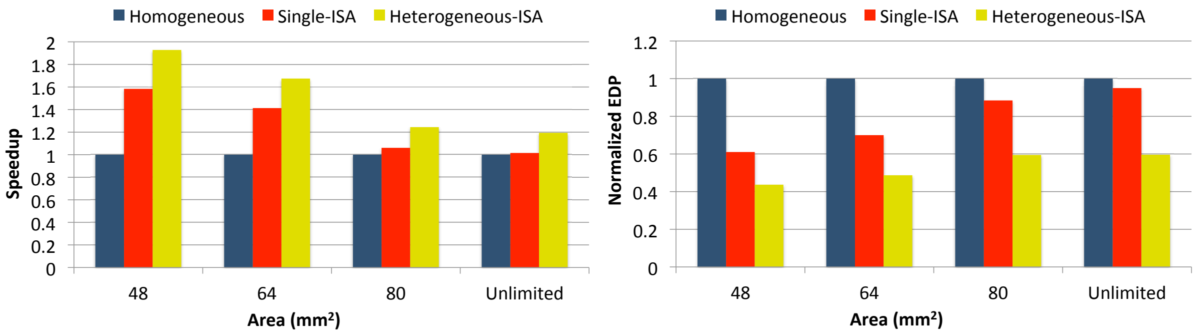

Figure 4.6: Single Thread Performance and EDP evaluation under different area budgets

18.8%. However, the maximum speedup that a single-ISA heterogeneous CMP can provide over a homogeneous CMP in such a topology, is just 1.5%. So again we see that ISA heterogeneity continues to provide gains in regions where hardware heterogeneity is less effective. Figure 4.6 shows the performance and EDP evaluation on designs optimized for single-threaded workloads, under different area budgets. Such designs are typically composed of multiple small cores and one large core optimized to provide high single thread performance. When highly power constrained, single-ISA heterogeneous CMPs use three small Alpha inorder cores and one powerful out-of-order core. Combining again the dual benefits of ISA affinity and the area benefits of Thumb, the best heterogeneous-ISA CMPs provide more balanced cores (to better exploit ISA affinity) yet still enable the same large Alpha core as the single-ISA design. That configuration contains two small Thumb cores, the same out-of-order Alpha core, and a medium-end x86-64 core. We observe that a heterogeneous-ISA CMP can improve single-thread performance by 20.8% over a single-ISA heterogeneous CMP, or achieve 23% more energy savings and 31.8% reduction in EDP, again with no loss in performance.

4.5.2 Framework for ISA-Microarchitecture co-design

In this section, we present inferences from our design space exploration that can serve as a framework for future ISA-microarchitecture co-design in the context of a heterogeneous-ISA CMP. We consider the best designs from all experiments carried out in the previous section.

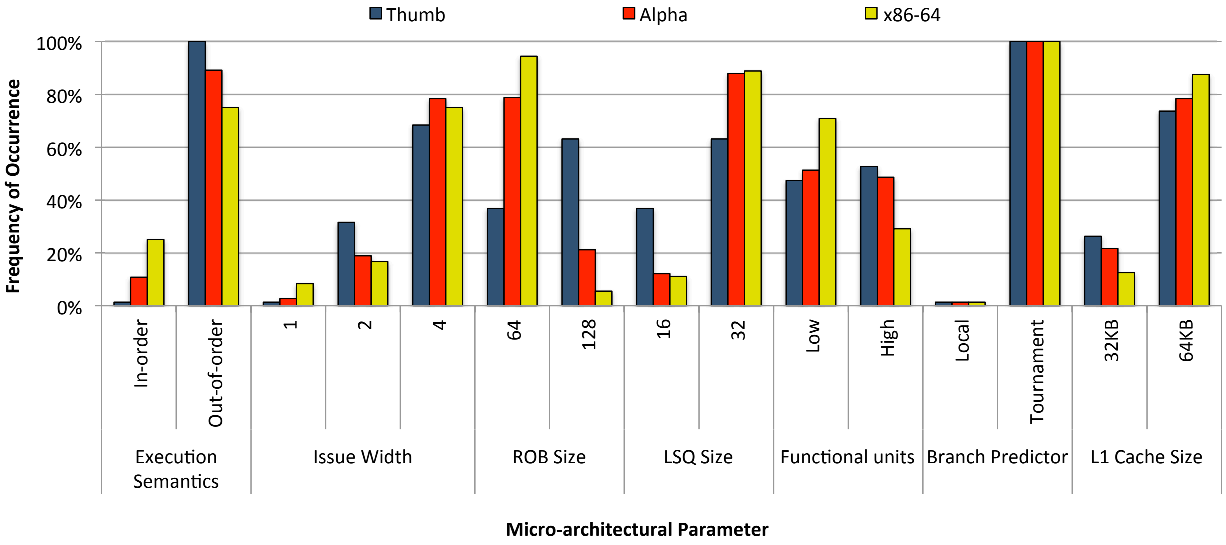

Figure 4.7: Inferences from the Design Space Exploration

Figure 4.7 shows the frequency of occurrence of different micro-architectural parameters in a heterogeneous-ISA design. We analyze the influence of ISA on each of these micro-architectural parameters.

Execution Semantics. We find that out-of-order execution semantics is favorable in general. However, due to conservative peak power budgets in some designs, we select inorder cores 10.8% of the time on Alpha, and 25% of the time on x86-64. Due to the enormous peak power and area benefits of Thumb, we always select out-of-order Thumb cores because they are so cheap. This impacts a number of our results, because it means even our smallest designs always have an out-of-order core available.

Issue Width. In general, higher fetch width enables higher issue width. Since the instruction fetch units of Thumb and Alpha dissipate less power than x86-64, we seldom choose a small issue width for these ISAs. In fact, only 2.7% of our designs choose a single-issue Alpha core and none of our designs use a single-issue core for Thumb. Because code compression ensures our minimum fetch bandwidth is 2 instructions with Thumb, a scalar Thumb processor would always have fetch and issue out of balance.

ROB Size. We find that the number of ROB entries and register file size are highly influenced by the register pressure of an ISA. The low register pressure of x86-64 (see Section 4.1) results in small ROBs and physical register files being configured. Conversely, the high register pressure of Thumb has the opposite effect, while Alpha finds a middle ground between the two.

Load/Store Queue Size. Although we do not see a direct correlation between load/store queue sizes and ISA traits, ISAs more likely to be configured out-of-order and with wide issue, not surprisingly, also demand large LSQs.

Number of Functional Units. Because the x86-64 cores we model have more basic functional unit types (integer, floating-point, and SIMD), the cost of going from the low to the high configuration is higher, and that step is taken less often.

Branch Predictor. Interestingly, all ISAs almost always choose the tournament branch predictor. This implies that it is always worthwhile to invest transistors on the branch predictor. ISA traits such as predication support have little influence on the selection of branch predictor type.

L1 Cache Size The size of L1 cache largely depends on the working set of applications, rather than a specific trait of an ISA. In general, all ISAs favor higher L1 cache sizes.

4.5.3 ISA Affinity

To determine the ISA affinity of each application, we simulate both single-threaded and multi-threaded workloads on two types of designs: (1) optimized for performance, and (2) optimized for EDP. In all scenarios, designs are constrained by a peak power budget of 40 W. Each scenario provides some interesting insights. The design optimized for single-threaded performance is the true indicator of ISA affinity, since only one application is in execution at a time, and each application is allowed to freely migrate between the cores. In case of multi-programmed workloads, due to contention between applications, some applications may execute on ISAs of second preference. On designs optimized for energy efficiency, applications may

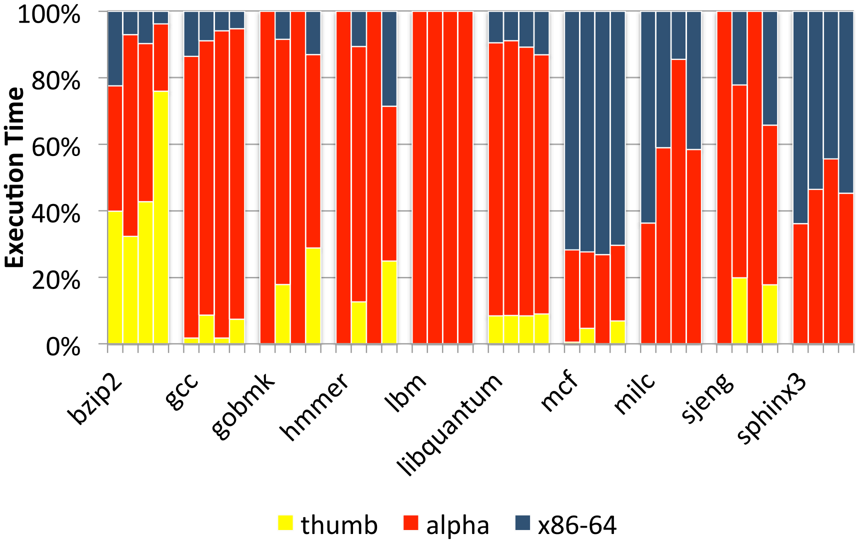

Figure 4.8: ISA affinity for different applications on designs optimized for (left to right) - (a) Single-thread performance, (b) Multi-programmed workload performance, (c) Single-threaded workload EDP, (d) Multi-programmed workload EDP

choose to execute on ISAs that provide energy efficiency but don't maximize performance.

Figure 4.8 shows that each application exhibits a different degree of ISA affinity, and most use all ISAs. In our experiments, benefits arise due to a combination of ISA factors, some synergistic, some interacting negatively – trying to separate those effects is difficult and not always intuitive. However, we are able to make a few high level observations. (a) No floating-point benchmark prefers execution on Thumb due to floating-point emulation. (b) The floating-point benchmark *lbm* prefers execution on Alpha instead of x86-64, because Alpha requires about 34% fewer dynamic floating-point instructions. (c) The high ILP benchmarks *bzip2*, *hmmer*, and *sjeng* prefer execution on Alpha over x86-64, because Alpha offers lower register pressure during phases of high instruction-level parallelism (see Section 4.1). (d) *bzip2* prefers execution on the 32-bit Thumb ISA during phases that involve 32-bit unsigned integer arithmetic. Alpha incurs 27% more dynamic instructions to emulate 32-bit arithmetic using 64-bit registers. In such phases, x86-64 emerges as the ISA of second preference due to sub-register addressing. (e) The benchmarks *libquantum*, *milc*, and *sphinx3* take advantage of x86-64's SIMD functionality at different execution phases, and revert back to Alpha/Thumb during the scalar phases.

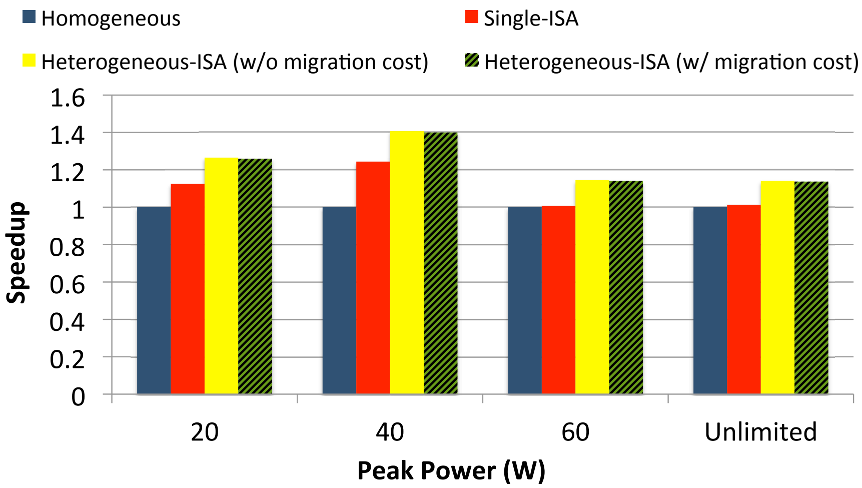

Figure 4.9: Overall Speedup due to migration.

Not immediately clear from the results so far is to what extent the gains are a result of broad differences in feature sets (e.g., SIMD vs no SIMD support) as opposed to the more subtle differences. Further experiments show that the former are a surprisingly small component. For example, if we consider x86-64 with vs without SSE, that heterogeneity provides a gain of 1.3% over the best single-ISA configuration – significantly lower than the 15.8% speedup from a fully heterogeneous-ISA design.

Finally, we note that there is little deviation in ISA affinity due to contention amongst multi-programmed workloads, or due to optimization for EDP instead of performance.

The prior results, primarily concerned with the discovery of the best core configurations, do not account for the cost of migration. We next account for the cost of the actual migrations encountered in an earlier experiment. That is, we incur the cost of migration between two ISAs when a phase change causes a new core/ISA combination to be preferred. Figure 4.9 shows the result of this experiment. Here we see that we sacrifice negligible performance (about 0.4-0.7%) for migration, meaning that virtually all of the performance gain from heterogeneous ISAs is retained. This comes from two factors. First, in most cases, migration overhead is very low. Second, phase changes are relatively infrequent, infrequent enough that even our few cases of high migration overhead are not significant.

4.6 Conclusion

This chapter explores the design space of heterogeneous-ISA chip multiprocessors. It shows that adding an extra axis of heterogeneity by considering multiple ISAs significantly increases the performance and energy efficiency of a heterogeneous processor. Specifically, a heterogeneous design that allows cores with distinct ISAs outperforms the optimal heterogeneous single-ISA design by as much as 20.8% and improves energy efficiency over the most efficient single-ISA design by 23%. This work opens the door for more diverse, and therefore more efficient, architectures, greatly expanding the tools available to hardware and system architects.

Acknowledgements

Chapter 4, in full, is a reprint of the material as it appears in proceedings of ISCA 2014. Venkat, Ashish; Tullsen, Dean M., Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor, 41st ACM International Symposium on Computer Architecture (ISCA), June, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 5

HIPStR: Security Defense via ISA Diversification

The previous chapter describes the design of a heterogeneous-ISA CMP that synergistically complements architectural heterogeneity with micro-architectural heterogeneity, and allows an application (compiled to each ISA as a fat binary) to dynamically identify the ISA of its preference and migrate execution at any given point of time. In this chapter, we leverage this architecture to demonstrate significant new security benefits, and in particular, showcase its ability to defend against an evasive class of buffer overflow exploits called Return-oriented Programming (ROP) [RBSS12, Sha07].

5.1 Background and Motivation

Return-oriented Programming chains together short code snippets in the program (called gadgets) that end with a return or an indirect jump instruction, by overflowing the stack with a carefully constructed sequence of return addresses, and other data required for malicious computation. ROP has been shown to be Turing-complete for multiple ISAs, and over a wide range of applications [RBSS12, BJFL11, BRSS08, CDD$^+$10, CF09, Kor10]. Several exploit mitigation techniques have been described in the literature to thwart ROP. These mitigations can be broadly classified as (a) control flow integrity (CFI) techniques [ABEL05, CPM$^+$98, DSW11, Eto03, KOAGP12, ZS13, ZWC$^+$13, KSP$^+$14] that constrain execution to a predefined control flow graph, or (b) randomization techniques [MBSN14, DLS$^+$15, HNTC$^+$12, KKP03, PPK12, PT03a, SKIH12, WMHL12] that enable a system to exist in one of many random states such that it is hard to predict the exact location or manifestation of a gadget.

The success of randomization techniques is directly proportional to the *entropy* (number of randomizable states) they provide, and the extent to which they are resistant to entropy reduction attacks [RMPB09, SPP$^+$04, Wev04]. In their most powerful form, entropy reduction attacks called just-in-time return-oriented programming (JIT-ROP) [SMD$^+$13], completely bypass all randomization, using a single leaked memory disclosure. Therefore, it is critical to design robust and performance-efficient randomization techniques that provide an entropy that is beyond the reach of state-of-the-art exploit generation. In this work, we find that the low overhead of execution migration in a heterogeneous-ISA CMP makes it a natural candidate to repel such attacks, and therefore propose a novel defense mechanism called Heterogeneous-ISA Program State Relocation (HIPStR), that performs dynamic randomization of run-time program state, both within and across ISAs.

First, we leverage a heterogeneous-ISA CMP composed of an ARM core and an x86 core, and non-deterministically migrate execution of a vulnerable process between the two ISAs, in such a way that we render JIT-ROP attacks extremely hard to execute, while still retaining inherent performance gains offered by the heterogeneous-ISA architecture. Consequently, we remove one of the last remaining “constants” available to the attacker – knowledge of the ISA the program is executing on.

Second, we note that any program, including a return-oriented program, requires a certain amount of program state, in the form of registers and memory, to perform any computation in the target ISA. To this end, we employ a dynamic binary translation engine on both cores that moves the run-time program state of an application to random and attacker-unknown locations, such that legitimate execution that preserves control flow is guaranteed to function as expected, but an attacker-crafted malicious exploit is highly unlikely to function as intended.

5.2 Architectural Overview

In this section, we lay out our security and performance guarantees and discuss strategies to harness and re-purpose the cross-ISA process migration techniques described in Chapter 3 as a security defense for ROP.

5.2.1 Security and Performance Guarantees

Security. One of the main goals of HIPStR is to defend against and reduce the attack surface of a wide array of attacks, including but not limited to return-into-libc, ROP, JOP, brute force attacks, JIT-ROP, and JIT-spraying. For any program in execution, HIPStR dynamically randomizes the location of its program state (registers and stack objects) in order to render brute-force attacks infeasible. Furthermore, HIPStR has the ability to detect a potential break-in attempt via JIT-ROP, and when detected, probabilistically migrates execution to a different ISA, thereby imposing serious limitations on JIT-ROP attacks.

Performance. HIPStR makes several careful performance-related decisions in order to provide security guarantees, at an acceptable degradation in performance, and outperform Isomeron, the only other JIT-ROP defense in the literature. First, unlike Isomeron which diversifies execution at every function call and return, HIPStR migrates execution to a different ISA only when a potential security breach is detected, thereby enjoying full security benefits at virtually zero performance overhead due to migration. Second, HIPStR implements several optimizations described in Section 4.2 in order to speed up the underlying dynamic binary translation framework.

5.2.2 Instruction Set Randomization

From a security standpoint, heterogeneous-ISA CMPs have two major advantages. First, ROP attacks are highly target-ISA dependent. An application that migrates between multiple heterogeneous-ISA cores executes instructions from different instruction sets. If a migration is forced upon execution of every ROP gadget, a successful attack would require chaining gadgets from different ISAs, and yet produce a meaningful result (e.g., spawn a shell). Furthermore, if we make migration probabilistic, we remove the most fundamental assumption of the attacker – knowledge of what ISA the gadget will execute on. The second advantage is that execution migration in a heterogeneous-ISA CMP requires stack transformation. This especially constrains ROP gadgets to save all intermediate state in locations that are immune to run-time stack transformation (e.g., heap memory), thereby significantly reducing the attack surface.

Several fine-grained randomization techniques proposed in prior work have been shown to be broken by JIT-ROP [SMD$^+$13] that exploits a single leaked memory disclosure to reconstruct the entire memory image of the process, and thereby bypass all randomization. Instruction Set Randomization in a heterogeneous-ISA CMP, however, severely inhibits JIT-ROP. This is because the decision to migrate execution to a different ISA is made probabilistically at run-time, thereby limiting an attacker's ability to chain gadgets reliably.

While randomization across heterogeneous-ISAs systematically removes the knowledge of what architecture the attacker is executing on, in the next section, we show how randomization within an ISA could further extend the effectiveness of our technique.

