Popcorn Linux: A Compiler and Runtime for Execution Migration Between Heterogeneous-ISA Architectures

Robert F. Lyerly

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Changhee Jung
Cameron D. Patterson
Paul E. Plassmann
Haibo Zeng

March 22, 2019
Blacksburg, Virginia

Keywords: heterogeneous architectures, compilers, runtime systems

Popcorn Linux: A Compiler and Runtime for State Transformation Between Heterogeneous-ISA Architectures

Robert F. Lyerly

ABSTRACT

In recent years there has been a proliferation of parallel and heterogeneous architectures. As chip designers have hit fundamental limits in traditional processor scaling, they have begun rethinking processor architecture from the ground up. In addition to creating new classes of processors, chip designers have revisited CPU microarchitecture in order to target different computing contexts. CPUs have been optimized for low-power smartphones and extended for high-performance computing in order to achieve better performance and energy efficiency for heavy computational tasks. Although heterogeneity adds significant complexity to both hardware and software, recent works have shown tremendous power and performance benefits obtainable through specialization. It is clear that emerging systems will be increasingly heterogeneous.

Many of these emerging systems couple together cores of different instruction set architectures (ISA), due to both market forces and the potential performance and power benefits in optimizing application execution. However, differently from symmetric multiprocessors or even asymmetric single-ISA multiprocessors, natively compiled applications cannot freely migrate between heterogeneous-ISA processors. This is due to the fact that applications are compiled to an instruction set architecture-specific format which is incompatible on other instruction set architectures. This creates serious limitations, as execution migration is a fundamental mechanism used by schedulers to reach performance or fairness goals, allows applications to migrate between heterogeneous-ISA CPUs in order to accelerate parallel applications or even leverage ISA-heterogeneity for security benefits.

This dissertation describes system software for automatically migrating natively compiled applications across heterogeneous-ISA processors. This dissertation describes the implementation and evaluation of a complete software stack on commodity scale heterogeneous-ISA CPUs, emulating datacenters with heterogeneous-ISA systems or future systems that tightly integrate heterogeneous-ISA CPUs via point-to-point interconnect. This dissertation describes a compiler which builds applications for heterogeneous-ISA execution migration. The compiler generates machine code for every architecture in the system and lays out the application's code and data in a common format. In addition, the compiler generates metadata used by a state transformation runtime to dynamically transform thread execution state between ISA-specific formats, allowing application threads to migrate between different ISAs.

The compiler and runtime is evaluated in conjunction with a replicated-kernel operating system, which provides thread migration and distributed shared virtual memory across heterogeneous-ISA processors. This redesigned software stack is evaluated on a setup containing and ARM and an x86 processor interconnected via point-to-point interconnect over PCIe. This dissertation shows that sub-millisecond state transformation is achievable. Ad-

ditionally, it shows that for a datacenter-like workload using benchmarks from the NAS Parallel Benchmark suite, the system can trade some performance for up to a 66% reduction in energy and up to an 11% reduction in energy-delay product.

This dissertation then describes an exploration into using hardware transactional memory (HTM) to maximize scheduling flexibility. Because applications can only migrate between ISAs at program locations with specific properties, there may be a significant delay between when the scheduler wishes to migrate an application and when the application can respond to the migration request. In order to reduce this migration response time, this dissertation describes compiler instrumentation which uses HTM to allow the scheduler to force applications to roll back to the most recently encountered program location suitable for migration. This is evaluated both in terms of overhead and responsiveness to migration requests.

In addition to showing the viability of the infrastructure for optimizing workload placement in a heterogeneous-ISA datacenter, this dissertation also demonstrates utilizing the infrastructure to accelerate multithreaded applications. This dissertation describes a new OpenMP runtime named `libopenpop` that is optimized for executing applications in heterogeneous-ISA systems with distributed shared virtual memory. The runtime utilizes synchronization primitives that enable scale-out execution across rack-scale systems and new work distribution mechanisms that predict the best partitioning of parallel work across CPUs with diverse architectural characteristics. `libopenpop` demonstrates sizable improvements over a naïve OpenMP implementation – a 38x improvement in multi-server barrier latency, a 5.4x improvement in multi-server data reductions and a geometric mean speedup of 4.04x for scalable applications in an 8-node x86-64 cluster. For a heterogeneous system composed of a highly-clocked x86 server and a highly-parallel ARM server, `libopenpop` delivers up to a 4.7x speedup and a geometric mean speedup of 41% across benchmarks from several benchmark suites versus the best single-node homogeneous execution.

Finally, this dissertation describes leveraging the compiler and state transformation runtime to provide enhanced security for applications. Because the compiler provides detailed information about the stack layout of applications, it can be leveraged to defend against exploits such as stack smashing attacks and return-oriented programming attacks. This dissertation describes Chameleon, a runtime which uses the compiler and state transformation infrastructure to continuously re-randomize the stack layout and code of vulnerable applications to thwart attackers. Chameleon attaches to applications using existing operating system interfaces and periodically switches the application to new randomized stack layouts and code by rewriting the stack. Chameleon enhances security with little overhead – it disrupts a geometric mean 76.32% of code gadgets in benchmark binaries, randomizes stack element locations with geometric mean 3 potential randomized locations, and has 1.1% overhead when re-randomizing every 50 milliseconds, making it extremely difficult for attackers to exploit target applications.

This work is supported in part by ONR under grants N00014-13-1-0317, N00014-16-1-2711, and N00014-18-1-2022, and NAVSEA/NEEC under grants 3003279297 and N00174-16-C-0018.

Popcorn Linux: A Compiler and Runtime for State Transformation Between Heterogeneous-ISA Architectures

Robert F. Lyerly

GENERAL AUDIENCE ABSTRACT

Computer processors have experienced unprecedented performance improvements over the past 50 years. However, due to physical limitations of how processors execute, in recent years this performance growth has started to slow. In order to continue scaling performance, chip designers have begun diversifying processor designs to meet different performance and power consumption targets. Processors specialized for different contexts use various instruction set architectures (ISAs), the operations made available for use by the hardware. Programs built for one instruction set architecture are not compatible with others, requiring developers to build complex applications to manually bridge the gap. This leads to brittle applications and prevents the system software managing the processors from adapting workloads to match processor characteristics.

This dissertation presents the Popcorn Linux system software which provides transparent support for running applications across computers composed of processors of multiple ISAs. Popcorn Linux provides the ability to migrate applications between these processors without requiring developers to add any application instrumentation – the system software manages all the details of building and migrating applications. An evaluation of Popcorn Linux shows that transparently migrating applications between diverse processors provides power and performance benefits in a variety of scenarios. Additionally, this dissertation describes leveraging the Popcorn Linux software infrastructure to harden applications against attackers seeking to hijack applications for malicious purposes.

Dedication

To my parents David and Mary Denton; my brother Matt; and Rebecca.

Acknowledgments

I would like to thank my advisor, Professor Binoy Ravindran, for helping to guide me throughout my Ph.D. In addition helping me grow as a software engineer and researcher, he taught me how to deal with setbacks throughout life. He showed me how to laugh when things are not going as planned and he always knew when to provide support, empathy and encouragement for the path forward. The road was full of ups and downs, but he was always there to support me through it all.

In addition to Professor Ravindran, I would like to thank my committee members Professor Changhee Jung, Professor Cameron Patterson, Professor Paul Plassmann and Professor Haibo Zeng for providing valuable insights and feedback on my work. Additionally I would like to thank Professor Changwoo Min and Professor Christopher Rossbach for providing an outside perspective and new ideas about my work.

I also would like to thank the many SSRG friends I've made over the years – in rough chronological order, Dr. Alastair Murray, Dr. Antonio Barbalace, Ben Shelton, Kevin Burns, Dr. Sachin Hirve, Curt Albert, Duane Niles, Dr. Pierre Olivier, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Ian Roessle, Josh Bockenek, Dr. (now Professor) Sang-Hoon Kim and Yihan Pang. The camaraderie helped all of us through the late nights and tough times – without you all it would have been a much tougher and much less fun journey.

I would like to thank my friends who are too innumerable to name but whom all made sure to laugh at the fact that I was still in school.

I would like to thank Rebecca, without whom I do not think I would have made it all the way through. Her kindness and support is unmatched; she was always ready with encouragement and was willing to sacrifice when I had to prioritize work. She is more than I could hope for. I would also like to thank George, who always made me smile, even when he was chewing through cords.

Finally I would like to thank my parents and my brother. The reason I have made it this far is due to them trailblazing a path in too many ways to count. They have always been there for encouragement and support. I can count on them for anything, and they have given me the opportunity to pursue my goals without obstacle.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a shift towards increasing parallelism and heterogeneity in processor design [188, 189]. As traditional uniprocessors have hit the clock speed, power, instruction-level parallelism and complexity walls, chip designers have been forced to re-think computer architecture from the ground up. This has led to an explosion in new architectures such as graphics processing units (GPUs), digital signal processors (DSPs) and field-programmable gate arrays (FPGAs). Additionally, general-purpose CPUs have been re-architected in order to meet energy and performance goals for varying form factors [107, 59, 15, 74, 17, 16]. It is clear that emerging computer systems will be increasingly heterogeneous in order to achieve better energy efficiency and higher performance.

Recently there has been a tremendous amount of change in CPU microarchitecture in order to reach different power and performance targets. With the advent of smartphones, CPU designers have built processors that strike a balance between low power and reasonable performance [107, 47]. The high-performance computing (HPC) community has embraced CPU heterogeneity, with several of the top supercomputers in the Top500 list [193] mixing symmetric chip-multiprocessors (CMP) with general-purpose and OS-capable [149] many-core accelerators. Additionally, the HPC community has begun to include energy efficiency as a primary design goal as they realized they could not continue scaling the number of cores at current power consumption levels [75]. Chip designers have even begun to include heterogeneous CPU cores together on a single die in order to achieve high performance and energy efficiency for a variety of workloads [95, 141, 107, 15].

Due to the history of how different CPUs were created and the technology limitations of their time, many commodity scale CPUs utilize different instruction set architectures (ISA) [157]. The ISA defines the hardware-software interface and provides a fundamental definition of how software can execute on a given processor. This definition includes how data is encoded

into binary representations, how memory is arranged for execution and what instructions are available, among other aspects. The ISA is fixed for a particular CPU and thus the job of a compiler is to map an application written in a source code language like C onto a processor's ISA. ISAs are not interoperable and therefore it is impossible for applications compiled for one ISA to be run on another ISA with today's compilers, operating systems and runtimes.

However, because CPUs that target different power and performance goals often use different ISAs, systems composed of such heterogeneous-ISA CPUs provide an attractive means for optimizing a variety of workloads. For example, application migration is desirable in such systems in order to achieve higher performance and improved energy efficiency [155, 123, 204, 197, 127, 27]. Application migration allows the system software to optimize how a given workload executes in the system to best utilize the available compute resources, e.g., placing applications in consideration of architectural characteristics or multiprogrammed environments. Without application migration across heterogeneous-ISA CPUs, the system has limited ability to adapt to application or workload characteristics and may miss out on significant benefits. Thus, it is imperative that new techniques are developed to enable execution migration across heterogeneous-ISA CPUs as they become increasingly interwoven into the same systems, i.e., racks, servers or even systems-on-chips.

1.1.1 Heterogeneous Datacenters

The x86 instruction set architecture is the most widely used processor in datacenters today [136, 166, 106]. Recently, however, there has been a push to introduce the ARM ISA into the server space. Multiple chip vendors including AMD [11], Qualcomm [148], Ampere [74] and Cavium [48] are producing ARM processors for datacenters and the cloud. Additionally, there is increasing vendor support behind the POWER ISA, with IBM forming the OpenPOWER foundation by partnering with companies such as Google, NVIDIA, Mellanox and others [80]. Interest in alternative processor architectures is driven by increasing availability of ARM and POWER cloud offerings [137, 126, 55, 56] in addition to traditional x86 services. These new processor architectures promise higher energy proportionality [28], meaning more performance per watt and increased computing power per rack (i.e., compute density).

Reducing electricity costs has become one of the most important concerns for datacenter operators today [209]. Datacenter hardware and software designers have proposed many techniques for improving energy efficiency while maintaining acceptable computational capacity [192, 209, 204, 198]. There are several software-based approaches that are effective for conserving energy, including load balancing and consolidation. Load balancing spreads applications evenly across nodes so that no nodes are over-saturated and each server consumes a reduced amount of power. Consolidation instead groups tasks on the minimal number of nodes required so that service-level agreements (e.g., latency requirements) can be met. The remaining servers are subsequently placed in a low-power state. Both solutions require

migrating applications between nodes to dynamically adjust the computational capacity of the datacenter with time-varying workloads. How can datacenter operators leverage these techniques in datacenters with increasing ISA diversity?

1.1.2 Heterogeneous-ISA CMPs and Tightly-Coupled Systems

Recent works have demonstrated significant advantages for execution migration between tightly-coupled cores that utilize the same ISA but with heterogeneous microarchitectures [95, 141, 187, 107, 123, 158, 108, 163, 179]. Existing mechanisms for execution migration in symmetric multiprocessors (SMP) work without modification for these new processors because all cores share the same ISA and are interconnected via cache-coherent shared memory. In asymmetric chip multiprocessors (ACMP), execution migration can be used to accelerate both serial and parallel portions of applications with higher energy efficiency [158, 108, 163, 179].

More recent works by DeVuyst et al. [70] and Venkat et al. [197, 195] show that there are further performance and energy benefits obtained by migrating between heterogeneous-ISA cores versus ACMPs. Applications may exhibit affinities for certain ISAs based on characteristics of code generated by the compiler, such as register pressure, memory addressing modes, floating-point and SIMD computation, etc. Additionally, because emerging heterogeneous-ISA CPUs have vastly different macro- and micro-architectures [74, 159, 97, 59, 100], they also provide different levels of performance, energy efficiency and parallelism to accelerate applications with diverse execution profiles – tightly coupling such processors together can provide significant performance benefits [27]. Finally, migrating execution between heterogeneous-ISA cores can provide a defense against security exploits such as return-oriented programming attacks [196]. However, past works simulate a cache-coherent shared memory processor with heterogeneous-ISA cores [70, 196] or couple together overlapping-ISA CPUs [27]. How are applications built and migrated between fully-diverse heterogeneous-ISA processors in commodity scale systems?

1.1.3 Challenges

These fundamental changes in processor design have forced developers to rethink how emerging heterogeneous systems are programmed. Utilizing heterogeneous-ISA CPUs places a large burden on developers because they can no longer use a shared-memory programming model [8]. Instead, developers must reason about application structure and memory layout in order to obtain maximum performance [150, 92, 91]. Because these processors have distinct ISAs, source code compiled for one processor is not able to be run on another. This harms programmability because developers must manually partition applications into pieces and coordinate computation and data movement across architectures. It also hinders system adaptability because the system software cannot freely schedule applications to meet performance or fairness goals [155, 209].

One solution for heterogeneous-ISA execution is to use a language-level virtual machine, e.g., a Java virtual machine [128]. When using language VMs, the application is maintained in an architecture-independent intermediate format which the VM interprets to execute the application. Because the VM has complete knowledge of the application's execution, including code and data format, it can migrate applications between architectures [83, 84, 88, 54]. However, using these approaches requires applications be rewritten in the interpreted language. Many datacenter applications, e.g., Redis [164], are written using natively-compiled languages such as C and C++ in order to apply aggressive optimizations. Re-writing the application in an interpreted language is a non-starter due to the loss of control – for example, Java applications are required to use garbage collection for memory management. Additionally, many VM-level techniques for migration rely on language-level mechanisms (e.g., object serialization [154]), which are demonstrated to have high overheads.

Therefore, as heterogeneity becomes ubiquitous in all computing contexts it becomes increasingly important to develop new techniques for seamless execution migration across heterogeneous-ISA processors for natively-compiled applications.

1.2 Thesis Contributions

This dissertation presents a full software stack for enabling execution migration across heterogeneous-ISA architectures. The prototype, named Popcorn Linux, includes an operating system, compiler and runtime which seamlessly migrates applications between an ARM and an x86 processor interconnected over a high speed network. This work describes the design and implementation of the compiler and runtime components of Popcorn Linux, named the Popcorn compiler toolchain and state transformation runtime. These components are presented, which build applications and enable migration between heterogeneous-ISA CPUs using capabilities provided by Popcorn Linux's OS. In addition to the core infrastructure, this dissertation describes leveraging Popcorn Linux for accelerating multithreaded applications and for hardening applications against security exploits. This dissertation makes the following contributions:

- The design and implementation of the Popcorn compiler toolchain. The toolchain builds applications suitable for migration by adjusting data and code layout, and by automatically inserting migration points into the generated machine code. Additionally, the compiler performs offline analysis to provide metadata for dynamic state transformation. The toolchain builds multi-ISA binaries which the OS uses to recreate an application's virtual address space across heterogeneous-ISA CPUs.
- The design and implementation of the state transformation runtime. The state transformation runtime transforms execution state between ISA-specific formats so that threads of an application can migrate between architectures. It additionally provides

a mechanism for initiating migration and for bootstrapping execution after the application has migrated to the destination architecture. The runtime provides sub-millisecond transformation latencies for benchmarks from the NAS Parallel Benchmark (NPB) suite on an x86 and an ARMv8 CPU. Using Popcorn Linux (compiler, runtime, OS), the dissertation demonstrates a 30% reduction in energy and an 11% reduction in energy-delay product when load-balancing a multiprogrammed workload on top of server-class x86-64 and ARMv8 CPUs.

- An exploration of using hardware transactional memory (HTM) to improve scheduler responsiveness. Because applications cannot migrate at arbitrary locations, extensions to the Popcorn compiler instrument generated code with transactional execution. This allows the scheduler to abort speculative execution and roll back to the most recently encountered migration point, enabling high responsiveness to scheduling requests. This dissertation shows that using HTM reduces migration response time to 1.9 microseconds but adds a geometric mean 13.45% overhead for benchmarks from NPB.

- The design and implementation of `libopenpop`, an OpenMP runtime optimized for running multithreaded applications parallelized using OpenMP across systems running Popcorn Linux. `libopenpop` optimizes multithreaded synchronization for distributed shared virtual memory systems like Popcorn Linux and utilizes new workload distribution mechanisms to ideally leverage the compute capabilities of heterogeneous CPUs. Using OpenMP benchmarks from NPB, Rodinia and PARSEC, `libopenpop` achieves a geometric mean 4.04x speedup for scalable application on a small homogeneous cluster. For a heterogeneous system composed of an x86-64 server and a high core count ARMv8 server connected via InfiniBand, `libopenpop` achieves up to a 4.7x speedup and a 41% geometric mean speedup.

- The design an implementation of Chameleon, a runtime re-randomization framework for preventing stack smashing and return-oriented programming attacks. Chameleon leverages the Popcorn compiler infrastructure to continuously randomize the stack layout and code of target applications, transforming a thread's execution state to match the new randomization. Using Chameleon on benchmarks from SPEC CPU 2017 and NPB, Chameleon disrupts a geometric mean 76.32% of code gadgets, randomizes stack elements to on average one of three possible locations, and randomizes with an overhead of 1.1% for a 50 millisecond re-randomization period.

Previous works present compiler and runtime systems for cross-ISA execution migration in order to perform a design space exploration for heterogeneous-ISA chip multiprocessors [70, 196]. These works simulate a heterogeneous-ISA CMP with cache-coherent shared memory, allowing the authors to demonstrate power and performance benefits of leveraging multiple ISAs. However, no such CMP exists at the commodity-scale at the time of writing this dissertation. Many ISAs are proprietary [13, 14] and even for open-license ISAs, their

cache implementations have compatibility issues due to ISA-specific memory consistency semantics [176, 10]. This dissertation instead proposes system software for cross-ISA execution migration in systems composed of commodity scale hardware. While some of the Popcorn compiler and run-time system components have similarities with DeVuyst et al. [70] and Venkat and Tullsen [197], there are significant differences. In particular, the Popcorn compiler and state transformation runtime are co-designed with the Popcorn Linux OS to implement thread migration and memory consistency across non-cache-coherence heterogeneous-ISA CPUs. This requires new low-level mechanisms for interacting with the OS, including insertion of migration points, performing state transformation, initiating thread migration and bootstrapping execution post-migration. The compiler also differs in that it does not attempt to create a common stack layout but instead fixes up references to stack elements at runtime. Additionally, the Popcorn compiler toolchain and state transformation handles runtime migration for multithreaded applications, which are not explored by previous works including [197]. A detailed discussion of the differences between the dissertation and [70, 197] is presented in Chapter 2. Thus, to the best of our knowledge Popcorn Linux (OS, compiler, runtime) is the first complete software architecture providing the ability to transparently migrate threads of execution between commodity scale heterogeneous-ISA CPUs at runtime without any application changes.

Using Popcorn Linux allows developers to more easily target future heterogeneous-ISA CPU systems. In particular, because Popcorn Linux extends the shared memory abstraction across non-cache-coherent CPUs, developers can re-use existing parallel programming models (e.g,. OpenMP [38] or Cilk [37]) and easily gain the benefits of heterogeneity. Existing multithreaded applications work as-is on Popcorn Linux; developers do not have to rewrite applications in a new programming model or environment to target new architectures. However, tuning applications to best take advantage of heterogeneous CPU systems poses a substantial challenge, as applications (and phases within applications) map differently to each architecture and cause different amounts of memory consistency communication over the network. `libopenpop` helps developers overcome these challenges by both minimizing cross-node synchronization traffic and by automatically distributing parallel work in consideration of system characteristics. Thus, Popcorn Linux helps developers regain programmability while simultaneously allowing them to easily benefit from advances in computer architecture.

Finally, the prevalence of security exploits is leading to new ideas on how to thwart attackers. In particular, security experts have begun devising new methods to prevent attackers from gaining control over applications or leaking sensitive information. One successful approach is to use randomization [34, 66, 203] to prevent the attacker from utilizing program structure to attack vulnerable applications. Because the Popcorn Linux compiler generates rich stack layout metadata and the state transformation runtime is proven to rewrite thread execution state with small latencies, this infrastructure can be repurposed into a security context to implement efficient and robust randomization.

1.2.1 Popcorn Compiler Toolchain

This dissertation presents the Popcorn compiler toolchain, which builds multi-ISA binaries suitable for migration across heterogeneous-ISA boundaries. The toolchain natively compiles applications written in C and C++ for all ISAs in the system using a common frontend and ISA-specific backends. The compiler automatically inserts migration points into the source code at function call sites. The compiler runs several analyses over an intermediate representation of the application to gather live data that must be transformed between ISA-specific formats. The compiler generates metadata (added as extra sections in the multi-ISA binary) describing the code and live data locations emitted for each architecture. The linker aligns global data in a common format (including thread-local storage), and a final post-processing step optimizes the application for efficient state transformation. The compiler is built using clang and LLVM [160] for compilation and GNU gold [86] for linking. The Popcorn compiler builds multi-ISA binaries with minimal changes to the core data layout mechanisms of the compiler, which allows our implementation to be more easily ported to new architectures unlike previous works [70, 197].

1.2.2 State Transformation Runtime

This dissertation presents a state transformation runtime for efficiently translating execution state of threads between ISA-specific formats. The runtime cooperates with the operating system scheduler to decide at which points to migrate. After the scheduler requests a migration, the runtime attaches to a thread's stack and begins state transformation. Using the metadata generated by the compiler, the state transformation runtime efficiently reconstructs the thread's current live function activations in the format expected by the destination ISA, including transforming a thread's register state, call frames and pointers to other stack objects. After reconstructing the stack, the runtime invokes the OS's thread migration mechanism and bootstraps on the destination architecture to resume normal execution. This dissertation also develops a methodology for invoking migration for multi-threaded applications in a real system. This dissertation describe how threads cooperate with the OS both before and after migration for seamless migration. It describes how the state transformation runtime attaches to and transforms an individual thread's state. Using this setup, this dissertation demonstrates that state transformation can be performed in under a millisecond, and oftentimes under several hundred microseconds, for real applications from the NAS Parallel Benchmarks suite [23]. Additionally, this dissertation presents an evaluation of Popcorn Linux that demonstrates up to a 66% reduction in energy and up to an 11% reduction in energy-delay product [119] for a multiprogrammed, datacenter-like workload.

1.2.3 Scale-Out and Heterogeneous OpenMP

This dissertation presents the design of an OpenMP [38] runtime named `libopenpop` optimized for systems composed of non-cache-coherent CPUs connected via distributed shared memory. In particular, `libopenpop` rebuilds many of the core components of OpenMP to prevent excessive overheads when running across multiple non-cache-coherent CPUs, where each CPU is designated as its own *domain*. `libopenpop` establishes a hierarchy of threads across CPUs and breaks OpenMP functionality down into local and global components. Using the hierarchy allows `libopenpop` to minimize the number of threads synchronizing on global data and therefore minimizes the amount of data movement required for synchronization. Using this thread hierarchy, `libopenpop` optimizes synchronization primitives like barriers, reductions and work distribution mechanisms. On a small cluster, `libopenpop` demonstrates a 38x speedup in multi-server barrier latency, a 5.4x speedup in multi-server reduction latency, and a geometric mean speedup of 4.04x for scalable applications.

In addition to refactoring the OpenMP runtime for scalability across non-cache-coherent CPUs, `libopenpop` introduces new parallel work distribution primitives that allow the OpenMP runtime to adapt parallel execution to best leverage the heterogeneous CPUs comprising the system. `libopenpop` monitors data movement (i.e., page transfers in distributed shared memory systems) and execution characteristics during parallel execution. Using this information, `libopenpop` determines whether to execute parallel computation across multiple CPUs or distribute work to only a single CPU. In the former case, `libopenpop` determines how much work to give each CPU to balance performance and minimize execution time. In the latter case, `libopenpop` automatically determines which CPU is best suited for a given computation. For an x86 machine and ARM machine interconnected via Infiniband, `libopenpop` demonstrates up to a 4.7x speedup and a geometric mean speedup of 41% over the best single-node homogeneous execution.

1.2.4 State Transformation for Runtime Re-randomization

This dissertation presents Chameleon, a runtime re-randomization framework that utilizes the Popcorn compiler to continuously re-randomize the stack layout and code of applications. Chameleon is an out-of-band framework, meaning that it executes in an entirely separate context from the target application and attaches to it via existing operating system interfaces. Chameleon's goal is continuously change the application's state so as to thwart exploits such as stack smashing attacks [152] and return-oriented programming (ROP) exploits [177]. Chameleon continuously generates new sets of randomized application code for target applications. Periodically, Chameleon pauses the target application and atomically switches it to the newly randomized code, transforming the target application's execution state from the previously randomized layout to the newly randomized layout. In this way, would-be attackers have a diminishing window of time in which to discover how Chameleon has laid out the application's state, craft an exploit and launch the attack. Chameleon dis-

rupts a geometric mean 76.32% of gadgets discovered by a gadget finding tool in benchmark binaries. Additionally, Chameleon can randomize the locations of stack elements to an average of 3 different locations per stack element, forcing the attacker guess where buffers are located with low probability. Finally, Chameleon provides these security benefits with low overhead – a geometric mean 1.1% overhead when re-randomizing the target application every 50 milliseconds. This is significantly better than other dynamic binary instrumentation (DBI) solutions, which add 14.9% or greater overhead [66, 203, 196].

1.3 Thesis Organization

This dissertation is organized as follows. The dissertation first describes the core Popcorn compiler infrastructure, including compiler and state transformation runtime. Next, the dissertation describes how the infrastructure is leveraged for accelerating multithreaded applications. The dissertation finally describes how the infrastructure is leveraged for enhancing the security of applications. Chapter 2 summarizes related work in each of the aforementioned areas, including execution migration in heterogeneous-ISA systems, scale-out/heterogeneous parallel execution and security. Chapter 3 describes Popcorn Linux, the replicated kernel operating system used to provide execution migration across ISA boundaries. It also formalizes the state of an application and describes the requirements for the compiler and state transformation runtime. Chapter 4 describes the Popcorn compiler toolchain which is used to analyze and build applications for cross-ISA migration. Chapter 5 describes the state transformation runtime and how threads migrate between architectures. Chapter 6 evaluates overheads associated with the state transformation runtime and energy benefits obtained when using execution migration in a datacenter context. Chapter 7 describes an exploration into using HTM to reduce migration response time. Chapter 8 describes `libopenpop`, including how it restructures OpenMP execution for cross-node execution. Chapter 9 evaluates scaling out OpenMP execution on a cluster. Chapter 10 describes how `libopenpop` makes workload distribution decisions in heterogeneous CPU systems. Chapter 11 evaluates `libopenpop`'s ability to leverage diverse CPU architectures. Chapter 12 describes Chameleon and how it uses the Popcorn compiler infrastructure to implement continuous re-randomization. Chapter 13 evaluates the security and performance properties of Chameleon. Finally, Chapter 14 concludes and describes future work in each of these areas.

Chapter 2

Related Work

2.1 Compiler and Runtime Support for Heterogeneous Architectures

Traditionally, developers have programmed heterogeneous architectures using a variety of programming models and languages. NVIDIA's CUDA [150] provides a programming language for NVIDIA GPUs. Using CUDA, developers partition their application into host (CPU) and device (GPU) code. Device code is offloaded to the GPU, and users must provide memory consistency by manually moving data between host and device memory spaces. More recently, CUDA offers managed shared memory between the host and device, but provides limited consistency guarantees. Thus, execution is offloaded to devices only at predefined locations and cannot be adapted in the face of changing workload conditions. OpenCL [92], OpenMP 4.0 [38] and OpenACC [153] offload computation to different target processors, but suffer from the same limitations as CUDA. Popcorn Linux provides strong memory consistency guarantees using distributed shared virtual memory and does not require applications to be partitioned between devices.