5.2.3 Program State Relocation

Program State Relocation (PSR) comprises a set of dynamic binary code transformations that can be easily deployed in any JIT-based system. The major goal of program state relocation is to shuffle program state (registers and memory) such that it is always found at the expected

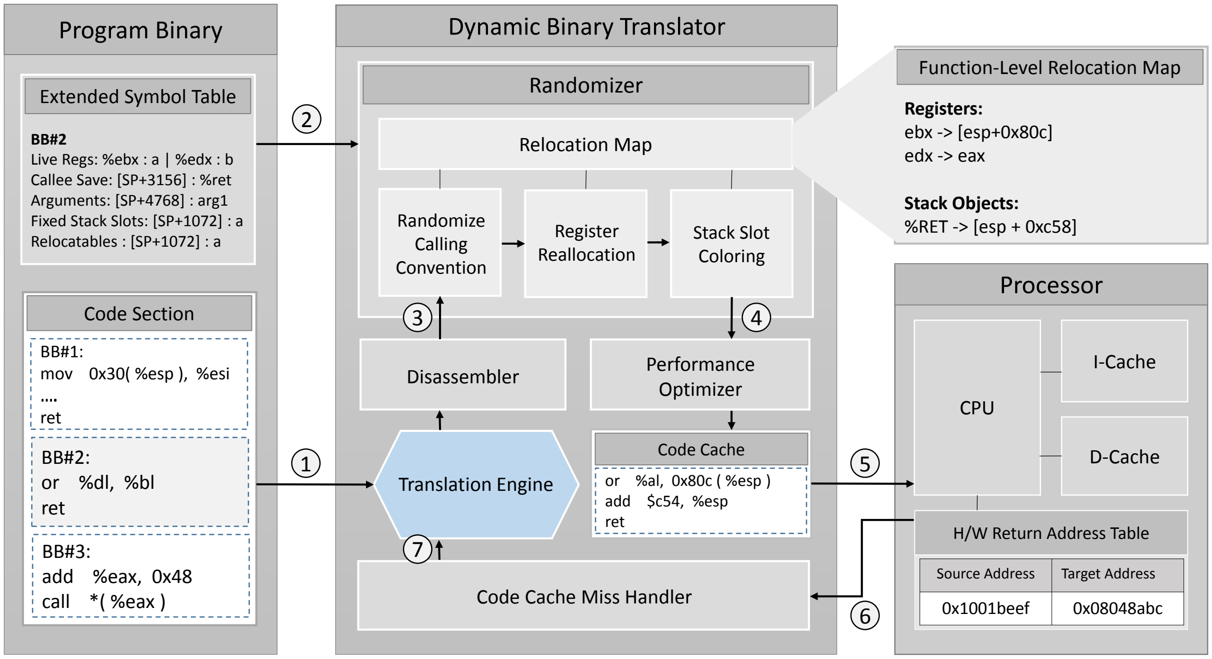

Figure 5.1: Program State Relocation Architecture

location during legitimate execution, but it is highly unlikely to be found by a ROP gadget that strays away from the legitimate control flow path.

As shown in Figure 5.1, the PSR runtime operates in a classic just-in-time dynamic translation mode, processing one basic block at a time. For each basic block in translation, it gathers information about the parent function, which is available from static analysis. Irrespective of the point of entry, the PSR runtime constructs a *relocation map* for every function, if it is being entered for the first time. The relocation map specifies the randomized calling conventions to be followed while calling the function, along with a set of randomized register allocation and stack slot coloring rules to be followed within that function. In Section 4.2, we describe each transformation in detail. The figure shows an example PSR transformation – BB#2 in the code section is transformed to the basic block shown in the code cache. Note that PSR just requires a simple change in the addressing mode of the instruction *or* in order to relocate its original operands *dl* and *bl* to *al* and *[esp+0x80c]*, as indicated by the relocation map respectively. Note that the return address is also moved to a random location on the stack.

As with classic DBT [SN05, Bel05], translation is performed until an indirect or conditional jump is reached, at which point control is transferred to the translated code in the code cache. If a translation for the jump target is not available (a code cache miss), necessary transformations are applied as described above, and control is relinquished to the translated code. To ensure the code cache does not get compromised, we mandate that all return addresses stored on the stack point to original source code instead of the translated version. Furthermore, we make minor changes to the call and return instructions (macro-ops) to perform an extra cycle look-up in a hardware-maintained *Return Address Table* (RAT), in order to translate the source-level address to its corresponding translated version before making the actual control transfer.

The effect of program state relocation is that an object previously found in a register may be relocated to a different register or a random location on the stack, and vice-versa. Due to the sheer number of stack locations available to use for relocating an object, the number of possible dynamic code transformations (entropy) explodes, thereby rendering classic brute force attacks such as Blind-ROP [BBM$^+$14] practically impossible on a system implementing PSR. Moreover, since the transformations happen at run-time rather than load-time, a PSR system will always re-randomize upon a crash or reboot, further strengthening its effectiveness.

5.2.4 Heterogeneous-ISA PSR

Instruction Set Randomization and Program State Relocation each represent strong defenses independently. However, we find that there is significant synergy between the two techniques, and one technique only amplifies the effectiveness of the other. Therefore, we combine them into one solid defense called "Heterogeneous-ISA Program State Relocation"(HIPStR). Figure 5.2 shows the high level architecture of Heterogeneous-ISA PSR.

The defense leverages, in this particular implementation, a heterogeneous-ISA CMP composed of a low-power ARM core and a high-performance x86 core, that each run a virtual machine capable of performing program state relocation. To continue to reap the full perfor-

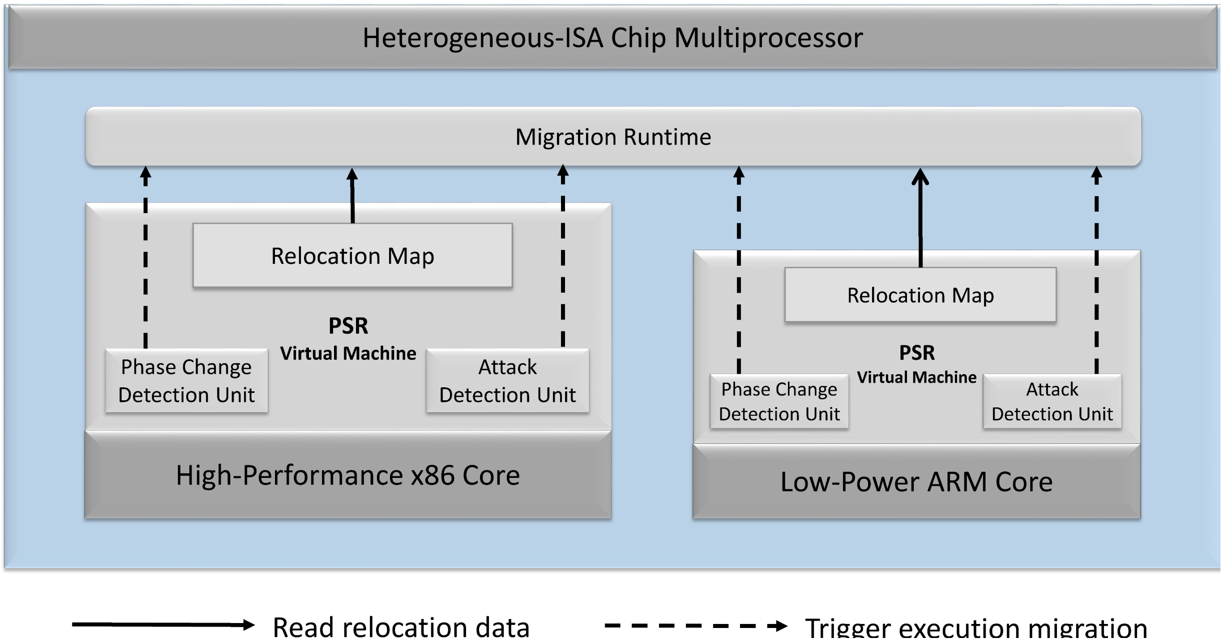

Figure 5.2: Heterogeneous-ISA Program State Relocation

mance/energy benefits of the heterogeneous-ISA CMP, we perform task migration only when an application phase change profits from migration to a different ISA. Additionally, we perform non-deterministic execution migration between the two ISAs only when the PSR runtime detects a possible attempt to compromise security.

In our evaluation, we find that a code cache miss resulting from an indirect control transfer (including returns) is one of the key characteristics of a possible security breach. A code cache miss could result from one of two scenarios. In the legitimate execution scenario, the jump target is valid, but has not been translated yet (compulsory miss), or a translation for it was previously evicted from the code cache (capacity miss). In an attack scenario, the jump target points to a ROP gadget, and therefore a mapping does not exist in the PSR data structures. The PSR virtual machines make no effort to distinguish between the two scenarios. They instead migrate execution to a different ISA (with some probability) on every indirect control transfer that misses the code cache.

Like any JIT system with a sufficiently large code cache, one would expect code cache misses to be infrequent once the application reaches a steady state in execution. Therefore, legitimate execution should experience no meaningful degradation in steady state performance. Furthermore, we perform multiple translations, one for each ISA, when an indirect control transfer results in a compulsory miss, further reducing miss events.

In theory, an attacker could avoid migrating to a different ISA by using gadgets that are already translated indirect jump targets or function call sites, for which the PSR virtual machines already have a mapping in their internal data structures. In our evaluation, we find that the number of such gadgets is insufficient even for the simplest *execve* exploit.

5.3 Assumptions and Threat Model

JIT Engine. We model a JIT engine such as a browser environment that performs dynamic binary transformations. Like most browser environments and other code randomization defenses [DLS+15, HNTC+12] that employ dynamic binary instrumentation, we assume that the JIT engine is checked for vulnerabilities and its address space is protected by memory protection mechanisms such as code signing [MSD], sandboxing [AME+11], and Intel Software Guard Extensions (SGX) [Int14].

Fine-grained Randomization. We require no fine-grained randomization techniques (including ASLR) to protect our system, although they would only further strengthen the system since HIPStR is orthogonal to most existing defense mechanisms.

Complete Disclosure. We assume that the attacker has full knowledge of the inner workings of our defense mechanisms. We also assume that the attacker has unfettered access to the binary, source code, and complete control flow graph of the program in execution. Consequently, the attacker has a complete list of all potential ROP/JOP gadgets in the binary, and is capable of mounting attacks ranging from classic ROP [Sha07] to just-in-time code reuse (JIT-ROP) [SMD+13] attacks.

Just-in-time Code Reuse. In addition to the ability to snoop into a program's memory, we assume the program in execution exhibits one or more vulnerabilities that allow an attacker to (a) write to memory (by means of a stack/heap based overflow), and (b) read an arbitrary number of bytes from any memory location, using a single leaked memory disclosure.

Brute Force Attacks. We also assume that the system is susceptible to brute force attacks such as Blind-ROP [BBM$^+$14]. To this end, we model a system as described by Shacham, et al. [SPP$^+$04] that assumes a program executing as a child thread, whose parent re-spawns it upon on a crash. We do not assume any defense mechanism that monitors the frequency of such an event to detect ROP attacks. We instead use it as a metric to demonstrate the effectiveness of PSR against brute force.

5.4 Design and Implementation

In this section, we present the design and implementation details of Program State Relocation, discuss how our system behaves under different execution scenarios, and finally describe techniques to optimize our system for performance.

5.4.1 Program State Relocation

As discussed in Section 5.2, Program State Relocation is a set of transformations that relocate program state (registers and stack objects) within the same ISA. In our implementation, these transformations essentially randomize calling conventions, register allocation, and stack slot coloring. While most of these transformations can be accomplished by a mere change in the addressing mode, some transformations (e.g., procedure call/return) are slightly more involved and might require insertion of a small number of *move* instructions.

Addressing Mode Transformation. Each instruction in a basic block is modified to access its source and destination operands at their new locations, as specified by the function's relocation map. In most cases, this transformation is rather trivial and involves mere changing of addressing modes. If the ISA does not expose a certain addressing mode, the PSR virtual machine emulates it using additional instructions and register temporaries. For example, owing to the variety of addressing modes in x86, we use additional instructions only when more than one operand of an instruction is relocated to memory.

Procedure Call Transformation. The PSR virtual machine instruments all procedure call instructions to perform argument relocation and register spill/restore as specified by the callee's relocation map and the target ABI, respectively. As an optimization, the PSR virtual machine eliminates any redundant caller/callee register save and restore instructions. Furthermore, the virtual machine allocates 2 to 16 pages of randomization space on the stack in addition to the space already used by the callee's locals, temporaries, and spills, effectively providing 13 to 16 bits of entropy for every register or memory access. Note that return addresses are also relocated to random offsets, and therefore even a *nop* gadget that just performs a return incurs an entropy of at least 13 bits.

One of the biggest challenges with procedure call transformation is to preserve the live-ins and live-outs across function call sites, and correctly compute the caller/callee saves upon every function invocation. We take advantage of a single basic block look-ahead liveness analysis to accurately compute this information, and incorporate them into the randomized calling convention. A major source of ROP gadgets include the callee restore sequence that pops a bunch of callee save registers before returning back to the caller. To circumvent this, we perform a randomized scatter of callee saves (spray callee saves to random locations on the stack) at the function call site, and a randomized gather after return.

Indirect Control Transfer. Like any DBT system [SN05, Bel05], the PSR virtual machine traps all indirect jumps into the translator. This ensures there exist absolutely no indirect jumps translated into the code cache. As a software fault isolation measure, we terminate the process in case we find an indirect jump target within the code cache's address range. Similarly, we disallow pointers to the code cache to exist as function pointers or return addresses on the stack. We handle function pointers in the same way as indirect jumps.

For function returns however, we always push the source return address on the stack, and take advantage of the return address table (RAT) that contains a mapping from source address (address of the function call site in the native binary) to target address (address of the function call site in the code cache). The call macro-op in the processor is modified to update the RAT with the right mapping, while the return macro-op is modified to perform return address translation as an extra step with a 1-cycle penalty. Upon a RAT miss, we conclude that there was a code cache miss and trap into the translator, for re-translation of that basic block.

5.4.2 PSR-aware Execution Migration

Our migration policy allows execution migration across heterogeneous ISAs in two specific scenarios. First, we migrate execution whenever an application phase change or the processor's current operating condition demands migration to another core. This is essential because it preserves the performance and energy advantages of a heterogeneous-ISA CMP. On the other hand, we also migrate execution, although probabilistically, when the PSR virtual machine suspects a security breach (specifically, when an indirect control transfer results in a code cache miss).

Prior work on heterogeneous-ISA execution migration suggests that we can be migration-safe at only 45% of the basic blocks [VT14]. To support instantaneous migration, they employ dynamic binary translation until a point of execution is reached where the stack can be safely transformed. This implies that a ROP exploit that is composed entirely out of the remaining 55% of the basic blocks could completely bypass instruction set randomization.

To circumvent this, we re-purpose the original multi-ISA compilation infrastructure to support an on-demand execution migration. In essence, we transform only those objects on the stack that are absolutely necessary for executing instructions until the next control transfer (jump, call or return), and revert back to the original ISA to execute the next basic block. By doing so, we manage to be migration-safe 78% of the time. Furthermore, we completely avoid jumps to *unintentional* gadgets upon a code cache miss. We do this by taking advantage of an attack detection unit that disassembles from the last seen nearest address (or function boundary) to the program counter, up until the program counter itself. This is a minor change to the PSR virtual machine, which already does sophisticated liveness analysis.

Finally, we ensure that our migration strategy is PSR-aware, which means we not only transform an object from one ISA-form to another, but we fetch the object from its randomized location on one ISA and move it to its new randomized location on the other ISA.

5.4.3 Execution Scenarios

Legitimate execution. In a legitimate execution scenario, the procedure call transformation ensures that functions are always presented with relocated arguments. Furthermore, basic blocks are also presented with relocated live-ins since execution starts at the intended entry point of the function, thereby preserving the integrity of legitimate program execution.

Stack Unwinding. Libraries such as *libunwind* rely on compiler generated stack frame layout information to unwind the stack in exceptional scenarios such as *setjmp and longjmp*, and C++ exceptions. PSR seamlessly works with *setjmp and longjmp* due to the temporary register spill/restore, performed as a part of the procedure call transformation.

However, C++ exceptions and other debugger routines unwind the stack frame-by-frame, inspecting stack objects at each frame, until the unwind target is reached. Performing PSR on such routines might lead to inconsistent program state. To prevent such inconsistencies, the PSR virtual machine instruments these unwind routines to use the same relocation map as the function that owns the frame being processed. This guarantees that frame objects are always accessed from their appropriate relocated addresses, irrespective of the control flow. Furthermore, we force migration (and thus stack transformation) in the rare event when a *longjmp* is taken, but the corresponding *setjmp* was performed on a different ISA.

ROP attack. In the event of a ROP attack, the buffer overflow itself happens at a relocated stack address. Therefore, there is no guarantee that the return address is overwritten with the gadget address. In case the attacker manages to successfully overwrite the return address, she will find that the gadget at that address fails to work as intended. This is because the PSR virtual machine dynamically transforms every instruction in that gadget to access data from their randomized locations. Note that this is not just true for ROP attacks, but holds for jump-oriented programming, v-table hijack, and other variants. PSR inherently defeats return-into-libc because of the randomized calling conventions.

Crash/Reboot scenarios. To guarantee high quality of service and robustness, most servers re-spawn worker threads upon a crash or a reboot. Several brute force attacks such as Blind-ROP exploit this property of servers to mount repeated attacks until they become compromised. These attacks typically bank on using information leaked in a previous attempt, in order to reduce the overall time-to-attack. This is possible because a process randomized at load-time typically does not get re-randomized every time it spawns a thread. However, a PSR virtual machine performs randomization at run-time, which means we have the ability to re-randomize upon re-spawn. Note that this extends to the PSR virtual machines on both ISAs. Therefore, each time a worker thread re-spawns, the attacker is presented with a re-randomized version of the code cache on both ISAs.

5.4.4 Performance Optimizations

Machine Block Placement. As with any JIT engine, we take measures to carefully place translated basic blocks in the code cache, so that we incur as few conflict misses in the instruction cache as possible. To further improve the instruction cache performance and fetch bandwidth, we align tight single-entry single-exit loops to cache block boundaries.

Branch Inlining and Superblock Formation. Next, we compose our translated basic blocks into *superblocks* that have a single entry-point, but multiple exit-points. We form superblocks in two steps. First, we fold branches whenever possible. This includes both direct unconditional branches and fall-through cases in conditional branches. Second, we avoid backward branches by inlining a direct branch instruction. Note that this results in code duplication, but it both improves the locality in the instruction cache, and reduces pressure on the branch predictor.

Global Register Cache. While PSR provides extremely high entropy, the sheer number of stack operations could potentially cause severe performance degradation. To optimize for performance, we use a global register cache that holds the most frequently used registers that are relocated to stack objects. We mandate that this cache be only three entries long so that we provide high performance in tight loops, and at the same time provide security guarantees by still spilling frequently to random locations.

PSR with a Register Bias. In this final optimization, we perform PSR with a *register bias*, i.e, at all times, we ensure at least three registers are always relocated to other registers, albeit randomized for each function.

5.4.5 Prototype Implementation of PSR

Owing to the complex addressing modes in x86 and the possibility of *unintentional* gadgets (unaligned sequence of bytes that end with the byte *c3* indicating a *ret* opcode), x86 not only exhibits greater susceptibility to vulnerabilities, but also presents greater challenges in terms of design and implementation. Specifically, we note that the attack space on ARM is 52X smaller than x86 (measured using Galileo [Sha07] ported to ARM), since ARM enforces strict alignment of instructions. Moreover, the simplicity of the instruction set and lower register pressure on ARM facilitate a smaller engineering effort and better opportunity for performance optimization. Therefore, we choose to implement our more complete PSR prototype, and do most of our PSR measurements, in x86. By doing so, we not only demonstrate high coverage (as the vast majority

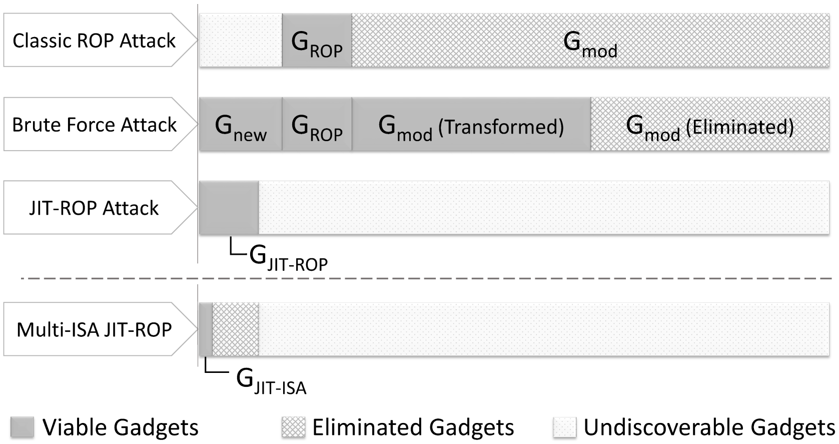

Figure 5.3: Attack Surface of a Victim Program

of the gadgets exist in the x86 code), but also report conservative estimates in both performance and security evaluation.

5.5 Experimental Methodology

Security Evaluation. An important characteristic of a security attack is that it requires the victim program to expose a reasonable *attack surface* to exploit. In the context of ROP, the attack surface is represented by the number of gadgets available in a program that facilitate the construction of a successful exploit. The goal of every randomization defense is to reduce the attack surface (both in terms of availability and functionality), in order to limit the attacker's ability to construct meaningful exploits. In our evaluation, we not only subject our defense to state-of-the-art attack mechanisms [RBSS12, BJFL11, SMD$^+$13, Moo10, SD97], but also measure its effectiveness against potential attacks that are computationally beyond the reach of today's attacker. For each attack, we report the degree to which the attack surface is reduced by our technique. Figure 5.3 represents a victim program's attack surface for different types of attacks while running on our architecture. Table 5.1 introduces a list of symbols and their definitions that we use to represent key elements of an attack surface through the rest of this section.

Table 5.1: Attack Surface: Symbols and Definitions

Symbol	Definition
G_{ROP}	Size of the attack surface for a classic ROP attack.
G_{mod}	Number of gadgets modified by PSR.
G_{new}	Number of gadgets introduced by PSR.
$G_{JIT-ROP}$	Size of the attack surface for a JIT-ROP attack.
$G_{JIT-ISA}$	Size of the attack surface for a JIT-ROP attack in Heterogeneous-ISA PSR.

We use the 'Galileo' algorithm described by Shacham, et al. [Sha07] to mine a benchmark for every possible instruction sequence that ends with a return instruction. Since every exploit requires some program state in the form of either registers or stack objects, we designate any gadget that successfully populates a register with an attacker supplied value from the stack as *viable*. We evaluate every gadget for its viability on a system, without and with PSR, to measure the attack surface for four major classes of attacks: (a) classic ROP, (b) brute-force, (c) JIT-ROP, and (d) tailored heterogeneous-ISA attacks to defeat HIPStR.

Owing to the sheer number of stack locations available for program state relocation, the number of possible manifestations of a gadget explodes. To evaluate the system against brute force attacks while keeping the experiment tractable, we analyze each gadget to gather data about every perturbation it produces on the state of the program, at a randomly chosen point in its execution. We then simulate a brute force attack by running this data through Algorithm 1. Cheng, et al. [CZY$^+$14] showed that the shortest aligned gadget chain generated by gadget compilers such as Q [SAB11] is 17, but to establish the effectiveness of PSR, we consider a much smaller four-gadget shellcode exploit that performs the system call $execve()$, which in theory should be easier to brute force by several orders of magnitude. Although the run-time nature of PSR transformations involve re-randomization upon crash, to keep the experiment tractable, we make the conservative assumption that a failed attempt does not result in re-randomization, and thereby tip the scales in the attacker's favor.

Algorithm 1 Brute Force Simulation

```
$R = \{r_1, r_2 ... r_m\}$ /* Set of $m$ registers to load. */
$P = \emptyset$ /* Set of successfully populated registers. */
$X = ()$ /* List of chosen gadgets for the attack. */
$Y = ()$ /* List of return address locations for chosen gadgets. */
$A(g)$ is the randomized return address for gadget $g$

for all $i = 1$ to $m$ do
    $r_i$ is the register to populate
    find $g_j$ in $G$ s.t. $g_j$ populates register $r_i$,
        does not clobber any register $s$ in $P$, and
        $A(g_j) = \min_{k=1...n} A(g_k)$
    $P = P + \{r_i\}$
    $X = X + \{\mathrm{j}\}$
    $Y = Y + \{A(g_j)\}$
end for

Let $B$ be the number of attempts to populate all registers, then
    for an average frame size of $f$
$B = Y[0] + f.X[0] + nf.Y[1] + nf^2.X[1] + ... + n^3 f^4.X[3]$
```

Algorithm 1 simulates a brute force attack to populate the four registers (*eax*, *ebx*, *ecx*, and *edx*) necessary to perform the *execve*() system call with attacker provided values on the stack. On a system protected by PSR, all program state (registers and stack objects, including the return address) is relocated to a random register or a stack location. Therefore, such an attack should brute force three independent variables in the system: (a) the gadget to execute, (b) relative position(s) of data on the stack, as required by the gadget, and (c) relative position of the return address on the stack, required to chain the next gadget. The attacker should brute force the gadget itself, because it is difficult to determine the potential *viability* of a gadget that will inevitably be subject to PSR. Therefore, we brute force every gadget discovered by the Galileo algorithm. The data for each gadget (the value to load into a register) and the return address both share the same stack frame. In an unsecured system their locations can be easily determined, but with PSR, they can lie anywhere within a stack frame.

To maximize the success of a gadget, our attack *sprays* the data for the gadget on the

Table 5.2: Architecture detail for ARM and x86 cores

ARM core			
Frequency	2 GHz	I cache	32 KB, 2 way
Fetch width	2	D cache	32 KB, 2 way
Issue width	4	ROB size	20 entries
LQ/SQ size	16/16 entries	Functional Units	Int ALU(2), IntMult/Div(1), FP ALU/Mult/Div(2)
x86 core			
Frequency	3.3 GHz	I cache	32 KB, 2 way
Fetch width	4	D cache	32 KB, 2 way
Issue width	4	ROB size	128 entries
LQ/SQ size	48/96 entries	Functional Units	Int ALU(6), Mult/Div(1), FP ALU/Mult/Div(2), SIMD(2)

entire stack frame and brute force the location of the return address within the frame. We model our attack to populate one register at a time, in order to *spray* an entire stack frame with the data for one register, thereby increasing its chances of being read by a gadget. Since we assume the attacker has insight into the inner workings of PSR, we assume a frame size of 8KB, at which PSR provides substantial security benefits at an acceptable degradation in performance. In our algorithm, we also account for register and stack clobbering to ensure that a gadget does not destroy previously established (by an earlier gadget in the exploit) state. The algorithm stops searching for more *viable* gadgets as soon it finds a four-gadget shellcode exploit.

It is worth noting that our method of simulating brute force loosely resembles the Blind-ROP algorithm [BBM$^+$14] that finds *viable* gadgets when an attacker has no knowledge of the binary or source code. The key difference is that Blind-ROP relies on the target binary respecting traditional calling conventions and stack layout, whereas in a PSR-protected system, we can make no such assumptions. Therefore, a Blind-ROP attack on a system secured with PSR essentially translates to a version of our brute force attack and will require a similar number of attempts to succeed.

Performance Evaluation. We use the SPEC CPU2006 integer and floating-point C benchmarks to evaluate the proposed defense. We exclude *gcc* and *sjeng* from this set because they perform dynamic memory allocation on the stack either using the *alloca* library function, or by passing variable-length array parameters. While our multi-ISA compilation and runtime infrastructure is capable of working with variable-size stack frames, our PSR implementation does not support this feature yet. All benchmarks are compiled using an LLVM-based multi-ISA compiler at the -O3 optimization level. To model a heterogeneous-ISA CMP, we use the gem5 [BDH$^+$06] architectural simulator. The processor model of the ARM core is based on the low-power Cortex A-9, while the x86 core is modeled after the high performance Intel Xeon. Table 5.2 shows the details of each core.

Correctness. To ensure that we preserve the semantics of a program at all times, we perform two types of sanity checks. First, we periodically examine the register and stack contents of a randomized program in execution, and compare it against the unrandomized version. Our test infrastructure has the ability to tune the frequency of this sanity check at the function, basic block, and instruction levels. Second, we ensure that the migration runtime has appropriately transformed the program's architectural state by comparing it against a checkpoint of the same program that has been executing on the migrated-to ISA from the time of its instantiation.

To evaluate the steady state performance and study the effect of various contributing factors, we simulate a portion of the program's execution at different optimization and entropy levels, with different code cache and hardware Return Address Table (RAT) sizes, as follows. We fast forward execution for the first one billion instructions to skip initialization code, and perform cycle accurate simulation for another one billion instructions[SPHC02], while running in the context of a PSR virtual machine and with heterogeneous-ISA migrations enabled.

To evaluate the migration overhead at random execution points, we skip the initalization phase and fast forward execution of each benchmark to a random checkpoint and force migration to a different ISA, and report results averaged across ten random checkpoints.

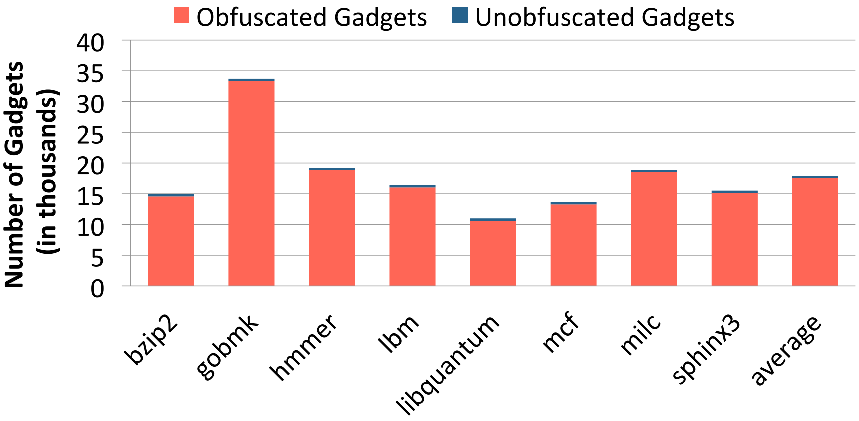

Figure 5.4: Classic ROP Attack Surface

5.6 Evaluation

5.6.1 Security Evaluation

Classic ROP-Style Attacks. Figure 5.4 shows the extent to which PSR reduces the attack surface for classic ROP-style attacks, including return-into-libc, jump-oriented programming, and v-table hijack. We observe that the sheer amount of randomization that each gadget undergoes guarantees that only a very small portion of the attack surface remains unaltered. To be precise, PSR reduces the attack surface of classic ROP (G_{ROP}) by an average of 98.04%. We note that although the remaining 1.96% is unobfuscated by PSR, the attacker has no way of determining which gadgets they are beforehand, since their randomized version is only generated on execution, thereby rendering classic ROP attacks infeasible.

Brute Force Attacks. As illustrated in Figure 5.4, PSR modifies a majority of the gadgets that were previously available for ROP. These gadgets, (G_{mod}) by virtue of PSR's transformations, have either been obfuscated in a way that they no longer perform the attacker intended action, or have been completely eliminated. The former of these are *viable* candidates for a brute force attack since they perform useful computation, just not what an attacker expects them to. Even under the assumption of full memory disclosure, it is impossible to determine the transformations that will be applied on a gadget, without executing it. Also *viable* for a brute force attack are any

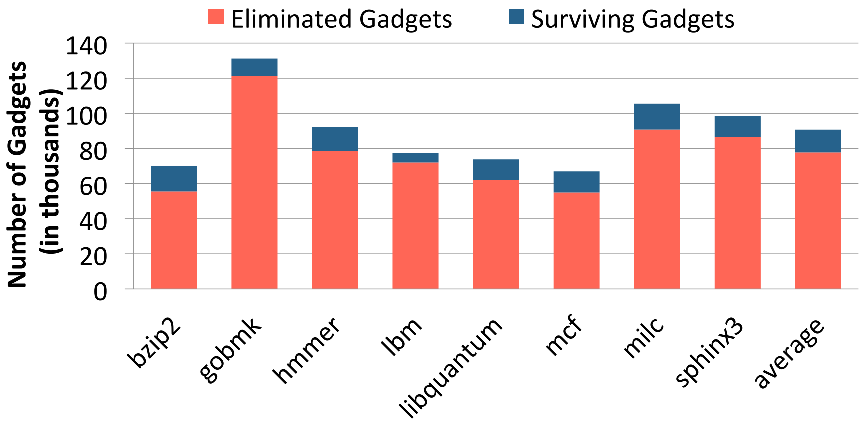

Figure 5.5: Brute Force Attack Surface

Table 5.3: Inferences from Brute Force Simulation

Benchmark	**Randomizable Params (avg)**	**Entropy**	**Attempts (no reg-bias)**	**Attempts (reg-bias)**
bzip2	6.76	88	9.11 x 10^{33}	2.34 x 10^{33}
gobmk	6.53	85	2.34 x 10^{34}	2.87 x 10^{34}
hmmer	6.69	87	1.37 x 10^{34}	1.16 x 10^{34}
lbm	6.92	90	3.33 x 10^{34}	3.90 x 10^{34}
libquantum	6.76	88	1.05 x 10^{34}	6.45 x 10^{33}
mcf	6.69	87	3.10 x 10^{33}	1.71 x 10^{34}
milc	6.46	84	1.86 x 10^{34}	2.92 x 10^{34}
sphinx3	6.92	90	1.14 x 10^{34}	8.68 x 10^{33}

gadgets introduced by the randomization itself. The attack surface for brute force comprises every gadget available in the program, since there is no way to ascertain which ones will transform to be *viable* gadgets. Note that the set of *viable* gadgets for brute force includes G_{ROP}, G_{mod} (transformed gadgets only), and G_{new}. As shown in Figure 5.5, we observe that a sizable portion (an average of 15.83%) of all gadgets are *viable* for brute force, and therefore require thorough evaluation. Although some existing ROP defenses dismiss brute force as impossible assuming that an operating system would detect multiple crashes, or that the user would not re-run a crashing application, it has been proven that brute force remains a viable option if the application is vulnerable to repeated attacks [BBM+14, SPP+04].

Table 5.3 shows the results of our brute force simulation described in Section 5.5. We

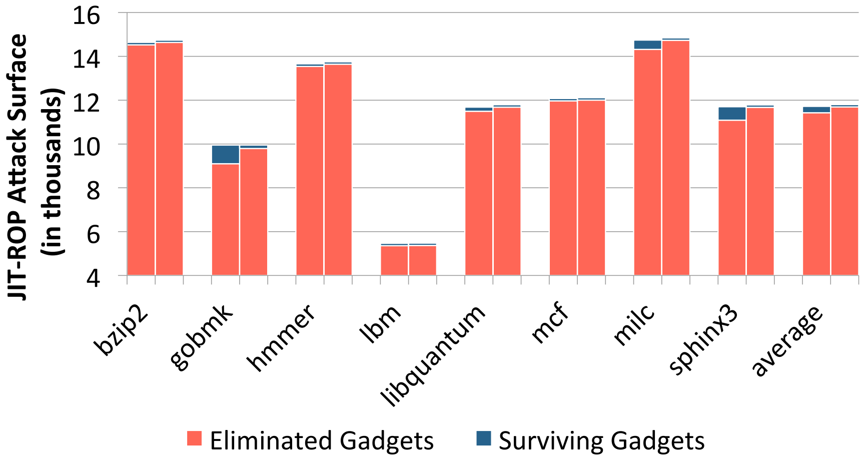

Figure 5.6: JIT-ROP Attack Surface on (a) PSR, (b) HIPStR

observe that PSR successfully renders brute force attacks computationally infeasible, by a considerably large margin. We find that, on an average, a gadget has between six and seven randomizable parameters which could potentially include registers, stack objects, and at least one address on the stack to place the (return) address of the next gadget. In our configuration of 8KB sized stack frames, each parameter has 2^{13} randomizable states, resulting in an average entropy of 87 bits per gadget. Even if a vulnerability allowed an indefinite number of attempts, with each attempt only taking a nanosecond, we find that it is computationally infeasible to perform such a brute force attack with state-of-the-art computing infrastructure. In fact, such an attack would remain computationally infeasible on future processors targeted at exascale computing.

Just-In-Time Code Reuse. Figure 5.6 shows the reduction in attack surface for each benchmark under both single-ISA and heterogeneous-ISA PSR. Owing to the just-in-time nature of PSR, only the steady state program code that has already been randomized by PSR and is present in the code cache remains vulnerable to JIT-ROP. We find that the number of gadgets already randomized by PSR accounts for only 1.45% of all classic ROP gadgets and 1.92% of those *viable* for brute force, thereby severely constraining the attack surface. Figure 5.3 shows this reduction in attack surface for JIT-ROP. Note that a majority of gadgets are now *undiscoverable*, since they lie outside the code cache.

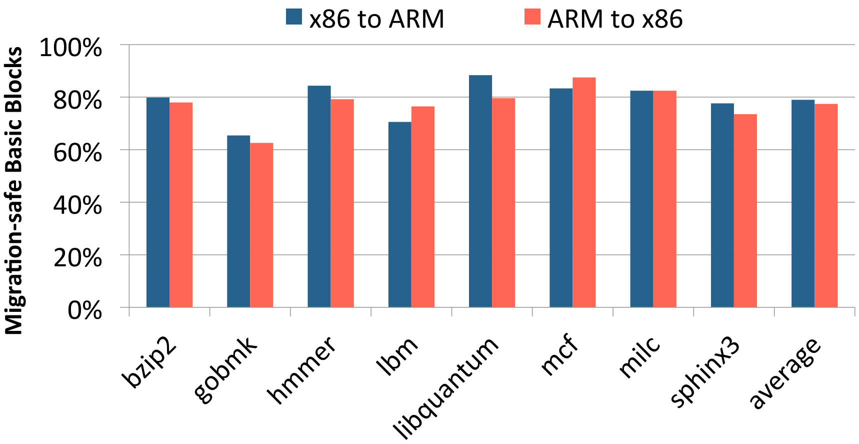

Figure 5.7: Percentage of Migration-Safe Basic Blocks

Although the attack surface has been considerably reduced, the surviving 294 gadgets could potentially be enough to mount a JIT-ROP attack. Recall from Section 5.2 that the PSR virtual machines suspect a security violation when an indirect control transfer (including returns) misses the code cache, and subsequently migrate execution to a different ISA, albeit probabilistically. Note that the PSR virtual machine can find in its internal structures only those indirect jump targets and function call sites that have been translated so far, and will result in a code cache miss for all others. Any surviving gadget that is *viable* for JIT-ROP must ideally avoid migration to a different ISA, and therefore begin at an already translated indirect jump or function call site. This imposes serious limitations on the JIT-ROP attack surface that has already been weakened by PSR. Figure 5.3 represents this as $G_{JIT-ISA}$, the true size of the attack surface for a JIT-ROP attack on heterogeneous-ISA PSR.

We find that out of the 294 surviving gadgets from PSR, 267 gadgets cause a security breach violation in the PSR virtual machine, thereby triggering a probabilistic migration to a different ISA. This leaves the attacker with only 27 gadgets, on average, that do not flag a violation and could potentially bypass migration to a different ISA. Furthermore, as shown in Figure 5.7, we note that our infrastructure is capable of being migration-safe on an average of 78% of the time, in either direction. This implies that gadgets in the remaining 22% of the basic blocks are

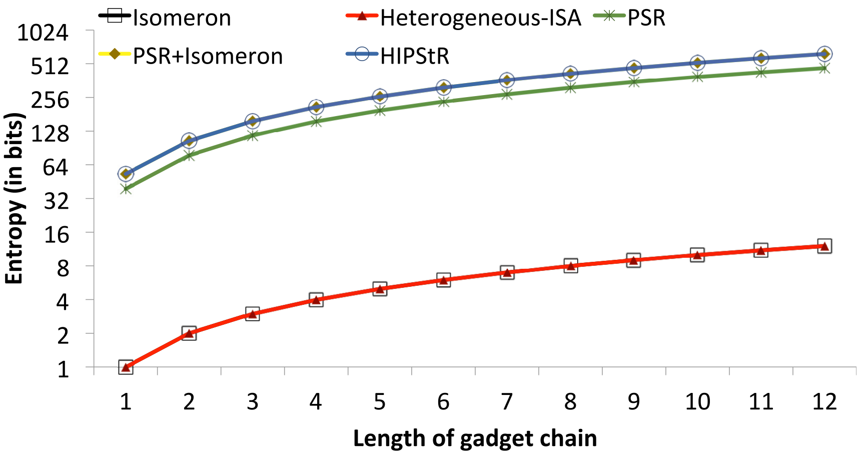

Figure 5.8: Entropy Comparison

still *viable* candidates for JIT-ROP. However, we find that these remaining gadgets are insufficient to even construct a four-gadget shellcode exploit, let alone complex exploits.

Tailored attacks. To further explore the synergy of the HIPStR components, we compare the combined entropy of HIPStR with the two individual components of HIPStR (PSR and heterogeneous-ISA migration) alone, as well as a hybrid of Isomeron and our PSR approach (See Figure 5.8). We make two important observations. First, any system that implements only Isomeron or only Heterogeneous-ISA migration suffers from extremely low entropy, which cannot be amortized unless the gadget chain is long enough. For example, every one out of as low as 256 attempts will succeed for a gadget chain that is 8 gadgets long. Second, the just-in-time nature of PSR inherently enables re-randomization upon a crash. Therefore, the attacker is always presented with a re-randomized version of the code cache on both ISAs, for every brute force attempt. Although PSR by itself is susceptible to JIT-ROP, this characteristic of PSR makes brute forcing a JIT-ROP attack significantly harder on a system that implements both diversification and PSR.