Saha et al. [170] describe an OS mechanism for shared memory between single-ISA heterogeneous cores interconnected over PCIe. Their programming model allows developers to open shared memory windows between the interconnected processors. These windows have a relaxed consistency, requiring developers to insert synchronization points to make memory writes visible across the PCIe bus. However, this programming model does not enable execution migration between interconnected processors, but rather uses a similar partitioning approach to CUDA. Popcorn Linux provides stronger consistency guarantees and flexible execution migration.

The Message Passing Interface (MPI) [91] provides a portable API for parallel processing using message passing for communication between processes. Processes execute in sepa-

rate address spaces but can share memory by manually sending and receiving data. The OpenMPI implementation [81] of the MPI standard supports serializing and de-serializing memory into ISA-specific formats, hiding cross-architecture data representation issues behind the interface. However MPI does not support execution migration at arbitrary points – developers manually insert data transfers and coordinate execution across processes on different machines within the application source code. Similarly to the programming models listed above, this hinders programmability and the flexibility of the system to adapt to changing workload conditions. PC^3 [76] uses a modified C/MPI compiler to instrument MPI applications for execution migration in a cluster and uses checkpointing to transfer state. However developers must manually annotate checkpointing locations and the compiler only accepts MPI applications that have well-typed code. Furthermore, the checkpointing system requires annotating data with descriptors as the data comes into and goes out of scope, adding significant runtime overhead for metadata collection in addition to checkpointing costs. Popcorn Linux allows efficient and flexible execution migration between processors and distributed shared virtual memory.

The Lime programming language [21] and the Liquid Metal runtime [20] together implement a language system for seamless execution across heterogeneous architectures. Developers build data-flow applications in a Java-based language. The runtime distributes computation nodes of the data-flow graph across architectures and uses serialization coupled with message passing to automatically send data between architectures. The system is limited in that developers must use a data-flow programming model (they cannot use traditional SMP semantics) and they must manually annotate properties of data types so that the runtime can transfer state. The Dandelion compiler [168] and PTask runtime [167] are similar in that programmers develop data-parallel applications in a high-level language (e.g., C#) which is decomposed into a data-flow execution model. The runtime then distributes computation nodes to devices in a cluster, automatically managing communication between the different contexts. Like Lime and Liquid Metal, developers must use a restrictive programming language, and the system is designed solely for data-parallel applications. Popcorn Linux lets programmers develop applications using a shared memory programming model across heterogeneous-ISA architectures.

2.2 State Transformation

Various techniques have been developed to translate state between machine-specific formats. Dubach and Shub [73] and Shub [180] describe a user-space mechanism for single-threaded processes to migrate themselves between heterogeneous machines. They describe modifications to executables needed for migration, including multiple code sections, data padding (using the greatest common denominator of data sizes and alignments), and how to translate data types between architecture-specific formats. However this approach is completely user-controlled, and furthermore incurs large overheads for state transformation. Work by

Zayas [208] shows that state transformation can also be applied as pages are migrated between machines, rather than in bulk at migration time. Theimer and Hayes [191] describe an alternative translation approach where a program's execution state is lifted into a machine-independent format and recompiled to recreate the state on the target machine. All of these approaches were designed assuming the main bottleneck in process migration was communication and not state translation. With newer high-bandwidth networking technologies such as PCIe point-to-point connections [183] or Infiniband [18], this is no longer the case. The Popcorn compiler toolchain and state transformation runtime avoid most state transformation overheads by construction – applications runs on architectures which use the same primitive data sizes and alignments. Additionally, the compiler and runtime minimize overheads through alignment and by only transforming a small portion of application state.

Attardi et al. [19] describe a number of user-space techniques for heterogeneous-ISA execution migration. They describe running the program in a machine-independent format via interpretation, re-compiling the application on the fly for a different target ISA, and translating runtime state between machine-specific formats. The TUI system [181] implements a combination of these approaches – it lifts the application's state into an intermediate format and then lowers it to the target machine's format. Additionally, TUI implements migration of I/O descriptors using a custom standard C library and an external remote server. These approaches incur significant translation overheads, however. As mentioned previously, Popcorn elides much of this overhead through careful data layout and minimal runtime transformations. Popcorn Linux also pushes cross-ISA I/O functionality into the kernel.

More recently, Ferrari et al. [77] propose a mechanism for state checkpointing and recovery using introspection. They implement a source-to-source compiler which modifies applications to periodically save stack data in an architecture-independent format. The compiler also refactors functions to be able to restore this state after a migration. This technique is very invasive in terms of source code modifications, and incurs significant overhead for periodic state saving procedures which record information for all functions on the stack. The Popcorn compiler toolchain makes minimal transformation to code, other than inserting migration points.

Makris and Bazzi [135] present a mechanism for stack transformation to be used for in-place software updates. A compiler performs source-to-source transformation so that threads recursively save their stack (including all variables within call frames) before migrating. The threads then reconstruct their stack with the new version of the application. Their approach attempts to solve a harder problem of reconstructing state for a different version of the application, and thus requires user-driven help. The state transformation runtime focuses on transforming state between machine-specific versions of the same application, rather than a modified application for the same ISA.

2.3 Heterogeneous-ISA Execution Migration

von Bank et al. [200] formalize a model of procedural applications executing in a system. They identify the various components of an application, including program data and machine code, that must be equivalent in order for execution to be migrated between architectures at points of equivalence. They define these locations as program points where a transformation exists between different representations of an application, i.e., compilations for different targets. The Popcorn compiler toolchain builds upon their definition of points of equivalence.

Many works use language-level virtual machines to perform heterogeneous-ISA migration. Heterogeneous Emerald [184] implements a TUI-like heterogeneous migration system for the Emerald language. PadMig [83] and JnJVM [84] migrate threads of execution between Java virtual machines (JVM), using Java's reflection capabilities to serialize/de-serialize objects between architecture-specific formats. COMET [88] and CloneCloud [54] also use the JVM to transparently offload portions of applications from mobile devices to the cloud over the network. COMET additionally uses a DSM system to ship data between the device and the cloud. Neither approach implements full execution migration, but only offloads a portion of the application to the cloud. The drawbacks with all language-level approaches is that applications must be implemented using the specified language. A significant amount of legacy code is therefore not suitable for migration in these systems. For languages like Java, applications may experience severe performance degradation versus being written in a compiled language like C. Finally, language introspection mechanisms have high latency, meaning translation costs may dominate execution migration overheads. Virtual machines like QEMU [31] also enable heterogeneous-ISA migration, but experience unacceptably high performance losses. Popcorn Linux provides cross-ISA execution migration for natively compiled applications, allowing native-execution speeds and low migration overheads.

More recent works explore process migration in heterogeneous-ISA systems for native applications. Lee et al. [121] propose a compiler and runtime for refactoring applications to offload computation from ARM smartphone CPUs to x86 server CPUs. Their work is restricted to only offloading portions of smartphone applications and requires expensive runtime translation between ISA-specific data layouts. Barbalace et al. [27] describe an operating system and compiler for offloading application computation from an x86-64 Xeon to an overlapping-ISA Xeon Phi processor. The compiler prepares applications for execution on both architectures, but there is no mechanism to perform state transformation – migrated threads must return to the host after executing the offloaded computation. DeVuyst et al. [70] and Venkat and Tullsen [197, 196] implement process migration in simulated heterogeneous-ISA CMPs in order to perform a design space exploration. All three works use a custom compiler and runtime to migrate threads between heterogeneous-ISA cores which shared cache-coherent shared memory. The compiler generates metadata describing a state transformation function for individual call frames. The runtime performs dynamic binary translation (DBT) when a migration is requested until the application reaches a location where state can be translated and native execution can resume. Popcorn Linux, the Popcorn compiler toolchain and the

state transformation runtime differ in several ways:

1. [70, 197, 196]'s prototype uses a simulated heterogeneous-ISA CMP with cache-coherent shared memory. Furthermore, [70, 197, 196]'s prototype does not incorporate an operating system. Popcorn Linux demonstrates execution migration on real hardware using an ARM and an x86 processor interconnected via high-speed networking using a complete software stack.

2. [70, 197, 196]'s prototype does not support multi-threaded applications. [70, 197, 196]'s compiler does not support aligning thread local storage, and [70, 197, 196]'s runtime does not provide a solution for performing state transformation in a multi-threaded environment. The Popcorn compiler toolchain includes a linker which lays out thread local storage in a common format for all ISAs in the system, and the state transformation runtime is designed to be thread safe so that threads in multi-threaded applications can migrate between architectures without blocking.

3. In order to perform stack transformation between ISA-specific formats, [70, 197, 196]'s compiler modifies each function's call frame layout to adjust the size, layout of individual sections of the call frame, and layout of objects within the call frame. [70, 197, 196]'s compiler generates a mostly-identical call frame layout across different compilations of the application. This adds complexity to the compiler including changing the flow of the compilation pipeline, making porting [70, 197, 196]'s toolchain to new architectures difficult. The Popcorn compiler toolchain instead minimizes changes to the compilation pipeline and pushes handling of pointers to stack elements into the state transformation runtime. The evaluation demonstrates that even with handling pointers to stack elements at runtime, state transformation latencies are low.

4. [70, 197, 196]'s work does not describe how machine code is loaded into memory, and in particular how after migrating to another ISA, a thread is able to locate its ISA-specific code without rewriting function pointers. Popcorn Linux provides this mechanism transparently to application threads.

5. [70, 197, 196] do not describe how a migration or state transformation is invoked, but rather only mention that a migration is triggered through some external event. In our system, the Popcorn compiler toolchain inserts migration points into the source code, trigger migrations using the operating system, and use a library which lets threads transform their own stack.

6. [70, 197, 196]'s prototype allows migration at arbitrary points by performing dynamic binary translation (DBT) up until an equivalence point. Popcorn Linux does not have this ability, but rather the OS and application cooperate to migrate threads. Although this hinders the scheduler's flexibility, it significantly reduces migration costs and runtime complexity. [70, 197, 196]'s results show that DBT can cause up to a several millisecond delay when migrating.

2.4 Scaling Applications to Rack-Scale Systems

Traditionally, developers have used the message passing interface (MPI) to distribute execution across domains [91]. Deemed the "assembly language of parallel processing" [118], MPI forces developers to orchestrate parallel computation and manually keep memory consistent across domains through low-level send/receive APIs, which leads to complex applications [27]. Partitioned global address space (PGAS) languages like Unified Parallel C [57] and X10 [49] provide language, compiler and runtime features for a shared memory-esque abstraction on clusters. How threads access global memory on remote domains is specific to each language, but usually relies on a combination of compiler transformations, runtime APIs, and user-specified memory consistency semantics. Additionally, PGAS languages require users to define thread and data affinities, i.e., which threads access what data. This hinders system flexibility in adapting to multiprogrammed workloads. More recently, many works have re-examined distributed shared memory abstractions in the context of new high-bandwidth interconnects. Grappa [145] provides a PGAS programming model with many runtime optimizations to efficiently distribute computation across a cluster with high-speed interconnects. Grappa relies on a tasking abstraction to hide the high costs of remote memory accesses through massive parallelism, meaning many types of applications may not fit into their framework.

Previous works evaluate OpenMP on software distributed shared memory systems [140, 29, 96]. These approaches require complex compiler analyses (e.g., inter-procedural variable reachability) and transformations (software DSM consistency boilerplate, data privatization) in order to translate OpenMP to DSM abstractions, which limit their applicability. OpenMP-D [118] is another approach whereby the compiler converts OpenMP directives into MPI calls. This process requires sophisticated data-flow analyses and runtime profiling/adaptation to precisely determine data transfers between domains. Additionally, OpenMP-D limits its scope to applications that repeat an identical computation multiple times. OmpCloud [207] spans OpenMP execution across cloud instances using OpenMP 4.5's offloading capabilities [38]. However, computation must fit into a map-reduce model and developers must manually keep memory coherent by specifying data movement between domains.

All of these previous works have limitations in that either the developer must rewrite applications in a new programming model or have limitations when extending existing shared memory parallel programming models (e.g., OpenMP) into multi-domain settings. Popcorn Linux instead provides the ability to run existing shared memory applications across multiple domains. However, naïvely executing multithreaded applications across multiple domains can cause excessive traffic in software distributed shared memory systems.

2.5 Work Distribution in Heterogeneous Systems

Currently, developers have limited options in terms of programming models to support execution across heterogeneous-ISA systems. Shared-memory parallel programming models like OpenMP [38] and Cilk [37] provide source code annotations to automate parallel computation, but do not support execution across cache-incoherent, heterogeneous-ISA CPUs. MPI [91] gives developers low-level primitives to distribute execution, manage separate physical memories and marshal memory between heterogeneous-ISA CPUs. However for asymmetric CPUs, developers must manually assign parallel work and transfer the required data to maximize performance, leading to complex and verbose applications with static, non-portable workload distribution decisions. PGAS frameworks like UPC [57], X10 [49] and Grappa [145] support cross-node execution and memory accesses, but do not support sharing data between heterogeneous-ISA CPUs. Even if heterogeneous-ISA execution was possible, changing workload distribution decisions in light of system characteristics is cumbersome – data is not migrated between nodes for locality, meaning re-balancing work distribution decisions can cause additional network transfers and thus more overhead. Cluster frameworks like SnuCL-D [112] and OmpSs [44] provide coarse-grained work distribution in clusters by assigning multiple independent parallel computations to individual heterogeneous processors. They do not consider fine-grained work-sharing of a single parallel computation or automatic workload placement, and require developers to specify data movement (device data transfer commands for SnuCL-D, `in/out/inout` clauses for OmpSs). In comparison to these works, `libopenpop` automatically distributes work in consideration of platform characteristics and leverages transparent and on-demand DSM to manage memory consistency for flexibility and programmability.

Several works explore fine-grained work distribution in CPU/GPU systems. Qilin [133] is a compiler and runtime that enables CPU/GPU workload partitioning but requires developers to rewrite computation using a new API. Unlike `libopenpop`, Qilin does not make distribution decisions online but must profile multiple full executions before determining the optimal workload split. Kofler et al. [114] present a machine learning approach to determining workload distribution, but require sophisticated analyses with a custom compiler and the machine learning model must be retrained for each new hardware configuration. Similarly, Grewe and O'Boyle [90] present a machine learning approach that requires per-system retraining. Scogland et al. [174] present CPU/GPU workload distribution approaches for accelerated OpenMP. However their approach only works for dense array-based computations and developers must manually specify data movement between devices. All of these approaches only explore CPU/GPU systems. Additionally, these approaches are limited by the visible split in CPU and GPU memory and require developer intervention to help marshal data. Additionally, none of these approaches provide optimized cross-node synchronization primitives and none consider situations where cross-node execution may not be beneficial.

There are a number of schedulers designed to improve task-parallel workloads (as opposed to data-parallel workloads targeted by `libopenpop`) on single-ISA heterogeneous systems,

e.g., ARM big.LITTLE [89]. The Lucky scheduler [158] measures the energy efficiency of multiprogrammed workloads via performance counters and uses lottery scheduling to time multiplex applications across big and little cores. The WASH AMP scheduler [108] classifies threads in applications written in managed languages (e.g., Java) using performance counters and schedules threads to remove bottlenecks (e.g., critical sections). Other works like meeting point thread characterization [163] and X10Ergy [179] propose other means for characterizing and accelerating individual threads on single-ISA heterogeneous platforms. All of these works focus on determining the "critical" task in task-parallel workloads and placing it on the most performant core. Additionally, none deal with cache-incoherent heterogeneous-ISA CPUs, meaning they do not consider data marshaling and cross-node memory access costs.

None of these works fully automate workload distribution for multithreaded applications across heterogeneous-ISA CPUs. This dissertation describes how `libopenpop` is extended to analyze execution characteristics and automatically distribute work to leverage the compute capabilities of diverse CPUs.

2.6 Runtime Re-randomization

In recent years there has been a large amount of security research focused on analyzing and defending against return-oriented programming attacks [177]. These attacks stitch together small "gadgets" from existing code inside the application to build arbitrary functionality. This obviates the need for injecting malicious code into the target application and has spawned a whole field of security research.

There are two classes of defenses that have emerged to disrupt ROP-style attacks: control-flow integrity [7] and code diversity [120]. In the former, the compiler or runtime system instruments the application to only allow control flow transfers that were originally encoded in the application, e.g., no jumps to arbitrary instructions as used by ROP attacks. However, CFI defenses must make sacrifices in both performance and completeness, as instrumentation can add significant overhead (Abadi et al. report 16% overhead on average for SPEC CPU 2000 [7]) and new types of attacks build exploits out of correct control flow transitions [172]. Instead, many defenses propose using code diversity (i.e., utilize multiple semantically-equivalent code variants) such as code randomization to disrupt gadgets and gadget chains used by ROP attacks. Address space layout randomization (ASLR) [178] uses position-independent code to randomize the locations of application sections (code, data, heap, etc.). Unfortunately, ASLR only provides coarse-grained randomization – memory leaks allow attackers to discover the base addresses of sections and de-randomize an application's layout. Newer forms of code diversity apply ASLR-like principles at a finer granularity, e.g., randomizing the locations of functions (ASLP [111] and Oxymoron [22]) or basic blocks within a function (binary stirring [202]). Readactor [64] uses a custom hypervisor to map code pages (including specially-generated function trampolines) with read-only permissions and mitigates memory disclosures by forcing all control flow to go through the trampolines.

This leads to an average overhead of 6.4%, but uses a complex software architecture and is specific to x86-64. Other defenses instead randomize the code in-place, re-arranging instructions within a single basic block (i.e., sequence of instructions ending in control flow), replacing sequences with semantically equivalent but different instructions and re-assigning registers and changing the location of stack slots [156, 115]. All of these approaches only perform one randomization at target application load time.

With the advent of attacks such as JIT-ROP [182] that dynamically discover gadgets (i.e., post-randomization), new defenses were proposed that provide other forms of randomization during runtime. RuntimeASLR [131] tracks code pointers and randomizes the code layouts of forked children (rewriting the pointers to reflect the new code locations) to thwart Blind-ROP attacks on web servers. However, RuntimeASLR is only applicable to server-style applications with a master/worker model such as web servers. Additionally, RuntimeASLR adds significant overhead in the master process for tracking pointers. Isomeron [66] creates two copies of each function that are semantically equivalent but implement functionality using different sets of instructions and dynamically selects between them at runtime, forcing gadget compilers to guess which version of each function is being used with decreasing odds of success. HIPStR [196] uses a similar idea but also adds the ability to switch between heterogeneous ISAs to add extra entropy to the randomization. Both works require the use of a dynamic binary instrumentation (DBI) or dynamic binary translation (DBT) framework, and thus add significant overheads to normal execution. TASR [34] and Shuffler [203] continuously re-randomize the locations of code; TASR uses compiler and kernel modifications to add code pointer tracking, whereas Shuffler uses a layer of indirection to capture all code pointer references. Again, however, these approaches create significant performance overhead – TASR requires complete and correct debugging information, limiting which compiler optimizations can be applied, and Shuffler's code transformations add significant normal overhead. These works add 30%-40% and 14.9% overhead, respectively, to normal baseline execution. Smokestack [9] continually randomizes the layout of stack frames by permuting stack slot elements for every invocation of a function. While it incurs low overheads for less predictable permutation selection algorithms, Smokestack only targets data-oriented programming attacks and hence is susceptible to other forms of code-reuse attack. Additionally, it utilizes Intel-specific AES instruction extensions (although other vendor-specific AES instructions could potentially be used). CodeArmor [52] decouples the code address space into virtual and concrete instances. Code is instrumented at compile time to use a linear translation to convert virtual code references to concrete addresses which are continuously randomized; addresses simply use the updated linear translation to switch between randomizations. However, CodeArmor incurs a 6.9% average overhead for compute-intensive applications or a 14.5% overhead for server applications and uses x86-64-specific segment registers.

The large number of randomization frameworks suggest that diversity-based defenses are more popular than control flow integrity defenses. However, previous randomization approaches either only perform an initial randomization [202, 111, 156, 115] and are thus

susceptible to dynamically constructed exploits or require invasive, complex and slow instrumentation frameworks [66, 196, 34, 203]. Instead, Chameleon uses in-place code randomization (similarly to Pappas et al. [156] and Koo et al. [115]) and performs re-randomization outside the context of the target, providing both code diversity and low overhead.

Chapter 3

Background

This work presents compiler and runtime support for seamlessly running applications across heterogeneous-ISA CPUs in emerging systems. There are many benefits to exploiting these systems, including higher performance, better energy efficiency, increased scalability, and stronger security mechanisms [70, 197, 196, 27, 127, 151]. All of these benefits require thread migration between processors in the system. Thread migration is the act of moving a thread's execution context (including live register state, runtime stack, page mappings, etc.) between different processor cores in a system [186]. Current monolithic kernel OSs like Linux provide thread migration in SMP systems through hardware and OS mechanisms [155]. However, thread migration across heterogeneous-ISA processors requires additional compiler and runtime support due to the fact that the compiler builds the application specifically for a processor's ISA.

This work provides several important components for Popcorn Linux, a replicated-kernel operating system designed to provide OS support across diverse processors. This work describes the design of the Popcorn compiler toolchain and state transformation runtime for Popcorn Linux, all of which work together to replicate an application's execution environment across a tightly coupled heterogeneous-ISA system.

Section 3.1 describes the design of Popcorn Linux's OS and the facilities it provides for execution migration. Section 3.2 provides a formal definition of application state and how the compiler, runtime and OS cooperate to ensure it accessible across processors of different ISAs. Finally, Section 3.3 describes the expectations of the compiler and runtime when constructing an application's execution state.

3.1 Replicated-Kernel Operating Systems

Traditional process-model monolithic operating systems such as Linux maintain all operating system services and state in a single kernel instance, which operates as a single process in the system. The kernel is responsible for managing all devices in the system, many of which require interacting with system- or architecture-specific interfaces. The kernel provides a series of abstractions which hide low-level hardware details from applications executing in the system. The kernel must handle virtual memory management, disk access, networking, etc., which require ISA-specific implementations. Because of this, the kernel is heavily tied to and must be compiled specifically for the underlying architecture.

Recent work has begun to question traditional OS architecture due to increasing core counts and heterogeneity. The multikernel [30] is a new OS design which treats a high core count shared memory machine as a distributed system. The multikernel is designed to address scalability and heterogeneity barriers by distributing pieces of the system across multiple kernels. The multikernel boots several instances of the kernel, each of which owns a partition of the physical memory and a subset of available devices. Kernels communicate via message passing to share access to devices, but applications execute in a distributed fashion across the kernel instances. Because of this, shared-memory applications must be rewritten to take advantage of the multikernel. Unlike microkernels [124] which move kernel services into separate processes that communicate via message passing, each kernel instance in a multikernel is a full-fledged monolithic kernel capable of moderating all devices which it owns.

The replicated-kernel OS [27] is an extension of the multikernel which expands shared-memory programming support to a multiple-kernel OS. Figure 3.1 shows the architecture of a replicated-kernel OS, including the interface presented to applications. The replicated-kernel OS is similar to the multikernel in that multiple kernel instances run simultaneously and system resources are distributed among them. However rather than exposing the distributed nature of the OS, the kernel instances work together to present a single system image to applications executing in the system. Threads of an application can migrate between kernels, and the application's address space and OS state are replicated so that threads execute in an identical operating environment. Because the OS mediates all access to devices (requiring applications to use the system call interface), applications can use traditional POSIX interfaces for disk, networking, process control, etc. The kernels coordinate access to devices in order to provide services regardless of where the application executes. This architecture allows applications to continue to use a shared-memory programming model, while the OS architecture can be adapted to suit different levels of parallelism and heterogeneity.

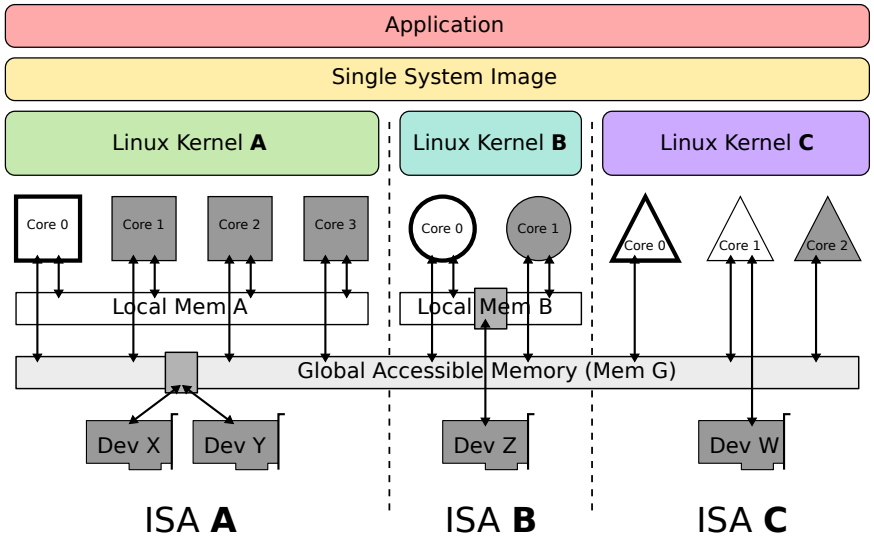


Figure 3.1: Replicated-kernel OS architecture and application interface

3.1.1 Thread Migration

In a replicated-kernel OS, each kernel owns and is run on a subset of the available processors in the system. Because kernels have a number of ISA-specific components, in heterogeneous-ISA systems a kernel instance is run on each set of same-ISA processors (called a **processor island**). For example, in a heterogeneous-ISA platform containing an x86 CMP interconnected to an ARM CMP, the replicated kernel OS would run one kernel instance on the x86 processor island and another instance on the ARM processor island. The scheduler can migrate application threads between processors of different kernels, or threads can migrate themselves by setting their CPU affinity to a processor owned by a specific kernel.

The replicated-kernel OS enables thread migration between kernels through the use of shadow threads. When a thread migrates from a source to a destination kernel, the destination kernel spawns a new thread and the original thread is put to sleep on the source kernel. In this scenario, the original thread that is put to sleep is known as a shadow thread. The newly spawned thread is populated with the original thread's execution context and resumes execution on the destination kernel. The replicated-kernel OS keeps track of which shadow threads correspond to which new threads executing on the kernels in the system. All thread contexts are kept alive until the application exits, at which time the kernels broadcast teardown messages that trigger a cleanup of all thread contexts associated with the application [109].

At which program locations threads are able to migrate depends on which ISAs are available in the system. If all processors use the same ISA, then threads can migrate between kernels at arbitrary locations due to the fact that all threads execute using the same implementation of the application, i.e., the same data layout and machine code. From the application's

point of view, this is equivalent to migrating between cores in an SMP multiprocessor. If kernels execute on processor islands of different ISAs, then threads can only migrate at pointwise-equivalent program locations [200], known as **equivalence points**, in the application. Equivalence points are matching program locations in two separate implementations of an application (i.e., two compilations of the application for different ISAs) that satisfy three properties:

1. At the specified program location, the set of live variables for both implementations are equivalent. This means that there are the same number and types of live variables at the program location.

2. All variables have been stored to memory, i.e., no variables are stored in registers. While seemingly very strict, the ISA's calling convention satisfies this requirement. Any values required to be saved will have been saved as part of the register save/restore procedure except for the outermost frame. At an equivalence point, a runtime can take a snapshot of current registers, thereby placing all live values into memory.

3. The structure of the two computations must be similar, i.e., the result of a set of computations must be equivalent. The granularity of this sub-computation equivalence can be adjusted from a single instruction up to the entire application's execution. A finer granularity reduces possible compiler optimizations, while a coarser granularity limits the number of equivalence points.

At equivalence points, there exists a state transformation function between ISA-specific versions of the application's state. The compiler, OS and runtime cooperate to perform this translation, after which the thread can resume execution post-migration.

3.1.2 Distributed Shared Virtual Memory

Although several efforts have explored cache-coherent shared memory for simulated heterogeneous processors [70, 197, 98], no commodity scale heterogeneous-ISA CMPs currently exist that support cache-coherent shared memory. In order to sidestep this issue, the replicated-kernel OS provides distributed shared virtual memory (DSVM or DSM). In DSVM systems, a runtime or operating system provides a single view of addressable memory to applications executing across multiple computing nodes, each of which has its own physical memory. The DSVM system mediates access to memory objects which are either stored in a node's local memory or in a remote node's memory. The DSVM system provides access to remote memory objects either by direct reads and writes to remote physical memory regions [57, 49] or by migrating memory objects between memory regions to increase data access locality [12, 127, 25]. The DSVM system provides the illusion of a single shared memory region overlaid across multiple physical memory regions, allowing applications to be developed using a shared-memory programming model [162].

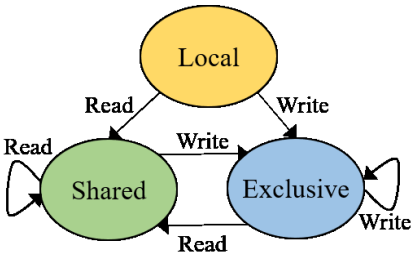


Figure 3.2: Page coherency protocol. Pages permissions are maintained similarly to a cache-coherency protocol to provide consistent views of memory across processor islands. Multiple nodes may map a page as readable, but only a single node may map the page as writable.

The replicated-kernel OS provides DSVM for threads of an application executing on different kernels. As threads migrate between different kernels (and therefore, different processor islands) in the system, the kernels communicate to migrate pages on-demand so that threads are able to access code and data. After a thread migrates, it resumes execution at an equivalence point in user-space. However there are no pages mapped into the application's address space on the destination kernel – the thread causes a page fault as soon as it accesses any code or data. The destination kernel sends a message to the source kernel requesting the page and any mapping information for the faulting address. The page is transferred from the source to the destination kernel, which maps the page into the application's address space and returns from the page fault. The thread continues execution as normal, most likely causing more page faults which get resolved in a similar fashion. This mechanism allows the kernels to reconstruct the application's address space regardless of where threads execute.