This might suggest that a system implementing a combination of PSR and Isomeron is as effective as HIPStR. However, we note that entropy as a metric by itself does not completely capture the effect of randomizing the instruction sets. To observe this effect, we turn to tailored

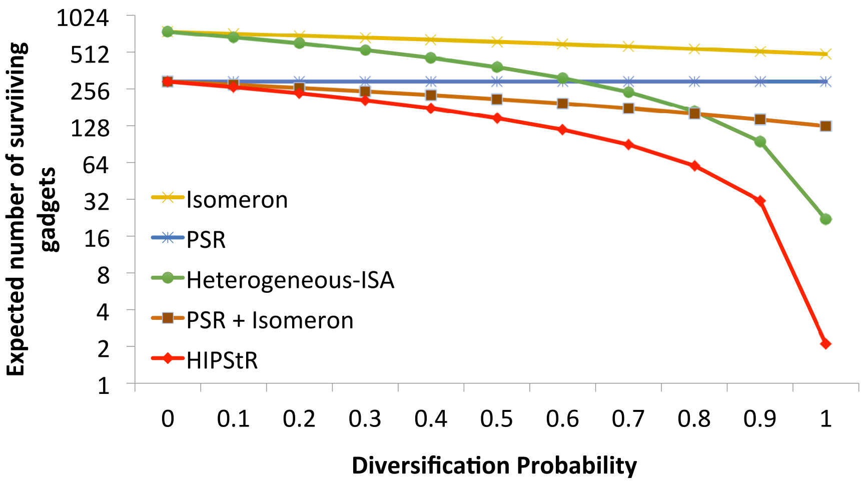

Figure 5.9: Effect of diversification on attack surface

attacks that bypass both same-ISA (Isomeron) and heterogeneous-ISA diversification. An attacker who is aware of the diversification could construct exploits that interleave gadgets from both the original and the diversified versions. For example, on HIPStR, one could craft an exploit that alternates gadgets between x86 and ARM, such that the all meaningul computation is performed by x86 gadgets, while the ARM gadgets are all *nop*s that switch execution to x86 without clobbering already established state. Another example of such a tailored attack would be to use those gadgets that are unaffected by diversification, i.e, those gadgets that perform the same intended malicious operation regardless of what ISA/software version they execute.

Figure 5.9 examines the JIT-ROP attack surface of each technique in the face of such tailored attack mechanisms. When the probability of diversification (the probablility of switching ISAs or program variants between gadgets) is zero, such an attack surface will include all gadgets present in the code cache. As the diversification probability increases, the attacker would find it increasingly harder to brute force a JIT-ROP attack.

We note that the system implementing both PSR and Isomeron is as effective as HIPStR when the diversification probability is zero, but the two systems rapidly diverge in their effectiveness as the probability increases. In fact, at a probability of one, at which we diversify execution for every gadget, HIPStR manages to have an average of just two surviving gadgets in its attack

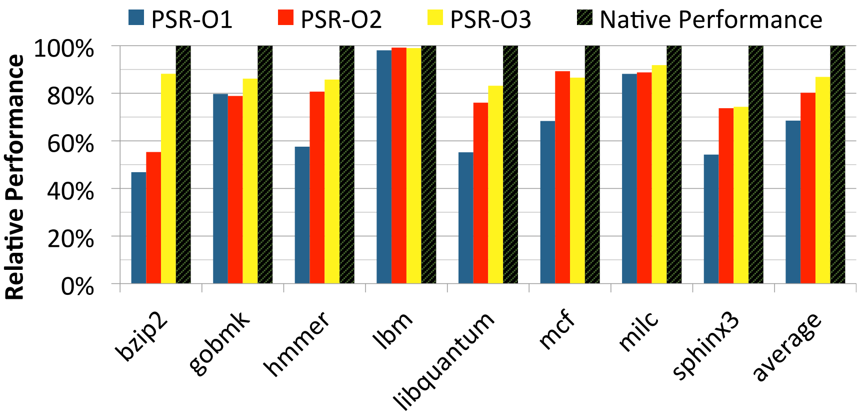

Figure 5.10: Performance at different optimization levels

Table 5.4: Performance Optimizations for PSR

Level	**Optimizations**
-O0	No Optimization.
-O1	Machine Block Placement, Branch Inlining and Superblock Formation.
-O2	-O1 optimizations, Global Register Cache.
-O3	-O2 optimizations, PSR with a register bias.

surface, while the attack surface of the system implementing PSR and Isomeron still comprises hundreds of gadgets. It is worth noting here that on HIPStR, we failed to find any surviving gadgets in five out of the eight applications we benchmark, thereby completely thwarting such tailored attacks.

Aside from these eight SPEC applications, we also evaluate the effectiveness of HIPStR on the network facing daemon *httpd*, a classic target of ROP attacks. Our evaluation shows that the attack surface of *httpd* is composed of 169,272 gadgets. PSR successfully obfuscates 99.7% of the gadgets, requiring 1.8x10^{39} attempts to brute-force (still computationally infeasible). Furthermore, while 84 gadgets are available for JIT-ROP, only two survive heterogeneous-ISA migration. These are insufficient to generate even the simplest shellcode exploit.

In our evaluation, we find that it is more likely to find large gadgets that populate multiple registers at a time, and are unaffected by diversification on the same ISA, rather than different ISAs. Note that our experiments only measure the number of surviving gadgets in the face

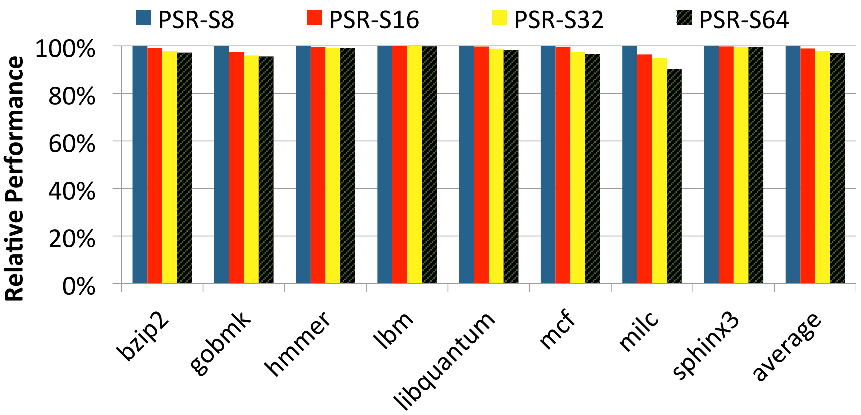

Figure 5.11: Effect of additional stack memory overhead

Table 5.5: Entropy Levels for PSR

Level	**Description**
-S8	Stack frame size is expanded by 8KB.
-S16	Stack frame size is expanded by 16KB.
-S32	Stack frame size is expanded by 32KB.
-S64	Stack frame size is expanded by 64KB.

of such tailored attacks. We expect that chaining together these gadgets to craft an exploit payload is a much more daunting task, because not only is brute-forcing such an attack under PSR extremely hard, but heterogeneous-ISA migration involves stack transformation that could potentially clobber the exploit payload on the stack.

The strength of PSR lies in its ability to defeat brute force attacks, while simultaneously reducing the attack surface. It amplifies the entropy of heterogeneous-ISA migration making a brute force attack on the combined defense infeasible. Conversely, heterogeneous-ISA migration has the ability to shield PSR from a JIT-ROP attack attempting to bypass randomization. Together, they form a formidable defense.

5.6.2 Performance Evaluation

Steady State Performance. Figure 5.10 shows the steady state performance overhead of PSR and the effect of each performance optimization. We do not gain a significant performance

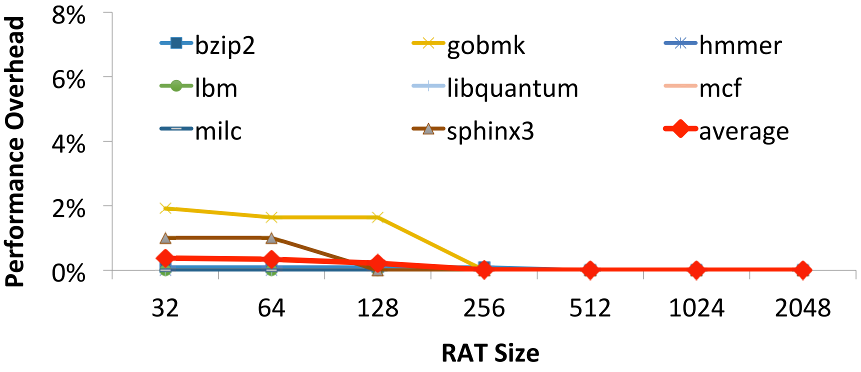

Figure 5.12: Effect of RAT size on Performance

boost from the O1 level of optimizations that we apply to improve the instruction cache performance. However, we observe an average of 13% improvement in performance due to a small global register cache that holds just 3 registers. This, in large part, is due to the fact that short and tight loops operate on a small set of input registers, but dominate most of a program's execution. Finally, we find that operating PSR in a register-bias mode can further improve performance by an average of 5.5%, thereby reducing the overall performance degradation from native execution to just 13.14%.

Since PSR relies on large stack frames to provide higher entropy, it is important to measure the performance effects due to the extra stack memory used. Figure 5.11 shows the steady state performance overhead at different entropy levels. The performance only drops by an average of 2.96% even after expanding individual stack frame sizes by as much as 64KB. This is because the stack frames become very sparse, and large empty spaces between items (e.g., larger than a cache line) do not place pressure on the cache.

PSR takes advantage of a return address table (Section 5.2) to protect the code cache from becoming compromised. Figure 5.12 shows the effect of the RAT size on performance. We incur an average of 0.37% performance overhead even with as small as a 32-entry RAT. In fact, we observe no noticeable degradation in performance with a RAT that can hold just 512 entries. This is because the distance between a *call* and a *return* instruction is generally so short that we seldom incur a RAT miss.

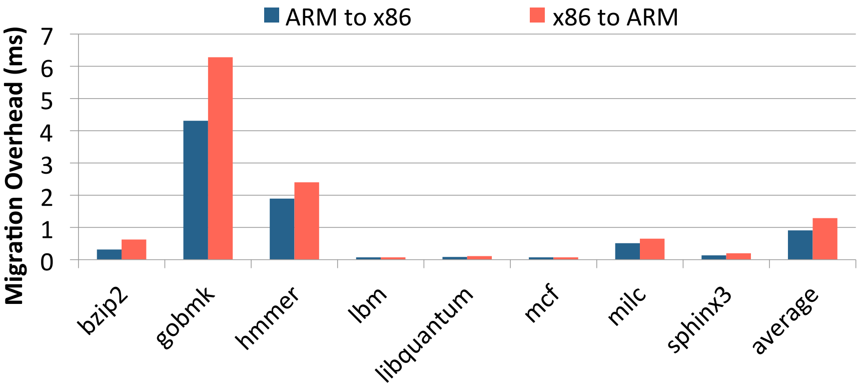

Figure 5.13: Migration Overhead

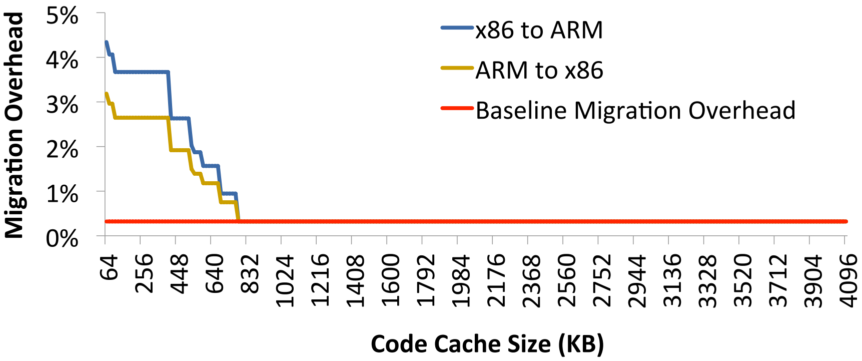

Figure 5.14: Effect of code cache size on Performance

Migration Overhead. Figure 5.13 shows the state transformation overhead due to heterogeneous-ISA process migration. We report an average migration overhead of 909 microseconds when migrating from ARM to x86, and 1.287 ms in the other direction, resulting in an overall baseline migration overhead of 0.32% resulting from the migrations initiated to improve performance. Our migration policy ensures that we switch ISAs only when a program's ISA preference changes as it enters a new program phase, or when the PSR virtual machine suspects a security breach, i.e, when we encounter an indirect control transfer that results in a code cache miss.

Figure 5.14 shows the effect of code cache size on migration overhead. We record zero indirect control transfers that miss a code cache as small as 768 KB, resulting in no measurable overhead for security-induced migrations. In steady state execution, it is highly unlikely that we

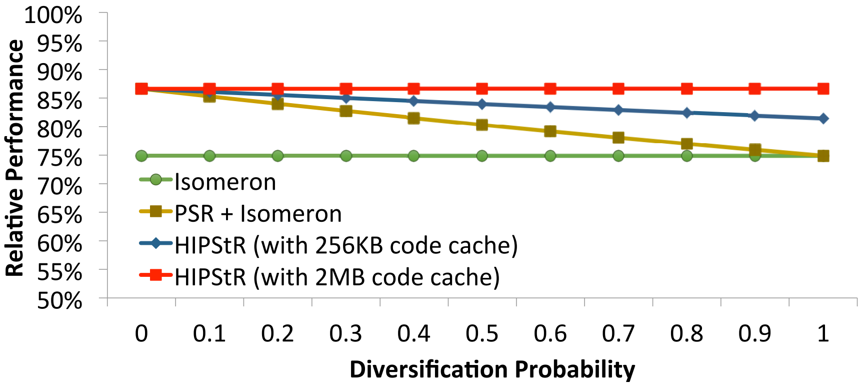

Figure 5.15: Performance Comparison with Isomeron

return to a function whose translation has been evicted, or we make an indirect jump or a function pointer call to an evicted region. For example, *gobmk* makes 65,746 function pointer calls within a span of one second, but none miss the code cache.

Finally, Figure 5.15 compares the performance of HIPStR with Isomeron, the only other technique that defends against JIT-ROP, along with a system that implements both PSR and Isomeron, for the six common applications that we benchmark. Since Isomeron invokes execution path diversification for each function call and return, they report a higher performance overhead. In fact, they mention that this degradation in performance is expected since their program shepherding renders CPU optimizations like branch prediction ineffective [DLS+15]. As the diversification probability increases, HIPStR performance only slightly decreases with the smaller code cache, but still manages to provide higher performance than both Isomeron, as well as the combination of PSR and Isomeron. Overall, HIPStR outperforms Isomeron by an average of 15.6% while simultaneously providing signficantly higher entropy, and thereby higher resistance to brute force, JIT-ROP, and tailored attacks that bypass diversification.

5.7 Conclusion

Heterogeneous-ISA CMPs have been shown to provide significant performance and energy gains. In this chapter, we showcase their security benefits through a novel defense called HIPStR

(Heterogeneous-ISA Program State Relocation) that combines dynamic randomization of program state with non-deterministic process migration between heterogeneous-ISAs. To demonstrate the full potential of the proposed security defense, we subject it to a series of malicious ROP-style attacks, including classic return-into-libc, jump-oriented programming, simple brute force, and just-in-time code reuse. Consequently, we make the following major observations:

- The sheer amount of entropy provided by our defense renders brute force attacks such as Blind-ROP [BBM^{+}14] practically impossible, for current or even distant future microprocessors.

- Our ability to perform seamless and instantaneous execution migration across heterogeneous ISAs significantly inhibits just-in-time code reuse attacks, forcing an attacker to construct heterogeneous-ISA exploits that are extremely hard to execute, as shown in Section **??**.

- Our performance focused migration policy and our optimization techniques help us outperform Isomeron [DLS^{+}15], the only other JIT-ROP defense, by an average of 15.6%, while simultaneously providing greater security guarantees against brute force, JIT-ROP, and more tailored attacks geared to break execution path diversification.

- We defend against all variants of classic ROP-style attacks [RBSS12, BJFL11, Moo10, SD97], and reduce their overall attack surface to such an extent that it is difficult to construct a four-gadget shellcode exploit, let alone achieve Turing-completeness.

Acknowledgements

Chapter 5, in full, is a reprint of the material as it appears in proceedings of ASPLOS 2016. Venkat, Ashish; Shamasunder, Sriskanda; Shacham, Hovav; Tullsen, Dean M., HIPStR: Heterogeneous-ISA Program State Relocation, 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April, 2016. The

dissertation author was the primary investigator and author of this paper.

The security evaluation part of chapter 5 (subsection 5.6.1) is joint work with Sriskanda Shamasunder. The dissertation author is also particularly grateful to Arvind Krishnaswamy, Koichi Yamada, and Rajan Palanivel for their numerous suggestions during the early days of PSR's formulation, and for their contributions to the United States Patent US009135435B2 [VKYP15].

Chapter 6

Composite-ISA Architectures

Chapter 4 demonstrates substantial advantages of the synergistic combination microarchitectural [KFJ$^+$03, KTR$^+$04, KTJ06] and ISA heterogeneity [DVT12, VT14, VSST16]. The advantages of varying the ISA is that we can tune not just microarchitectural features (cache sizes, window sizes, etc.), but can also tune features such as virtual register size, vector support, addressing mode availability, etc. to the needs of the code executing, improving single-thread performance, multithreaded workload throughput, and overall energy efficiency. However, despite their potential for greater performance and energy efficiency, the deployment of heterogeneous-ISAs on a single chip is non-trivial due to a number of practical concerns. First, heterogeneous-ISA multicores incur particular overheads owing to fat binaries and a complex process migration scheme that includes binary translation and stack transformation. Second, integration of multiple vendor-specific commercial ISAs on a single chip is fraught with significant licensing, legal, and verification costs and barriers.

This chapter significantly alleviates these concerns by presenting a design paradigm, the *composite-ISA architecture* that can recreate the effects of multi-ISA heterogeneity using a single composite-ISA. The composite-ISAs are derived by leveraging a large superset ISA that resembles the Intel x86 and offers customization along five axes of diversity: (1) register depth (8 to 32 programmable registers), (2) register width (32 vs 64-bit), (3) instruction complexity (1:1 vs 1:n micro-op encoding), (4) predication (full vs partial), and (5) specialized support (vector vs scalar). This chapter also features a suite of efficient code generation techniques that can take advantage of the underlying composite-ISA features, and migration strategies to switch between composite-ISAs with overlapping feature sets in a programmer-transparent manner. In this way, we can achieve far greater ISA diversity, enabling the full performance and energy benefits of a multi-ISA design, without the issues of multi-vendor licensing, fat binaries, and complex migration schemes. Furthermore, this work features a comprehensive analysis of the hardware implications of the custom feature set options, including a full synthesized RTL design of multiple versions of the x86 decoder.

6.1 ISA Feature Set Derivation

In this section, we describe our superset ISA. It resembles x86, but with an additional set of features that can be customized along 5 different dimensions: register depth, register width, opcode and addressing mode complexity, predication, and data-parallel execution. While we construct the superset ISA using extensions and mechanisms completely consistent and compatible with the existing Intel x86 ISA, we note that greater levels of customization can be achieved by creating a new (superset) ISA from scratch. We start with x86 because it not only already employs a large set of the features we want, but it has a clear history and process for adding extensions. We further study, in this section, the code generation impact, processor performance, power, and area implications of each dimension.

Register Depth. The number of programmable registers exposed by the ISA to the compiler/assembly programmer constitutes an ISA's register depth. The importance of register depth as an ISA feature is well established due to its close correlation to the actual register pressure (number of registers available for use) in any given code region [CAC$^+$81, AH82, CH90, CFR$^+$91]. While most compiler intermediate representations, including GCC's Register Transfer Language [DF80, SD09] and LLVM's bitcode [LA04a], allow for a large number (potentially infinite) of virtual registers, the number of architectural registers is limited, resulting in spills and refills of temporary values into memory, and limiting the overall instruction-level parallelism [AS17, Akr17].