The DSVM system provides coherency at the granularity of a page of memory. The replicated-kernel OS uses a page coherency protocol [169] across kernels that acts like a multiple-reader, single-writer lock on pages – Figure 3.2 shows the state transition diagram for page access permissions. When application threads executing on a single kernel access a page, there is no coherency required, hence the page is mapped with *Local* permissions. When threads executing on different kernels access a page with read-only permissions, the page is mapped with *Shared* permissions and replicated across both kernels. This allows both concurrent across multiple processor islands (and hence improved scalability) and data access locality. However, when a thread writes to a page and thus has both read and write permissions, only one kernel may own the page at a time. When a thread migrates to a new node and writes to a page, the source kernel unmaps the page from the application's address space (only on the source kernel) and migrates it to the destination kernel, where it is subsequently mapped into memory. If a thread on the source kernel tries to access the same page, the process is reversed – the page is unmapped from the application's address space on destination kernel and migrated to the source kernel. This prevents consistency issues from multiple writes to the same page of memory and supports ISA-specific locking mechanisms across architectures (e.g., compare-and-swap instructions in x86 versus load-link/store-conditional instructions in ARM). However it can lead to pathological behavior and poor performance when threads

spread across multiple processor islands access the same pages [169].

Using these mechanisms, the replicated-kernel OS allows threads to migrate between processors of different ISAs while executing in a replicated working environment. Popcorn Linux implements thread migration and DSVM through a series of distributed kernel services between kernels on different processors.

3.2 Application State

As mentioned in Section 3.1.1, there exists a state transformation function at equivalence points that can convert between ISA-specific formats of an application's state. In order to understand how application state can be transformed by the compiler and runtime in a replicated-kernel OS, a formal model of application state is defined. A model allows us to understand which parts of the application can be laid out in a common format across ISAs, and which parts of the application should be transformed at runtime between ISA-specific formats. For application state laid out in a common format, no transformation is required and the replicated-kernel OS can simply migrate the state between kernels. Special handling is required for state that must be transformed, however.

3.2.1 Formalization

We consider a model in which applications execute as a single process in a replicated-kernel operating system, and may utilize several threads of execution. We do not consider multi-process applications, although the model can be extended to support them. Additionally we do not support self-modifying applications, or applications which generate or modify their machine instructions. Applications executing using a traditional von Neumann architecture are comprised of data and code, both of which are stored in the same region of addressable memory[1]. In process-model monolithic operating systems, the OS creates a virtual address space V_A for each application A. An application's virtual address space V_A is composed of per-process state P and per-thread state T_i, where $1 \leq i \leq k$ for an application which has k threads of execution. The compiler, linker and OS work together to construct V_A so that threads of execution are able to access required code and data.

The application's per-process state P consists of code memory P_C, statically-allocated data memory P_D, and dynamically-allocated data memory P_H. Code memory P_C includes all machine code generated by the compiler for a target ISA, and is included as the `.text` section in ELF binaries. Statically allocated global data memory P_D is created by the compiler and linker, and is included as `.data`, `.rodata` and `.bss` sections in ELF binaries (which

[1] Popcorn Linux's DSVM blurs the notion of a single region of memory, but it provides the abstraction that threads executing on different kernels are able to address code and data in the same address space.

correspond to initialized data, read-only initialized data, and uninitialized/zero-initialized data, respectively). Code memory P_C and statically-allocated data memory P_D are laid out in the binary by the compiler and linker, which may optimize placement for cache locality [46, 138, 85]. Dynamically-allocated global memory P_H is created on-demand by standard memory allocation routines, e.g., `malloc`, in the process' heap.

The per-thread state T_i is composed of a set of registers R_i, a thread's execution stack S_i, and a block of thread-local storage (TLS) L_i. The compiler is responsible for laying out all components of T_i. The compiler allocates storage for function-local data across R_i and S_i, aggressively optimizing the layout to take advantage of the ISA's resources and capabilities. The compiler also lays out L_i by optimizing placement of variables declared with a thread-local qualifier (such as `__thread` in GCC) for cache locality, similarly to P_C and P_D. All TLS variables for a single instance of L_i are collected into ELF sections such as `.tdata`, `.trodata` and `.tbss` to create an initialization image. L_i is instantiated by creating a copy of the initialization image for every thread in the application.

Each application also has associated kernel state maintained by the replicated-kernel OS, e.g., open files, network sockets, IPC handles, etc. In this model we omit definitions for kernel-specific application state – the kernels keep the state consistent via message passing, but from the application's point of view, the kernel reproduces a single system image. Thus, the application does not need to know about how kernel-side state is organized.

In order to achieve seamless execution migration, an application's virtual address space $V_A = \{P, < T_1, T_2, ..., T_k >\}$ (where $P = P_C, P_D, P_H$ and $T_i = \{R_i, S_i, L_i\}$ for $1 \leq i \leq k$) must be constructed so that threads executing on any ISA in the system can locate code and data. To create V_A, the compiler and linker can either align code and data in a common format so that no transformation is required, or the compiler can extract application metadata so that a runtime dynamically translates state between architecture-specific layouts. In this context, translating program data refers to both changing the content of the data between ISA-specific formats (**reification**) and changing the location of the data (**relocation**). In practice a combination of common layout and transformation is applied in order to minimize translation costs caused by application migration while simultaneously allowing applications to achieve highly optimized execution [70, 197].

3.2.2 Laying Out Application State

Attardi et al. [19] and Smith and Hutchison [181] describe mechanisms that enable heterogeneous-ISA execution migration by either maintaining program state in a target-agnostic intermediate format, such as Java bytecode, or by directly translating the application's entire address space V_A between target-specific formats during migration. Whole-program interpretation and translation are suitable for highly diverse targets, including targets which have differences in primitive data type sizes and alignments, differences in pointer sizes, and differences in endianness. However these mechanisms incur significant overheads, either due to the cost of

interpreting applications for an ISA-agnostic abstract machine or due to the cost of translating the entire address space of applications between formats. More recent work by DeVuyst et al. [70] and Venkat and Tullsen [197] describes techniques for minimizing translation costs by imposing stricter requirements for all target ISAs in the system, i.e., equivalent data sizes, alignments, pointer sizes, endianness. Additionally, their modified compiler toolchain aligns code and data in a common format across all ISAs on which threads execute, side-stepping translation costs due to relocating data. This work is extended by the Popcorn compiler toolchain and state transformation runtime.

Because the ISAs used for Popcorn Linux have identical data types and sizes, application state P_D and P_H do not need to be reified between ISA-specific formats. Conceptually, L_i is a per-thread "global storage" meaning that it too does not need to have its content transformed. However, code memory P_C is not compatible across architectures, as the ISA defines the machine code format. Because P_C does not change at runtime, its reification between formats is performed offline by the compiler. Specifically, the compiler generates multiple versions of P_C offline by compiling the application for each target ISA in the system. Runtime transformation simply becomes a problem of mapping the correct version of P_C into memory depending on which architecture threads are executing. As threads migrate between processor islands, the kernels map the appropriate version of P_C into V_A, making P_C an **aliased** region of memory.

Relocating data to different areas of memory causes all references to that data to be invalidated. In order to eliminate relocation costs, including the difficult task of finding all such references wherever they are stored (e.g., function pointers in C++ vtables), the compiler and linker lay out symbols in P_C and P_D at common addresses across all compilations of the application so that global data and function pointers are valid for all ISAs in the system[2]. References to P_H are also valid across all architectures – the DSVM system keeps the heap pages consistent, including all heap object metadata. The page coherency protocol ensures that accesses to P_D and P_H are replicated and coherent between kernels, and the OS automatically maps the correct version of P_C. Thus, data objects in P_C, P_D and P_H are **aligned** across executions on all ISAs.

The remaining parts of the execution state V_A are dictated by the ISA (e.g., registers R_i) or are highly tuned for each architecture (e.g., the stack S_i). For these parts of the execution state, it is either impossible to lay data out in a common format or doing so would cause severe performance degradation. Instead of using a common format and aligning data across compilations, runtime state transformation is applied to convert R_i and S_i between architecture-specific formats. Thus, the compiler must generate metadata so that the state transformation runtime can both reify and relocate R_i and S_i.

[2]Language semantics prevent function pointers into the body of a function, meaning that only the beginnings of functions must be aligned.

3.2.3 ISA-specific State

A thread's register set R_i and runtime stack S_i are partially specified by the architecture-specific application binary interface (ABI), which describes how applications represent, access and share data in the system. One component of the ABI is the function call procedure, which specifies how threads execute functions in an application. The function call procedure describes how to set up per-function R_i and S_i state, how to pass arguments to called functions using R_i and S_i, how to save and restore live registers (i.e., those parts of R_i which contain live values) in S_i, and how to pass return values back to the calling function. Each instance of a called function creates a **function activation** that becomes part of a thread's execution state. According to the DWARF debugging information standard [58], there are three pieces of information that define a function activation:

1. A **program location** within the function, either in a program counter register or saved in a child function's activation as a return address. The program location indicates the machine instruction currently being executed, or the instruction at which execution will resume after a returning from the child function, respectively.

2. A contiguous block of memory on the thread's stack S_i named the function's **call frame**. The call frame contains a function's live values and information connecting a function activation to surrounding activations, including saved registers and arguments to child functions.

3. A set of active or **live registers** in R_i. These registers might contain variables, control flow information, condition codes, etc. Registers are dictated by the ISA and cannot be changed by the compiler. The compiler does, however, have some flexibility in specifying what values are stored in which registers.

As functions execute, they modify their register state to read and write memory and to perform computations on data. When calling functions, some or all of this register state is saved onto the stack (as dictated by the ABI) – the calling function saves **caller-saved** registers, while the called function saves **callee-saved** registers. Each invoked function allocates space on a thread's stack which also adheres to the architecture's ABI. As functions return back up the call chain, call frames are removed from the stack and register state is restored from its saved format. A state transformation runtime must be able to observe registers and call frames for each activation on a thread's stack, and in particular must know how execution state is mapped onto them for each architecture. The compiler generates metadata describing the register and call frame state at equivalence points within functions.

The state transformation runtime needs to be able to access and understand register state R_i for each activation. A thread's register state is dictated by the ISA and can be grouped into several categories [125, 99]:

- **General-Purpose Registers** – These registers are used for integer and boolean logic operations, as well as addressing memory and control flow. A subset of these may be used for special purposes, e.g., to maintain a return address.
- **Floating-Point/SIMD Registers** – These registers are used for floating-point arithmetic, and are usually combined with ISA-specific SIMD extensions for data parallel computation.
- **Program Counter** – The register containing the address of the next machine instruction to be executed. It usually cannot be accessed like general-purpose registers, but must be changed using control-flow operations (branches, calls, etc.).
- **Stack Pointer (SP)** – The register pointing to the current top of the stack (which is the lowest stack address for architectures that have downward-growing stacks). It can usually be manipulated like general-purpose registers, and may have special semantics for other operations, e.g., on x86 a `call` instruction decrements the stack pointer and writes a return address to the new top-of-stack.
- **Frame Base Pointer (FBP)** – The register pointing to the beginning of the current call frame. It, together with the SP, identifies a function's call frame[3].

The state transformation runtime must be able to traverse call frames on the stack, and thus must have information regarding how to adjust the stack and frame base pointer in order to access a given function activation. Additionally, the ABI dictates which portion of the register state is saved onto the stack (and by whom), meaning the runtime must understand the register save and restore procedure in order to observe the correct register state for each activation.

Much of a thread's execution state is placed in call frames on the stack, in a format created by the compiler (but adhering to the ABI). Figure 3.3 shows a generalized view of a thread's stack of call frames, hereafter referred to as the stack. In this figure, a thread's call stack contains call frames for function `foo`, which has called function `bar`. Because the stack grows downward, `bar`'s call frame is below `foo`'s. Each function call frame is composed of several areas:

- **Return Address** – The machine instruction address at which execution will resume after the current function has finished execution. Upon entering a function from a call instruction, the return address is pushed it onto the stack (or it may be pushed automatically by the call instruction). In Figure 3.3, `bar`'s call frame saves the instruction address at which execution will resume when returning to `foo`.

[3]The FBP register can be used as a general purpose register for call frames which have a statically known size, e.g., those which do not perform operations like `alloca`.

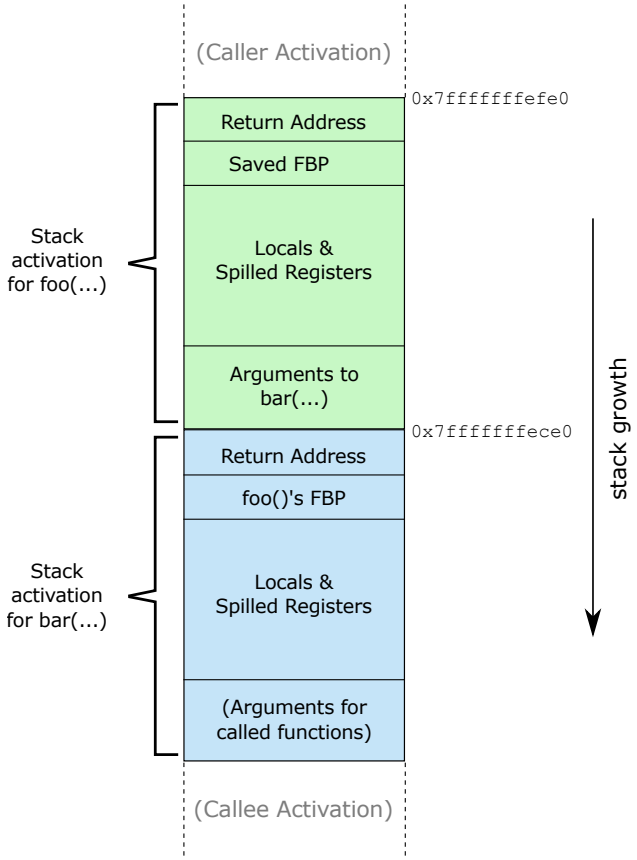


Figure 3.3: Stack frame layout. The stack includes call frames for function `foo(..)`, which calls function `bar(...)`.

- **Saved Frame Base Pointer** – The FBP of the calling function. The old FBP is saved so that the frame of the calling function can be restored after finishing execution of the current function. This is usually saved after the return address on the stack. In Figure 3.3, `bar`'s call frame saves `foo`'s FBP before setting its own FBP.
- **Locals and Spilled Registers** – This portion of the stack frame contains the callee-saved registers, local variables allocated on the stack, and registers that are spilled to the stack by the register allocator. In Figure 3.3, `bar` saves a subset of `foo`'s registers as dictated by the ABI before allocating local variables and spill slots.
- **Argument Area** – Storage on the stack to be populated with arguments to be passed to called functions. `foo`'s call frame has an area for arguments to `bar`, which in turn has an argument area for any functions it may call.

The state transformation runtime must be able to locate call frames for each function activation on the stack. It must also be able to find each of these areas of the call frame so

that they can be transformed between architecture-specific formats. The compiler generates metadata describing the call frame layout for each function in the application, and how each function can be unwound from the stack.

3.3 Expectations of the Compiler and Runtime

At equivalence points, a state transformation runtime is given the register set R_i. By reading the stack pointer register, the runtime can discover the stack S_i of a thread. The state transformation runtime must be able to do the following:

1. Given a program location, i.e., an instruction address in a program counter register, find the function encapsulating that address.
2. Given a stack pointer, frame base pointer and location within a function, locate each of the call frame areas identified above.
3. Given a call frame and register set, know which portions of the call frame and register set contain live values so that the runtime may copy them to the appropriate location within a transformed call frame and register set.
4. Given a relocated variable in either R_i or S_i, reify references to the variable in order to reflect its new relocation.
5. Given a call frame and register set, be able to unwind the call frame from the stack in order to access the frame of the calling function.
6. Given a return address in code compiled for one architecture, find the corresponding return address in the code generated for another architecture.

The compiler is responsible for generating metadata providing all of this information, which it injects into the binary for the runtime. Note that the compiler does not need to synthesize this metadata for all instruction addresses in an application, but only at equivalence points. Our prototype uses function call sites as equivalence points, as they satisfy all requirements listed in Section 3.1.1. Thus, transformation metadata is only needed at function call sites – by definition the stack is composed of function activations for functions that are paused at a call site and will resume when the child function returns. The only activation which is not paused at a function call site is the outermost activation, i.e., the activation of the currently executing function. The state transformation runtime implements a special function which carefully handles bootstrapping and initiating transformation, allowing the runtime to begin transformation at a specific known function call site. Thus threads only need to call this special function to begin the process.

The compiler, described in Chapter 4, generates the state transformation metadata needed at runtime to convert R_i and S_i between ISA-specific formats. Additionally, the linker is directed to lay out P_C, P_D and L_i in a common format to avoid transformation costs. Finally, a state transformation runtime (described in Chapter 5) applies the compiler-directed transformation when threads migrate between processor islands.

Chapter 4

Popcorn Compiler Toolchain

The Popcorn compiler toolchain is responsible for preparing applications for seamless migration across heterogeneous-ISA architectures. The toolchain generates **multi-ISA binaries**, binaries containing modified data and code sections along with state transformation metadata, built for migration on Popcorn Linux. Multi-ISA binaries lay out data and code in a common format, which Popcorn Linux uses to replicate a shared virtual address space across kernels (and thus, heterogeneous-ISA processors). For execution state that cannot be laid out in a common format due to ISA or performance reasons, the toolchain generates metadata so that a transformation runtime can switch state between ISA-specific formats. Using information from the multi-ISA binary, Popcorn Linux migrates threads of execution between architectures in a replicated environment so that threads see a single system image across all kernel instances.

4.1 Building Multi-ISA Binaries

The Popcorn compiler toolchain builds multi-ISA binaries by compiling the application source for each ISA available for execution in the system. The toolchain uses a modified LLVM [160] as the compiler and a modified GNU gold [86] as the linker. The toolchain also uses several custom-built tools for post-processing binaries in preparation for state transformation. Figure 4.1 shows an overview of how application source code flows through the toolchain to produce a multi-ISA binary. Different phases of compilation are encapsulated in boxes, with Popcorn-specific additions listed inside.

Application binaries are built through a standard compilation procedure augmented with several additional steps. The source is first parsed into an ISA-agnostic intermediate representation (IR) by Clang, the C-language frontend for LLVM. The IR is analyzed and optimized, then is compiled once for each ISA in the system using an ISA-specific back-end. After linking, which generates a binary per ISA, post-processing modifies the binaries by

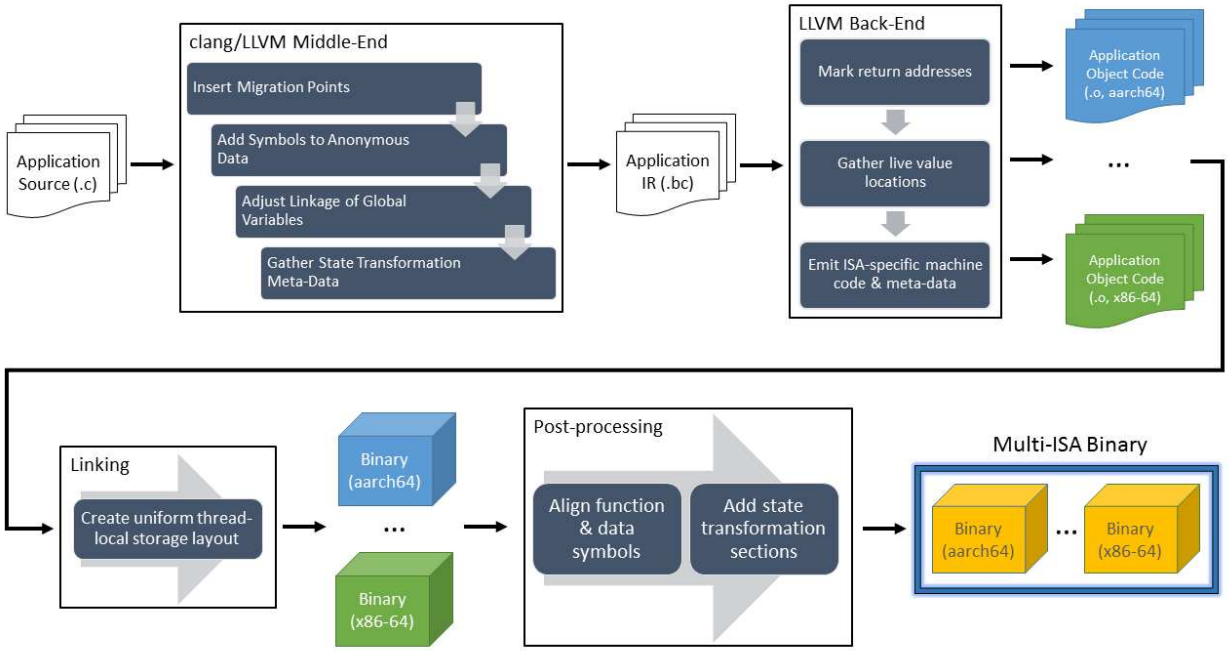


Figure 4.1: Popcorn compiler toolchain

aligning function and data symbols at identical virtual addresses across all binaries. Additionally, post-processing adds and organizes state transformation metadata. At this point the multi-ISA binary has been built and is ready for execution migration across kernels in Popcorn Linux.

There are many custom analyses and transformations added to the compilation process in order to build multi-ISA binaries:

- **IR Modification** (LLVM middle-end) – Clang generates LLVM bitcode, an intermediate representation of lowered source code in single-static assignment (SSA) form [65]. Popcorn's compiler modifies the IR by inserting migration points at the beginning and end of functions (Section 4.2). Several passes adjust data linkage in preparation for alignment. Finally, an analysis pass and an instrumentation pass find and record live values at all potential transformation sites throughout the IR in preparation for runtime state transformation (Section 4.3).

- **Back-end Analysis** (LLVM back-end) – Several back-end analyses are run which mark return addresses from function calls, gather live value locations in function activations, and generate metadata needed for state transformation (Section 4.4).

- **Linking** – Thread-local storage (TLS) layout is modified to conform to a single layout across all generated binaries. The current implementation forces all TLS to be identical

to the ARMv8 layout.

- **Alignment** (post-processing) – After generating a binary per ISA, a linking tool gathers symbol location and size information in order to align data and function symbols at identical addresses across all binaries. Symbols are placed in an identical order in all binaries (space is added for symbols that only exist in one binary). Data symbols do not need to be padded, because the architectures used in our prototype have identical data sizes and alignments for primitive data types. Function symbols do require padding, however, because the machine code implementing a function may be different sizes for different ISAs [24].
- **State Transformation Metadata** (post-processing) The binaries are post-processed to set up the state transformation metadata needed to transform execution state at runtime (Section 4.5).

The Popcorn compiler currently supports applications written in C and C++. The toolchain builds multi-ISA binaries for POSIX- and Popcorn-compliant programs, meaning that all traditional POSIX interfaces supported by Popcorn Linux, such as the standard C library and pthreads, are supported by the compiler. Additionally, the compiler has almost no restrictions on program optimization, meaning applications can be aggressively optimized for each architecture in the system (see Section 4.3). There are currently several limitations – the current prototype only supports 64-bit architectures whose primitive data types have both the same sizes and alignments. The toolchain does not support applications that use inline assembly, as analyses in the middle-end do not understand machine-code level semantics. Architecture-specific features such as SIMD extensions or language level features that have architecture-specific implementations such as `setjmp/longjmp` and variable-argument functions are not supported. Functions that have dynamically sized frames (e.g., functions that use `alloca` or variable-length arrays [101]) are not supported. Finally, applications cannot migrate during library code execution (e.g., during calls to the C standard library).

Other works focus on aligning global state to replicate the same virtual address space across kernel instances [27, 24, 134, 26]. This dissertation analyzes and solves the problem of transforming execution state between ISA-specific formats to enable seamless thread migration at runtime. Section 4.3 describes analyses and transformation over the application's IR needed to capture state transformation metadata. Section 4.4 describes back-end changes for converting IR-level metadata into machine code metadata. Section 4.5 describes the final post-processing step which adds state transformation metadata to the multi-ISA binary for a state transformation runtime.

4.2 Inserting Migration Points

Because threads cannot migrate between heterogeneous-ISA architectures at arbitrary locations, threads must check to see if the scheduler has requested a migration. **Migration points** are inserted by the compiler at the beginning and end of functions, which corresponds to the equivalence point at the call site of the function. Recall from Section 3.1.1 that there are three properties that must be satisfied for a program location to be an equivalence point. Function call sites satisfy all three properties:

1. **Identical number and type of live variables** – this is satisfied by construction. LLVM compiles the application for each ISA using the same LLVM bitcode. Architecture-specific back-ends are tasked with allocating storage for the live values described by the IR. The individual back-ends can introduce new per-architecture live values, although higher optimization levels tend to remove these.

2. **Live values must be in memory** – this requirement is satisfied by the function call procedure. In order for a live value in a register to be preserved across a function call, it must be stored in a callee-saved register. This means that if the calling function uses the register, it is required by the ABI to spill the register into the callee-saved register section of its call frame. Otherwise, the live value remains untouched in the register while the called function executes. Therefore, all live values are either stored in memory or are live in the register set of the outermost function activation. To bootstrap transformation, the state transformation runtime stores a snapshot of the current register set, thus capturing all live values in memory.

3. **Semantically-equivalent computation** – this is again satisfied by construction. The back-ends generate machine-specific code which corresponds to a single set of IR. The back-ends may perform architecture-specific optimization, including both basic-block level and function-level code movement. However, code movement is prevented across function call sites as described in Section 4.3, meaning that computation completed up until a function call site is semantically-equivalent across all versions of the machine code.

Migration points are implemented as a call-out to a migration library. The library contains APIs for querying information about nodes participating in the Popcorn single system image such as architecture and number of CPUs, APIs for querying information about the current thread such as the node on which it is currently executing and whether a migration has been requested, and APIs to perform thread migration. At application startup, the main thread reads information about all nodes participating in the single system image. When the scheduler requests that a thread migrate, it writes the node ID of the requested destination node inside the thread's descriptor in kernel space via a migration request system call. At migration points, threads check whether a node ID has been set via a query system call.

If it has been set, the thread queries the ISA of the destination node and begins the state transformation and migration process described in Chapter 5.

4.3 Instrumenting the IR of the Application

The Popcorn compiler toolchain is responsible for capturing execution state information at **rewriting sites**, i.e., function call sites at which stack transformation may occur, during the compilation process. The toolchain must generate metadata describing the makeup of generated function activations, including instruction addresses and locations of live values at rewriting sites. The toolchain collects this information while the application is in an intermediate representation in order to determine program locations and liveness information in an architecture-agnostic fashion. Additionally, recording liveness information in the middle-end captures IR-level semantic information (such as data type, size, etc.), which is stripped away when lowering the IR to machine code. An LLVM pass was built that implemented the algorithm presented by Brandner et al. [39], an optimized version of the standard data-flow analysis algorithm for SSA-form programs, for the Popcorn compiler toolchain. Another pass was built which instruments the application IR to capture program and live value locations using the results from this liveness analysis.

The transformation pass instruments the IR with stack map intrinsics [161]. Stack map intrinsics appear as function calls in the application IR with a set of live values as function arguments. As the IR is lowered to machine code, stack maps record function activation information at the stack map instruction's location. Stack maps are inserted into the IR at rewriting sites – in our prototype, at function call sites. As they are lowered by the back-end, stack maps are converted into metadata stored in an extra ELF section in the generated object code. Each stack map intrinsic generates a record in the ELF section and is composed of several fields:

- **ID** – Each stack map has a unique 64-bit ID, allowing the state transformation runtime to find matching stack map records for each ISA-specific version of the generated machine code.
- **Function Record Index** – Stack maps mark specific program locations inside of functions. Multiple stack maps may map to the same function, thus the function's metadata can be shared among them. Each stack map contains an index into the function record metadata referencing the function containing the stack map.
- **Program Location** – The stack map record contains a machine instruction offset from the beginning of the function, which denotes the stack map's program location. This is used to locate the return address for function calls when transforming the stack.

- **Location Records** – The record encodes the locations of live values specified in the stack map intrinsic in the IR. Values can be stored on the stack (as an offset from the frame base pointer), in a register (encoded using architecture-specific DWARF register numbers), or they may be a constant not stored anywhere. The record also contains information about the live value's type, described in more detail in Section 4.4.

Stack maps prevent frame pointer elimination optimization because they use offsets from the frame pointer to locate stack-allocated variables. This is only an implementation artifact, however, and not a design requirement. Additionally, stack maps prevent code movement around the intrinsic's location in the LLVM back-end, which ensures that all three properties of equivalence points are satisfied.

Figure 4.2 shows an example of LLVM bitcode for a simple basic block:

```
bb1:
  %mydata = alloca i32, align 4
  store i32 5, i32* %mydata, align 4
  ...
  %call = call void (...) @do_compute()
  ...
  %res = load i32, i32* %mydata, align 4
  ret i32 %res
```

Figure 4.2: Uninstrumented LLVM bitcode

In this basic block, integer `mydata` is allocated on the stack and is initialized to 5. Sometime later in the basic block, the function `do_compute` is called. At the end of the block, `mydata` is loaded into integer `res` and returned as the result of the function. Figure 4.3 shows the result of running Popcorn's liveness analysis and instrumentation pass over the basic block:

```
bb1:
  %mydata = alloca i32, align 4
  store i32 5, i32* %mydata, align 4
  ...
  %call = call void (...) @do_compute()
  call void (i64, i32, ...) @llvm.experimental.stackmap(i64 0, i32 0, i32* %mydata)
  ...
  %res = load i32, i32* %mydata, align 4
  ret i32 %res
```

Figure 4.3: Instrumented LLVM bitcode

The transformation pass places a stack map intrinsic directly after the call to `do_compute` to capture transformation metadata at the rewriting site. The stack map has an ID of 0 (the first argument), which uniquely identifies this function call site across all per-ISA versions of the application. Liveness analysis determines that `mydata` is live across the call to `do_compute`, so the transformation pass adds the value as an argument to the stack map. Stack map 0's instruction address and `mydata`'s storage location will be recorded after the basic block has been lowered to machine code, after instruction scheduling and register allocation.