Register depth not only affects efficient code generation, but significantly impacts machine-independent code optimizations due to (register pressure sensitive) redundancy elimination and re-materialization (re-computation of a temporary value to avoid spills/refills) [BCT92, BGM$^+$89, GB99, CS98, ZA03]. For example, decreasing the register depth from 32 to 16 registers in our custom feature sets results in a increase of 3.7% in stores (spills), 10.3% in loads (refills), 3.5% increase in integer instructions and 2.7% in branch instructions (rematerialization) on the SPEC CPU2006 benchmarks compiled using the methodology described in Section 6.2.

Furthermore, the backend area and power is strongly correlated to the ISA's register depth potentially impacting the nature and size of microarchitectural structures such as the reorder buffer and the physical register file. In processors that support register renaming (e.g. dynamically scheduled), power and area will scale with physical register file size rather than the architectural register depth. However, the physical register file size scales with the virtual register depth. In our superset ISA, we allow register depth to be customized to 8, 16, 32, and 64 registers. A composite-ISA design that customizes each core with a different register depth alleviates the register pressure of impacted code regions by migration to a core with greater register depth, and at the same time saves power by enabling smaller microarchitectural structures in other cores.

Register Width. Like register depth, the register width of an architecture impacts performance and efficiency in several different ways. First, wider data types implies wider pointers allowing access to larger virtual memory and avoiding unnecessary memory mapping to files. However, wide pointers potentially expand the cache working set when stored in memory as part of an aggregate object, thereby negatively impacting performance on applications that otherwise have smaller working sets [ARM]. Second, wider registers can often be addressed as individual sub-registers enhancing the overall register depth of the ISA. Most compilers' register allocators take advantage of sub-registers (e.g., *eax*, *ax*, *al* etc) and perform aggressive live range splitting and sub-register coalescing [CS98, BGM^{+}89, GA96, LGAT00, BDEO97, LA04a]. Third, emulating data types wider than the underlying ISA/core's register width not only requires more dynamic instructions, but could potentially use up more registers and thereby adversely impact register pressure.

Finally, wider registers imply larger physical register files in the pipeline which impacts both core die area and overall power consumption. In our experiments, doubling the register width from 32 bits to 64 bits could impact processor power by as much as 6.4% across different register depth organizations. Our superset ISA supports both 32-bit and 64-bit wide registers like x86, but we modify the instruction encoding to eliminate any restrictions on the addressing of a particular register, sub-register, or combination thereof.

Instruction Complexity. The variety of opcodes and addressing modes offered by an instruction set controls the mix of dynamic instructions that enter and flow through the pipeline. Incorporating a reduced set of opcodes and addressing modes into the instruction set could significantly simplify the instruction decode engine, if chosen carefully. In particular, we strive for 1:1 instead of a 1:n macro-op to micro-op translation, saving as much as 9.8% in peak power and 15.1% in area. However, for some code regions, such a scheme could increase the overall code size potentially impacting both the overall instruction cache accesses and instruction fetch and energy.

To derive such a reduced feature set, we carve out a subset of opcodes and addressing modes from our superset ISA that can be implemented using a single micro-op, essentially creating custom cores that implement the x86 micro-op ISA, albeit with variable-length encoding. The reduced feature set, called *microx86* in this work, adheres to the *load-compute-store* philosophy followed by most RISC architectures. As a result, we could view this option as RISC vs CISC support. While one could conceive a more aggressive low-power implementation of *microx86* that implements fewer opcodes and further recycles opcodes for a more compact representation [LAR$^+$15, BBWA07, BABJW11, BABW14, Bau17], we keep all the same opcodes, and thus follow x86's existing variable-length encoding and 2-phase decoding scheme. This not only maintains consistency with existing implementations of x86, but prevents us from incurring the binary translation costs associated with multi-vendor heterogeneous-ISA designs. However, this does mean we cannot completely replicate the instruction memory footprint of a theoretically minimal representation.

Predication. Predication converts control dependences into data dependences in order to eliminate branches from the instruction stream and consequently take pressure off the branch predictor and associated structures [AKPW83, MLC$^+$92, MHB$^+$94, MHM$^+$95, AHM97], while also removing constraints on the compiler's instruction scheduler. Modern ISA implementations of predication can be classified into three categories: (a) partial predication that allows only a subset of the ISA's instructions to be predicated, (b) full predication that allows any instruction to be predicated using a predefined set of predicated registers, and (c) conditional execution that allows any instruction to be predicated using one condition code register.

The x86 ISA already implements partial predication via conditional move instructions that are predicated on condition codes. In this work, we add full predication support to our superset ISA, allowing any instruction to be predicated using any available general-purpose register using the if-conversion strategy described in Section 6.2. While predication eliminates branch dependences, aggressive if-conversion typically increases the number of dynamic instructions, thus placing more pressure on the instruction fetch unit and the instruction queue. In our custom feature sets that offer predication, we observe an average increase of 0.6% in the number of dynamic instructions with a reduction of 6.5% in branches.

Data-Parallel Execution. Most modern instruction sets offer primitives to perform SIMD operations [Fly72, Dun90, ARM, Int] to take advantage of the inherent data-level parallelism in

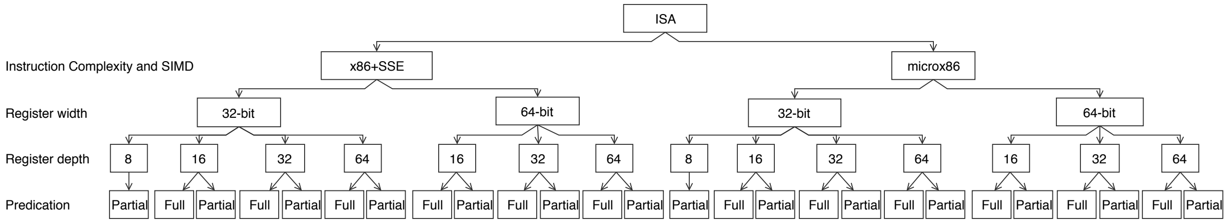

Figure 6.1: Derivation of Composite Feature Sets from a Superset ISA.

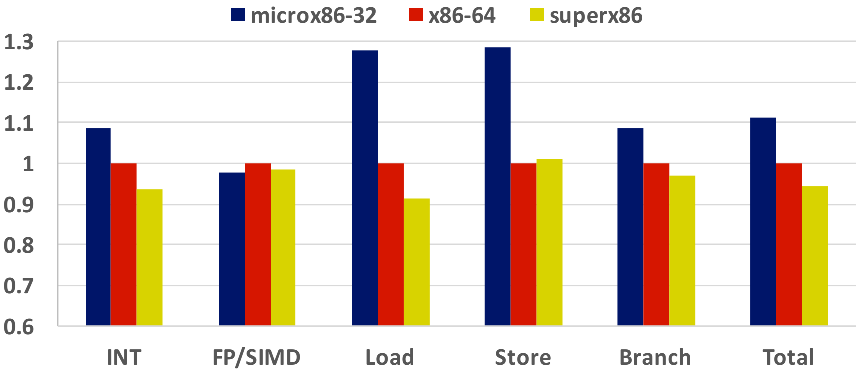

Figure 6.2: Instruction Mix (normalized to x86-64) for SPEC2006

specific code regions. The x86 ISA already supports multiple feature sets that implement a variety of SIMD operations. We include the SSE2 feature set in our superset ISA that can compute on data types that are as wide as 128 bits as implemented in the gem5 simulator [BDH$^+$06]. Furthermore, we constrain our microx86 implementations to not include SSE2 since more than 50% of SIMD operations rely on 1:n encoding of macro-op to micro-op. In our composite-ISA design, cores that do not implement SSE2 save 7.4% in peak power and 17.3% in area. They execute a precompiled scalarized version of the code when available, and in most cases, migrate code regions that enter intense vector activity to cores with full vector support.

In summary, we derive 26 different custom feature sets along the five dimensions described above, as shown in Figure 6.1. We exclude full predication from our 32-bit feature sets that have less than 16 registers since they already suffer from extremely high register pressure. Similarly, we constrain 64-bit ISAs to have a register depth that is greater than or equal to 16.

Figure 6.2 shows the dynamic instruction (micro-op) breakdown for the SPEC CPU2006 benchmarks on three different custom ISAs: (a) the 32-bit version of microx86 with a register depth of 8 and no additional features (smallest feature set in our exploration), (b) the x86-64 ISA with SSE and no other customizations, and (c) the superset ISA which implements all the features described above. Due to the high register pressure in microx86-32, it incurs an average of 28% higher memory references than x86-64 and an overall expansion of 11% in the number of micro-ops. Also compared to the x86-64, we find that the superset ISA, owing to the diverse set of custom features added, sees an average reduction of 8.5% in loads (spill elimination), 6.3% in integer instructions (aggressive redundancy elimination), and 3.2% in branches (predication).

6.2 Compiler and Runtime Strategy

One of the major goals of this chapter is to provide the benefits of ISA diversity without incurring binary translation and state transformation costs, by leveraging custom feature sets of the same ISA. In this section, we describe our compilation strategy that generates code to efficiently take advantage of the underlying custom feature sets, and our runtime migration strategy that allows code regions to seamlessly migrate back and forth between different custom feature sets, without the overhead of full binary translation and/or state transformation.

6.2.1 Compiler Toolchain Development

Compilation to a superset ISA or a combination of custom feature sets allows different code regions to take advantage of the variety of custom feature sets implemented by the underlying hardware. For example, code regions with high register pressure could be compiled to execute on a feature set with greater register depth, and code regions with too many branches could be compiled to execute predicated code. We leverage the LLVM MC infrastructure [LA04a] to efficiently encode the right set of features supported by the underlying custom design and further propagate it through various instruction selection, code generation, and machine-dependent optimization passes. We further take advantage of the MC code emitter framework to encode feature sets such as register depth and predication that require an extra prefix.

In order to convert the existing x86 backend to that of the superset ISA, we first include the additional 48 registers in the ISA's target description and further associate code density costs with it. This enables the register allocator to always prioritize the allocation of a register that requires fewer prefix bits to encode it. Furthermore, we allow each of these registers to be addressed as a byte, a word, a doubleword, and a quadword register, thereby smoothly blending into the register pressure tracking and subregister coalescing framework.

We next implement full predication in x86 by re-purposing the existing machine-dependent *if-conversion* framework of LLVM that implements if-conversions in three scenarios: (a) *diamond* – when a true basic block and a false basic block split from an entry block and rejoin at the tail, (b) *triangle* – when the true block falls through into the false block, and (c) *simple* – when the basic blocks split but do not rejoin, such as a break statement within a conditional. For every such pattern in the control flow graph, if-conversion is performed if determined profitable. The profitability of if-conversion is based on branch probability analysis, the approximate latency of machine instructions that occur in each path, and the configured pipeline depth. We further add the if-conversion pass as a pre-scheduling pass for the X86 target – this allows the scheduling pass to take full advantage of the large blocks of unscheduled code created by the if-conversion.

To implement *microx86*, we modify the existing x86 backend to exclude all opcodes, addressing modes, and prefixes that decode into more than one micro-op during machine instruction selection. While LLVM's instruction selector, for the most part, identifies a replacement *ld-compute-st* combination for the excluded addressing mode, certain IR instructions such as tail jumps and tail call returns require explicit instruction lowering.

For each code region (where we roughly define a region as the set of code that dominates a simpoint phase), the compiler must now make a global (or regional) decision about which features to use and which to skip. Further, the compiler should make these decisions with some knowledge of the features of the cores for the processor on which it will run. Because we examine so many core combinations in our design space exploration, we assume the compiler makes good decisions about which features to include in compilation in all cases. So despite the fact that features included in compilation are not static for a region (across experiments), we can still see clear trends in code affinity for features. For example, the highly register-pressure constrained benchmark *hmmer* is consistently compiled to use all 64 registers across all code regions. In contrast, only one phase of the benchmark *bzip2* is compiled with a register depth of 64, with the rest of the seven regions typically compiled with a register depth of 32. Similarly, the compiler employs predication in four regions of the benchmark *milc*, but does not find it to be profitable in two other regions of the same benchmark. Furthermore, due to the high register pressure and lack of free registers on *hmmer*, the compiler harnesses the full suite of complex addressing modes offered by x86 and seldom employs predication. These decisions propagate through the rest of the optimization passes resulting in carefully optimized code for the underlying feature set.

6.2.2 Migration Strategy

Process migration across overlapping custom feature sets could involve two scenarios. In an upgrade scenario, a process is compiled to use only a subset of the features implemented by the core to which it is migrated, and therefore can resume native execution immediately after migration (no binary translation or state transformation). Conversely, in a downgrade scenario, the core to which a process is migrated implements only a subset of the features being used by the running code, which necessitates minimal binary translation of unimplemented features. We outline the following low-overhead mechanisms to handle feature downgrades.

Owing to the overlapping nature of the feature sets (same opcode and instruction format), feature emulation entails only a small set of binary code transformations, in contrast to full blown cross-ISA binary translation. First, when we downgrade from *x86* to *microx86*, we perform simple addressing mode transformations by translating any instruction that directly operates on memory into a set of simpler instructions that adhere to the *ld-compute-st* format. Second, we downgrade to a feature set with a smaller register depth by translating higher (unimplemented) registers into memory operands using a register context block [SH98, Bel05, DVT12], a commonly used technique during binary translation to register pressure-constrained ISAs. Third, we perform long-mode emulation [Bel05, VT14] and use *fat pointers* implemented using *xmm* registers in order to emulate wider types on a 32-bit core. Finally, we employ simple reverse if-conversions to translate predicated code back to control-dependences.

6.3 Decoder Design and Implementation

In this section, we describe our customizations to the x86 instruction encoding and decoder implementation in order to support the 26 feature sets derived in the Section 6.1. We show that, due to the extensible nature of the x86 ISA, the decoder implementation requires minimal changes to support the new feature sets and has a small impact in terms of overall peak power and area.

6.3.1 Instruction Encoding

Feature extensions to the x86 instruction set are not uncommon. In accordance with its code density and backward compatibility goals, major feature set additions to x86 (e.g., REX, VEX, and EVEX) have been encoded by exploiting unused opcodes and/or by the addition of new (optional) prefix bytes. We use similar mechanisms to encode the specific customizations we propose as shown in Figure 6.3.

In order to double/quadruple the register depth of x86-64, we add a new prefix – REXBC, similar to the addition of the REX (register extension) prefix that doubled both the register width and depth of x86-32, giving rise to the x86-64 ISA. In particular, the REXBC prefix encodes 2 extra bits for each of the 3 register operands (input/output register, base, and index), which is further combined with 4 bits from the REX, MODRM, and SIB bytes, to address any of the 64 programmable registers. Furthermore, we use the remaining 2 bits of the REXBC prefix to lift

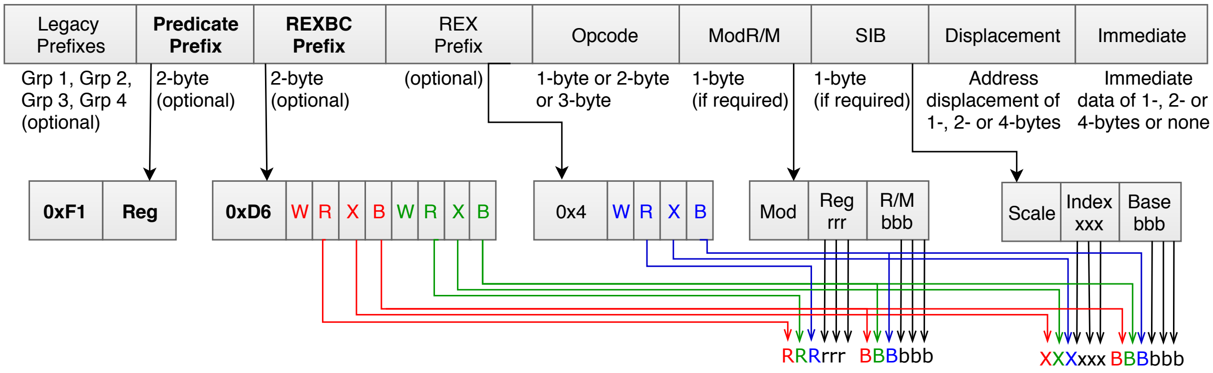

Figure 6.3: Customizations to the x86 Instruction Encoding

restrictions in x86 that do not allow certain combinations of registers and subregisters to be used as operands in the same instruction. Finally, we exploit an unused opcode **0xd6** in order to mark the beginning of a REXBC prefix.

Similarly, to support predication, we use a combination of an unused opcode **0xf1** and a predicate prefix. In order to efficiently support diamond predication (described in Section 6.2), we use the predicate prefix to encode both the nature (true/not-true) of the conditional (bit 7) and the register (bits 0-6) the instruction is predicated on.

All of the insights in this paper apply equally (if not more so) to a new superset ISA designed from scratch – such an ISA would allow much tighter encoding of these options.

6.3.2 Decoder Analysis

Figure 6.4 shows the step-by-step decoding process of a typical x86 instruction. Owing to the variable length encoding, x86 instructions typically go through a 2-phase decode process. In the first phase, an instruction-length decoder fetches and decodes raw bytes from a prefetch buffer, performs length validation, and further queues the decoded macro-ops into a macro-op buffer. These macro-ops are fused into a single macro-op when viable, and further fed as input into one of the instruction decoders that decode it into one or more micro-ops. The decoded micro-ops are subjected to fusion again, in order to store them in the micro-op queue and the micro-op cache

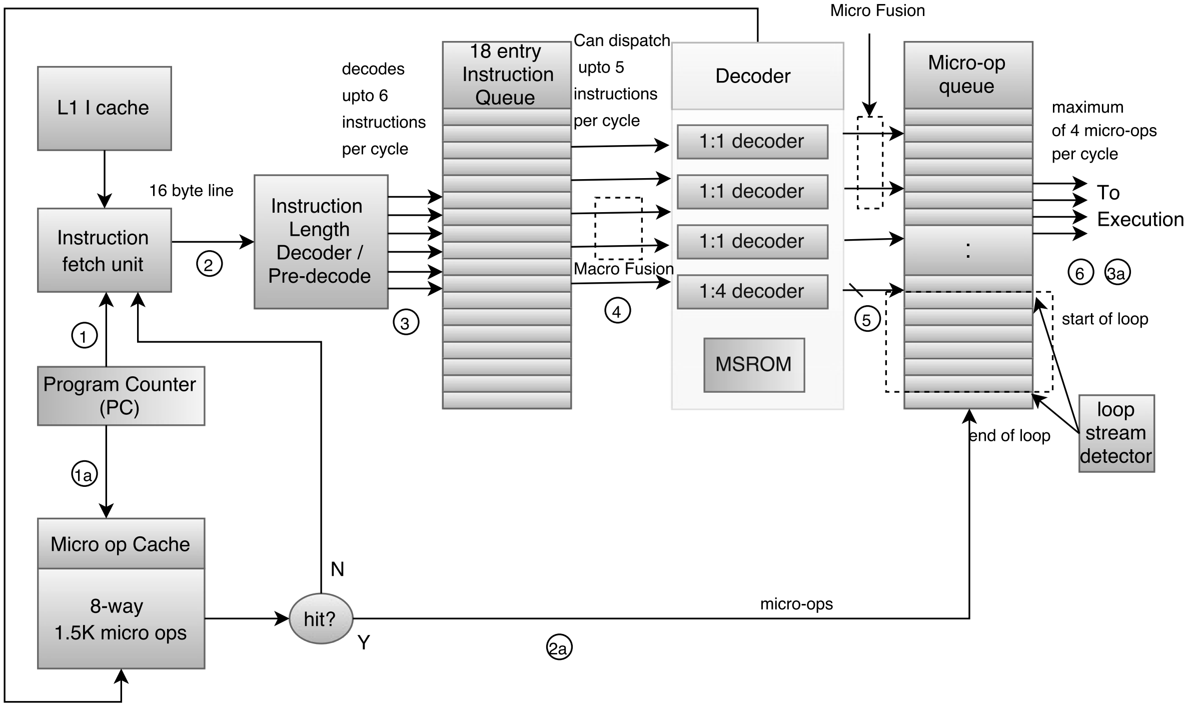

Figure 6.4: x86 Fetch/Decode Engine

in a compact representation, and are later unfused and dispatched as multiple micro-ops. The micro-op cache is both a performance and power optimization that allows the decode engine to stream decoded (and potentially fused) micro-ops directly from the micro-op cache, turning off the rest of the decode pipeline until there is a micro-op miss.