4.4 Augmenting Compiler Backend Analyses

The application IR is lowered to machine code for each target ISA in the system on which Popcorn Linux runs. As the IR is transformed, special handling converts stack map intrinsics into records which contain concrete details about the rewriting site, such as program location and live value locations within function activations. Several additional analyses were integrated into the LLVM back-end to add pieces of information not visible in the middle end. LLVM implements IR lowering to machine code using a set of target-independent analyses and transforms, meaning the Popcorn compiler's modifications are available for all targets supported by LLVM. Unlike previous works [196, 197, 70], the Popcorn compiler toolchain does not change the size or layout of call frames to be compatible across architectures. The toolchain minimizes the number of changes to the architecture-specific portions of the back-end so that applications can take advantage of extensive architecture-specific compiler optimizations and be easily ported to any architecture that LLVM supports.

4.4.1 Program Location

Stack maps are inserted into the IR directly after function calls to record return addresses from those function calls. LLVM IR encapsulates the entire function call procedure into a single IR instruction, which is expanded during instruction selection and register allocation to adhere to the ISA's function call procedure defined in the ABI. Because this procedure is not visible in the middle-end, it is not possible to directly capture a call's return address by adding stack map intrinsics. Instead, in the back-end stack map intrinsics are matched to the appropriate function call site. This allows the stack map machinery to encode the return address irrespective of the architecture-specific function call procedure.

4.4.2 Live Value Locations

Stack map intrinsics were designed for online compilers, and as such were designed so that a set of values could be captured at the intrinsic call site and execution could be transferred to an optimized version of the function (i.e., moving from an interpreter to compiled machine code). Stack maps capture the function activation state specified as arguments to the intrinsic – they do not capture the entire function activation itself. An artifact of this design is that a value may be live in several locations (e.g., in a register and backed by a slot in the call frame) but the stack map mechanism only records one of these locations. For example, consider the AArch64 assembly in Listing 4.1:

```
0x410000: ldr x20, [sp,#32] ; stack slot 4
0x410004: add x0, xz, x20
0x410008: mul x0, x0, 2
0x41000c: bl do_compute
          <stack map records metadata here>
0x410010: add x20, x0, x21
```

Listing 4.1: Live values across call to `do_compute` in AArch64 machine code. The value is live in stack slot 4 and register `x20`.

In this assembly, a live value is loaded from stack slot 4 into register `x20`, which is a callee-saved register for AArch64. The value is then used to compute an argument for the call to `do_compute`. After returning from the function call, `x20` is overwritten using the return value from `do_compute` and another callee-saved register `x21`. The stack map intrinsic inserted after the call requests that the back-end record the location of this live value at `do_compute`'s return address. The back-end only records that the value is stored in register `x20`, although it is also stored in stack slot 4. Without additional analysis, the metadata at this rewriting site is incomplete, meaning that the transformation runtime will not be able to fully rewrite the activation and the application will likely fail after migration. Note that in addition to live values being in both a register and call frame slot, values may also be live in multiple registers depending on the types of optimizations applied. Live values stored in multiple locations are more prevalent on RISC architectures because live values must be loaded from and stored to memory in order to do computation on them. The compiler tries to keep as many values as possible in registers so that it does not have to continually re-materialize them. However, this re-materialization behavior also arises on CISC architectures, depending on the results of register allocation.

The Popcorn compiler back-end implements liveness range checking for live values in stack maps to determine if they are stored in multiple locations. This analysis uses liveness ranges for registers and stack slots which are already calculated by LLVM for register allocation. At this point in the compilation, the application has been lowered to another form of IR which is close to machine code. The IR is still in pseudo-SSA form, however, and values have use-def chains which point to instructions where the value is defined and used.

After register allocation, the definitions of all live values stored in registers are checked[1]. If the register is defined by a copy (e.g., a load from a stack slot or a copy from another register), the liveness range of the source of the copy is searched. If the source value's live range overlaps with the stack map, then the source is determined to be a duplicate location for the live value and extra metadata is added to account for the duplicate. Similarly, if

[1]It is not necessary to check live values stored in stack slots, because if they are marked as stored in the call frame by the stack map machinery then they are never also in a register – either they are required to be on the stack or the register allocator decided that they are to be spilled to the stack. Duplicate locations only arise when promoting values from stack slots to registers or when copying values between registers.

the register in the stack map is used as the source of a copy and the copy location is live across the stack map, metadata is added to account for the duplicate location. This process is repeated exhaustively up and down the use-def chain to find all duplicate locations of the live value.

4.4.3 Live Value Semantic Information

Stack maps were designed so that execution could be transferred to an optimized version of a function on the same architecture. Because of this, the live value information needed to jump to optimized execution is simpler than what is required by the state transformation runtime for Popcorn Linux. Stack maps encode the following information about live values and their locations:

- **Storage Type** – where the value is stored, i.e., a register, a stack slot or if it is a constant, nowhere.
- **Register Number** – if the value is stored in a register, which register it is stored in. Stack maps use DWARF register numbers as specified by each ISA's ABI.
- **Offset from frame base pointer** – if the value is stored on the stack, the offset from the frame base pointer where the value is stored. The frame base pointer is ISA-specific, e.g., `rbp` on x86-64 or `x29` on AArch64.
- **Constant** – if the value is a constant, the stack map will directly encode the value. Note that our implementation of liveness analysis ignores constant values because they are, in general, materialized right before use in the machine code rather than being held in storage.

The following fields were added to location records using extra semantic information gathered from the LLVM IR for the live value in order to provide a complete state transformation:

- **Pointer** – flag indicating if the value is a pointer. The state transformation runtime requires special handling for pointers to the stack (Section 5.2.3), although pointers to global data and functions are valid because of symbol alignment.
- **Alloca** – variables allocated to the stack are instantiated using the `alloca` IR intrinsic in LLVM bitcode[2]. This flag indicates that the live value is allocated to the stack.

[2]LLVM's alloca is semantically different from the C language `alloca` API, although the latter is implemented using the former in LLVM bitcode.

- **Size of Stack Variables** – the stack map fields described above only indicate how to locate the beginning of a stack-allocated variable, but do not specify their size. If the value is allocated to the stack, this field encodes how large the allocated data is in the call frame.
- **Duplicate** – flag indicating if this location record is a duplicate, meaning that it describes another location for the same live variable (as determined by the analysis described in Section 4.4.2).
- **Temporary** – some IR-level live values may be either re-materialized when needed or held in a register (e.g., reference to a stack slot) depending on the register allocator's decisions for each ISA. This flag is set if the back-end materialized a value only to satisfy the stack map.

The application IR is converted to machine code, which is emitted into object files. Stack maps records are added to a special section within the object file, but are not yet in a suitable format for state transformation.

4.4.4 Architecture-Specific Live Values

Depending on the results of the register allocator, the back-end may create extra architecture-specific live values, e.g., references to global symbols or constant data. These decisions are specific to each ISA – for example, materializing references to global symbols may take multiple instructions on AArch64 and therefore the back-end may save a reference in a register, in contrast to x86-64 which can encode large addresses into instructions. Rather than prevent this behavior, the Popcorn compiler captures these values in order to allow the back-end to optimize the generated machine code as much as possible. The compiler produces metadata describing both the location of the architecture-specific live values and how to re-generate them, e.g., constant data, addresses or simple math operations. It is important to note that architecture-specific live values are statically calculable, i.e., the compiler knows how to produce the values at compile time. This allows the compiler to capture constants, references to global data and even references to stack data – although the particular address of a given stack slot is not known until runtime, the method of calculating the address (e.g., add offset to frame base pointer) is known at compile time and thus the compiler can emit metadata describing how to recreate the value at runtime.

4.5 Generating State Transformation Metadata

At this point, the LLVM back-end has generated object code and added stack map metadata to the binaries. Additionally, the alignment tool has aligned code and data symbols across

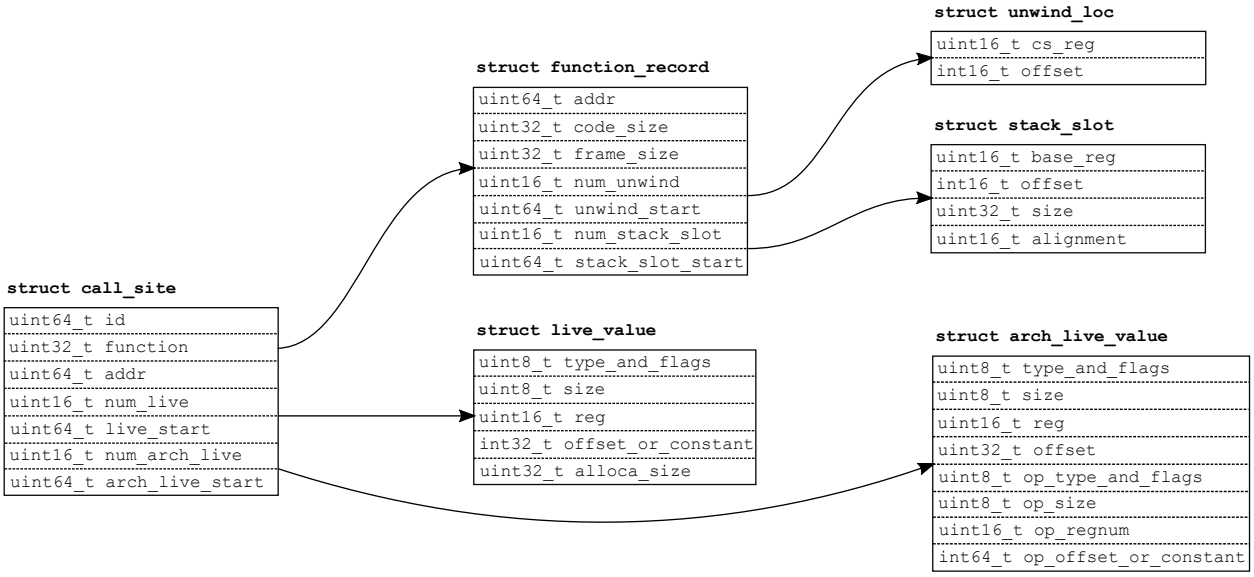


Figure 4.4: Metadata emitted by the compiler. Each type of structure (e.g., `call_site`, `function_record`) is contained in its own section.

each of the generated versions of the binary. The final step in the toolchain is to convert the emitted stack map records into the format the state transformation runtime uses to rewrite the stack. There are several downsides to the default format emitted by LLVM:

- **Stack map records are variable-sized** – there are a variable number of live value location records per stack map record. This means searching through stack map records is a sequential process because it requires jumping across differing numbers of location records per stack map.
- **There are multiple stack map sections per binary** – LLVM generates a stack map section per source file. Stack map records are not combined during linking, but are rather appended one after another into a larger ELF section. This compounds the problem of searching for records, as searching for a stack map from a particular source file requires first finding the beginning of the stack map records for that file and then searching sequentially through the records.

A final post-processing step reorganizes stack map records into a format amenable for efficient lookups of stack maps and location records of live values at the rewriting site. A post-processing tool parses the LLVM-generated stack map sections and breaks the metadata out into multiple extra sections added to the multi-ISA binary. Each of the extra sections contains equal-sized records, meaning the transformation runtime can directly jump to a record given an offset or can use a binary search for efficient look-ups. Figure 4.4 illustrates all of the metadata emitted by the compiler.

The first two sections added to the binary contain stack map/call site records (`struct call_site`). The first two sections provide stack map records sorted by ID and program location, respectively. These sections provide a dictionary lookup between stack map IDs and program locations, which is used by the state transformation runtime to look up and correlate call stack map records for the source and destination versions of the activation. The call site records contain the following fields:

1. ID of the the stack map/rewriting site.
2. Index into the function record section referencing the metadata for the function enclosing the call site.
3. Program location, i.e., return address of the function call defining the rewriting site.
4. Number of live values at the rewriting site.
5. Offset into the live value location record section
6. Number of architecture-specific live values
7. Offset into the architecture-specific live value record sections

In order to correlate stack map records between metadata generated for each ISA, the transformation runtime uses a return address on the source stack to look up its stack map record, which is tagged with a unique ID. The transformation runtime next looks up the destination stack map record using the unique ID. The runtime then uses the source and destination stack map records to locate live variables and to correlate return addresses found on the source stack to the appropriate return addresses for the destination ISA.

The function field references a function record (`struct function_record`) which contains the following information:

1. Address of the function
2. Size of the code comprising the function's body
3. Size of the function's on-stack activation size
4. Number of locations storing callee-saved registers that must be restored during stack unwinding
5. Locations of callee-saved register locations, including the offset from the frame-base pointer and the register stored at that location
6. Number of stack slots[1]

[1]Only used by Chameleon to update references to stack slots in the code – see Chapter 12

7. Stack slot records, each of which is denoted by its location (base register plus offset), size and alignment[1]

The live values section added to the multi-ISA binary (containing `struct live_value` and `struct arch_live_value` entries) contains live value location records for all stack maps. The transformation runtime finds live variables at a call site by reading the offset and the number of location records from the stack map record and pulling the records from relevant sections. Figure 4.4 shows the location record fields:

1. Type of live value and the aforementioned flags providing extra semantic information
2. Size of the value
3. The register either containing the value or the register used as the base to form a reference to the containing stack slot
4. A displacement if the value is in a stack slot
5. Size of on-stack slot if the value is an alloca

Finally, the architecture-specific live value metadata contains the same fields as normal live values, but contains extra fields describing how the value must be created:

1. Type of materialized value and extra semantic flags
2. Size of materialized value
3. Register used in materializing value (for slot reference types)
4. Offset or constant used in materialization operation

At this point, the compiler has finished building the multi-ISA binary, which is ready for execution on Popcorn Linux.

[1] Only used by Chameleon to update references to stack slots in the code – see Chapter 12

Chapter 5

State Transformation Runtime

At runtime, applications compiled by the Popcorn compiler toolchain execute as normal on a single architecture until the Popcorn Linux scheduler requests a migration. At that point, a state transformation runtime built into the multi-ISA binary (hereafter referred to as **the runtime**) co-opts execution in user-space and transforms thread state into the format required by the destination ISA. After transformation, threads invoke a Popcorn Linux-specific system call which migrates the threads to the destination ISA. Special handling is required to set up for migration and to bootstrap execution on the new architecture after migration.

The runtime is built to minimize end-to-end state transformation latency as a primary design goal so that the scheduler can react to changing workload conditions without significant delay. The runtime is implemented in a standalone library linked into multi-ISA binaries. The compiler hooks applications into the library by inserting migration points, which check for migration requests and perform state transformation. The runtime is written in C in order to aggressively optimize its performance and so that it does not drag external dependencies (e.g., the C++ standard library) into applications.

The runtime operates at the granularity of threads of execution, which enables the OS scheduler to migrate individual threads of an application. Threads execute normally, checking at migration points to see if the scheduler has requested a migration. When the scheduler requests a migration, the thread takes a snapshot of its register state R_i and calls into the runtime. The runtime uses the stack pointer from R_i to attach to the thread's stack S_i and convert all live function activations from the source ISA format to the destination ISA format. After transformation, the thread makes a system call into the Popcorn Linux kernel which migrates it to the new architecture using the thread migration service. One of the arguments to the system call is the transformed register state for the outermost activation, which the destination kernel uses to set the destination thread's initial register state. The kernel sets the transformed register state and the thread returns back into user-space inside of the runtime. The runtime performs a few housekeeping steps, and the thread resumes

normal application execution.

When transforming the thread's execution state, the runtime divides the thread's stack into two halves – one half which the thread is currently using, and another half for transformation[1]. The runtime transforms the thread's execution state in its entirety – the entire stack is rewritten from the current ISA's format to the destination ISA's format. Register state, including state for the current function activation and all state saved on the stack as part of the register save procedure, is transformed along the way. In the current implementation, threads transform their own stacks before migrations. Threads takes a snapshot of their own registers and call the stack transformation runtime, which allows the runtime to attach to and traverse the thread's stack. Then, the runtime rebuilds the stack in the other half of rewriting memory.

The runtime operates in user-space to provide a cleaner separation of responsibilities. Pushing state transformation into the kernel requires integrating application-specific logic into kernel space (including parsing application-specific transformation metadata), even though the kernel should only be an arbiter of resources. By keeping state transformation in user-space, applications are responsible for their own state and there is less complexity in the kernel, which ultimately makes the kernel more robust to faulty or malicious applications. The downside of transformation in user-space is that rewriting is not opaque to the application – state transformation is visible to application threads. Nevertheless, the runtime performs state transformation in user-space due to the aforementioned benefits. The state transformation runtime differs from previous works [70, 197] in that the runtime reconstructs the destination stack from the source stack in a separate region of memory, whereas their implementation does in-place modification. This is an artifact of the design decision not to unify call frame layout in the compiler. Because of this design decision, the runtime must handle reifying references to stack elements during transformation.

Section 5.1 describes how the runtime prepares for transformation at application startup by loading in metadata and preparing stack pages. Section 5.2 describes the state transformation process, which rewrites the thread's registers and stack in their entirety. Finally, Section 5.3 describes how a thread invokes and resumes execution after the OS migrates it to another ISA. Section 5.4 describes how developers can debug applications in a heterogeneous-ISA environment.

5.1 Preparing for Transformation at Application Startup

In order to reduce state transformation latency, the runtime loads rewriting metadata into memory when the application begins execution. At startup, the main thread creates **state transformation descriptors** for all ISAs in the system. These descriptors contain the following ISA-specific metadata needed for transformation:

[1]The default stack size on Linux systems is 8MB, meaning the runtime divides it into two 4MB regions.

- **ISA ID** – a numeric ID uniquely identifying the architecture, as defined by the ELF standard.
- **Pointer Size** – size of pointers as defined by the ISA's ABI. This is always 8 bytes (64-bit) in the current prototype.
- **Register Operations** – a set of function pointers which implement register access operations for the ISA. All register access operations in the runtime use an architecture-agnostic interface, and architectures provide ISA-specific implementations via function pointers[2].
- **ISA Properties** – a set of properties which describe ISA-specific register behavior (register size, which registers are callee-saved) and stack properties (stack pointer alignment).
- **State Transformation Metadata** – all of the metadata emitted by the compiler as described in Section 4.5 required for converting the stack between ISA-specific formats. This includes function records, unwinding location records, call site records (sorted by call site ID and address, respectively), live value location records and architecture-specific live value location records for call sites.

At startup, the application creates descriptors for all ISAs in the platform by reading the metadata added to the multi-ISA binary by the Popcorn compiler toolchain. This information is subsequently available for threads to perform transformation when the scheduler requests migrations. All of the metadata contained in the binary read into memory as read-only, meaning threads can concurrently access the information and transform their stacks in parallel.

For implementation on Linux, the runtime must also prepare the main thread's stack for transformation due to how Linux handles stack memory growth. In Linux, the main thread is given a system-defined stack size on application startup (usually 8MB). However, this memory is allocated on demand by observing page faults as the stack grows. When stack growth causes a page fault, Linux checks to see if the access is on the same page as or a page adjacent to the current stack pointer. If so, Linux maps the new stack page into the application's page table and returns to user-space to continue normal execution. However, if the stack access is not close to the stack pointer, Linux raises a segmentation fault and ends the application. Because the runtime may be rewriting to a part of the stack not adjacent to the current stack pages (due to splitting the stack in half), the runtime forces Linux to pre-allocate the entire stack area by moving the stack pointer to the bottom of the stack region, performing a memory access at the stack pointer, and resetting the stack pointer to its original value. The runtime does not need to force Linux to pre-allocate stack pages for

[2]Essentially, a C version of object-oriented programming where a base class defines the register access API and child classes implement the API for each ISA.

threads forked by the threading library, as the library allocates stack memory by using `mmap` or `malloc`, both of which sidestep this issue.

5.2 Transformation

Application threads execute normally, checking to see if the scheduler has requested a migration at compiler-inserted migration points (Section 4.2). When a thread sees that the scheduler has requested a migration, the thread stores a copy of all of its register state into memory and calls into the transformation runtime, operating on a snapshot of the thread's registers at the migration point and therefore its stack state up until the migration point.

First, the thread determines which half of the stack it is currently using and computes the bounds for the other half. It then passes the snapshot of the register set, the stack bounds for the two halves of the stack, and the rewriting handles for the current and destination ISAs to the core of the runtime. The runtime begins by allocating **rewriting contexts** for the thread's execution state on the current and destination ISA. Rewriting context store information about the thread's current execution, including the following:

- **Stack Bounds** – the beginning and end of the stack.
- **Register Set** – register set for the outermost activation, i.e., the current activation. For the thread's current execution state, this is the snapshot taken at the migration point. Another register set will be populated for the transformed execution state, which will be used by the kernel to initialize the thread when it resumes execution on the destination architecture.
- **Function Activations** – metadata about the function activations in the execution state, including call site information, call frame bounds, current register state and frame unwinding information.
- **Stack Pointers** – a list of pointers to the stack that have yet to be resolved (Section 5.2.3).
- **Memory Pools** – pools of memory needed for per-activation data, an optimization to reduce the number of memory allocation calls in the runtime similar to a slab allocator. Because the runtime does not know for which ISA the context will be used, it does not know which registers require unwinding per activation. Rather than dynamically allocating a buffer for this information as activations are discovered, the runtime allocates a single chunk of memory and sets a pointer into it for each activation.
- **State Transformation Descriptor** – ISA-specific version of the state transformation descriptor which contains transformation metadata.

After initializing contexts for the current and transformed execution state, the runtime begins rewriting activations. There are three components in the transformation process:

1. **Find activations on the current stack in order to determine the size of the transformed stack (Section 5.2.1)** – the thread's current stack is unwound to find which activations are currently active. This information is used to locate call site records for the rewritten stack, which allow the runtime to calculate the size of the rewritten stack.
2. **Transform activations from the current ISA's format to the destination ISA format (Section 5.2.2)** – the runtime transforms one function activation at a time from the source to the destination context, for all live activations.
3. **Fix up pointers to the stack (Section 5.2.3)** – pointers to the stack require special handling, and may not be resolved within a single activation. The runtime keeps track of and fixes up pointers as the pointed-to data is discovered. This component is intertwined with function activation transformation, but is a separate mechanism.

5.2.1 Finding Activations on the Current Stack

Using the call frame unwinding metadata contained in the current ISA's state transformation handle, the runtime unwinds all call frames from the source stack. This lets the runtime cache metadata about the current live activations for the source and destination execution state, but more importantly it lets the runtime calculate the size of the rewritten stack. Algorithm 1 shows pseudo-code for the unwinding procedure.

The runtime begins by initializing the sets of live activations for both the source (i.e., current) and destination (i.e., rewritten) execution state. The outermost activation for the source is added to the set of live activations. The runtime checks if the current activation is the first live activation for the thread, i.e., if it is the activation for the first function executed by the thread. If not, a matching empty activation is created for the destination. The runtime then uses the program counter from the source activation to look up the rewriting site record in the rewriting metadata for the source (Section 4.5). The runtime then uses the ID of the record to find the corresponding rewriting site record for the destination. Note that the runtime cannot simply use the call site address to look up the destination call site record, as the call site may be at a different address in the destination binary due to the size of the machine code generated for each ISA. The rewriting site records are cached in the source and destination rewriting contexts as they are needed when transforming individual activations. The destination stack size is updated using the destination record, which contains the size of the call frame for the function in which it is contained. Finally, the source activation is unwound from the source stack, which sets the source activation to its caller.

Algorithm 1: Algorithm to unwind current stack and calculate size of rewritten stack

Data: Handle for source rewriting metadata H_S, handle for destination rewriting metadata H_D, outermost activation for source a_S

Result: Set of activations for source A_S, set of empty activations for destination A_D, stack size for destination stack S_D

$S_D = 0$;
$A_S = \{a_S\}$;
$A_D = \{\}$;
while *!FirstActivation(a_S)* **do**
    a_D = CreateEmptyActivation();
    $A_D = A_D \cup a_D$;

    $CallSite_{a_S}$ = GetSiteByAddress(H_S, GetPC(a_S));
    $CallSite_{a_D}$ = GetSiteByID(H_D, GetID($CallSite_{a_S}$));

    SetCallSite(a_S, $CallSite_{a_S}$);
    SetCallSite(a_D, $CallSite_{a_D}$);
    $S_D = S_D$ + GetCallFrameSize($CallSite_{A_D}$);

    a_S = UnwindActivationToCaller(a_S);
    $A_S = A_S \cup a_S$;
end
return A_S, A_D, S_D

This process is repeated until reaching the initial activation for the source, which is either a starter function in the standard C library for the main thread, or a thread start function in a threading library such as pthreads. The algorithm returns a set of activations for the source execution state, a set of empty activations for the destination state (which will be filled as described in Sections 5.2.2 and 5.2.3), and the stack size of the destination stack. The runtime then moves on to transforming live function activations.

5.2.2 Transforming Activations

After discovering live activations, the runtime resets to the outermost activation and works up the stack, transforming activations as it goes. In order to fully transform an activation, the runtime must populate a destination activation with the following information:

- **Call Frame Bounds** – the runtime must determine the beginning and end bounds of the activation's call frame on the stack. This consists of setting the canonical frame address (i.e., highest address of a frame for stacks that grow downwards) and stack pointer, which denote the beginning and end of the call frame, respectively.

- **Live Values** – the runtime must copy live values, as gathered by IR and back-end analyses (Sections 4.3 and 4.4) from the source to the destination activation.
- **On-Stack Arguments** – similarly, the runtime must set up arguments that were passed on the stack.
- **Saved FBP** – the runtime must set the saved frame base pointer from the calling activation in the called activation's call frame.
- **Return Address** – the runtime must also set the return address in the current activation's call frame to the rewriting site in the calling function.

In addition to the above pieces of information, the runtime must adhere to the register save procedure by forward propagating values in callee-saved registers to the activations where they have been saved onto the stack. The runtime only needs to handle callee-saved registers – the stack map mechanism automatically handles caller-saved registers as it records where LLVM's register allocator spills them around call sites.

The runtime begins with the outermost activations and works inwards. It steps through all live value location records in the stack map record to find where live values are stored in both the source and destination activations. Figure 5.1 shows an example of the runtime copying live values from the AArch64 to the x86-64 version of a function activation.

The runtime uses stack map records to locate live value location records in the transformation metadata for each binary. The runtime parses a live value's location record for both the source and destination format to find its location, e.g., an offset into the call frame or a particular register. The location record also provides the size of the data, which the runtime uses to copy the value from the source to the destination activation. The runtime also applies the same procedure for any duplicate location records that may exist. The runtime repeats this process for all live values at the rewriting site.

The runtime must take special care to adhere to the ISA-specific register save procedure. Because of this, the runtime keeps track of which callee-saved registers are stored in each function's call frame as frames are unwound from the stack. When the runtime finds a live value in a callee-saved register, it searches down the call chain (i.e., towards the most recently called function) to find the nearest activation which saves the register onto the stack. If the runtime finds an activation that saves the register, it stores the value in the appropriate call frame slot. If none of the called functions save the register, then the value is still live in the thread's register set in the outermost activation and the runtime stores the value in that activation's register set.

It is important to note that it does not matter that each ISA defines a different number of registers (and different numbers of different classes of registers, e.g., general-purpose or floating-point). The Popcorn compiler determines the live values at a given rewriting site in the architecture-agnostic IR. Each architecture-specific back-end is handed the same version

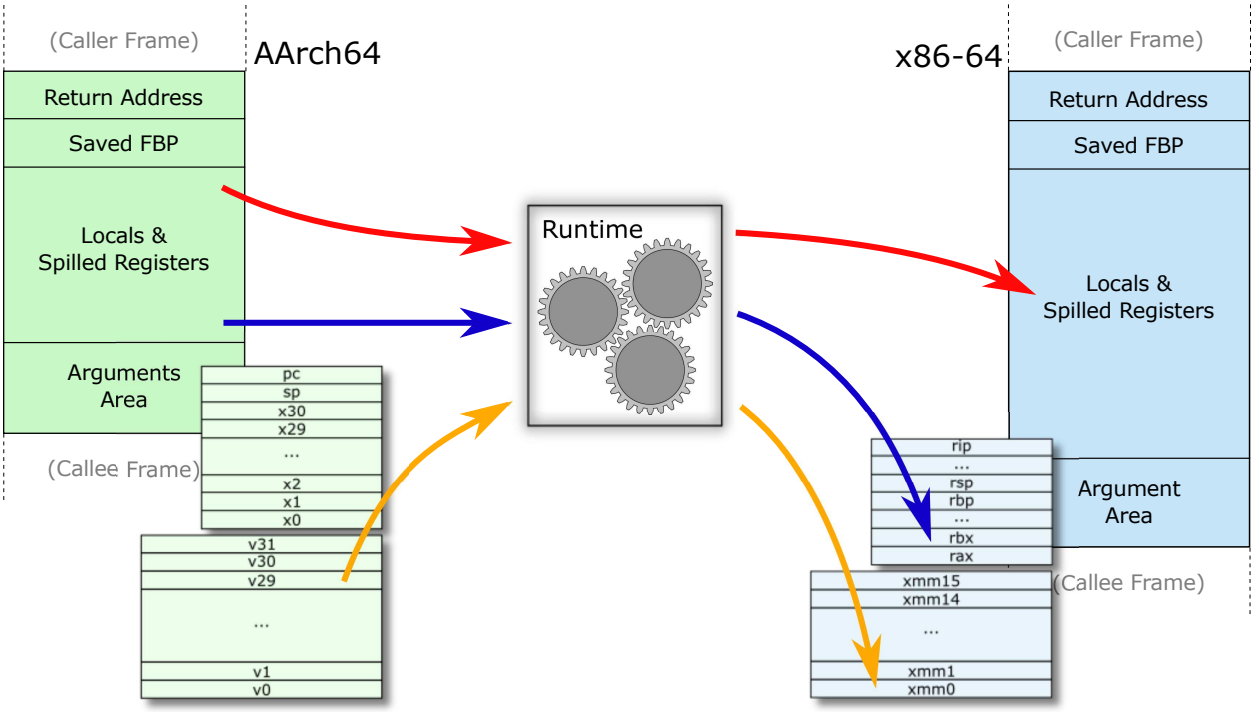


Figure 5.1: An example of the state transformation runtime copying live values between source (AArch64) and destination (x86-64) activations.

of the IR, and therefore the register allocator is responsible for allocating storage for the same set of live values regardless of the ISA. The register allocator is handed a set of parameters describing the numbers and types of registers for each target and makes allocation decisions for each live value. Therefore it is only necessary for the runtime to copy data between these different storage locations in order to rewrite the live values for a function activation. The runtime may copy values between call frames, between registers, from a call frame to a register, or from a register to a call frame. Where values are stored only depends on the register allocator's decisions.

After rewriting the live values, the runtime must set the saved frame base pointer and return address before it can move to the caller's activation. However in order to set this information it must unwind to the caller's frame to read its call frame size and program location. The runtime restores callee-saved registers from the destination activation (they have already been restored on the source as described in Section 5.2.1) to access the caller's activation. It then sets the saved frame base pointer and return address from the caller's stack map record into the callee activation, and sets the current activation to the caller activation.

The runtime repeatedly transforms activations until it reaches the thread's starting function. At this point transformation has finished and the runtime copies out the transformed execution state (including register state and stack pointer) in preparation for migration.

5.2.3 Handling Pointers to the Stack

While transforming activations, the runtime must also take care to transform pointers to stack-allocated data. Pointers to global data do not need to be transformed – symbol alignment and a replicated virtual address space ensure that pointers to global data and heap memory remain valid before and after migration. However, pointers to the stack require special handling. Note that in previous work [70, 197], because they align pointed-to data in call frames across all ISAs, they do not have to reify pointers to the stack. However, Popcorn's state transformation runtime must be able to rewrite pointers to stack-allocated data to instead refer to the data's location post-transformation. This process is named reifying or fixing up pointers to the stack.

The runtime does not have a-priori knowledge about pointers to stack memory, and must discover where these pointers exist during transformation. When copying live values between the source and destination activations, the runtime checks the rewriting metadata to see if the live value is a pointer (which is encoded by the back-end as described in Section 4.4). If the live value is a pointer, the runtime checks to see if it points to stack memory. The runtime then records a fix-up memo which is resolved when it finds the pointed-to data.

Figure 5.2 shows an example of the runtime transforming pointers to the stack from a source activation to a destination activation. As the runtime copies live values from call frame 3 on the source to the destination stack, it finds live value `mydata_ptr` which points to `mydata` in call frame 1. Because the rewriting metadata indicates that `mydata_ptr` is of pointer type, the runtime does a stack bounds check to see if it points to the stack. It concludes that the pointed-to address is within the stack bounds, and because it has not yet begun transforming call frame 1, adds a pointer fix-up memo to the rewriting context. The memo saves metadata about `mydata_ptr`'s location in destination activation 3 and the address to which it points in call frame 1 on the source stack.

The runtime continues transforming activations until it reaches activation 1. When copying `mydata` from the source to destination activation, the runtime observes that `mydata_ptr` points to `mydata`. The runtime first copies `mydata` to the destination activation. It then writes `mydata`'s new address on the transformed stack into the location record stored in the fix-up memo (e.g., call frame 3, slot 6). The fix-up has been handled, so the runtime deletes the fix-up memo and continues transforming activation 1.

The runtime must also handle the case where the pointed-to data is not a scalar, e.g., if `mydata` were an array of integers and `mydata_ptr` pointed to the middle of the array. In this case, the runtime calculates the offset from the beginning of the stack storage location (`mydata_ptr` − &`mydata`) using the saved source stack address and update `mydata_ptr` on the destination stack with the appropriate offset into `mydata`.

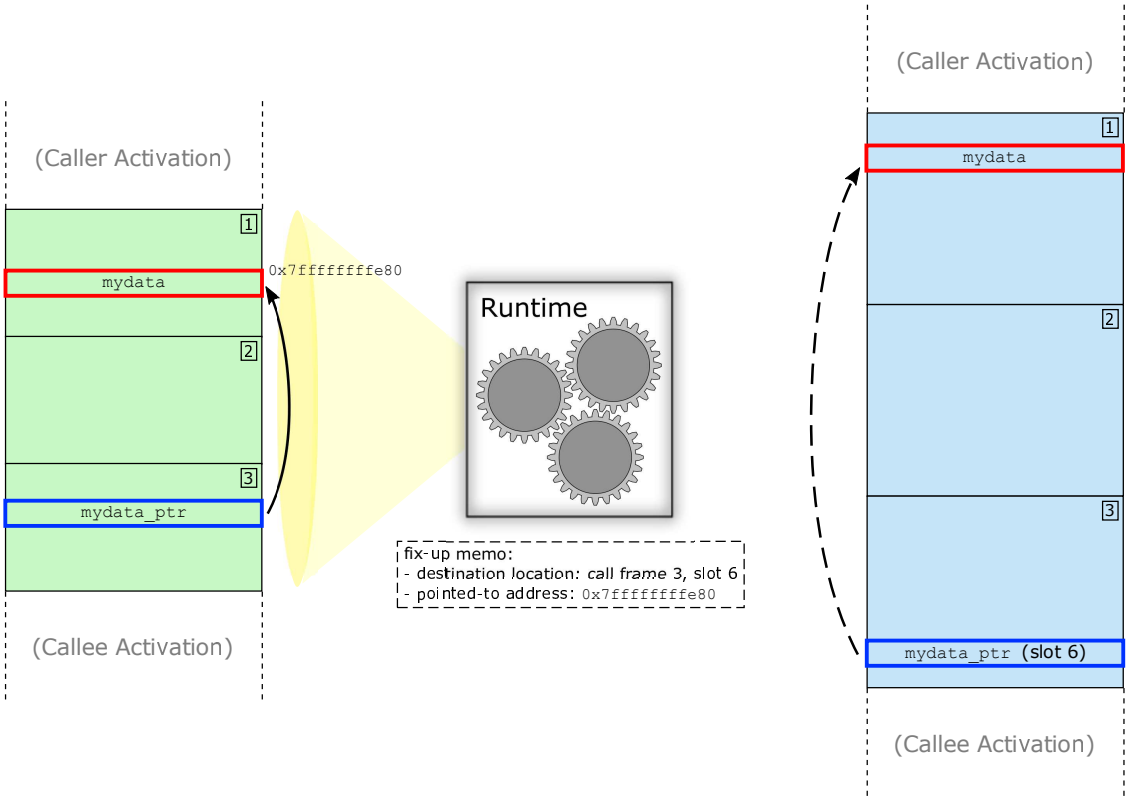


Figure 5.2: Example of the runtime observing and transforming a pointer for the destination activation.

5.3 Migration and Resuming Execution

After the state transformation runtime has finished converting execution state to the destination ISA format, it copies out the transformed register set for the outermost function. The final step of state transformation is translating the thread local storage (TLS) pointer. The TLS pointer is maintained in an ISA-specific register defined by the ABI and points to an ABI-defined location within a thread's TLS memory. The runtime converts the TLS pointer between ABI-specific locations and sets the pointer in the transformed register set. After translating the TLS pointer, the thread returns to the migration point and initiates migration. The thread saves a pointer to the transformed register set into a known location that can be retrieved post-migration, then invokes a Popcorn Linux-specific system call to migrate to another kernel. The thread passes to the kernel a node describing the destination processor island, the program counter at which to resume execution, and a pointer to the transformed register set. The source kernel passes the register set and PC value to the destination kernel, which switches the thread's stack pointer, frame base pointer, PC, and any architecture-specific registers (e.g., on AArch64 the kernel must handle setting up the link register). The destination kernel sets the thread's register state and returns from the system

call to the specified PC, resuming execution at a known-good location on the destination architecture. Before the thread resumes application execution, it initializes any registers that were not able to be set by the kernel (i.e., callee-saved registers, floating point registers) and cleans up the migration data. The thread then returns back to application code as normal on the destination architecture.

5.4 Debugging Cross-ISA Execution

Debugging applications when executing and migrating between heterogeneous-ISA CPUs proceeds similarly to debugging in cache-coherent shared memory systems. The replicated-kernel OS is a process-based kernel, meaning processes (i.e., tasks in Linux) are atomic units of execution. Debuggers use OS interfaces (e.g., `ptrace` on Linux [4]) to inspect and modify processes being debugged. Because of this, Popcorn Linux allows existing debuggers to attach to threads migrating between different CPUs[3]. As described in Section 3.1.1, the OS creates a new thread on the destination node when migrating from the source node. Debuggers can attach to these remote threads on the destination CPUs similarly to their counterparts on the origin.

When debugging a thread before migration, debugging proceeds as normal – debuggers either start the application with the debugger tracing or attach to a running application. When executing the migration system call, the OS puts the thread on the source node to sleep inside the kernel. To the debugger, it looks like the thread is sleeping inside of a blocking system call. On the remote node, however, the Popcorn kernel starts a new thread and returns it to user-space. Developers can attach to the new thread post-migration and continue debugging on the destination node as normal. To help facilitate attaching post-migration, Popcorn's migration library forces the thread to spin indefinitely on a flag before returning to normal execution. After migrating to the new node, the thread will spin until the developer attaches a debugger and clears the flag. The thread can then continue back to normal execution. When the thread migrates back to the original node, the thread on the remote node exits (the debugger will receive an exit message from the kernel) and the original thread on the origin returns back to user-space, waking the debugger. This gives developers a way to follow the thread across nodes to aid in debugging.

Note that re-using the existing process interfaces in the Linux kernel provides the flexibility to reuse many existing tools. For example, developers can use kernel-exposed process counters to profile and tune execution across heterogeneous CPUs.

[3]The developer must specify the binary to be used as the source of debugging information emitted by the compiler; the Popcorn compiler emits a binary per ISA, each of which contains that ISA's debugging information

Chapter 6

Evaluation

In this chapter the costs associated with runtime state transformation and Popcorn Linux's ability to utilize execution migration for different scheduling goals are evaluated. State transformation costs are analyzed using microbenchmarks and real applications from the NAS Parallel Benchmark suite [23]. Additionally, Popcorn Linux's thread migration costs are compared against a Java-based implementation. Finally, Popcorn Linux's ability to use migration execution to achieve higher energy efficiency and energy-delay product is analyzed using different scheduling policies for a datacenter-like workload. In Section 6.1 the experimental setup used in our evaluation is described. In Section 6.2 the cost of the state transformation process is analyzed using a set of microbenchmarks. In Section 6.3 state transformation latencies are analyzed for real applications. In Section 6.4 Popcorn Linux's state transformation and execution migration efficiency is compared versus a Java-based implementation. Finally, in Section 6.5 Popcorn Linux's efficiency is evaluated using several different scheduling policies in a datacenter-like environment.

6.1 Experimental Setup

Table 6.1 shows specifications for the processors used in our evaluation. Our experimental setup consists of an ARM64 machine interconnected to an x86-64 machine via a PCIe bridge. Our setup used an APM883208 X-Gene 1 processor (referred to as "X-Gene") which implements the ARMv8 ISA. The X-Gene was connected to an Intel Xeon E5-1650v2 processor (referred to as "Xeon"), which implements the x86-64 ISA. Because there are no single-chip or single-node heterogeneous-ISA system, our setup approximated a cache-coherent shared memory system by interconnecting the X-Gene and Xeon systems over PCIe. A pair of Dolphin PXH810 PCIe adapters were used, which provide a point-to-point connection between the two machines at 64Gbps bandwidth. Although these adapters do provide transparent shared memory windows across systems, Popcorn Linux instead uses them for message-

passing between kernels. Popcorn Linux was implemented using Linux kernel version 3.12 for both ARM64 and x86-64. The Popcorn compiler was built using LLVM 3.7.1 and GNU gold version 2.27.

	APM X-Gene 1	Intel Xeon E5-1650 v2
Clock Speed	2.4GHz	3.5GHz (3.9GHz boost)
Number of Cores	8	6[1]
Last-level Cache	8MB	12MB
Process Node	40nm	22nm
Thermal Design Power (TDP)	50W	130W
RAM	32GB	16GB

Table 6.1: Specification of Processors in Experimental Setup
[1]There are two hardware threads per core, but hyperthreading was disabled for our experiments.

The on-board sensors and an external system were used to measure instantaneous power consumption for the two machines. The X-Gene has an on-board power monitor which can be queried via I^2C. This sensor provides instantaneous power for the motherboard's power regulator chips. The Xeon implements Intel's Running Average Power Limit (RAPL) [62], which exposes a machine-specific register that keeps a running count of energy consumed. RAPL can be used to measure power for both the core (ALU, FPU, L1 and L2 caches) and the uncore (L3 cache, cache-coherent interconnect, memory controller). An external power-monitoring system was built using a National Instruments 6251 PCIe data-acquisition device (DAQ), which was used to validate the measurements obtained via on-board sensors.

6.2 State Transformation Microbenchmarks

The state transformation runtime described in Chapter 5 is designed to transform a thread's register state R_i and stack S_i with as low latency as possible. Minimizing state transformation latency enables more frequent migrations, allowing the system to adapt application execution to changing system workloads at a finer granularity. The costs associated with state transformation for a thread's registers R_i and stack S_i were evaluated using a set of microbenchmarks.

The two main factors on which state transformation latency depends are the number of live activations for a thread and the number of live values in each of those activations. For each live activation, the runtime must both unwind it from the source stack and reconstruct it on the destination stack. For each live value, the runtime must find its storage location in both the source and destination activation and copy the value between those location. A microbenchmark was designed which varies both of these dimensions to see how they affect the transformation cost.

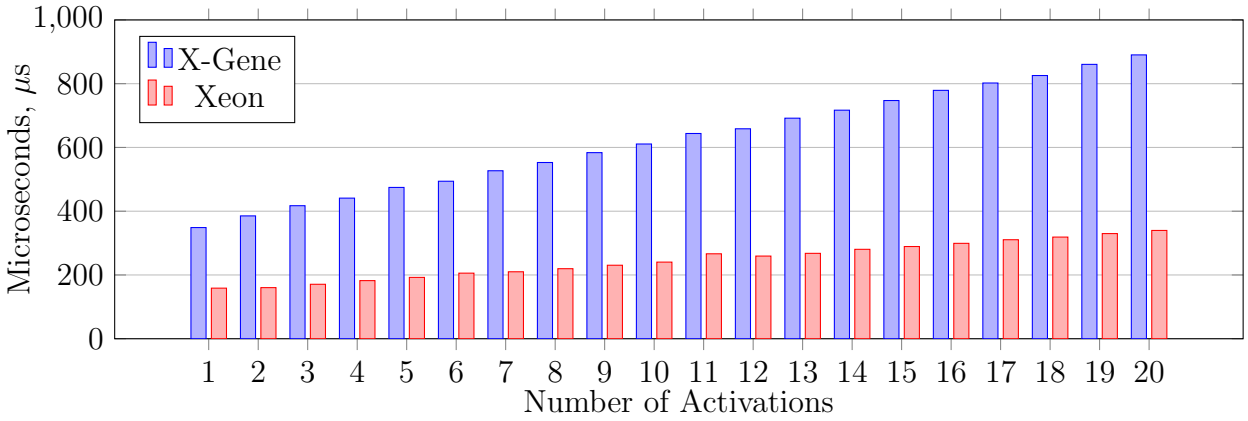
The microbenchmark recurses to a user-specified depth and then invokes the runtime to transform the thread's state R_i and S_i. There are three versions of the microbenchmark, each of which varies the number of live values per activation. The three versions have no live values per activation, 8 live values per activation, and 32 live values per activation that must be transformed between source and destination stacks. Each of these live values is a integer that must be copied between the two versions of the activation. In real applications live values can range in type from booleans to complex structures. The live value type is limited to integers in the microbenchmark in order to understand the costs associated with finding and applying the metadata to copy the live value between locations rather than the costs of memory copies. We observed that in real applications there were rarely more than 32 live variables at a call site. Similarly, the number of activations is varied from 1 to 20 in order to understand how costs increase with the number of open function calls. Although some applications may recurse into a deeper function call chain, analysis is limited to a maximum of 20 activations as it illustrates overhead trends associated with increasing stack depth. As shown in Section 6.3, however, applications in the NPB benchmark suite do not have deep recursion.

Figure 6.1 shows how state transformation latency rises with increasing numbers of activations for the three versions of the microbenchmark. The runtime is able to transform thread state on the Xeon with very low latencies. In all versions of the microbenchmark, threads are able to completely rewrite their state in under 400μs. As expected, the number of activations is directly proportional to the transformation latency, although costs rise slowly. The number of live variables per activation has a slight impact on performance, meaning that most of the cost comes from discovering live activations and unwinding frames.

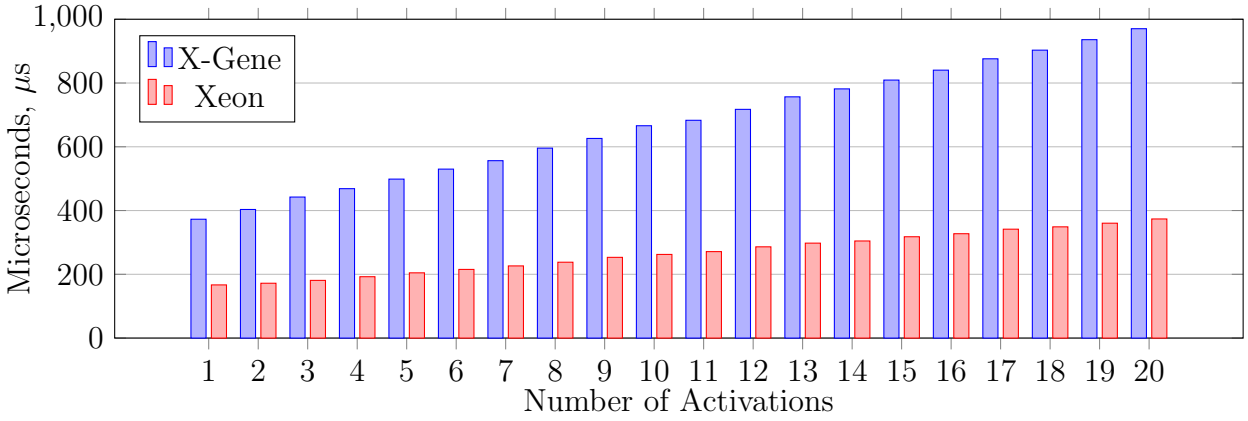
The X-Gene, as expected, has a higher latency versus the Xeon. This is due to both the lower clock speed, the smaller amount of cache and the relative immaturity of the X-Gene processor. Because it was built using a 40nm process, it has fewer transistors per chip and thus has fewer performance optimizations compared to the Xeon. Nevertheless, the runtime is still able to transform state on the X-Gene with low latency – all except one configuration of the microbenchmark has a transformation cost of less than 1ms. The effects of increasing numbers of activations are more exaggerated on the X-Gene, as are the costs for rewriting more live values.

The runtime was instrumented with fine-grained timing information in order to get a clearer understanding of which phases of transformation dominate execution time. Figure 6.2 shows the breakdown of execution time into four phases:

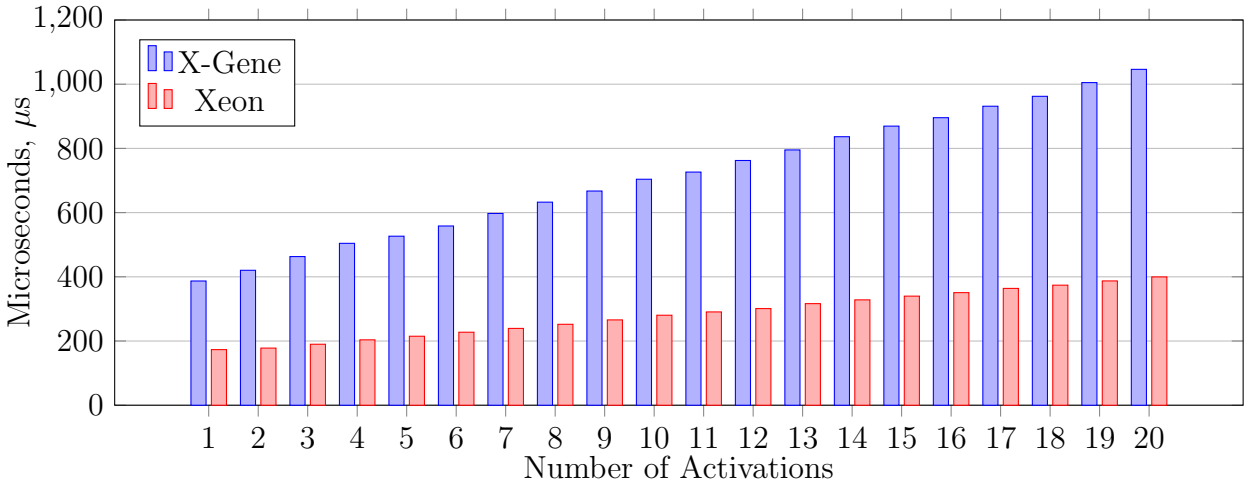
1. **Initialization** – time required to allocate and initialize rewriting contexts for both the source and destination thread state.
2. **Unwind and size** – time required to unwind the source's stack and allocate space for the destination stack as described in Section 5.2.1.
3. **Rewrite** – time required to rewrite the state, as described in Section 5.2.2.



(a) Transformation latency, no variables in each activation



(b) Transformation latency, 8 variables per activation



(c) Transformation latency, 32 variables per activation

Figure 6.1: State transformation latency

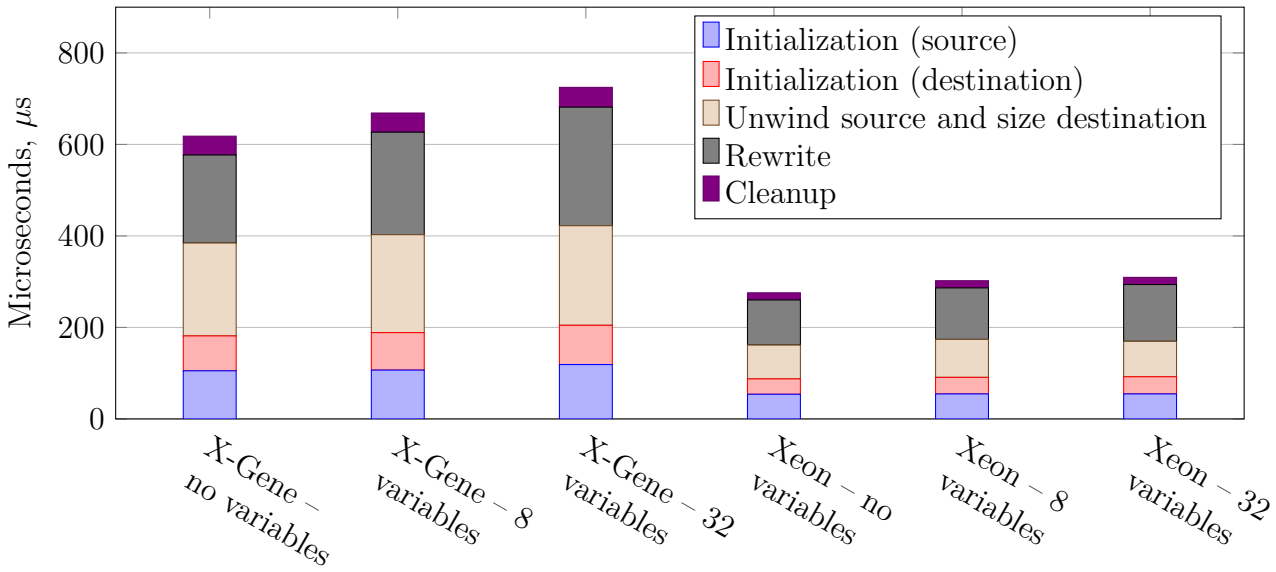


Figure 6.2: Time spent in each phase of state transformation

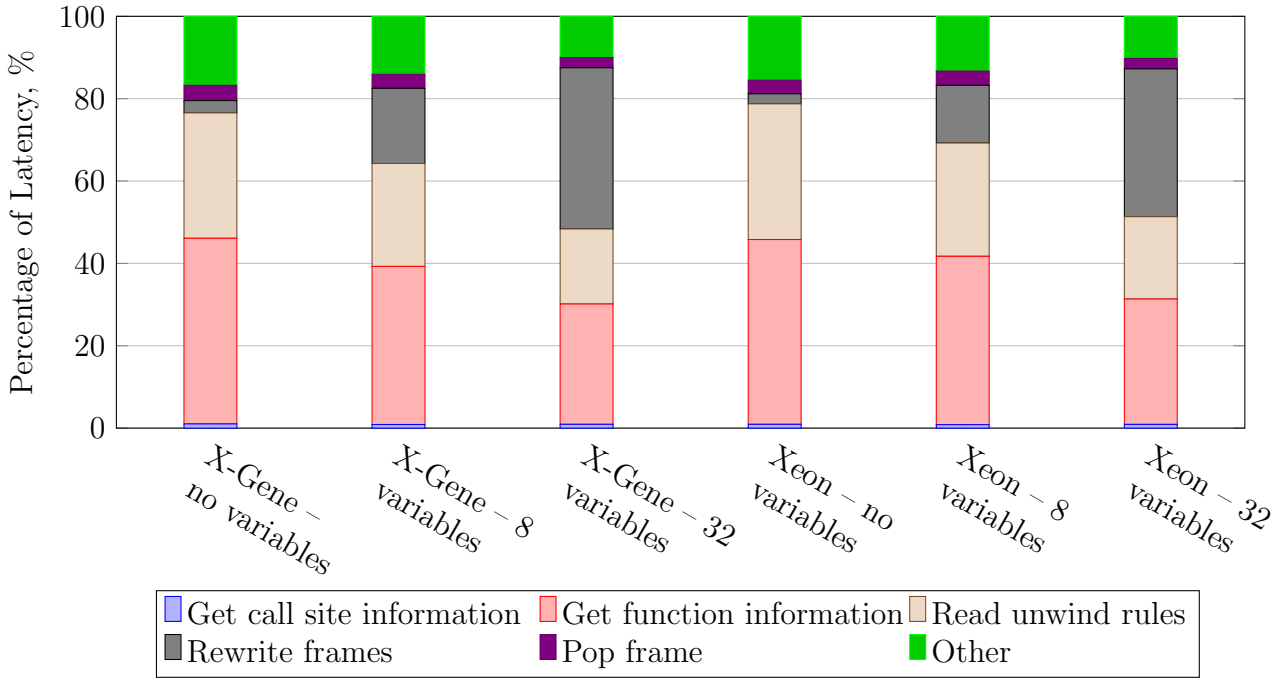


Figure 6.3: Percentage of time spent executing different actions during state transformation

4. **Cleanup** – time required to tear-down and free the rewriting contexts for both the source and destination contexts.

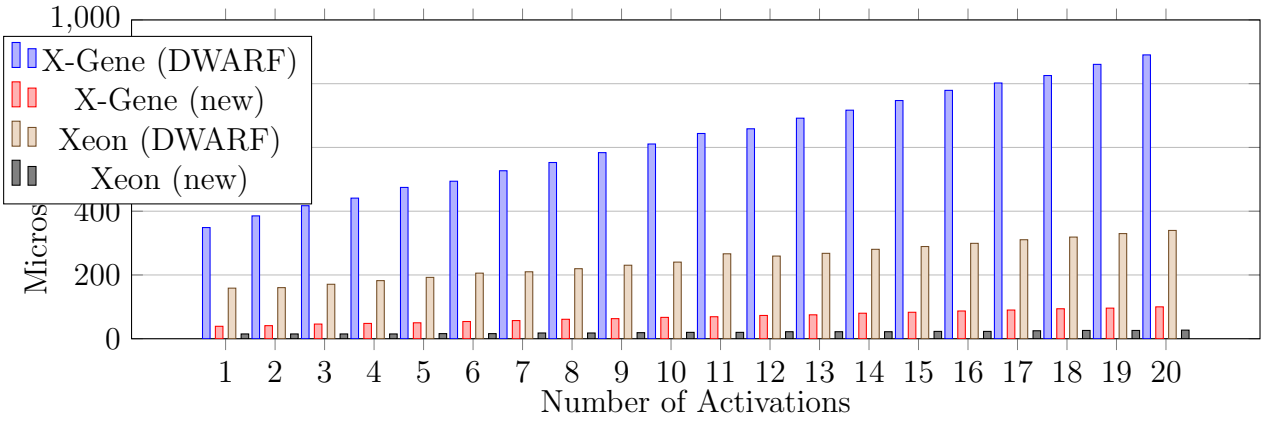
Figure 6.2 shows the timing breakdown into the four phases for each of the three versions

of the microbenchmark with 10 live activations. As shown in the figure, initialization and cleanup take only about a third of the total transformation time. Unwinding and sizing the destination stack takes about a third of the time and rewriting state takes up the remaining third. Note that as the number of live variables increases, the amount of time spent rewriting activations takes up a larger proportion of the time. With larger numbers of variables, theres a larger metadata lookup and copying cost between frames. This is more evident for the X-Gene, but is also present on the Xeon.