In our RTL implementation, we model an instruction length decoder based on the parallel instruction length decoder described by Madduri, et al [MST]. The instruction length decoder has three components: (a) an instruction decoder that decodes each byte of an incoming 8-byte chunk as the start of an instruction, decoding both prefixes and opcodes, (b) a speculative length calculator that speculatively computes the length of the instruction based on the decoded prefixes and opcodes, and (c) an instruction marker that checks the validity of the computed lengths, marks the begin and end of an instruction, and detects overflows into the next chunk.

Since our customizations affect the prefix part of the instruction, we modify the eight decode subunits of the instruction decoder to include comparators that generate extra decode signals to represent the custom register depth and predicate prefixes. These decode signals propagate through the speculative instruction length calculator and the instruction marker requiring wider multiplexers in the eight length subunits, the length control select unit, and the valid begin unit. These modifications to the instruction length decoder result in an increase of 0.87% in total peak power and 0.65% in area for our superset ISA.

Furthermore, we increase the width of the macro-op queue by 2 bytes to account for the extra prefixes. Since predication support and greater register depth in our superset ISA could potentially require wider micro-op ISA encoding, we increase the width of the micro-op cache and the micro-op queue by 2 bytes. Finally, for our *microx86* implementations, we replace the complex 1:4 decoder with a simple 1:1 decoder and forgo the microsequencing ROM. From our analysis, a decoder that implements our simplest feature set *microx86-32* consumes 0.66% less peak power and takes up 1.12% lesser area than the x86-64 decoder, and our superset decoder consumes 0.3% more peak power and takes up 0.46% more area than the x86-64 decoder. These variances do not include the increases or savings from the instruction length decoder.

6.4 Experimental Methodology

In this section, we describe our experimental methodology for the design space exploration and process migration strategy.

As shown in Table 6.1, our design space consists of 5 dimensions of ISA customizations and 19 micro-architectural dimensions. After careful pruning of configurations that are not viable (e.g., 4-issue cores with a single INT/FP ALU) or unlikely to be useful (full predication with 8 registers), this results in 26 different custom ISA feature sets, 180 microarchitectural configurations, and 4680 distinct single core design points, that each are spread across a wide range of peak power (4.8W to 23.4W) and area (9.4mm^2 to 28.6mm^2) distributions. The goal of our feature set exploration is to find an optimal 4-core multicore configuration using the

Table 6.1: Feature Exploration Space

ISA Parameter	**Options**
Register depth	8, 16, 32, 64 registers
Register width	32-bit, 64-bit registers
Instruction/Addressing mode complexity	1:1 macroop-microop encoding (load-store x86 micro-op ISA), 1:n macroop-microop encoding (fully CISC x86 ISA)
Predication Support	Full Predication like IA-64/Hexagon vs Partial (cmov) Predication
Data Parallelism	Scalar vs Vector (SIMD) execution
Microarchitectural Parameter	**Options**
Execution Semantics	Inorder vs Out-Of-Order designs
Fetch/Issue Width	1, 2, 4
Decoder Configurations	1-3 1:1 decoders, 1 1:4 decoder, MSROM
Micro-op Optimizations	Micro-op Cache, Micro-op Fusion
Instruction Queue Sizes	32, 64
Reorder Buffer Sizes	64, 128
Physical Register File Configurations	(96 INT, 64 FP/SIMD), (64 INT, 96 FP/SIMD)
Branch Predictors	2-level local, gshare, tournament
Integer ALUs	1, 3, 6
FP/SIMD ALUs	1, 2, 4
Load/Store Queue Sizes	16, 32
Instruction Cache	32KB 4-way, 64KB 4-way
Private Data Cache	32KB 4-way, 64KB 4-way
Shared Last Level (L2) Cache	4-banked 4MB 4-way, 4-banked 8MB 8-way

fully custom feature sets derived out of the superset ISA. Our objective functions that evaluate optimality include both performance and energy delay product (EDP), for both multithreaded and single-threaded workloads.

Our workloads include eight SPEC CPU2006 benchmarks further broken down into 49 different application phases using the SimPoint [SPHC02, PHVB^{+}03] methodology. In order to create equivalent phases for 26 different feature sets across the 8 applications, we first create simpoints on the commonly used x86-32 ISA with a simpoint interval of 100 million dynamic instructions, and then find equivalent start and end basic blocks using a combination of IR-level and MC-level data obtained using LLVM, and simpoint metadata obtained using the fast (atomic) simulation in the gem5 architectural simulator [BDH^{+}06].

We use the gem5 [BDH$^+$06] simulator to measure performance in both our inorder and out-of-order cores. We modify the gem5 simulator to include micro-op cache and micro-op fusion support in order to measure the impact of our customizations in light of existing micro-op optimizations. However, we do not employ micro-op fusion in our *microx86* ISA because each instruction only decomposes into one micro-op and the micro-op fusion unit doesn't yet combine micro-ops from different macro-ops. Our implementations of the micro-op cache and micro-op fusion are consistent with guidelines mentioned in the Intel Architecture Optimization Manual [Int09].

We perform a full RTL synthesis using the Synopsys Design Compiler to measure the decoder area and power overheads of each of the customizations we employ in our feature sets, as described in Section 6.3. We also use the McPAT [LAS$^+$09b] power modeling framework to evaluate the power and area of the rest of the pipeline. The peak power and area measurements we obtain out of McPAT simulations are key parameters to our exploration of this massive design space, which involves 196,560 gem5+McPAT simulations resulting in 49,733 core hours on a large 2 petaflop cluster. Furthermore, our multicore design search involves finding an optimal 4-core multicore out of a 102.5 trillion combinations that run all permutations of simpoint regions, for several different objective functions and constraints. As of this writing, the results we report for the fully custom feature set design are local optima, and therefore conservative estimates.

Finally, process migration in a composite-ISA architecture could potentially involve binary translation of unimplemented features in case of a feature downgrade. We measure this cost by performing feature emulation for each checkpointed code region that represents a simpoint, on artificially-constrained cores that only implement a subset of the features the simpoint was compiled to. We also report the overall cost of migration on our designs optimized for multiprogrammed workload throughput where threads often contend for the cores of their preference, and therefore may not always run on the core of their choice.

Table 6.2: x86-ized versions of Thumb, Alpha, and x86-64

microx86-8D-32W	**microx86-32D-64W**	**x86-16D-64W**
Thumb-like Features Load/Store Architecture Register Depth: 8 Register Width: 32 No SIMD support	**Alpha-like Features** Load/Store Architecture Register Depth: 32 Register Width: 64 No SIMD support CMOV Support	**x86-64-like Features** CISC Architecture Register Depth: 64 Register Width: 64 SIMD support CMOV Support
Exclusive Features: FP Support	**Exclusive Features:** None	**Exclusive Features:** None
Thumb-specific Features: Code Compression Fixed-length instructions (one-step decoding)	**Alpha-specific Features:** 2-address instructions Fixed-length instructions (one-step decoding) More FP Registers	**x86-specific Features:** None

6.5 Results

In this section, we present detailed results from our design space exploration that seeks to identify optimal combinations of ISA and microarchitectural features.

6.5.1 Performance and Energy Efficiency

This section elaborates on the findings of our design space exploration. We identify custom multicore designs that benefit from both hardware heterogeneity and feature set diversity, providing significant gains over designs that exploit only hardware heterogeneity. Furthermore, these designs recreate the effects or in most cases, surpass the gains offered by a fully heterogeneous-ISA CMP that implements a completely disjoint set of vendor-specific ISAs and requires sophisticated OS/runtime support. We conduct multiple searches through our design space to model different execution scenarios and different budget-constrained environments.

In each search, we identify three optimal 4-core multicore designs: (1) homogeneous x86-64 CMPs that employ cores that implement the same ISA and the same microarchitecture, (2) x86-64 CMPs that exploit hardware heterogeneity alone, and (3) composite-ISA x86-64 CMPs

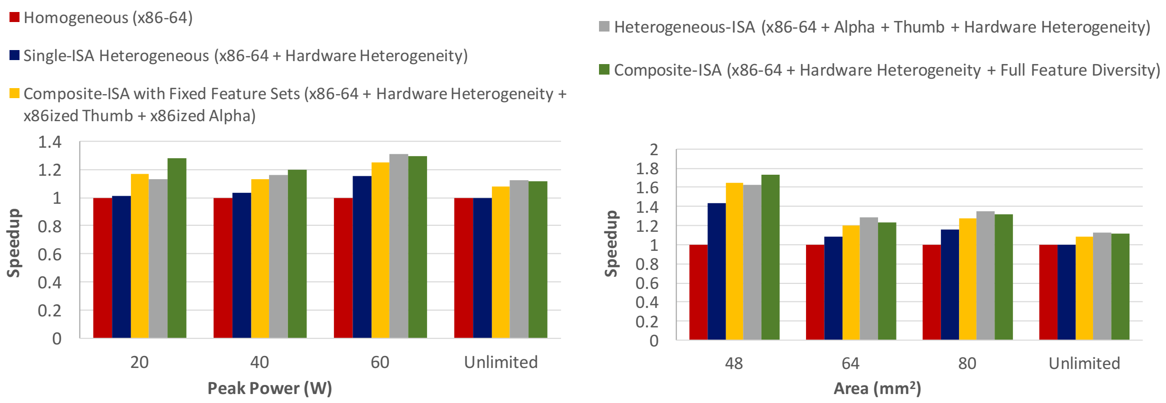

Figure 6.5: Multi-programmed workload throughput comparison (higher is better)

that exploit hardware heterogeneity and full feature diversity.

In addition, we also identify two intermediate design points of interest: (1) heterogeneous-ISA CMPs [VT14] that implement three fixed, disjoint, vendor-specific ISAs (x86-64, Alpha, Thumb), and (2) composite-ISA CMPs that exploit hardware heterogeneity and a limited form of feature diversity via three x86-based fixed feature sets that resemble the above vendor-specific ISAs. We use the latter design as a vehicle to demonstrate that vendor-specific ISA heterogeneity can be recreated to a large extent by carving out custom feature sets from a single sufficiently-diverse superset ISA. Table 6.2 offers a more detailed comparison.

Thus, we compare against a number of interesting configurations, but two are most revealing. Since our goal is to seek to replicate the advantage of multi-ISA heterogeneity over single-ISA heterogeneity, the single-ISA heterogeneous result is our primary baseline for comparison. However, the multi-vendor (x86, Thumb, Alpha) result represents our “goal” result that we are striving to match, yet with essentially a single ISA.

Figure 6.5 compares the performance and energy efficiency of the five optimal designs listed above, optimized to provide multi-programmed workload throughput when constrained under different peak power and area budgets. There are four major takeaways from this experiment. First, the designs that exploit feature diversity alongside hardware heterogeneity consistently and significantly outperform the designs that exploit only hardware heterogeneity. This is because

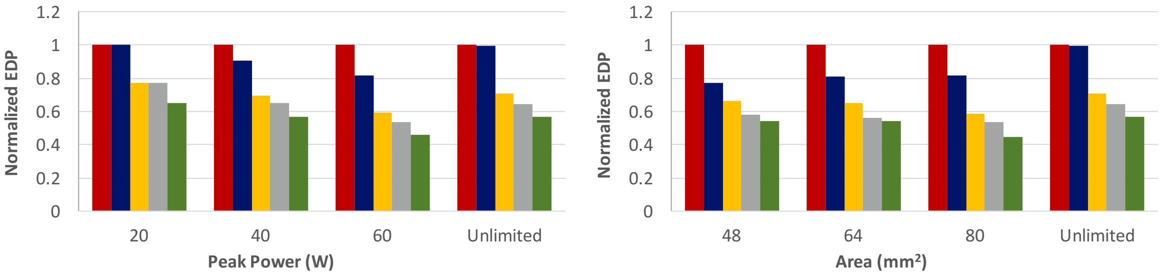

Figure 6.6: Multi-programmed workload EDP comparison (lower is better)

hardware heterogeneity tends to diminish when the amount of available chip real estate becomes too small or too generous. Second, we find in the resulting optimal architectures that in an especially tightly power/area constrained environment, every feature present in the superset ISA is implemented by at least one core in the composite-ISA design. When the constraints become more relaxed, the composite-ISA designs continue to implement at least 10 out of the 12 features described in Section 6.1. Third, the composite-ISA designs that implement the x86-ized versions of the vendor-specific ISAs trail slightly but generally match the gains of the fully heterogeneous-ISA designs. Fourth, the composite-ISA design with full ISA feature set diversity not only matches but frequently outperforms the fully heterogeneous-ISA design. This indicates that any loss due to the lack of specific ISA encoding and simplified hardware (e.g. decoders) is more than compensated for by the increased flexibility of the composable ISA features.

Overall, composite-ISA designs that exploit both hardware heterogeneity and full feature diversity outperform single-ISA heterogeneous designs that only exploit hardware heterogeneity by 17.6% on average, and by 30% under tight power constraints.

In Figure 6.6, we compare designs optimized to provide multi-threaded workload energy efficiency, measured as Energy Delay Product (EDP), while being constrained under different peak power and area budgets. We observe significant energy savings due to full ISA customization – an average of 31% savings in energy and 34.6% reduction in EDP over single-ISA heterogeneous multicore designs. This result was not necessarily expected, as the Thumb architecture still provides significant advantages over our most conservative microx86 core. However, many codes cannot use Thumb because of its limited features, while the composite-ISA architecture can combine microx86 with a variety of other features and make use of it far more often.

We next evaluate our designs optimized to provide high single thread performance and energy efficiency. That is, while the prior results optimized for four threads running on four cores, this exploration optimizes for one thread utilizing four cores via migration. When the designs are constrained by peak power budgets, we model them after the dynamic multicore topology [EBA$^+$11] where only one core is active at any given point of time, while the rest of the cores are powered off. Figure 6.7 compares designs optimized to provide high single thread performance and energy efficiency. Note that the peak power budgets are tighter in this case since we assume only one core to be powered on at a time. Due to the low power constraints, hardware heterogeneity provides only marginal improvements in performance and EDP. However, we observe that every feature of the superset ISA again manifests in at least one of the feature sets implemented by the composite-ISA design, allowing applications to migrate across different cores in order to take advantage of any required ISA feature. We observe an average speedup of 19.5% and an average EDP reduction of 27.8% over single-ISA heterogeneous designs. Moreover, owing to the many low-power and feature-rich *microx86* options available, we manage to outperform the fully heterogeneous-ISA design that implements vendor-specific ISAs, by 14.6% and reduce EDP by 3.5% under tight (5W) power constraints. The x86-ized versions of the fully heterogeneous-ISA designs again trail, but generally match up well, to the performance levels offered by vendor-specific ISA-heterogeneity. Once again, the composite-ISA design overall match the performance results of the fully heterogeneous-ISA design, while trailing slightly behind in terms of EDP.

When designs are constrained by area budget, the optimal multicore is typically composed of multiple small cores and one large core that maximizes single thread performance. In Figure 6.8, we evaluate the single thread performance and energy efficiency of the optimal designs under different area budgets. We observe that the composite-ISA designs sport two out-of-order

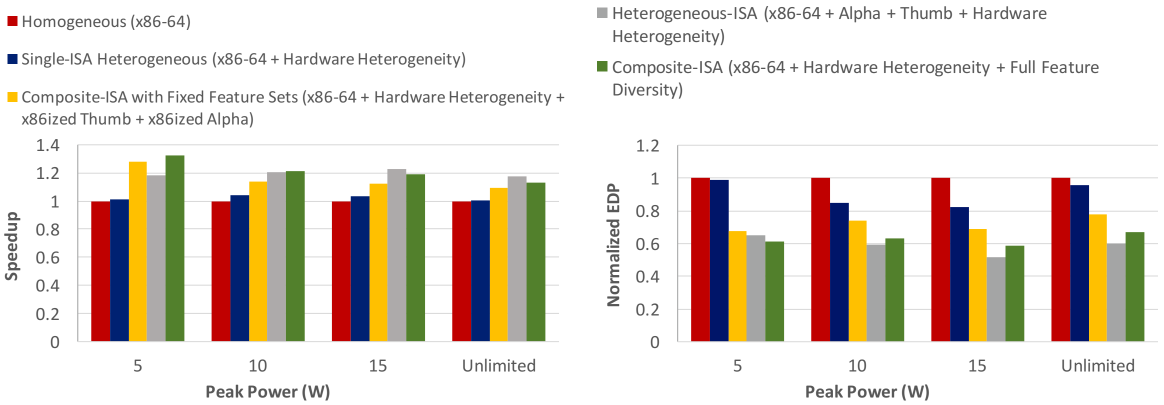

Figure 6.7: Single Thread Performance (higher is better) and EDP (lower is better) comparison under Peak Power Budget

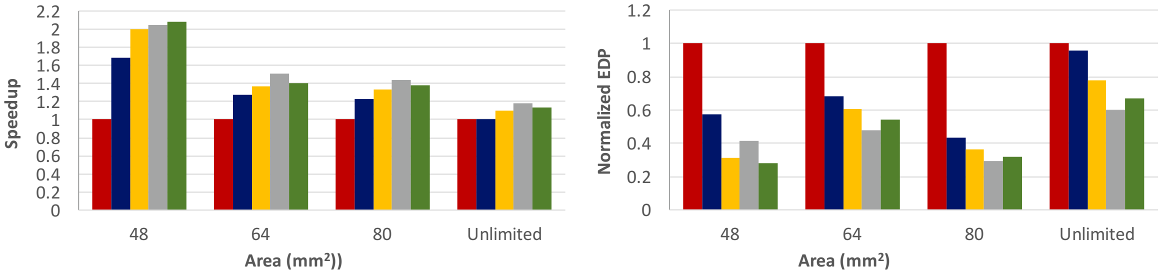

Figure 6.8: Single Thread Performance (higher is better) and EDP (lower is better) comparison under Area Budget

microx86 cores even under the most tightly area-constrained environment implementing different register depth and width features in each of them, allowing applications to migrate across cores and take advantage of the specific ISA features. While the fully heterogeneous-ISA design offers similar capabilities due to the area-efficient thumb cores, we note that migration across thumb and x86-64 cores is non-trivial and incurs significant overhead in comparison to the simpler migration across the two overlapping x86-based ISAs in our case. Moreover, under tight constraints, we are able to design more effective composite-ISA architectures due to the greater design options available (e.g., more efficient combination of 32-bit and 64-bit designs), saving an extra 13.2% in EDP when compared to fully heterogeneous-ISA designs.

Overall, we find that the composite-ISA design consistently outperforms the single-ISA heterogeneous design, resulting in an average speedup of 20% and a reduction of 21% in EDP.

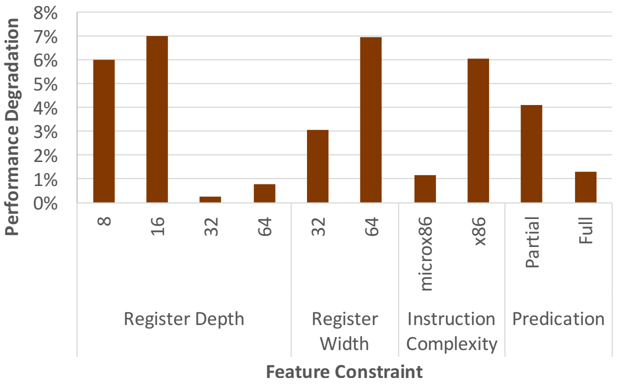

Figure 6.9: Performance Degradation over Composite-ISA Designs optimized for multi-programmed workload throughput at 48mm^2 budget, and under different Feature Constraints

6.5.2 Feature Sensitivity Analysis

Owing to their feature-rich nature, composite-ISA CMPs consistently offer significant performance and energy efficiency benefits, even in scenarios where hardware heterogeneity provides diminishing returns. One of the major goals of this design space exploration is to identify specific ISA features that contribute toward these benefits, and further help architects make more efficient design choices. However, since ISA features typically manifest as components of a larger feature set a core implements, it is generally non-trivial to measure the effect of a specific ISA feature in isolation.

In this section, we perform additional searches through our design space in order to understand the impact an ISA feature has over performance, energy, and transistor investment, by removing one axis of feature diversity at a time. As an example, consider the search for an optimal composite-ISA CMP that optimizes for multi-programmed workload performance under an area budget of 48mm^2, but constrained to only include designs that limit the number of architectural registers to 16 in all cores. If register depth is an important feature, the optimal design from this search is expected to perform worse than the one chosen through an unconstrained search.