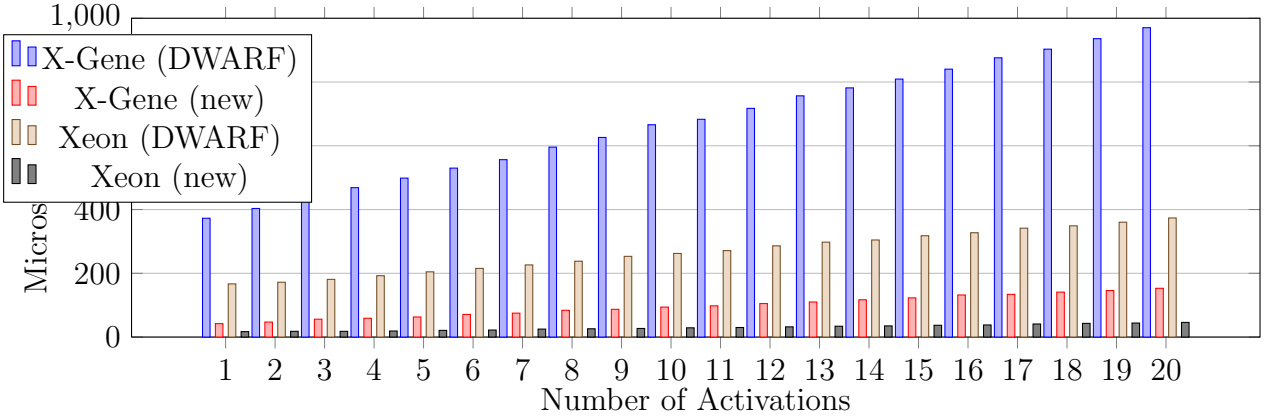
The transformation timing was broken into different actions required per activation. Figure 6.3 shows the percentage of the total transformation latency spent performing the following actions:

- **Get call site information** – given a program location for the thread, how long it takes the runtime to do a dictionary look up to find the call site record, and then use the ID from that record to do another dictionary lookup to find the corresponding program location for the destination.
- **Get function information** – given a program location obtained from the call site record, how long it takes the runtime to find DWARF debugging information for the surrounding function.
- **Read unwind rules** – given a program location and surrounding function, how long it takes the runtime to read the call frame unwinding information from the DWARF metadata.
- **Rewrite frames** – given a call site record, how long it takes the runtime to parse the location records for live values in both the source and destination activation, and the time required to copy the data between them.
- **Pop frame** – after a frame has been rewritten, how long it takes the runtime to apply the DWARF frame unwinding procedure to return to the caller frame.
- **Other** – the time to perform other miscellaneous actions.

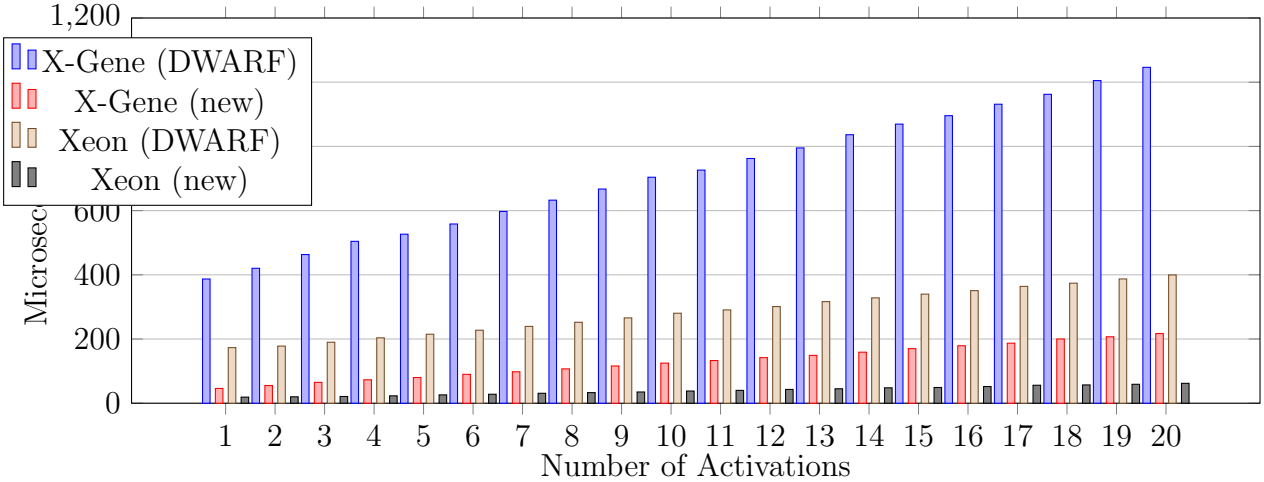
Figure 6.3 shows the timing breakdown in percentage of total transformation time for the aforementioned actions. The breakdowns for both the X-Gene and Xeon are virtually identical, demonstrating that although the total time required is different between the two processors, the costs of the different actions are proportionally similar. The majority of time required for transformation is spent performing DWARF-related actions. The DWARF library incurs significant overhead when searching for function information both because it dynamically allocates function descriptors and it performs a linear search over address ranges to find the function descriptor for a given program location. Additionally, reading frame unwinding metadata does significant memory copies between internal DWARF data structures and buffers allocated by the state transformation runtime.



(a) Transformation latency, no variables in each activation



(b) Transformation latency, 8 variables per activation



(c) Transformation latency, 32 variables per activation

Figure 6.4: State transformation latency after removing DWARF debugging information

As expected, Figure 6.3 shows that rewriting frames becomes a larger source of overhead as the number of live values per activation increases. There is still a slight overhead for rewriting even when there are no variables per activation. This is because the runtime must still populate the saved frame base pointer and the return address in each call frame on the stack.

In order to further reduce state transformation latencies, the Popcorn compiler was modified to emit the relevant DWARF information (frame records, unwinding information) in a format more amenable for state transformation. Refactoring the format of the metadata emitted by the compiler eliminates all of the aforementioned DWARF-related copying and speeds up several of the implementation mechanisms, e.g., replacing the linear function record search with a binary search, faster frame unwinding, etc. Figure 6.4 shows the new execution times of each of the aforementioned microbenchmarks when using the new metadata format. As seen from the results, state transformation without DWARF debugging information is 6x - 10x faster. This is due to removing all of the extraneous metadata parsing, copying and unoptimized implementation details. Across all stack depths in all microbenchmarks, the Xeon requires less than 65 microseconds and the X-Gene requires less than 220 microseconds for the entire state transformation process. These results demonstrate that dynamic state transformation is feasible for both register state R_i and stack state S_i. Furthermore, because the latencies are in the sub-millisecond range, Popcorn Linux can migrate threads between architectures at a fine granularity with minimal performance impact.

6.3 Single-Application Costs

In this section the state transformation latencies associated with real applications are analyzed. Four benchmarks from the NAS Parallel Benchmark (NPB) Suite [23, 175] were run, which represent computational fluid dynamics problems used by NASA. NPB applications are compute- and memory-intensive, with a focus on floating-point computation (except for the Integer Sort benchmark). NPB applications can be compiled with different class sizes, which scale the amount of computation from single-server workloads to cluster-size computation. The applications are written in C and are parallelized using OpenMP. For this evaluation, the benchmarks were run on both the X-Gene and the Xeon processors in single-threaded mode. They were run without any external workload in order to understand the performance characteristics of each application. The class A versions of the benchmarks, one of the smaller computation sizes, was used for state transformation analysis. This is because larger class sizes do not affect the transformation costs for thread state R_i and S_i, but only affect the amount of global computation to be performed (i.e., the number of loop iterations performed during computation).

Figure 6.5 shows the average and maximum stack depth for each of the applications. For these benchmarks, threads do not recurse into deep call stacks – the average and max stack depths are never larger than five call frames. Most the computation is nested in a for-loop

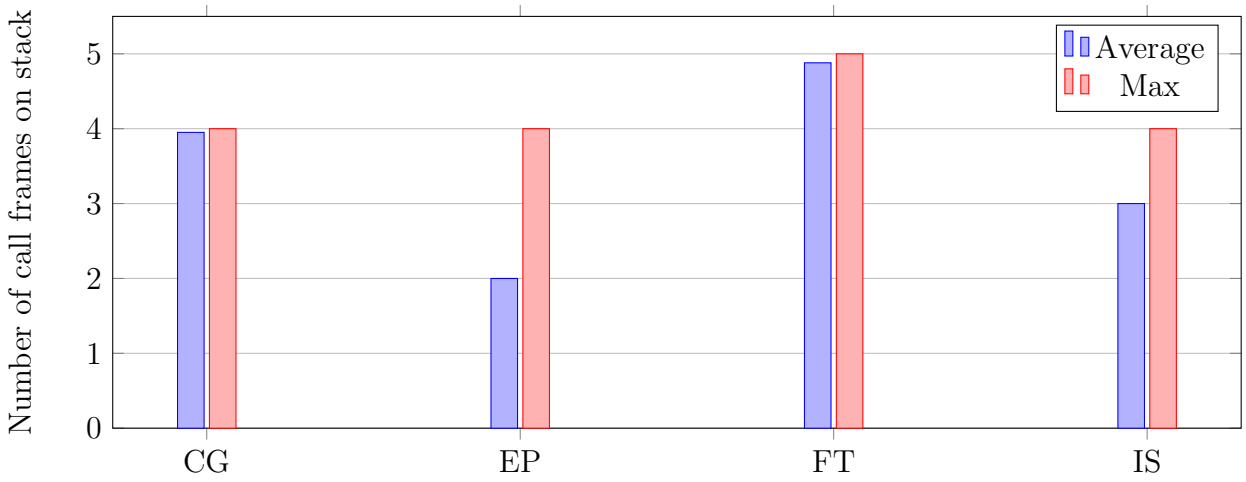


Figure 6.5: Average and maximum stack depths for benchmarks from the NPB benchmark suite

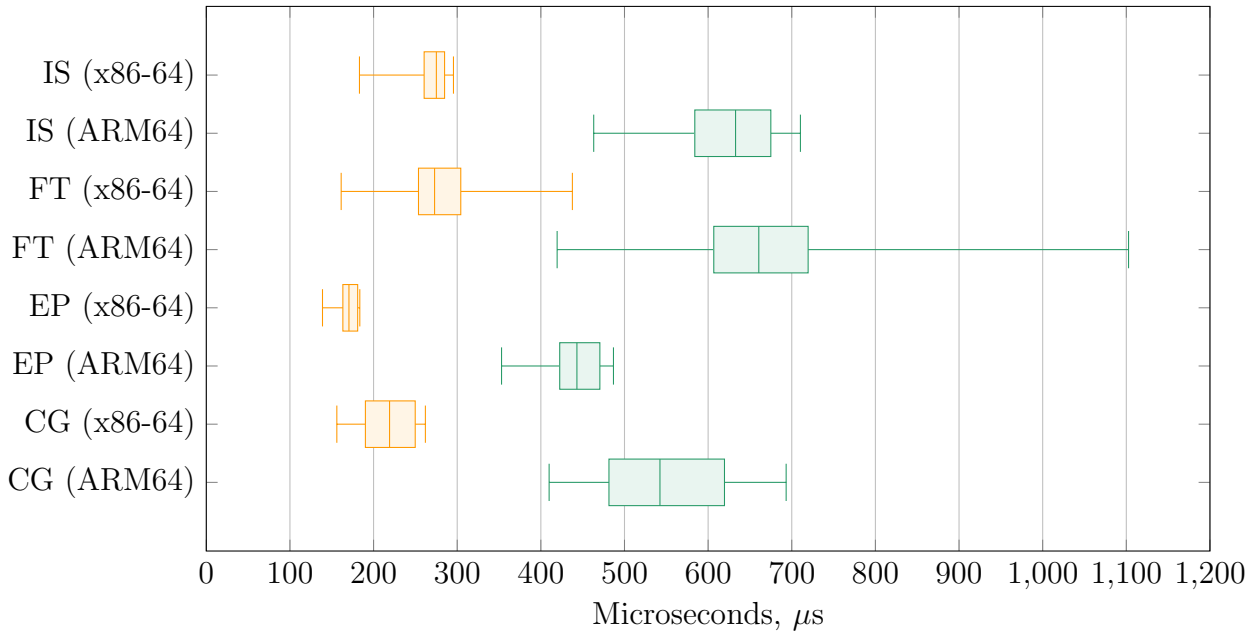


Figure 6.6: State transformation latency distribution for all migration points in real applications. Box-and-whisker plots show the minimum, 1st quartile, median, 3rd quartile, and maximum observed transformation latencies.

in the main function of each application, which call a few helper routines to do the heavy computation.

Figure 6.6 shows the distributions of state transformation latencies across all migration points, added to the binaries as discussed in Section 4.2, in each of the applications. The

plot contains a box-and-whisker plot for each benchmark on each processor, which shows the minimum, 1^{st} quartile, median, 3^{rd} quartile and maximum latencies observed across all migration sites. Once again, the Xeon exhibits smaller state transformation latencies compared to the X-Gene. However, the majority of transformation costs for both processors is well under one millisecond.

Interestingly, these benchmarks exhibit higher transformation costs than what would be expected based on the microbenchmark analyzed in Section 6.2. This is due to a larger amount of machine code being generated for real benchmarks, which leads to increased DWARF debugging metadata for function address ranges and a larger number of call site records. Finding a call site record and enclosing function for a given program location takes longer with more metadata, because the runtime must search through more call site records and more address ranges. Table 6.2 summarizes the difference in time required for executing individual actions for the microbenchmark versus FT on the Xeon processor. For each activation, the runtime must do two call site lookup queries (one to locate the call site ID on for the source, another to locate the call site record for the destination). It must also get the function information and read the unwind rules for both the source and destination activation. A 393% increase in finding function information and a 228% increase in reading call frame unwinding rules accounts for the significant increase in per-activation latency. Other benchmarks experience similar behavior to FT.

	Get call site information	Get function information	Read unwind rules
Microbenchark	0.127 μs	5.584 μs	3.384 μs
FT	0.888 μs	21.957 μs	7.701 μs

Table 6.2: Time required for executing individual actions on the Xeon

Even with this increased per-activation latency, state transformation costs are still small enough to enable fine-grained application migration.

6.4 Alternative Migration Approaches

In this section Popcorn Linux's state transformation and migration efficiency is compared to a Java-based approach. The Paderborn Thread Migration and Checkpointing Library (PadMig) [83] provides a compiler and runtime for migrating threads between Java virtual machines (JVM) running on separate machines. The library provides communication between JVMs over the network, and can automatically serialize a running application's object state for migrating a thread. PadMig does source-to-source transformation to insert migration points into the source Java source, similarly to the Popcorn compiler. At runtime, the library uses Java's reflection to automatically serialize and de-serialize application data, eliminating the need manually send and receive data like an MPI program.

Figure 6.7 shows a comparison between Popcorn Linux and PadMig in terms of power consumption and execution migration efficiency. IS class B was run on the Xeon and migrated the verification phase of the benchmark (`full_verify`) to the X-Gene. The x-axis shows the total execution time for the benchmark on both systems. The left y-axis shows instantaneous power consumption, and the right y-axis shows CPU load. The top row of graphs shows the power consumption of the X-Gene CPU over the course of execution, while the bottom row shows the same for the Xeon. System power represents the whole-system power as measured by the external power monitoring setup, while CPU power represents the power measured by on-board sensors. The load represents the total amount of CPU time spent executing the application.

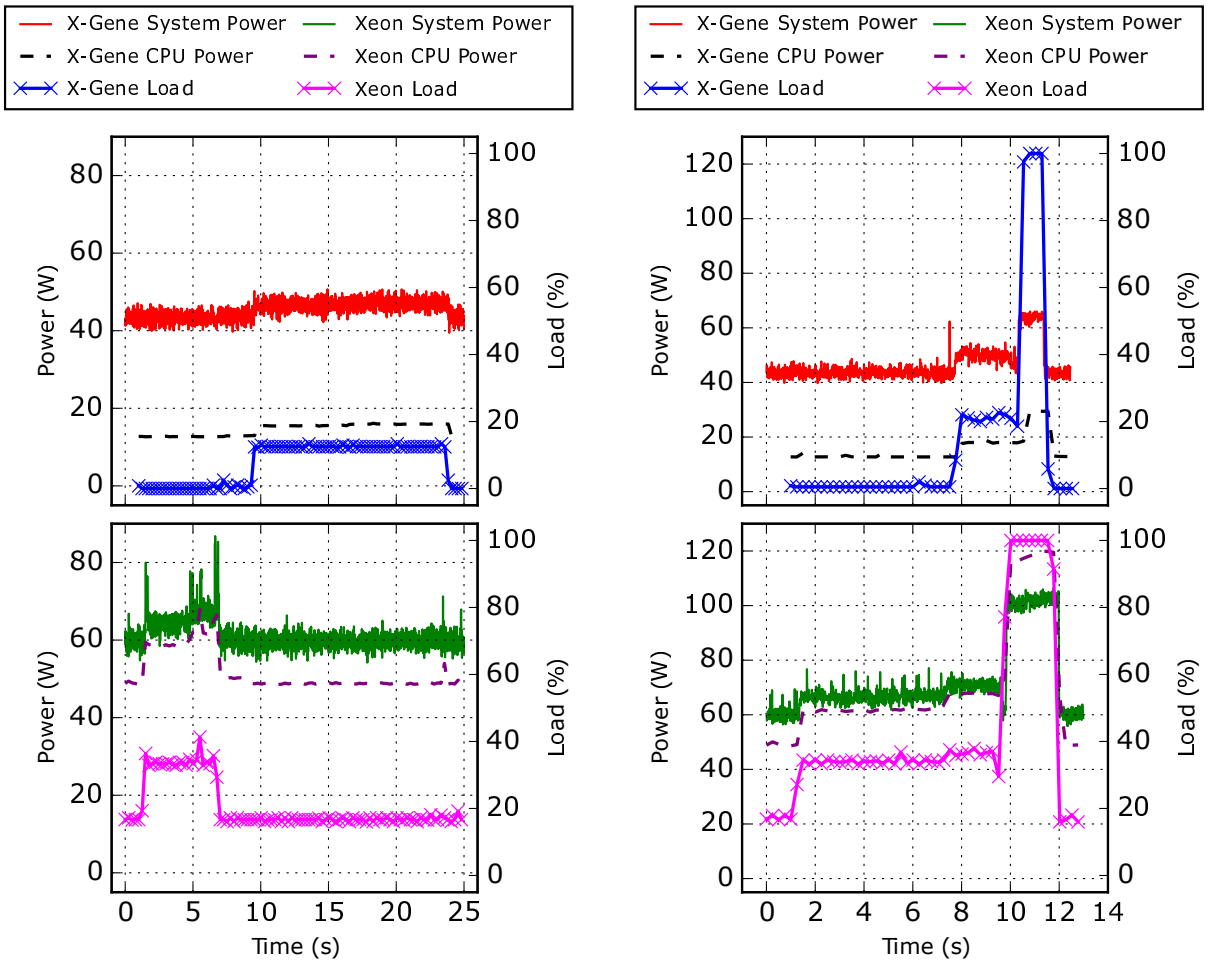
Figure 6.7 clearly shows the advantages of Popcorn Linux's state transformation and execution design versus a language-level based approach. Popcorn Linux takes approximately half as much time to execute IS, which translates into significant overall energy savings. PadMig spends a significant amount of time serializing (seconds 5-7 in Figure 6.7a) and de-serializing (seconds 9-13) data. Popcorn Linux, instead, benefits from laying out the majority of application state (P_C, P_D, P_H and L_i) in a common format, and only performing state transformation for a small portion of a thread's execution state. In general, the power consumption is roughly equal across the two executions of IS. However, Popcorn Linux incurs a significant load and power spike between seconds 8-12, as seen in Figure 6.7b. This is due to significant numbers of page transfers between the two kernels, as the Xeon transfers pages to the X-Gene so that calculations can be verified. Popcorn Linux's DSVM service (which implements the page coherency protocol) is multithreaded, meaning it can support a large number of in-flight page transfers.

These results clearly show that Popcorn Linux's design has significant power and performance advantages over virtual machine-based migration approaches.

6.5 Optimizing Multiprogrammed Workloads

Popcorn Linux's ability to migrate applications efficiently makes it possible to take advantage of different ISAs in a datacenter-like system, unlike current heterogeneous-ISA datacenters which must be partitioned into per-ISA zones. Using Popcorn Linux, applications are able to migrate at function boundaries between architectures that vary in terms of performance and power. In this section Popcorn Linux's ability to adapt changing workloads is evaluated. Previous work by Mars and Tang [136] and DeVuyst et al. [67] examine scheduling in homogeneous-ISA processors with heterogeneous microarchitectures at the cluster level and the chip-multiprocessor level, respectively. However to the best of our knowledge, no previous works have studied scheduling in heterogeneous-ISA datacenters.

Because vanilla Linux (referred to hereafter simply as Linux) cannot migrate applications between architectures at runtime, the scheduler can only provide an initial placement of



(a) PadMig execution time and power consumption

(b) Popcorn Linux execution time and power consumption

Figure 6.7: Comparison of Popcorn Linux and PadMig execution time and power consumption for IS class B. The x-axis shows the total execution time for each system. The left y-axis shows instantaneous power consumption in Watts and the right y-axis shows CPU load. The top row shows power consumption and CPU load for the X-Gene, while the bottom row shows the same for the Xeon.

applications across the X-Gene and Xeon processors. After the application has begun executing on one of the processors it cannot be migrated across ISA boundaries. Several baseline scheduling policies were developed for Linux, on both a homogeneous-ISA and a heterogeneous-ISA test setup:

- **Homogeneous Balanced** (homogeneous) – a two-x86 setup is considered where the scheduler places applications across two identical Xeon E5-1650v2 processors. The scheduler keeps the number of threads balanced across both processors. Note that

even though the processors are identical, there is no mechanism in Linux to migrate applications between kernels.

- **Static Balanced** (heterogeneous-ISA) – the scheduler balances the number of threads across X-Gene and Xeon processors. After an application (and its threads) have been assigned to an architecture, they cannot migrate to another architecture.
- **Static Unbalanced** (heterogeneous-ISA) – the scheduler assigns threads to the X-Gene and the Xeon according to some ratio. Because the Xeon has a much higher computational capacity than the X-Gene, the scheduler assigns twice or three times as many threads (a 2:1 or 3:1 ratio) to the Xeon.

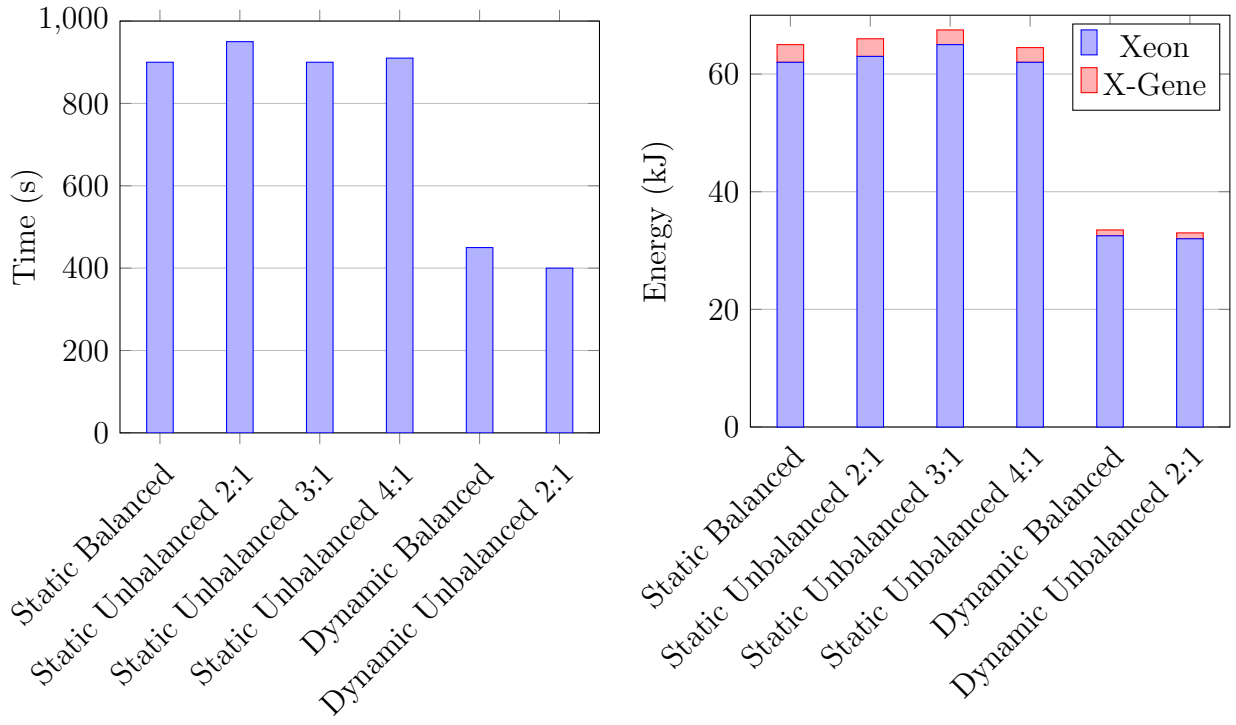
Popcorn Linux provides unique execution capabilities versus vanilla Linux. Without Popcorn Linux's thread migration and DSM support coupled with runtime state transformation, applications cannot migrate between heterogeneous-ISA architectures. This means that jobs can be scheduled onto the X-Gene or the Xeon machine, but cannot switch between them as the system load varies. Popcorn Linux can instead take advantage of execution migration to adjust the workload of each processor in the system. Several scheduling policies were developed based on system workload that balance load across the machines:

- **Dynamic Balanced** – this heuristic keeps the number of threads balanced across both the X-Gene and the Xeon processor. This is similar to the Static Balanced policy mentioned above, except that the scheduler can migrate jobs after they have begun execution.
- **Dynamic Unbalanced** – this heuristic keeps the number of threads assigned to each processor equal to a ratio. This is similar to the Static Unbalanced policy mentioned above, except that the scheduler can migrate jobs after they have begun execution.

The instantaneous power consumption for both the X-Gene and Xeon processors was measured using the on-board sensors, as the external power monitoring setup also measures power consumption of hard disks, peripherals (e.g., USB devices), etc., which is not directly correlated to the computation. Additionally, because the X-Gene is a first-generation processor, an optimized version is estimated would consume 1/10th the reported instantaneous power using McPAT [122]. A shrink in process node to a 22nm FinFET (similar to the Xeon) was estimated to not only allow the X-Gene to have significantly reduced power consumption, but to allow for aggressive power gating, dynamic voltage frequency scaling, and low power CPU states.

Figure 6.8 shows the first multiprogrammed workload run using the scheduling policies mentioned above. Sets of jobs were generated using the NPB benchmarks at class sizes A, B and C in a uniform distribution. The Static Balanced and Static Unbalanced policies were first evaluated against the Dynamic Balanced and Dynamic Unbalanced policies. There were also

two baselines used where the jobs were either all scheduled onto the X-Gene (All on ARM) or all onto the Xeon (All on x86).



(a) Total makespan ratio

(b) Energy consumption

Figure 6.8: Static vs. Dynamic scheduling policies in heterogeneous setup

As seen in Figure 6.8, execution migration using the dynamic policies provides enhanced flexibility which leads to half the energy consumption and half the runtime. With the static policies, the scheduler is not able to adjust decisions, meaning oftentimes the jobs on the X-Gene take significantly longer to execute while the Xeon becomes idle. With the dynamic policies, the scheduler pulls more workload onto the Xeon as it completes jobs while a smaller fraction continue execution on the X-Gene. This demonstrates that execution migration is a valuable mechanism for adapting workloads to changing conditions. Because of this, only the dynamic heterogeneous policies are evaluated in the remaining experiments.

Figure 6.9 shows the total energy consumption and the makespan ratio (i.e., the total time to completion for all benchmarks in the set) for each of the scheduling policies on each of the workload sets. Each of the sets consists of 40 jobs that arrive sequentially without overloading the machines, i.e., there is one application per core in the system. Once a job finishes, another is scheduled immediately in its place. This continues until all 40 jobs have finished.

As seen in Figure 6.9, execution migration allows the system to trade off performance versus

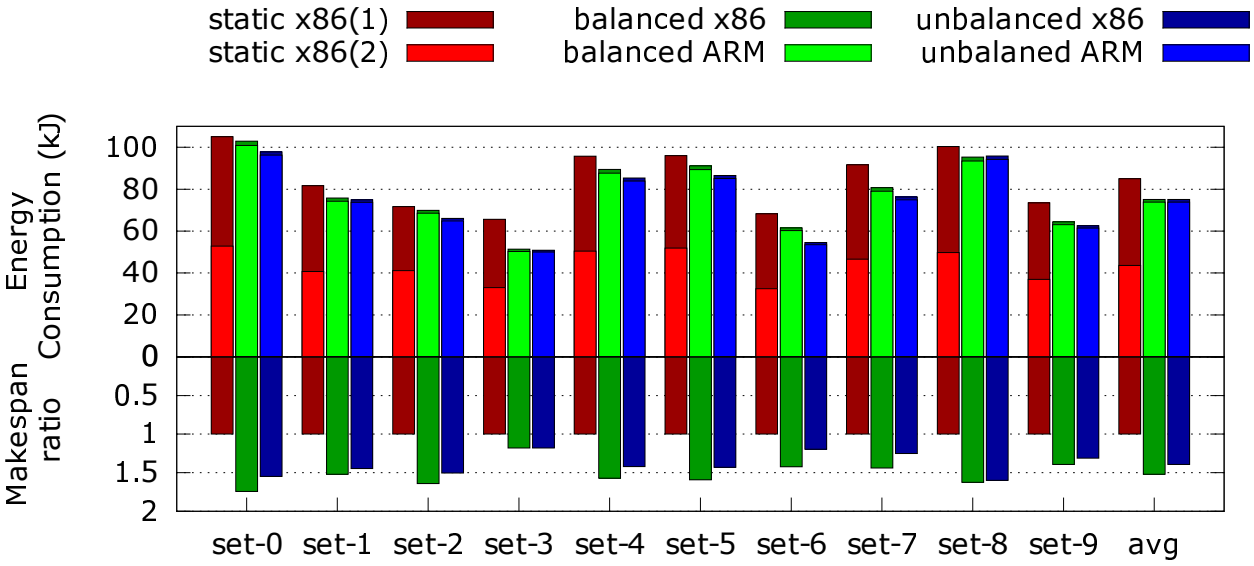


Figure 6.9: Energy consumption and makespan ratio for several single-application arrival patterns

energy savings. On average, both the Dynamic Balanced and Dynamic Unbalanced policies have a 22% reduction in energy consumed. The Dynamic Balanced policy has a 49% increase and the Dynamic Unbalanced policy has a 41% increase in makespan ratio, however.