Figure 6.9 shows the result of this experiment. We make several inferences here. First, constraining all cores to implement fewer than 32 architectural registers negates a significant

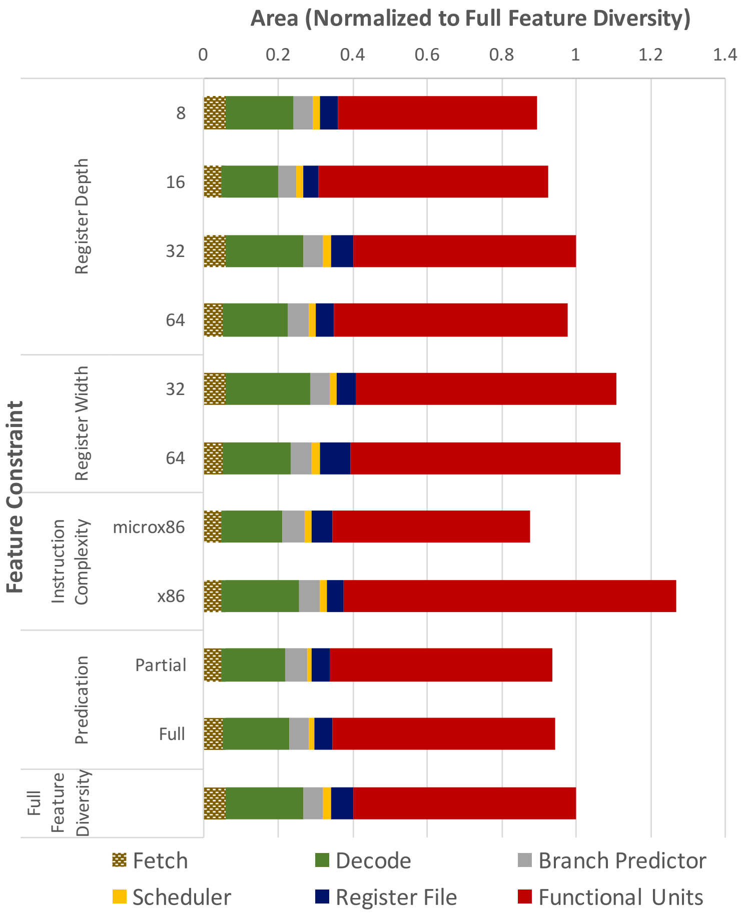

Figure 6.10: Transistor Investment by Processor Area normalized over that of Composite-ISA Designs optimized for multi-programmed workload throughput at 48mm^2 budget, and under different Feature Constraints

chunk of the performance gain due to feature diversity. Most optimal designs typically employ two or more cores with a register depth greater than or equal to 32, and seldom employ cores with fewer than 16 registers. Second, the best performing designs typically include a mix of both 32-bit and 64-bit cores. While 64-bit cores are more efficient at computing on wider data types, 32-bit cores employ smaller hardware structures, saving area for other features. Designs that exclude any one of them incur 3-7% loss in performance. Third, most optimal designs employ both microx86 and x86 cores. While constraining cores to only include microx86 cores marginally affects performance, excluding them limits performance considerably. Finally, most optimal designs include both partially predicated and fully predicated cores.

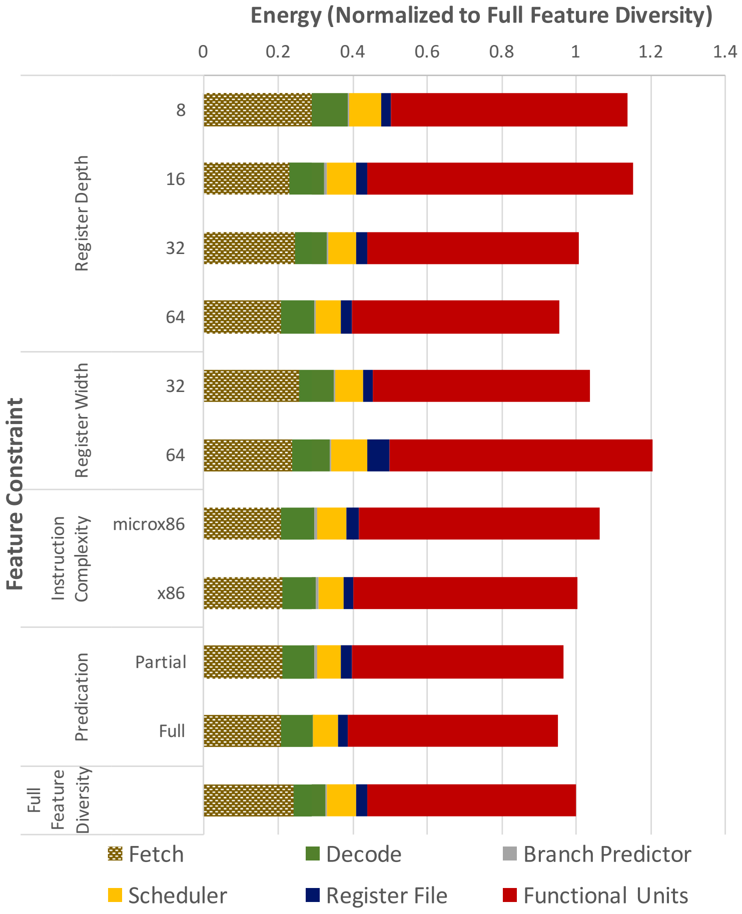

Figure 6.11: Processor Energy Breakdown normalized over that of Fully Custom Designs optimized for multi-programmed workload throughput at 48mm^2 budget, and under different Feature Constraints

We will examine these 10 constrained-optimal designs further. For example, this will allow us to compare the four-core design where all cores are *microx86* with the design where all cores are *x86*. Figure 6.10 shows the transistor investment for the processor part of the real estate for each of the best designs from the above experiment. These designs were all optimized for the same area budget, but here we plot combined core area, without caches; therefore, longer bars imply that the design needed to spend more transistors on cores and sacrifice cache area to get maximum performance. We make the following observations. First, the design that constrains all cores to *microx86* takes up the least combined core area. However, owing to the area efficiency of *microx86*, it is the only design among the 10 that employs all out-of-order cores, each sporting a tournament branch predictor. Second, the design constrained to exclude *microx86* takes up the highest processor area, investing most of its transistors on functional units. Note that we always combine SIMD units with *x86* cores, and that the *microx86* cores lack any SIMD units. Third, the 64-bit-only optimal design spends more transistors on the register file and the scheduler than any other design. In that design, two out of four 64-bit cores are configured with a register depth of 64, with the remaining two configured with 32.

Figure 6.11 shows the processor energy breakdown by stages for each of the best designs from the above experiment. We find that the energy breakdown shows significant deviation from the corresponding area breakdown. While the decoder requires a greater portion of processor area than the fetch unit, it is the fetch unit that expends more energy during run-time since the decode pipeline is only triggered upon a micro-op cache miss, and instructions are streamed out of the micro-op cache for the most part. Interestingly, the design that constrains all cores to be configured with a register depth of 8 spends significant energy in the Fetch stage. This is due to the artificial instruction bloat caused by spills, refills, and rematerializations – a direct consequence of high register pressure. Furthermore, although the x86-only designs invested significantly in SIMD units, the energy spent by the functional units is not nearly as proportional. This is due to relatively infrequent vector activity. Finally, the 64-bit-only design continues to dominate in terms of register file and scheduler energy.

6.5.3 Feature Affinity

In this section, we study the feature affinity of the eight applications we benchmark, in two specific execution scenarios. In the first scenario, we consider a composite-ISA heterogeneous design optimized for single thread performance under a peak power budget of 10W. Recall that in such a design, hardware heterogeneity provides only marginal benefits, and most of the gains come from the fact that an application is free to migrate across different cores that each implement a diverse feature set. Therefore, such a design captures the true ISA affinity of an application.

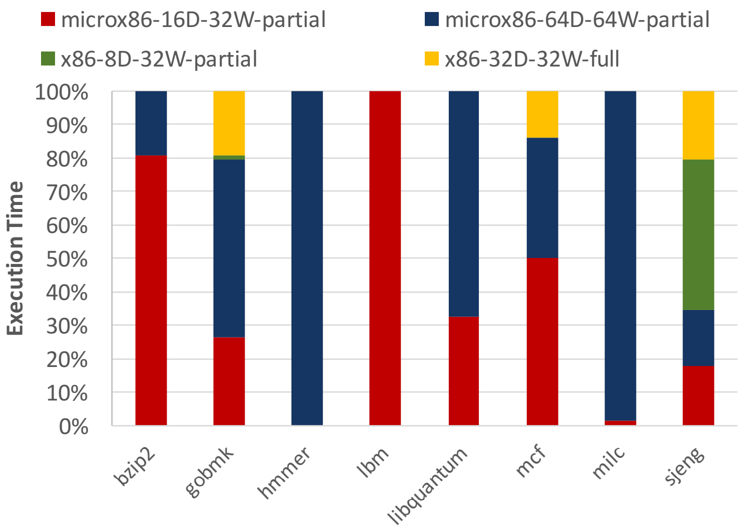

Figure 6.12: Execution Time Breakdown on the best composite-ISA CMP optimized for Single Thread Performance under a Peak Power Budget of 10W

Figure 6.12 shows the feature affinity in terms of the fraction of time an application spends executing on a particular feature set. First, we find that the multicore design that optimizes single thread performance exhibits significant feature diversity. In fact, by analyzing the component features of the multicore, we find that *all* features from our superset ISA have been used. Second, we find that there is significant variance in feature set preference across different applications. There is no single best feature set that is preferred by all applications. Third, we observe that there is some variance in feature set preference even within a single application's internal phases. We find that most applications migrate to a core with a different feature set at least once. Fourth, the benchmark *hmmer* that exhibits significant register pressure tends to always execute on a feature set with a register depth of 64. Interestingly, we found that phases with considerable irregular branch activity, due to indirect branches and function pointer calls, prefer full predication since it eases the pressure on the branch predictor by converting some of the control flow into data flow. This is evidenced by the benchmarks *sjeng* and *gobmk*.

In the second scenario, we consider a composite-ISA design optimized for multi-programmed workload throughput under an area budget of 48mm^2, in which case applications typically contend for the best feature set preference, and may sometimes execute on feature sets of second

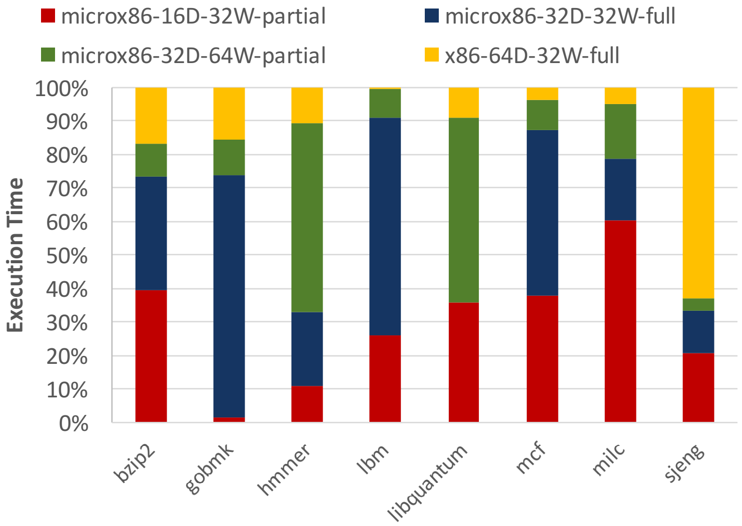

Figure 6.13: Execution Time Breakdown on the best composite-ISA CMP optimized for Multi-programmed Throughput under an Area Budget of 48mm^2

preference. Figure 6.13 shows the results of this experiment. In sharp contrast to the design optimized for single thread performance, where applications had clear preferences, we find that *all* applications in the multi-programmed workload execute on *all* feature sets at some point of time. However, we are still able to make some high level inferences about feature affinity. For example, the benchmark *sjeng* continues to show a clear preference to *x86* over *microx86*, and both benchmarks *sjeng* and *gobmk* prefer to execute on fully predicated ISAs during phases of irregular branch activity.

6.5.4 Migration Cost Analysis

Process migration across composite-ISA cores can involve two scenarios. In a feature upgrade scenario, the core which a process migrates to already implements a superset of the features the process was compiled to, in which case, there is *zero* binary translation or state transformation costs. On the other hand, in a feature downgrade scenario, the core to which the process migrates implements only a subset of the features the process is compiled to, necessitating minimal translation of unimplemented features. We first discuss the cost of a feature downgrade

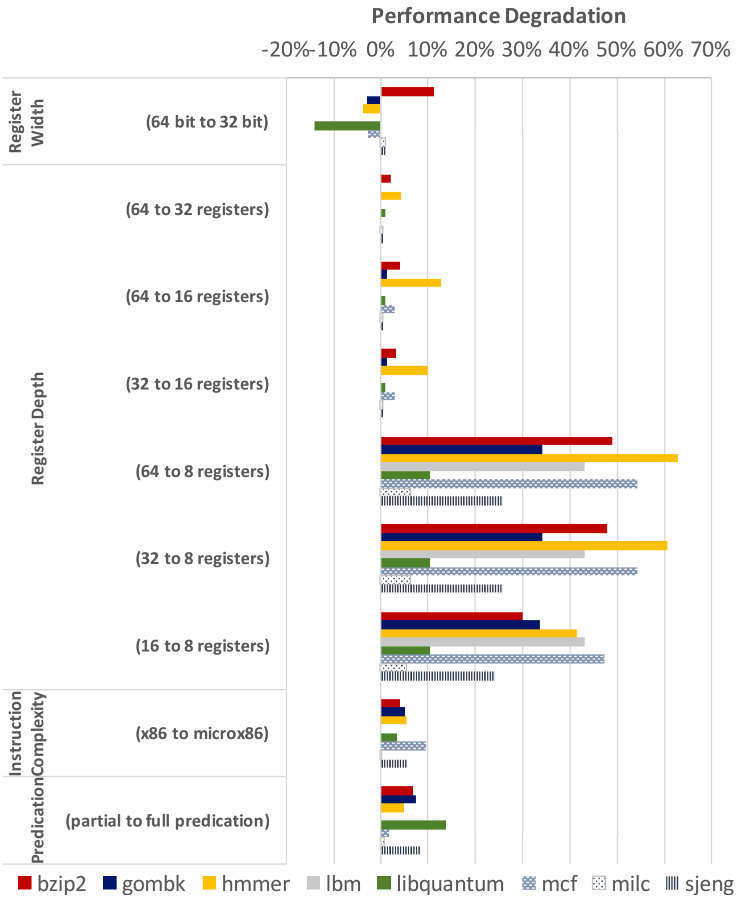

Figure 6.14: Feature Downgrade Cost

for any arbitrary code region, and then measure its performance impact on a design optimized for multi-programmed workload throughput.

We measure feature downgrade costs by running each code region that corresponds to a simpoint on an artificially constrained core that only implements a subset of the features the simpoint was compiled to. Figure 6.14 shows the result of this experiment. We make several important observations here. First, when we downgrade from 64-bit to 32-bit cores, most of the emulation cost is negated due to the cache-efficient 32-bit cores. In fact, we achieve a speedup for some applications when we downgrade them from 64-bit to 32-bit feature sets. Second, since most applications use 32 or fewer registers, there is little emulation cost incurred due to a register depth downgrade from 64 to 32 registers. While we incur some overhead (an average of

2.7%) when we downgrade to a feature set that implements only 16 registers, there is significant overhead (an average of 33.5%) in migrating to a feature set that implements only 8 registers. In all cases, we find that the benchmark *hmmer* incurs the highest emulation overhead due to a register depth downgrade, concurring with our prior feature affinity analysis. Third, we incur an average of 5.5% overhead when we downgrade to a feature set without full predication. We note that this is highly dominated by the outlier *libquantum*, which in our best designs always executes on a partially predicated feature set since the compiler tends to overestimate the cost of diamond predication for this benchmark. Finally, downgrade from *x86* to *microx86* comes at a cost of 4.2% on average. This can be attributed to the emulation of almost every arithmetic instruction that uses indirect addressing mode. However, this experiment does not tell us how often these downgrades are necessary.

In our design space analysis, we needed to assume optimal selection of compiler features to make the search tractable. In the next experiment, we explore a single instantiation of one of our optimal core configurations, and a single instance (single set of features) of each compiled binary. The set of features chosen for the binary is the most common one selected for that application (among all possible scheduling permutations). We then run (again, for all permutations of our benchmark set), an experiment with four cores and four applications for 500 billion instructions. Every time one of the applications experiences a phase change that would cause us to re-shuffle job-to-core assignments, we assume a migration cost for each application that moves, and a possible downgrade cost over the next interval for each job that moves to a core that doesn't fully support the compiled features. Migration cost is measured via simulation for each binary and set of features not supported.

Figure 6.15 compares the designs optimized for multi-programmed workload throughput with migration cost included. Recall that in such a design, threads often contend for the best core and may not always run on their core of first preference. We observe that the performance degradation due to migrations across composite ISAs is a negligible 0.42%, on average (max

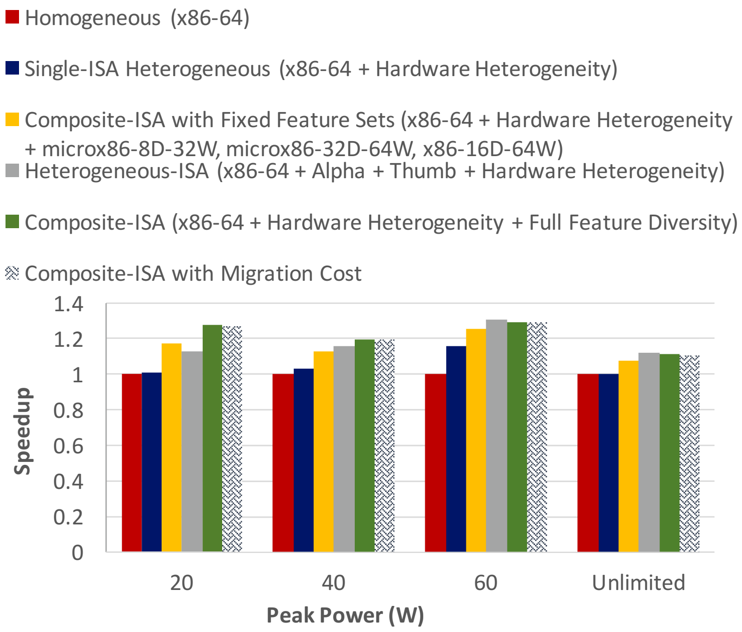

Figure 6.15: Multi-threaded Workload Throughput with Downgrade Cost

0.75%), virtually preserving all of the performance gains due to feature diversity. We attribute such a small migration cost to the fact that feature downgrades are infrequent and when there is one, the cost of software emulating it is minimal, both due to the overlapping nature of feature sets.

In summary, composite-ISA heterogeneous designs consistently outperform and use far less energy than single-ISA heterogeneous designs, generally matching or exceeding multi-vendor heterogeneous-ISA designs.

6.6 Conclusion

This chapter presents a composite-ISA architecture and compiler/runtime infrastructure that replicates the advantages of multi-vendor heterogeneous-ISA architectures. It does so along two dimensions. First, it enables the full performance and energy benefits of multi-ISA design, without the issues of multi-vendor licensing, binary translation, and state transformation. Second, it gives both the processor designer and the compiler a much richer set of ISA design choices, enabling them to select and combine features that match the expected workload. This provides richer gains in efficiency than highly optimized but inflexible existing-ISA based designs. Under certain design scenarios, this architecture gains 30% in performance and over 30% in energy-delay product over single-ISA heterogeneous designs. Further, it matches and in many cases outperforms the multi-vendor heterogeneous-ISA design, and consistently runs at lower levels of energy-delay product.

Acknowledgements

Chapter 6, in full, is currently being prepared for submission for publication of the material. Venkat, Ashish; Basavaraj, Harsha; Tullsen, Dean M., Composite-ISA Cores: Enabling Multi-ISA Heterogeneity using a Single ISA. The dissertation author was the primary investigator and author of this material.

Chapter 7

Concluding Remarks

The computing landscape already includes abundant heterogeneity. A single manufacturer targets different processes over time (different core sizes, different performance targets), and also targets different markets (high performance, embedded, mobile). Different manufacturers have targeted different niches of the market and developed ISAs and microarchitectures accordingly. Yet, each of these architectures has its own benefits and drawbacks. A heterogeneous system that could utilize these different compute engines seamlessly and on-demand offers substantial benefits. To make this vision a reality, we must (1) exploit the heterogeneity that already exists, and (2) design future systems that proactively create the right heterogeneity to maximize performance and energy efficiency, without abandoning the traditional models of programming and execution.