The X-Gene processor has much less computational capacity versus the Xeon, and therefore applications scheduled to the X-Gene take a longer time to execute. However, the X-Gene consumes significantly less power and thus effectively trades off performance for reduced energy consumption. This experiment shows that Popcorn Linux allows system administrators to trade off performance for increased energy efficiency. Administrators can tune the system according to how much energy they want to consume. If, for example a datacenter operator wanted to reduce the amount of computational capacity in the datacenter in order to conserve energy, the administrator could migrate applications to the lower-performing energy-efficient X-Gene servers. Alternatively, they could migrate applications to Xeon servers when increased computational capacity is needed.

Figure 6.10 shows the total energy consumption and the energy-delay product (EDP) for a clustered workload. In this experiment, workload sets are once again generated as described above. However rather than a single application arriving at a time, 5 waves of 14 applications arrive every 60 to 240 seconds. Thus, the scheduler must schedule all 14 jobs as the cluster arrives. Results for the Dynamic Unbalanced Policy are omitted as the results differ from the Dynamic Balanced policy by less than 1%.

Figure 6.10 shows significant benefits for using execution migration in this workload scenario. For all workload sets, using a dynamic policy with a heterogeneous-ISA system saves a significant amount of energy – the Dynamic Balanced policy saves 30% energy on average,

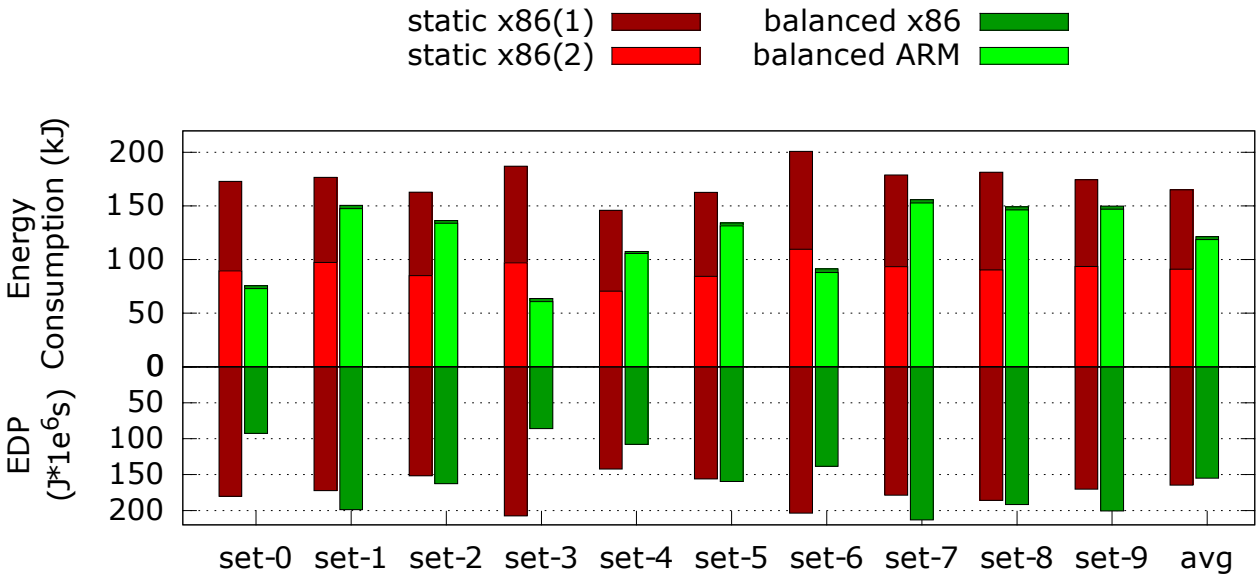


Figure 6.10: Energy consumption and makespan ratio for several clustered-application arrival patterns. Results for Dynamic Unbalanced policy are not shown as they differ by less than 1% from the Dynamic Balanced policy.

and up to 66% for set-3. Additionally, there is on average an 11% reduction in EDP versus a Homogeneous Balanced scheduler.

The reasons for lowered energy consumption and increased EDP are somewhat nuanced. As clusters of jobs arrive, all jobs are scheduled across the processors in the system. Because the waves arrive at 60-240 intervals, some applications from a previous wave are still running when the new wave arrives. Eventually both processors are over-saturated which leads to frequent context swaps, TLB flushing and cache thrashing. In the homogeneous setup, both Xeon processors are overloaded, meaning they are executing at full power while applications are competing for processing resources. In the X-Gene/Xeon setup, the same phenomena occurs but is handled more gracefully. The X-Gene consumes significantly less power while still making progress on application execution. The Xeon CPU completes job execution more quickly, and pulls jobs from the X-Gene when it has spare capacity. In this way the X-Gene gets computation started and the Xeon pulls jobs over to finish them more quickly. In essence, the degraded performance is less of an issue because the X-Gene consumes much less power.

These experiments validate the usefulness of heterogeneous-ISA execution migration in a datacenter. As datacenters become more heterogeneous, it becomes increasingly important for system software to be able to adapt workload execution across a pool of machines in order to meet power and performance goals. Popcorn Linux provides execution migration across ISA boundaries, enabling enhanced flexibility which leads to better server utilization and energy efficiency.

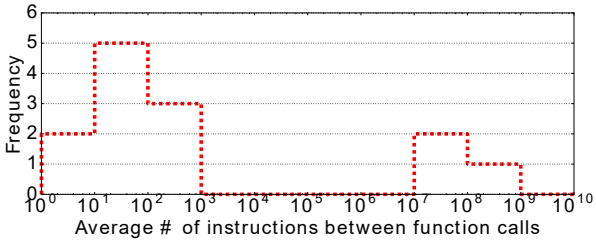
Chapter 7

Lower Migration Response Time Via Hardware Transactional Memory

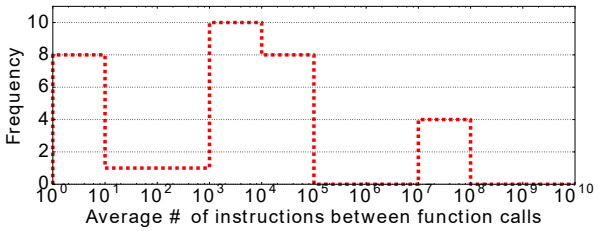
Popcorn Linux's compiler transparently inserts migration points into applications during compilation in order to enable migration between heterogeneous-ISA CPUs. The compiler requires migration points to be a subset of the application's equivalence points as described in Section 3.1.1. This is required as there only exists valid transformations between ISA-specific variants of the thread's execution stack at equivalence points. Currently the compiler selects function call boundaries as migration points, inserting call-outs to the migration library at the beginning and end of every function in the application.

However depending on the structure of the application, there can be significant delays between when the scheduler requests a migration and when the application reaches a migration point. This delay, defined as the *migration response time*, can lead to sub-optimal scheduling behaviors when trying to place applications to improve performance or power efficiency. Figure 7.1 shows the distribution of number of instructions between subsequent migration points for executions of three NPB applications, CG, FT and IS. For all three applications there are several clusters of instruction distances. While smaller distances (i.e., fewer than $10^5 = 100,000$ instructions) do not cause significant delays in migration response time, larger instruction distances between migration points may prevent the application from responding to migration requests for milliseconds or even seconds at a time. Each application has instances of both small and large migration response times, highlighting the need for the compiler infrastructure to increase the granularity at which applications can respond to migration requests.

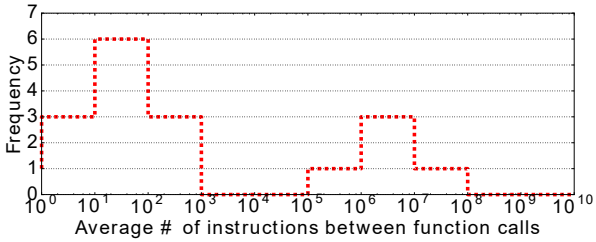
Previous works by DeVuyst et al. [70] and Venkat and Tullsen [197] use a dynamic binary translation (DBT) framework based on QEMU [31] to side-step this issue. When the scheduler wishes to migrate an application to a CPU of a different ISA in their framework, the application's state is imported into QEMU on the destination CPU and emulated up until an equivalence point. At the equivalence point, the transformation runtime translates the



(a) CG, class A



(b) FT, class A



(c) IS, class A

Figure 7.1: Distribution of number of instructions between migration points.

stack between ISA-specific formats and returns the application to native execution.

While using a DBT framework allows applications to respond immediately to scheduling requests (running with reduced performance until the next equivalence point), using such a complex framework in real systems has several downsides. First, it is not clear how state is imported into and out of the DBT framework. The emulator must load in all application state, including preparing registers and memory (data, page tables) for emulation. This can become an expensive process with large address spaces and multithreaded applications; the aforementioned works do not discuss these costs. Additionally, most emulators including QEMU [31] only support multithreaded execution by multiplexing all application threads onto a single emulator thread. One of the reasons is due to managing the code cache – attempting to synchronize multiple threads modifying and executing basic blocks from the code cache is a complex issue and can lead to trade-offs such as duplicating the code cache for increased performance at the expense of memory bloat. Additionally, cross-ISA emulation has the added burden of providing correct memory consistency semantics of one ISA on another architecture that may provide completely different semantics. Only recently have there been efforts to begin tackling semantically correct emulation of memory consistency models for multithreaded applications on different ISAs [61]. Finally, using a DBT framework to assist in migration requires integrating the entire DBT framework itself into all applications that may migrate, again leading to memory bloat and added application complexity.

Instead, this dissertation explores an alternative approach which uses *hardware transactional memory* (HTM) to roll back execution to the most recently encountered equivalence point upon receiving a scheduling request. Section 7.1 provides background information on HTM, including how it operates and its limitations. Section 7.2 describes how the Popcorn compiler

is augmented to insert HTM instructions into the application to blanket it in transactional execution and respond to scheduling requests. Section 7.3 describes how the Popcorn compiler was refactored to support this functionality. Section 7.4 evaluates the prototype system and Section 7.5 discusses findings and future work.

7.1 Background

Hardware transactional memory is an ISA extension used to implement transactional memory [94], a framework for optimistic concurrency control in multithreaded applications. When using transactional memory, the developer breaks down execution into *transactions* which are atomic units of functionality, e.g., converting a critical section protected by a lock into a transaction. Transactions execute in parallel by different execution units such as threads and may overlap in time. When executing transactions, the system providing transactional support (software or hardware) guarantees that either the entire transaction completes atomically or that none of the transaction takes effect, including reversing any side-effects of the transaction's execution. If the transaction completes successfully it is *committed* by the system; otherwise, the transaction is *aborted* and the system rolls back the application's state to before the transaction's execution. During execution of transactions, the system detects memory access conflicts, the main source of aborts. The transactional system maintains information describing which transaction accessed what data and how the data was accessed, i.e., whether it was read or written. Conflicts occur when multiple concurrent transactions access the same memory with incompatible access types, e.g., one thread reads from a memory location in one transaction that another thread writes in a concurrent transaction. In this situation, the system aborts the transactions and rolls back execution, which can either retry the transaction or fall back to other forms of concurrency control (e.g., locks).

Hardware transactional memory implements transactional memory by adding new registers and instructions to the ISA. The two main commodity scale HTM implementations are IBM's HTM extensions to the POWER ISA [45] and Intel's TSX-NI extensions to the x86-64 ISA [206][1] (IBM's System z mainframe ISA also has HTM extensions [143]). For processors implementing HTM, the ISA is augmented with instructions to start and end transactions[2]. In HTM implementations, the CPU maintains a buffer that both holds intermediate transaction data and detects conflicts. For example, Intel CPUs use the L1 cache as the HTM buffer whereas POWER CPUs use the L2 cache to buffer data and a small content addressable memory (CAM) to detect read/write conflicts. Both of these HTM implementations detect conflicts at the granularity of a cache line (64 bytes and 128 bytes for Intel and POWER CPUs, respectively), meaning that false sharing of data in the same

[1] We use restricted transactional memory (RTM) mode, as hardware lock elision (HLE) mode is intended as a transparent replacement for locks

[2] Some implementations additional instructions for more functionality, such as instructions to suspend and resume transactions in the POWER ISA

cache line can cause unnecessary aborts. As an implementation artifact of bounded buffer capacities, transactions may also abort due to buffers running out of space. Transactions may also abort due to conflict cache line evictions, i.e., cache lines evicted due to filling the ways of a given set in the cache [143]. Finally, there are a number of other ways transactions may be aborted such as mode switches or asynchronous traps. When aborting a transaction, the CPU discards all buffered state and returns to an implementation-defined location. For TSX-NI, aborted transactions will jump to an abort handler address specified in the transaction begin instruction. For POWER8, aborted transactions will return the instruction immediately after the transaction begin instructions and will set a flag indicating that the transaction was aborted. From the abort location, developers can specify an action, e.g., retry the transaction or fall back to conservative execution paths.

7.2 Design

When the scheduler is placing applications in heterogeneous-ISA systems, the applications in the workload must be responsive to migration requests in order to better optimize performance or power efficiency – long migration response times lead to sub-optimal placements as the workload dynamically changes. Therefore, applications must be instrumented with enough migration points to be responsive to scheduling requests. However, reducing the migration response time requires inserting more migration points, adding overhead from more frequent checks for migration requests. Thus when deciding where to insert migration points, the compiler must balance reducing the migration response time with adding additional overhead.

Previous works use a DBT framework to enable instantaneous migration by emulating up until a migration point, but using DBT in real systems causes complexity and performance issues. Instead of using a DBT framework, HTM can be used to instantly return execution back to the most recently encountered migration point. The compiler inserts transaction instructions at every migration point in an attempt to cover the entire application in transactional execution. At a migration point, the compiler inserts instrumentation to commit the current transaction and begin a new transaction. When no migration request is received, the application proceeds as normal, committing a series of transactions delineated by migration points. When a migration is requested, the scheduler signals the application, aborting the current transaction and diverting execution to the abort fallback path. The application then calls into the migration path from the fallback path, performing the migration procedure described in Chapter 5. Thus, the scheduler can instantaneously divert execution to a migration point.

Figure 7.2 shows an example of the instrumentation added to migration points to cover the application in transactional execution on x86-64. The instrumentation first ends the current transaction. The thread checks if it is inside a transaction using the `llvm.x86.xtest()`

```
entry:
  %0 = call i32 @llvm.x86.xtest()
  %htmcmp0 = icmp ne i32 %0, 0
  br i1 %htmcmp0, label %.htmend0, label %.htmendsucc0

.htmend0:
  call void @llvm.x86.xend()
  br label %.htmendsucc0

.htmendsucc0:
  %1 = call i32 @llvm.x86.xbegin()
  %2 = icmp eq i32 %1, -1
  br i1 %2, label %migpointsucc0, label %migpoint0

migpoint0:
  call void @migrate(void (i8*)* null, i8* null)
  br label %migpointsucc0

migpointsucc0:
  br label %for.body, !popcorn !2
```

Figure 7.2: LLVM bitcode instrumented with transactional execution at a migration point

intrinsic[3]. If so, it jumps to a basic block that uses the `llvm.x86.xend()` intrinsic to stop the current transaction. Note that this check is required on x86-64 because if a thread tries to end a transaction when not currently under transactional execution, the processor raises a segmentation fault. Next, the instrumentation starts the next transaction using `llvm.x86.xbegin()` – for x86-64, this intrinsic starts a new transaction and sets the abort handler path to the instruction immediately after the HTM begin instruction. The intrinsic returns a flag indicating the abort code. When starting a transaction, this flag is set to -1 to indicate that the thread successfully started a transaction. In this case, the thread jumps to basic block `migpointsucc0` and continues normal execution. At the next migration point, the thread will commit the current transaction and begin the next, continually jumping from migration point to migration point through transactions. However when the scheduler wishes to migrate the thread to another node, it sends a signal to the thread which causes the thread to abort the current transaction and return to the abort handler, i.e., the instruction directly after the HTM begin instruction. In this case, the flag returned by `llvm.x86.xbegin()` indicates that an abort occurred, which causes the thread to jump to basic block `migpoint0` and call into the migration library. In this way, applications respond very quickly to migration requests by immediately returning back to a migration point. Note that this is an instance of the classic polling mechanisms versus interrupt mechanism – checking for migration requests is a polling mechanism, whereas signaling migration and rolling back to a migration point is a way for the scheduler to interrupt execution.

[3]Intrinsics provide a way to inject architecture-specific functionality in LLVM's architecture-agnostic middle-end. Intrinsics look like function calls at the IR level, but get custom-lowered by the ISA's back-end to HTM instructions

7.2.1 Lightweight Instrumentation

At the heart of the approach is a best-effort algorithm which analyzes individual functions and places instrumentation inside the function's body. The goal is to place instrumentation throughout the function (and in general, the application) to both minimize overhead due to the instrumentation while still enabling the application to react in a timely manner to migration requests. The analysis algorithm exposes several tuning knobs which allow developers to adjust the granularity of instrumentation, and in turn the tradeoff between overhead and migration responsiveness.

The algorithm traverses the control flow graph (CFG) of functions, performing a forward data-flow analysis (i.e., basic blocks are visited before their successors) in order to understand the function's execution behavior. As the algorithm iterates over the basic blocks within a function, it analyzes individual instructions and keeps track of execution behavior according to a user-defined "weight". The algorithm uses this weight metric in addition to a user-specified weight capacity to make decisions about where to place instrumentation – once an execution path's weight becomes too "heavy" (i.e., exceeds the weight capacity) the analysis inserts instrumentation, which logically resets the weight to zero. Thus, the goal of the analysis is to understand how execution flows through a function's body and place migration points so that the weight of individual execution paths never exceeds the weight capacity.

The notion of a path weight is flexible and allows the algorithm to be tailored to different types of instrumentation. For example, in the case of HTM instrumentation the path weight is defined as the number of bytes read from and written to memory. A path is considered too heavy if the number of bytes read or written since the last instrumentation point (i.e., the start of the transaction) overflows the HTM buffers. For this particular type of instrumentation, the algorithm's primary duty is to prevent capacity aborts due to accessing too much memory within a single transaction.

The weight could also be defined so as to space instrumentation out temporally, e.g., threads should reach a migration point every $10ms$. For this type of instrumentation, the weight is defined as the latency in cycles per instruction and a path is considered too heavy when there are too many cycles between subsequent migration points. The instrumentation could be tailored to hit migration points with some user-specified frequency, allowing the application to meet system-specific responsiveness goals while minimizing instrumentation overhead. The same algorithm is thus used for both inserting HTM begin/end instructions to avoid overflowing transaction buffers and for temporally spacing migration points for responsiveness/overhead adjustments.

Basic execution of the algorithm. Algorithm 2 shows the analysis entry point called on each function in the program. The algorithm takes as inputs the function's bitcode and a resource capacity, and produces a set of program locations at which to insert instrumentation (i.e., migration points). The algorithm works by building up per-basic block and loop exit weights as part of the forward data flow analysis. This allows the analysis to calculate

Algorithm 2: Function analysis overview

Input: F, a function to be analyzed and C, a maximum resource capacity
Output: M, a set of migration points

```
/* Initialize migration points, block exit weights and loop exit weights */
```
$M = \emptyset$;
$W_{Blocks} = \emptyset$;
$W_{LoopExit} = \emptyset$;

```
/* Analyze loop nests contained in F */
```
foreach *Loop Nest* $N \in F$ **do**
  TraverseLoopNest(N, M, W_{Blocks}, $W_{LoopExit}$, C);
end

```
/* Analyze rest of the function's body in reverse postorder */
```
BB_{Topo} = **TopologicalSort**(**BasicBlocks**(F));
foreach *Basic block* $BB \in BB_{Topo}$ **do**
  W_{BB} = **IncomingWeight**(BB);
  $W_{Blocks}[BB]$ = **TraverseBlock**(BB, W_{BB}, M, C);
end

the incoming weight of a basic block and place instrumentation points at locations inside arbitrary basic blocks. Before describing how the algorithm analyzes loops (lines 4-6), it helps to understand how it traverses basic control flow (lines 7-11).

Figure 7.3 shows a simple if-else statement in LLVM IR. The algorithm starts by analyzing the entry basic block and follows control flow edges through the function body. In the case of HTM instrumentation, the algorithm is observing bytes loaded and stored by instructions in each basic block. The algorithm starts by traversing `entry` and records that it does not load or store any memory. The algorithm then traverses `if.then` and `if.else` in any order, as their mutual predecessor (`entry`) has been traversed. Both of these blocks have an incoming weight of 0 bytes loaded and stored from `entry`. The algorithm records that `if.then` has a weight of 8 bytes loaded and 8 bytes stored, while `if.else` has a weight of 4 bytes loaded and 4 bytes stored. Finally, the algorithm analyzes `if.end`. Because the algorithm cannot determine statically which path has been taken, it conservatively determines the incoming weight at the CFG join point to be the maximum weight over all predecessors of the block. Therefore, `if.end` has an incoming weight of 8 bytes loaded and 8 bytes stored.

For simple control flow, the algorithm traverses branches and joins in order to conservatively calculate path weights across all possible paths through the CFG. The algorithm can place instrumentation at arbitrary points inside of functions if the weight becomes too heavy. For example, in Figure 7.3 if the algorithm were to place instrumentation on line 13, the weight would reset and `if.then`'s final weight would be only 4 bytes loaded and stored. Note that function call sites inside of basic blocks are required instrumentation points – if the thread were to migrate inside of the called function, the current function's activation would need to be reconstructed at the call site for the destination ISA, meaning the compiler must emit stack transformation metadata. Thus, function calls are by definition instrumentation

```
@a = external global i32, align 4
@b = external global i32, align 4

define void @simplecfg(i32 %branch) {
entry:
  %tobool = icmp eq i32 %branch, 0
  br i1 %tobool, label %if.else, label %if.then

if.then:
  %0 = load i32, i32* @a, align 4
  %add = add nsw i32 %0, 1
  store i32 %add, i32* @a, align 4
  %1 = load i32, i32* @b, align 4
  %add1 = add nsw i32 %1, 2
  store i32 %add1, i32* @b, align 4
  br label %if.end

if.else:
  %2 = load i32, i32* @b, align 4
  store i32 %2, i32* @a, align 4
  br label %if.end

if.end:
  ret void
}
```

Figure 7.3: If-else control flow in LLVM bitcode

points.

Handling loops. Loops require more careful consideration, as they are both a source of uncertainty and can make up the bulk of an application's execution time. In general, loop iteration ranges, or the range of values over which a loop executes, are unknown at compile time. The algorithm is again designed to take a conservative approach and assume that a loop will execute for a large enough number of iterations as to require instrumentation.

The goals when analyzing loops are twofold – first, to add instrumentation into the loop body where appropriate, and two, generate correct analyses for basic blocks surrounding the loop. In particular, loop successors (i.e., successors of loop exiting blocks[4]) must properly incorporate loop execution behavior into their incoming weights. The algorithm calculates *loop exit weights*, or potential weights at loop exiting blocks based on iteration behavior, as part of its analysis. This allows the algorithm to incorporate a loop's behavior into the surrounding basic blocks. Additionally, this allows the algorithm to conceptually shrink all basic blocks that comprise the loop body into a single virtual node in the function's CFG, removing cycles from the forward data-flow analysis.

[4]Blocks inside the loop body which may branch to outside the loop's body

```
define void @vecadd(i32* %a, i32* %b,
                    i32* %c, i64 %nelem) {
entry:
  %cmp = icmp eq i64 %nelem, 0
  br i1 %cmp, label %for.end, label %for.body

for.body:
  %i = phi i64 [%inc, %if.end], [0, %entry]
  %and = and i64 %i, 15
  %cmp1 = icmp eq i64 %and, 0
  br i1 %cmp1, label %do.inst, label %if.end

do.inst:
  tail call void (...) @do_instrument()
  br label %if.end

if.end:
  %idx = getelementptr i32, i32* %a, i64 %i
  %0 = load i32, i32* %idx, align 4
  %idx2 = getelementptr i32, i32* %b, i64 %i
  %1 = load i32, i32* %idx2, align 4
  %add = add nsw i32 %1, %0
  %idx3 = getelementptr i32, i32* %c, i64 %i
  store i32 %add, i32* %idx3, align 4
  %inc = add nuw i64 %i, 1
  %ec = icmp eq i64 %inc, %nelem
  br i1 %ec, label %for.end, label %for.body

for.end:
  ret void
}
```

Figure 7.4: Transforming loop to hit instrumentation every 16 iterations

Unlike previous work which either places instrumentation outside of loops or in the loop's header to be executed every iteration [144], our design can choose to only execute instrumentation every N iterations, where N is chosen based on the loop body's weight and the capacity threshold. This is useful in cases where instrumentation is heavier than the cost of simple arithmetic and a branch. Figure 7.4 shows an example of this instrumentation, where threads execute the instrumentation only every 16 iterations. The header is changed so that the loop induction variable is compared against the analysis-selected N value of sixteen, and if zero, execution branches to the instrumentation (if the loop has no induction variable, one is added). Although this seems costly, simple arithmetic and advanced branch prediction takes only a few cycles on modern processors – for example, on Intel's Broadwell microarchitecture `and`/`cmp` instructions take one cycle and correctly predicted conditional jumps take two cycles [79]. This is minimal compared to the cost of a function call or TSX's `xbegin`/`xend` pair, which according to our microbenchmarking takes approximately

18.6 nanoseconds.

Function TraverseLoopNest – analyze a loop nest

Input: Loop nest N, set of migration points M, block weights W_{Blocks}, loop exit weights $W_{LoopExit}$, maximum capacity C
Output: Iterations between migration points for each loop, V

```
/* Analyze loops in nest with decreasing depth                              */
```
SortLoopsByDepth(N);
for *Loop* $L_i \in N$ **do**
```
    /* Analyze loop blocks in reverse postorder and with empty loop starting weight  */
```
    H = **Header**(L_i);
    W_H = **ZeroWeight**();
    $W_{Blocks}[H]$ = **TraverseBlock**(H, W_H, M, C);
    BB_{Topo} = **TopologicalSort**(**Blocks**(L_i));
    for *Basic block* $BB \in BB_{Topo}$ **do**
        W_{BB} = **IncomingWeight**(BB, W_{Blocks}, $W_{LoopExit}$);
        $W_{Blocks}[BB]$ = **TraverseBlock**(BB, W_{BB}, M, C);
    end
```
    /* Get loop's max divided and spanning path weight, iterations per migration point
       */
```
    $W_{L_i}^{Div} = \max_{P_{Div} \in L_i}$ **PathWeight**(P_{Div});
    $W_{L_i}^{Sp} = \max_{P_{Sp} \in L_i}$ **PathWeight**(P_{Sp});
    $V[L_i] = \lfloor C \div W_{L_i}^{Sp} \rfloor$;
    $W_{PrevIter} = \max\left(W_{L_i}^{Div}, W_{L_i}^{Sp} * (V[L_i] - 1)\right)$;
```
    /* Calculate loop exit weights by traversing paths through exit blocks      */
```
    foreach *Exiting basic block* $BB \in L_i$ **do**
        $W_{BB}^{Div} = \max_{P_{BB,Div} \in L_i}$ **PathWeight**($P_{BB,Div}$);
        $W_{BB}^{Sp} = \max_{P_{BB,Sp} \in L_i}$ **PathWeight**($P_{BB,Sp}$);
        $W_{LoopExit}[BB] = \max\left(W_{BB}^{Div}, W_{BB}^{Sp} + W_{PrevIter}\right)$;
    end
end
return V;

The algorithm traverses each loop nest within a function, analyzing individual loops within the nest one at a time. The algorithm proceeds from innermost loop outward, e.g., for sub-loop B inside of loop A, the algorithm analyzes B (removing cycles from A's body), then A (removing cycles from the function's body), and finally the function itself. Function TraverseLoopNest describes how the algorithm handles loop nests. For each loop in the nest (sorted by decreasing nesting depth), the algorithm calculates weights across control flow within the loop's body, identically to the function's body. This allows the analysis to place instrumentation inside the loop body, if necessary. After analyzing the loop's body (and potentially placing instrumentation within), the algorithm calculates loop exit weights so that successor blocks can incorporate the loop's execution behavior. In order to calculate loop

Function TraverseBlock – sequentially traverse instructions in basic block to analyze weight

Input: Basic block BB, incoming weight W, set of migration points M, maximum capacity C
for *Instruction* $I \in BB$ **do**
  if ***IsFunctionCall****(I)* || ***TooHeavy****(W, C)* **then**
    $M = M \cup I$;
    reset(W);
  end
  Analyze(W, I);
end

Function IncomingWeight –

Input: Basic block BB, block weights W_{Blocks}, loop exit weights $W_{LoopExit}$
Output: Incoming block weight W_{Start}
W_{Start} = **ZeroWeight**();
foreach *Basic block* $BB_{Pred} \in$***Predecessors****(BB)* **do**
  if BB_{Pred} *is sub-loop exiting block* **then**
    $W_{Start} = \max(W_{Start}, W_{LoopExit}[BB_{Pred}])$;
  end
  else
    $W_{Start} = \max(W_{Start}, W_{Blocks}[BB_{Pred}])$;
  end
end
return W_{Start}

exit weights, the algorithm categorizes individual paths through the loop into two types:

1. **Spanning paths** – paths that start at the first instruction in the loop header (i.e., loop entry point) and end at a loop backedge without any intervening instrumentation points.
2. **Divided paths** – paths that start at the loop entry point and end at an instrumentation point, that start at an instrumentation point and end at a loop backedge, or that begin and end at instrumentation points. In other words, divided paths are paths through the loop that have instrumentation points.