This dissertation demonstrates that substantial benefits arise by strengthening the hardware/software interface, specifically the ISA and the runtime system, with diverse capabilities. Although the advantages of single-ISA heterogeneity have been well established, there was no evidence, or even speculation, that extending that heterogeneity to the ISAs supported by the cores could be profitable. This work is the first to challenge the assumption that that boundary is necessary. By pulling down that boundary wall, this dissertation now allows the CPU and the system architect to finally harness the most underutilized configuration parameter for execution efficiency – the choice of the ISA. In particular, cross-ISA process migration strategy proposed by this dissertation has all but eliminated the tight coupling between the software application and the underlying instruction set without any impact on programmability, opening up a number of key opportunities in terms of performance, energy efficiency, and security.

The design space exploration effort described in this dissertation allows the CPU architect to now design more efficient heterogeneous chip multiprocessors with greater flexibility of ISA choices and microarchitectural options. The proposed multi-ISA heterogeneous chip multiprocessor design benefits substantially from a tighter ISA-microarchitecture co-design, resulting in greater single thread performance, multiprogrammed workload throughput and efficiency gains, especially under tight and relaxed power/area constraints where microarchitecturally heterogeneity alone provides diminishing returns.

By leveraging the seamless and instantaneous cross-ISA process migration capability offered by multi-ISA heterogeneous architectures, the proposed security defense HIPStR (Heterogeneous Program State Relocation) radically transforms the Return-Oriented Programming attack landscape. The defense substantially benefits from increased entropy due to program state relocation both within and across the heterogeneous ISAs, rendering several existing brute force attacks computationally infeasible. It also imposes serious restrictions on the highly evasive just-in-time code reuse attacks that have the ability to bypass fine-grained randomization, to an extent that it is difficult to construct a simple shellcode exploit, let alone Turing-completeness.

Finally, in addition to demonstrating the performance, energy efficiency, and security potential of heterogeneous-ISA architectures, this dissertation significantly alleviates the complexity concerns of multi-vendor ISA heterogeneity via the composite-ISA heterogeneous chip multiprocessor design. These architectures have the potential to recreate the gains of multi-ISA heterogeneity while essentially using a single overlapping set of composite ISAs, thereby eliminating the need for multi-vendor licensing, addressing differences in application binary interfaces, and significantly minimizing the need for binary translation.

This dissertation has unlocked several new heterogeneous-ISA processor architecture designs, and has further showcased substantial benefits over prior work on single-ISA heterogeneous architectures, while preserving the traditional programming and execution models. However, the expanding gap between the increasingly diverse software and the underlying heterogeneous hardware offers significant additional potential that is yet to be harnessed, and there still exist many underexploited levers in the hardware/software interface that could potentially transform the computing landscape in numerous worthwhile dimensions. The research that exploits these levers will not only enable greater levels of performance, energy efficiency, and security in state-of-the-art computing systems, but will further drive emerging technologies.

Bibliography

[ABEL05] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005.

[AEJE17] Almutaz Adileh, Stijn Eyerman, Aamer Jaleel, and Lieven Eeckhout. Mind the power holes: Sifting operating points in power-limited heterogeneous multicores. *IEEE Computer Architecture Letters*, 16(1):56–59, 2017.

[AH82] Marc Auslander and Martin Hopkins. An overview of the pl. 8 compiler. In *ACM SIGPLAN Notices*, volume 17, pages 22–31. ACM, 1982.

[AHM97] David I August, Wen-mei W Hwu, and Scott A Mahlke. A framework for balancing control flow and predication. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 92–103. IEEE, 1997.

[AKPW83] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189. ACM, 1983.

[Akr17] Ayaz Akram. A study on the impact of instruction set architectures on processors performance. 2017.

[AME+11] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. *ACM SIGPLAN Notices*, 2011.

[ARM] ARM Limited. *ARM Cortex-A Series Programmers Guide for ARMv8-A*.

[AS17] Ayaz Akram and Lina Sawalha. the impact of isas on performance". In *Proceedings of the 14th Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD) associated with ISCA*, June 2017.

[ASC$^+$16] Shoaib Akram, Jennifer B Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):4, 2016.

[BABJW11] Edson Borin, Guido Araujo, Mauricio Breternitz Jr, and Youfeng Wu. Structure-constrained microcode compression. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 104–111. IEEE, 2011.

[BABW14] Edson Borin, Guido Araujo, Mauricio Breternitz, and Youfeng Wu. Microcode compression using structured-constrained clustering. *International Journal of Parallel Programming*, 42(1):140–164, 2014.

[Bau17] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 132–137, New York, NY, USA, 2017. ACM.

[BBM$^+$14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking Blind. In *Security and Privacy*, July 2014.

[BBWA07] Edson Borin, Mauricio Breternitz, Youfeg Wu, and Guido Araujo. Clustering-based microcode compression. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 189–196. IEEE, 2007.

[BCT92] Preston Briggs, Keith D Cooper, and Linda Torczon. Rematerialization. *ACM SIGPLAN Notices*, 27(7):311–321, 1992.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *ACM SIGPLAN Notices*, 32(5):287–295, 1997.

[BDH$^+$06] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The M5 Simulator: Modeling Networked Systems. *Micro, IEEE*, 2006.

[BDL$^+$16] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Technical Conference*, April 2005.

[BGM$^+$89] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Programming Language Design and Implementation*, volume 24, pages 258–263. ACM, 1989.

[BHR^{+}15] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[BLJ^{+}17] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 645–659, 2017.

[BMF^{+}16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. *ACM SIGOPS Operating Systems Review*, 50(2):217–232, 2016.

[BMS13a] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A Detailed Analysis of Contemporary ARM and x86 Architectures. Technical report, University of Wisconsin - Madison, 2013.

[BMS13b] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *International Symposium on High Performance Computer Architecture*, February 2013.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, 2008.

[BS08] Sandeep Bhatkar and R Sekar. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2008.

[BSA^{+}15] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the 10th European Conference on Computer Systems*, April 2015.

[BSGG13] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. Performance analysis and predictability of the software layer in dynamic binary translators/optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 15:1–15:10, New York, NY, USA, 2013. ACM.

[BSR$^+$16] Sharath K Bhat, Ajithchandra Saya, Hemedra K Rawat, Antonio Barbalace, and Binoy Ravindran. Harnessing energy efficiency of heterogeneous-isa platforms. *ACM SIGOPS Operating Systems Review*, 49(2):65–69, 2016.

[CAC$^+$81] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[CAC$^+$08] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[CBD$^+$99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Proceedings of the 5th Linux Expo*, 1999.

[CBP$^+$15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[CCD$^+$15] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963. ACM, 2015.

[CDD$^+$10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, 2010.

[CF09] Stephen Checkoway and Edward W Felten. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. 2009.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CH90] Fred C Chow and John L Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, 1990.

[CHB+15] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11, 2015.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[CS98] Keith D Cooper and L Taylor Simpson. Live range splitting in a graph coloring register allocator. In *International Conference on Compiler Construction*, pages 174–187. Springer, 1998.

[CSH+12] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, PK Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, et al. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *International Symposium on High Performance Computer Architecture*, February 2012.

[cut17] Apple 2017: The iphone x (ten) announced. 2017.

[CW14] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[CWS+11] Niket K Choudhary, Salil V Wadhavkar, Tanmay A Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H Dwiel, Sandeep Navada, Hashem H Najaf-abadi, and Eric Rotenberg. Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 11–22. ACM, 2011.

[CWS+12] Niket Choudhary, Salil Wadhavkar, Tanmay Shah, Hiran Mayukh, Jayneel Gandhi, Brandon Dwiel, Sandeep Navada, Hashem Najaf-abadi, and Eric Rotenberg. Fabscalar: Automating superscalar core design. *IEEE Micro*, 32(3):48–59, 2012.

[CYH+08] Jiunn-Yeu Chen, Wuu Yang, Tzu-Han Hung, Hong-Men Su, and Wei-Chung Hsu. A static binary translator for efficient migration of ARM-based applications. In *Workshop on Optimizations for DSP and Embedded Systems*, April 2008.

[CZY+14] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[DB13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[DF80] Jack W Davidson and Christopher W Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):191–202, 1980.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[DK14] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.

[DLS$^+$15] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. July 2015.

[DLSM14] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Symposium on Principles of Programming Languages*, January 1984.

[DSW11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[Dun90] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[DVT12] Matt DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2012.

[EBA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *International Symposium on Computer Architecture*, June 2011.

[EFG$^+$15] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity1. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[Eto03] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. 2003.

[FCG00] Adam Ferrari, Steve J. Chapin, and Andrew Grimshaw. Heterogeneous Process State Capture and Recovery through Process Introspection. *Cluster Computing*, 2000.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[FT09] FreeBSD Team. FreeBSD 8.0 Release Notes. 2009.

[fus08] The future is fusion: The Industry-Changing Impact of Accelerated Computing. Technical report, AMD, 2008.

[GA96] Lal George and Andrew W Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.

[GABP14] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[GAP$^+$14] Enes Goktas, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[GB99] Rajiv Gupta and Rastislav Bodík. Register pressure sensitive redundancy elimination. *CC*, 99:107–121, 1999.

[Gre11] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, ARM, 2011.

[HHD16] William H Hawkins, Jason D Hiser, and Jack W Davidson. Dynamic canary randomization for improved software security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, page 9. ACM, 2016.

[HM08] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, July 2008.

[HNTC$^+$12] Jason Hiser, Anh Nguyen Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[Höl10] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[HS04] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 213–, Washington, DC, USA, 2004. IEEE Computer Society.

[HSA$^+$16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[HZL$^+$15] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ron Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, Washington, DC, USA, 2015. IEEE Computer Society.

[IJJ09] Ciji Isen, Lizy K John, and Eugene John. A Tale of Two Processors: Revisiting the RISC-CISC Debate. In *Computer Performance Evaluation and Benchmarking*. 2009.

[Int] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual.*

[Int09] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-018. March 2009.

[Int14] Intel. Software guard extensions programming reference. 2014.

[JTL14] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st International Symposium on Network and Distributed System Security*, February 2014.

[JYP$^+$17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[KDH$^+$05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, July 2005.

[Kes99] Richard E Kessler. The Alpha 21264 Microprocessor. *Micro, IEEE*, 1999.

[KFJ$^+$03] 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 *International Symposium on Microarchitecture*, December 2003.

[KG05] Arvind Krishnaswamy and Rajiv Gupta. Efficient Use of Invisible Registers in Thumb Code. In *International Symposium on Microarchitecture*, December 2005.

[KKCL13] M Kim, H Kim, Hyunkwon Chung, and Kyoungmook Lim. Samsung exynos 5410 processor-experience the ultimate performance and versatility. *White Paper*, 2013.

[KKP03] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, 2003.

[KOAGP12] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu Ghazaleh, and Dmitry Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[Kor10] Tim Kornau. Return oriented programming for the ARM architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[KSP$^+$14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[KTJ06] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture*, June 2004.

[LA04a] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LA04b] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, March 2004.

[LAR$^+$15] Bruno Cardoso Lopes, Rafael Auler, Luiz Ramos, Edson Borin, and Rodolfo Azevedo. Shrink: Reducing the isa complexity via instruction recycling. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 311–322. ACM, 2015.

[LAS$^+$09a] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture*, December 2009.

[LAS$^+$09b] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[Lat08] Chris Lattner. LLVM and Clang: Next Generation Compiler Technology. In *The BSD Conference*, May 2008.

[LBK$^+$10] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, January 2010.

[LCG$^+$14] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[LCG$^+$15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[LDDLARS14] Lucas Davi, Daniel Lehmann, and Ahmad-Reza Sadeghi. The Beast is in Your Memory: Return-Oriented Programming Attacks Against Modern Control-Flow Integrity Protection Te chniques. In *BlackHat USA*, Aug 2014.

[LGAT00] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):431–470, 2000.

[LHBF14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 276–291. IEEE, 2014.

[LLNB16] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to make aslr win the clone wars: Runtime re-randomization. In *NDSS*, 2016.

[LPD$^+$12] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 317–328. IEEE Computer Society, 2012.

[LPD$^+$14] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski Jr, Thomas F Wenisch, and Scott Mahlke. Heterogeneous microarchitectures trump voltage scaling for low-power cores. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 237–250. ACM, 2014.

[LPD$^+$16] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald G Dreslinski, Thomas F Wenisch, and Scott Mahlke. Exploring fine-grained heterogeneity with composite cores. *IEEE Transactions on Computers*, 65(2):535–547, 2016.

[LTPM15] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42nd International Symposium on Computer Architecture*, June 2015.

[Mat16] Craig Matsumoto. The case for arm in the data center. *SDXCentral*, 2016.

[MBSN14] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium*, Aug 2014.

[MFN$^+$17] Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Jonathan Balkind, Alexey Lavrov, Mohammad Shahrad, Samuel Payne, and David Wentzlaff. Piton: A manycore processor for multitenant clouds. *Ieee micro*, 37(2):70–80, 2017.

[MHB+94] Scott A Mahlke, Richard E Hank, Roger A Bringmann, John C Gyllenhaal, David M Gallagher, and Wen-mei W Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 217–227. ACM, 1994.

[MHM+95] Scott A Mahlke, Richard E Hank, James E McCormick, David I August, and Wen-Mei W Hwu. A comparison of full and partial predicated execution support for ilp processors. *ACM SIGARCH Computer Architecture News*, 23(2):138–150, 1995.

[MLC+92] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO Newsletter*, volume 23, pages 45–54. IEEE Computer Society Press, 1992.

[MNM+15] Nikola Markovic, Daniel Nemirovsky, Veljko Milutinovic, Osman Unsal, Mateo Valero, and Adrian Cristal. Hardware round-robin scheduler for single-isa asymmetric multi-core. In *European Conference on Parallel Processing*, pages 122–134. Springer, 2015.

[MNU+15] Nikola Markovic, Daniel Nemirovsky, Osman Unsal, Mateo Valero, and Adrian Cristal. Thread lock section-aware scheduling on asymmetric single-isa multi-core. *IEEE Computer Architecture Letters*, 14(2):160–163, 2015.

[Moo10] H. D. Moore. Microsoft Internet Explorer data binding memory corruption. 2010.

[Mor15] Timothy Prickett Morgan. Google will do anything to beat moores law. *The Next Platform*, 2015.

[MSD] MSDN. Introduction to code signing.

[MST] Venkateswara Madduri, Ross Segelken, and Bret Toll. Method and apparatus for variable length instruction parallel decoding. In *United States Patent Application 10/331,335*.

[MT13] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 619–630. ACM, 2013.

[NAM+17] Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal, and Adrian Cristal. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2017 29th International Symposium on*, pages 121–128. IEEE, 2017.

[NEE17] Ajeya Naithani, Stijn Eyerman, and Lieven Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 397–408. IEEE, 2017.

[Nic16] James Niccolai. Ibm's power chips hit the big time at google. *PC World*, 2016.

[NR16] Joel Nider and Mike Rapoport. Cross-isa container migration. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 24:1–24:1, New York, NY, USA, 2016. ACM.

[nvd] National Vulnerability Database.

[OBL$^+$10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[ope16] Ibm's power chips hit the big time at google. Technical report, 2016.

[OVB$^+$06] Hilmi Ozdoganoglu, TN Vijaykumar, Carla E Brodley, Benjamin A Kuperman, and Ankit Jalote. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers*, 2006.

[PCC$^+$14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture*, June 2014.

[PHVB$^+$03] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for Accurate and Efficient Simulation. In *ACM SIGMETRICS Performance Evaluation Review*, June 2003.

[PLD$^+$15] Vinicius Petrucci, Michael A Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258. IEEE, 2015.

[PLDM15] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Dynamos: dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 322–333. ACM, 2015.

[PLPI13] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 2013.

[PPK12] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[PT03a] PaX Team. PaX address space layout randomization. 2003.

[PT03b] PaX Team. PaX non-executable pages design and implementation. 2003.

[PZL06] Yong-Joon Park, Zhao Zhang, and Gyungho Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE Micro*, 2006.

[Qua11] Qualcomm. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. Technical report, October 2011.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 2012.

[RH97] Anton Chernoff Ray and Ray Hookway. DIGITAL FX!32 running 32-bit x86 applications on alpha NT. In *USENIX Windows NT Workshop*, August 1997.

[RI11] Chris Rohlf and Yan Ivnitskiy. Attacking clientside JIT compilers. *Black Hat, USA*, 2011.

[RMPB09] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib (c). In *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009.

[SAB11] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceddings of the 20th USENIX Security Symposium*, 2011.

[san08] 2nd Generation Intel Core vPro Processor Family. Technical report, Intel, 2008.

[SB09] Lukasz Strozek and David Brooks. Energy-and Area-Efficient Architectures through Application Clustering and Architectural Heterogeneity. *ACM Transactions on Architecture and Code Optimization*, 2009.

[SD97] Solar Designer. Getting around non-executable stack (and fix). 1997.

[SD09] Richard M Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3. 3.* CreateSpace, 2009.

[SH98] Peter Smith and Norman C Hutchinson. Heterogeneous Process Migration: The Tui System. *Software-Practice and Experience*, 1998.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[SKIH12] Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[SKKK16] Sudarshan Srinivasan, Nithesh Kurella, Israel Koren, and Sandip Kundu. Exploring heterogeneity within a core for improved power efficiency. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1057–1069, 2016.

[SLZD04] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[SMD$^+$13] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[SN05] James Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., June 2005.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SPP$^+$04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004.

[SRWB14] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 97–108. IEEE Press, 2014.

[SRWB15] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. The aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3):58–70, 2015.

[STL$^+$15] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming, May 2015.

[Sun13] Rahul Sundaram. Fedora 20 Security Features - Stack Smash Protection, Buffer Overflow Detection, and Variable Reordering. 2013.

[SWTB11] Lina Sawalha, Sonya Wolff, Monte P Tull, and Ronald D Barnes. Phase-guided scheduling on single-isa heterogeneous multicore processors. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 736–745. IEEE, 2011.

[teg10] The Benefits of Multiple CPU Cores in Mobile Devices. Technical report, NVidia, 2010.

[teg11] Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, NVidia, 2011.

[Ter11] Steve Terpe. Why Instruction Sets No Longer Matter. 2011.

[Tex] Texas Instruments Inc. *OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide*.

[UT06] Ubuntu Team. Ubuntu 6.10 Security Features - Stack Protector. 2006.

[VBSS94] David G. Von Bank, Charles M. Shub, and Robert W. Sebesta. A Unified Model of Pointwise Equivalence of Procedural Computations. *ACM Transactions on Programming Languages and Systems*, 1994.

[VCAH$^+$13] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 177–187. IEEE, 2013.

[VCE13] Kenzo Van Craeynest and Lieven Eeckhout. Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):32, 2013.

[VCJE$^+$12] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, June 2012.

[VdV04] Arjan Van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. 2004.

[vdVAG$^+$15] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[Ven01] Vendicator. StackShield: A Stack Smashing Technique Protection Tool for Linux. 2001.

[VKYP15] Ashish Venkat, Arvind Krishnaswamy, Koichi Yamada, and Rajan Palanivel. Binary Translation driven Program State Relocation. In *United States Patent Grant US009135435B2*, 2015.

[VPMPADK13] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[VSG$^+$10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[VSST16] Ashish Venkat, S Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2016.

[VT14] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, 2014.

[WA12] Daniel Wong and Murali Annavaram. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–130. IEEE, 2012.

[Wev04] Berend-Jan Wever. Internet Explorer IFRAME src&name parameter BoF remote compromise. 2004.

[WM09] Vincent M Weaver and Sally A McKee. Code Density Concerns for New Architectures. In *International Conference on Computer Design*, October 2009.

[WMHL12] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, 2012.

[WYZ+17] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. Enabling cross-isa offloading for cots binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 319–331. ACM, 2017.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K Iyer. Architecture support for defending against buffer overflow attacks. Workshop on Evaluating and Architecting Systems for Dependability, 2002.

[ZA03] Peng Zhao and José Nelson Amaral. To inline or not to inline? enhanced inlining decisions. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 405–419. Springer, 2003.

[ZS13] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[ZT00] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, March 2000.

[ZWC+13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.