In order to calculate weights at loop exit points, the analysis first determines weight that can be accumulated during loop execution due to previous iteration behavior. The analysis traverses individual paths through the loop, independently maintaining the maximum of both spanning and divided paths. This allows the analysis to generate a cumulative previous iteration weight by taking the maximum of the following:

- Divided paths by definition have an instrumentation point along the path, and therefore do not carry weight between loop iterations (the weight is reset at instrumentation

points). The loop's divided path weight consists of the maximum weight from instrumentation points to loop backedges.

- Spanning paths have no instrumentation points, and thus weight is carried between iterations. The analysis must consider as many iterations as can be executed without overflowing the capacity threshold, referred to as the *number of iterations between migration points*.

Using this previous iteration weight, the analysis can easily calculate loop exit weights at exiting blocks. For each exiting block, the analysis again calculates the maximum divided and spanning path weights over paths through the exit block. The loop exit weight for the block is then calculated as the maximum of the following:

- Maximum weight of divided paths through the exit block. Again, because they by definition have instrumentation points they do not include weight carried from previous iterations.
- Maximum weight of spanning paths through the exit block added to the previously calculated previous iteration weight. Spanning paths do not have instrumentation points and therefore include previous iteration behavior.

Consider again the vector addition in Figure 7.4[5]. The loop's body has two paths: one which flows through `do.inst` (a divided path due to the call to `do_instrument()`), and one which does not. The loop's divided path weight is the weight from the call to `do_instrument()` to the loop backedge, or 8 bytes loaded, 4 bytes stored. The loop's spanning path weight is 8 bytes loaded, 4 bytes loaded per-iteration. If, for example barring any cache line conflicts, an HTM implementation has a storage capacity of 8 kilobytes, the analysis determines that the loop can execute 2,048 iterations without overflowing the buffer's capacity. These two loop weights are then used to calculate the loop's exit weight at the exit branch in `if.end` – it is determined to be the maximum of the divided path's weight to the exit (8 bytes loaded, 4 bytes stored) and the spanning path's weight (8 bytes loaded, 4 bytes stored) plus the previous iteration weight (8 bytes loaded, 4 bytes stored multiplied by 2,047 iterations). The loop's exit weight is then used to calculate `for.end`'s incoming weight – the maximum of the loop exit weight (16Kb loaded, 8Kb stored) and `entry`'s exit weight (0 bytes loaded, 0 bytes stored).

Optimizations. There are several optimizations applied to improve analysis and instrumentation. First, if the loop trip count is known at compile time and is less than the number of iterations between migration points, the analysis can elide instrumenting the loop altogether. This requires re-analyzing the surrounding basic blocks and loop exit weights,

[5]Although this figure shows an example of how a loop could be instrumented, we also use it to illustrate loop analysis

and potentially adding instrumentation points around the boundaries of the loop. Second, analysis is restricted to only consider powers-of-2 for the number of iterations between migration points. This is so that a simple bit mask is all that is required in the loop header to determine whether to execute the instrumentation rather than more expensive remainder division for unconstrained values. This latter optimization reduced overhead in every evaluated benchmark evaluated by up to 11%.

Limitations The analysis is currently constrained to analyze instructions that can be modeled by LLVM IR, meaning it does not consider inline assembly. Additionally, the analysis currently only supports natural loops with irreducible control flow (i.e., loops with only header) although it could be extended to support more esoteric control flow.

7.2.2 Automatically Tuning Instrumentation

Because the algorithm statically analyzes applications at the IR level, it does not have information about an application's dynamic behavior, e.g., memory access patterns or control flow. Because the instrumentation is highly affected by dynamic application behavior, the algorithm exposes several knobs which can be used to fine-tune the instrumentation. For example, in HTM instrumentation the algorithm knows the transactional buffer sizes for the microarchitecture (L1 cache size for Intel's Broadwell microarchitecture) but because HTM leverages the cache coherency protocol for conflict detection, the amount of memory that can be buffered also depends on the cache associativity [104]. Thus, the algorithm has the following tunable parameters:

- **Capacity** – the overall weight capacity, e.g., memory bytes read and written for HTM instrumentation, or cycles for temporal-based instrumentation
- **Threshold** – what percentage of the capacity can be filled before an instrumentation point should be inserted, e.g., 80%

These parameters, which can be adjusted per function[6] allow fine-tuning the placement of instrumentation points based on an application's dynamic behavior. Users can adjust these parameters in order to meet various overhead and responsiveness goals.

Although instrumenting applications to hit migration points every number of cycles can be adjusted to meet a range of responsiveness versus overhead goals, instrumenting code with HTM execution has a much more limited design space. In fact, the goal of HTM instrumentation is simple – add as few HTM begin and end instructions as possible while simultaneously minimizing transactional aborts. In particular, the instrumentation should be tuned to avoid capacity aborts, one of the biggest limiting factors in current HTM implementations [143, 206, 71]. Because there is a fixed goal for HTM instrumentation, an auto-tuning

[6]More advanced analyses could tune these for each loop in a loop nest

framework was developed (hereafter referred to as the *auto-tuner*) to automatically refine the instrumentation granularity based on performance profiling.

The auto-tuner uses performance counters to iteratively profile and refine the instrumentation granularity in order to reduce capacity aborts while simultaneously minimizing the number of transaction begin/end instructions. The auto-tuner begins by setting the capacity threshold close to the hardware limits – previous studies [93] demonstrate that most HTM implementations will abort if the transactional capacity is completely filled. The auto-tuner runs the application several times and collects counters for the number of cycles, the number of cycles run under transactional execution, the number of cycles in committed transactions, the number of transactions started and the number of transactions aborted. These counters give the auto-tuner a general idea of the instrumentation quality – if there are a large number of transactional aborts or a large number of cycles not run under transactional execution, then the instrumentation granularity must be reduced (i.e., more transactions must added). The auto-tuner then reduces the capacity threshold and re-runs the application. Modern performance monitoring units also provide precise counter event locations [105], which allows the auto-tuner to pinpoint exactly which functions have a high number of aborts and should have their instrumentation granularity reduced; alternatively, if there are a large number of functions with capacity aborts, the auto-tuner can reduce instrumentation granularity for the entire application. This process is repeated until the auto-tuner finds a suitable instrumentation configuration, or reaches a maximum number of iterations.

7.3 Implementation

Although intrinsic instructions provide a way to abstract HTM functionality in the middle-end, the back-end will not accept intrinsics for another architecture. For example, the AArch64 back-end will throw an error for IR containing `llvm.x86.xbegin()` because it does not know how to custom-lower an intrinsic for another architecture. Therefore because the migration point implementations are ISA-specific, the IR-level passes in the Popcorn compiler that insert migration points are refactored to split the process into two pieces. The first piece occurs during the middle-end, similarly to how the compiler was originally structured. This pass is responsible for *selecting* but not inserting migration points. In this pass, the compiler runs all of the aforementioned analyses, estimating when extra migration points should be inserted to avoid spurious transaction aborts due to capacity or conflict misses in the HTM buffer. The pass tags selected migration points with IR metadata indicating that a particular instruction is a migration point and how the instruction should be refactored by the back-end, i.e., insert call-out, add transaction begin/end.

LLVM structures back-end execution as a series of ISA-specific lowering passes. At the very beginning of back-end execution, the Popcorn compiler inserts the second half of the migration point instrumentation passes, allowing them to work on the IR before it gets lowered to machine code. This pass identifies program locations tagged by the previously described

selection pass and rewrites the IR to implement ISA-specific migration point instrumentation (e.g., Figure 7.2). Because at this point the compiler has transitioned to ISA-specific lowering, the pass is free to insert HTM intrinsics or migration call-outs for ISAs that do not support HTM. As an implementation artifact of splitting migration point insertion into two halves, the pass that inserts stack-maps into the IR is also moved from the middle-end to the back-end because the migration point instrumentation inserts function calls to the migration library.

7.4 Evaluation

We evaluated instrumenting applications with HTM versus inserting more frequent call-outs to the migration library. We ran experiments to answer the following questions:

- How much overhead is added to applications when instrumented to run under HTM versus adding extra call-outs to the migration library? (Section 7.4.1)
- What is the migration response time latency for the different types of instrumentation? (Section 7.4.2)

Experimental Setup. We ran experiments on an Intel Xeon 2620v4 CPU and an IBM POWER8 CPU. The Xeon has 8 cores and 16 hardware threads, with a clock speed of 2.1 GHz (3.0 GHz boost). The POWER8 has 8 cores and 64 hardware threads, with a clock speed of 2.0 GHz (3.0 GHz boost). The Xeon has a 64-byte conflict detection granularity and uses the L1 cache for store buffering and conflict detection, although past works suggest that the total load and store capacities are 4MB and 22KB, respectively [143]. The POWER8 has a 128-byte conflict detection granularity and has an 8KB capacity for both loads and stores due to the use of a CAM attached to the L2 cache [143]. We used the single-threaded versions of benchmarks from the C version of the NAS Parallel Benchmarks [23, 175] to evaluate the instrumentation.

7.4.1 Overhead

In order to understand the cost of each of the mechanisms, we first measured how much overhead is added by the instrumentation at a single migration point. On the Xeon, a call-out to the migration library takes ˜80 nanoseconds. This includes the cost of the function call procedure and the cost of the system call to check the flag in the kernel to see if the scheduler has requested a migration. Call-outs on the POWER8 add a similar amount of overhead. On the Xeon, the HTM instrumentation at a migration point (i.e., HTM end/HTM begin) takes ˜18.6 nanoseconds, approximately 40 – 55 cycles. The POWER8 shows a similar latency –

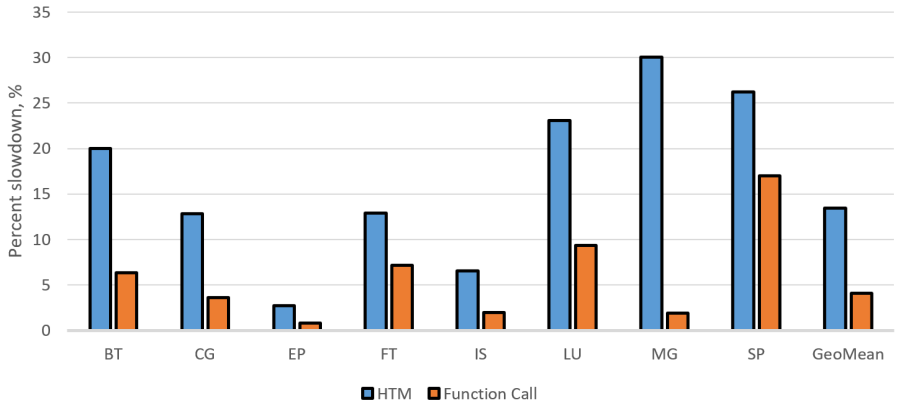


Figure 7.5: Overhead of instrumentation for HTM and inserting extra migration library call-outs

˜20.1 nanoseconds or approximately 40 – 60 cycles. Thus using HTM for migration point instrumentation incurs a lower per-migration point latency.

However, the true cost is the number of migration points added into applications. For HTM instrumentation, the compiler must insert enough migration points so as to avoid spurious aborts and thus wasted computation. This often leads to much higher overheads as the migration point granularity must be dramatically increased. Figure 7.5 shows the overhead for NPB applications for each of the different types of instrumentation on the Xeon. The y-axis shows the overhead added by instrumentation versus running the application without any instrumentation. As shown in the figure, HTM instrumentation add a significant amount of overhead, a geometric mean of 13.45%. In contrast, adding more frequent call-outs to the migration library adds a geometric mean 4.07% overhead, demonstrating that call-outs are relatively inexpensive. Several benchmarks experience large overheads with HTM instrumentation. BT, LU, MG and SP all contain tight loops with memory access patterns that cause a high rate of conflict and capacity cache misses, meaning the compiler must instrument the code at a fine granularity in order to avoid spurious HTM aborts. With a high number of transaction begin and end instructions, these applications each experience over a 20% slowdown versus uninstrumented execution.

It is also worth noting that the Intel optimization manual [102] states that transactional execution incurs some implementation-specific latency throughout the execution of the transaction:

> "There is an additional implementation-specific overhead associated with executing a transactional region. This consists of a mostly fixed cost in addition to a variable dynamic component. The overhead is typically amortized and hidden behind the out-of-order execution of the microarchitecture. ... The overhead of commits is reduced with processors based on the Broadwell microarchitecture."

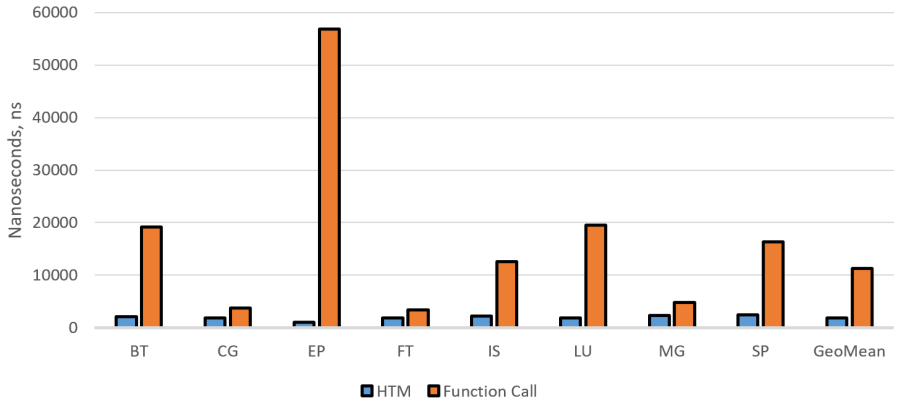


Figure 7.6: Median migration request response time

Although it is difficult to quantify the impact of this overhead, it most likely adds additional latency to the application's execution

7.4.2 Migration Response Time

Figure 7.6 shows the median migration request response time for the same applications and the same types of instrumentation (we present the median migration response time due to noise in results). Although using HTM to implement instantaneous migrations adds significant overhead, Figure 7.6 shows that it does provide a dramatic improvement in response time. It takes applications a geometric mean 1.9 microseconds to respond to migration requests when using HTM versus a geometric mean of 11.3 microseconds when inserting extra call-outs, a 5.92x improvement. This shows the efficacy of using HTM – applications roll-back immediately to the most previously encountered migration point, giving the scheduler fine-grained control over application placement.

However, these results bring into question what should be considered an acceptable migration response time. Although using HTM provides a significantly lower response time, adding extra migration library call-outs still provides latencies on the order of tens of microseconds with significantly less overhead. Typically, web servers target quality of service requirements on the order of milliseconds [139], meaning that migration response times on the order of microseconds are still several orders of magnitude smaller and give the scheduler enough control to optimize application placement. Because of these results, we conclude that using HTM to enable instantaneous migration does not provide enough benefit to outweigh the cost of instrumentation overhead. Simply inserting additional migration points provides a high enough granularity for schedulers to place applications in heterogeneous-ISA systems with low overhead.

7.5 Discussion

In order for an HTM-based approach for enabling instantaneous migration to become feasible, HTM implementations must overcome the significant number of transactional aborts due to, among other reasons, limited buffer capacity. In particular, CPUs must implement larger store buffers (e.g., use the L2 cache which provides 256KB per core on the Xeon) and avoid spurious aborts caused by memory access patterns (e.g., filling up the ways of a particular cache set). In addition, being able to turn off particular aspects ot transactional execution that are unnecessary may also improve performance. For example, POWER's rollback-only transactions only perform store-buffering and do not detect conflicts between threads. This mechanism is meant for speculative optimization of single-threaded applications [143], but applies equally to the migration instrumentation use case. The applications targeted by Popcorn Linux do not use HTM for concurrency control, and therefore do not need to detect inter-thread conflicts. Using HTM to implement instantaneous migration only requires the ability to buffer memory writes and roll back execution. These types of implementation improvements would allow HTM to become more generally useful and significantly reduce the overhead from large numbers of transaction begin and end instructions required to avoid spurious aborts.

Chapter 8

Scaling OpenMP Across Non-Cache-Coherent Domains

One of the main advantages of extending the POSIX shared memory programming model to non-cache-coherent systems is that developers can leverage existing abstractions and APIs built upon this programming model. For example, OpenMP [38] is a parallel programming model built upon shared memory semantics that allows developers to easily write multi-threaded applications by annotating source code with OpenMP `pragmas`. The compiler converts OpenMP annotations into API calls to an OpenMP runtime, which creates teams of threads to execute and synchronize parallel execution. Popcorn Linux transparently supports OpenMP execution across systems composed of multiple cache coherence *domains*, i.e., a set of cache-coherent CPUs. Figure 8.1 illustrates such a system, where CPUs inside a domain have hardware cache coherency and Popcorn Linux's OS provides memory consistency between domains. Because Popcorn Linux's page consistency protocol provides cross-domain sequential consistency, developers can utilize OpenMP to take advantage of the processing power of all domains by distributing threads to all CPUs participating in the single system image. However, software-provided memory consistency can cause large overheads due to page-level false sharing [194], synchronization using atomic operations on shared memory [105], or even normal cross-domain memory accesses. In this chapter, we describe the design and implementation `libopenpop`, an OpenMP runtime optimized for running multi-threaded applications across multiple incoherent domains. Additionally we identify several OpenMP anti-patterns that cause excessive overheads in multi-domain systems and how OpenMP usage can be optimized to remove those overheads.

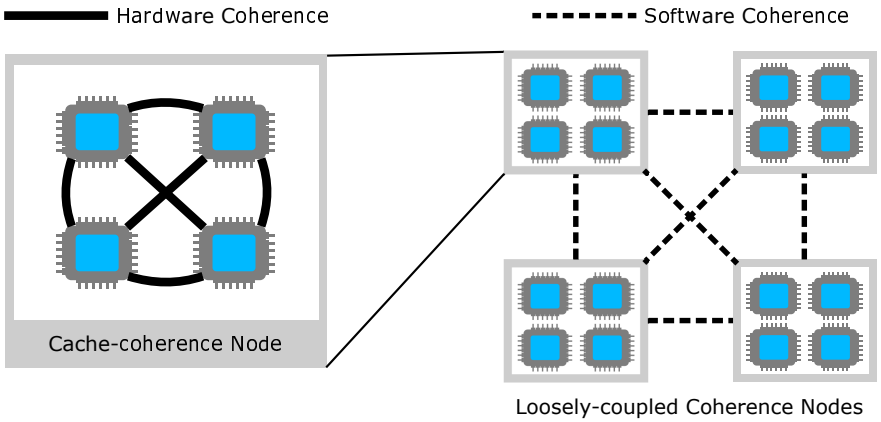


Figure 8.1: System composed of non-cache-coherent domains. Within a domain, the hardware provides cache coherency and a consistent view of memory. Between domains, however, software (e.g., user-developed, runtime, OS) must provide memory consistency – for `libopenpop`, Popcorn Linux's page consistency protocol provides sequential consistency between domains.

8.1 Profiling Software Memory Consistency Overheads

Data pages are migrated on demand between domains and mapped with different access permissions to optimize for locality similarly to a cache coherence protocol. Popcorn Linux uses a *multiple-reader/single-writer* protocol [169], which enforces sequential consistency between domains. Multiple domains may have read-only copies of a data page and may access it in parallel, but domains must acquire exclusive access to a page in order to write to it. Domains must invalidate other copies of the page before they may gain exclusive access, preserving the single-writer invariant. Cross-domain memory accesses take on the order of tens of microseconds versus tens or hundreds of nanoseconds for regular DRAM accesses [68], meaning excessive cross-domain memory accesses can cause large overheads.

Understanding which parts of an application cause significant DSM traffic allows developers to find and fix sub-optimal cross-domain memory access patterns. Popcorn Linux's OS exposes the DSM traffic through the kernel `ftrace` [3] mechanism to help developers understand how the page coherency protocol is behaving during execution. In particular, for each page fault (i.e. cross-domain page transfer) caused by the application, the DSM layer will emit a tuple with the following fields:

1. Time of the page fault
2. ID of the originating domain of the page fault
3. ID of the thread that caused the page fault
4. Type of memory access, i.e., read or write

5. Address of program instruction that caused the page fault
6. Memory access address that caused the fault

These tuples can be used to correlate page faults back to the application, giving users insights as to what parts of the application are causing cross-domain DSM traffic. Users first build the application with debugging information and then execute a profiling run, saving the `ftrace` output to a log file. Then, users supply the application binary and the log to a tool which parses the log and emits information about the application's behavior. Using the debugging information contained in the binary, the tool can identify which source code locations cause the most page faults, accesses to which program data causes the most faults and page fault frequency over time during the application's execution. The tool can also differentiate between read and write faults, helping the developer connect memory access patterns from different parts of the application, e.g., writes faults in one part of the application that cause read faults in another. Thus, this tool allows developers to focus on sub-optimal memory access patterns and optimize them for cross-domain execution. The tool was heavily used to detect and optimize `libopenpop`'s execution, including thread team initialization and synchronization. Additionally it was used to determine OpenMP anti-patterns within applications that cause excessive overheads.

8.2 Design of a Distributed OpenMP Runtime

`libopenpop` leverages Popcorn Linux's transparent thread migration and distributed shared memory to allow application developers to use a well-studied and simple shared memory parallel programming model in multi-domain systems. `libopenpop` handles migrating threads between domains inside the OpenMP runtime, giving developers the option of configuring where to place threads. Because the DSM is implemented transparently by the OS, existing OpenMP applications can execute across domains unmodified and the OS will migrate data pages to the accessing domain on demand. Because of the aforementioned page consistency protocol overheads, care must be taken when placing data across domains in the system in order to minimize permission update and page transfer traffic. `libopenpop` is designed to be domain-aware, organizing execution and data so as to minimize cross-domain communication by placing threads into a per-domain hierarchy during team startup (Section 8.2.1). During subsequent execution, `libopenpop` breaks OpenMP synchronization primitives into local and global components (Section 8.2.2). Even though this design implements OpenMP abstractions more efficiently than a domain-unaware runtime, the user can have a significant impact on performance. Several OpenMP anti-patterns are discussed, including how they can be refactored to optimize cross-node execution (Section 8.2.3).

```
int vecsum(const int *vec, size_t num) {
    size_t i;
    int sum = 0;
#pragma omp parallel for reduction(+:sum)
    for(i = 0; i < num; i++) sum += vec[i];
    return sum;
}
```

Listing 8.1: OpenMP vector summation. OpenMP directives instruct the runtime to spawn threads, distribute loop iterations to threads for execution and combine results from each thread. Additionally, there is an implicit end of region barrier.

8.2.1 Distributed OpenMP Execution

OpenMP consists of a set of compiler directives and runtime APIs which control creating *teams* of threads to execute code in parallel. In particular, the developer annotates source code with OpenMP pragmas, i.e., `#pragma omp`, which direct the compiler to generate parallel code regions and runtime calls to the OpenMP runtime. Developers spawn teams of threads for parallel execution by adding `parallel` directives to structured blocks, which the compiler outlines and calls through the runtime. Additionally, OpenMP specifies pragmas for *work-sharing* between threads in a team (e.g., `for`, `task`) and synchronization primitives (e.g., `barrier`, `critical`) among other capabilities. Listing 8.1 shows an example of parallelizing vector sum with OpenMP. The `parallel` directive instructs the compiler and runtime to create a team of threads to execute the for-loop. The `for` directive instructs the runtime to divide the loop iterations among threads in the team. The `reduction` clause informs the runtime that threads should sum array elements into thread-local storage (i.e., the stack) and accumulate the value into the shared `sum` variable at the end of the work-sharing region. Finally, the `parallel` and `for` directives include an implicit ending barrier.

Internally, OpenMP functionality is implemented by a combination of compiler transformations and runtime calls. The compiler outlines parallel blocks into separate functions and inserts calls to a "parallel begin" API to both fork threads for the team and call the outlined function. Other directives are also implemented as API calls – a `for` directive is translated into a runtime call which determines the lower and upper bounds of the loop iteration range for each thread and synchronization primitives are implemented as calls into the runtime to wait at a barrier or execute a critical section. While the developer can very easily parallelize and synchronize team threads using these pragmas, their implementation can drastically affect performance. OpenMP assumes a homogeneous memory hierarchy, where accesses to global memory are relatively uniform from all compute elements in terms of latency. However for multi-domain systems this assumption is broken and accesses to arbitrary global memory (e.g., reducing data or waiting at barriers) can cause severe performance degradations. In order to minimize cross-domain traffic, `libopenpop` refactors the OpenMP runtime to break functionality down into *local* (intra-domain) and *global* (inter-domain) execution.

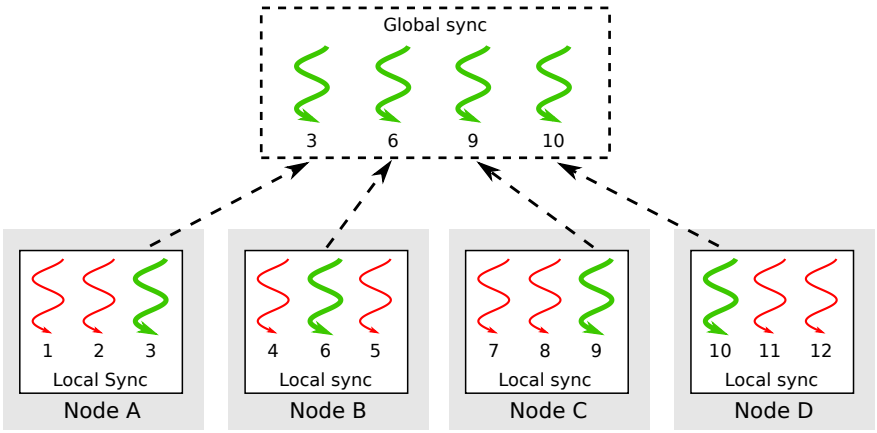


Figure 8.2: `libopenpop`'s thread hierarchy. In this setup, `libopenpop` has placed 3 threads (numbered 1-12) on each node. For synchronization, threads on a node elect a leader (green) to represent the node at the global level. Non-leader threads (red) wait for the leader using local synchronization to avoid cross-node data accesses.

Initializing thread teams. To begin a parallel region, the OpenMP runtime forks team threads which call the outlined parallel region to begin execution. During team startup, `libopenpop` creates a logical *thread hierarchy* to break OpenMP functionality into local and global computation. Threads operate on per-domain data structures whenever possible to avoid cross-domain data transfers. The first place this is utilized is when the runtime communicates parallel region startup information to threads. When the main thread starts a new parallel region, it must communicate both the outlined function and other execution state (references to shared variables, work sharing data structures, etc.) to all threads executing the new parallel region. Most OpenMP runtimes copy this information directly into each thread's thread local storage. However in a DSM-based system, this incurs two transfers for each thread – one for the main thread to write the startup data and one for each thread to read the data. Because this information is common to all threads in the team, `libopenpop` instead sets this data once per domain and threads synchronize per-domain to consume this information (see "Synchronizing Threads" below).

Migrating Team Threads. OpenMP provides facilities for specifying where threads should execute. In particular, OpenMP v4 [38] describes a method for mapping threads to physical "places" like sockets, cores and hardware threads. `libopenpop` extends this capability with a "`domains`" keyword that allows users to transparently distribute threads across domains, while internally initializing the thread hierarchy to match. `libopenpop` parses the places specification at application startup. Threads forked at the beginning of a parallel section enter a generic startup function inside `libopenpop` where the runtime applies the placement specification to migrate threads from the *origin* to *remote* domains according to the user specification. `libopenpop` calls into the kernel's thread migration service to transparently migrate to new domains. Threads execute as if they had never left the origin – data is brought

over on-demand and kept consistent using the DSM layer. Post-migration, threads call into the outlined parallel region and execute as if on a single shared memory machine. At the end of execution, threads migrate back to the origin for cleanup. Thus, developers can distribute threads across domains without changing a single line of code within the application – `libopenpop` encapsulates all the machinery necessary for interacting with the OS to migrate threads between domains. This also gives the runtime flexibility to redistribute threads between domains if needed; for example, to co-locate threads accessing the same memory.

Synchronizing threads. In order to facilitate optimizations listed in Section 8.2.2, `libopenpop` logically organizes threads into local/global hierarchy for synchronization. This enables optimizations that mitigate cross-domain traffic. `libopenpop` uses a *per-domain leader selection* process whereby a leader is selected from all threads executing on a given domain to participate in global synchronization (all other non-leader threads synchronize within a domain). As illustrated in Figure 8.2, this allows `libopenpop` to reduce contention while providing the same semantics as a normal synchronization. `libopenpop` provides two types of selection processes depending on whether a *happens-before* ordering is required:

1. **Optimistic selection.** The first thread on a domain to arrive at the synchronization point is selected as the domain's leader. The leader executes global synchronization while other threads on the domain continue in parallel, allowing all threads to perform useful work without blocking. After a global synchronization, leaders communicate results with local threads. This is useful for synchronization which does not require any ordering, e.g., `reduction` operations and grabbing batches of loop iterations using OpenMP's `dynamic` loop iteration scheduler.

2. **Synchronous selection.** The last thread to arrive at the synchronization point is selected as the leader and the last per-domain leader to arrive at the global synchronization point performs any global work required. This is useful for synchronization which requires a *happens-before* ordering, e.g., for `barrier`s all threads must arrive at the synchronization point *before* any threads are released.

8.2.2 Optimizing OpenMP Primitives

By controlling thread distribution across domains and organizing threads into a hierarchy, `libopenpop` can reduce several sources of cross-domain overhead. First, accessing remote memory takes two orders of magnitude longer than local DRAM accesses, meaning `libopenpop` organizes as much computation as possible to be performed locally. Second, the DSM layer operates at a page granularity which can cause pages to "ping pong" when threads on multiple domains access the same or discrete data on the same page, known as "false sharing". Logically organizing memory into per-domain partitions can yield large speedups. In this section we describe compiler and runtime optimizations to reduce these two sources of overhead.

```
struct barrier_t PAGE_ALIGN
{ int rem, total, sleep_key; };
struct hybrid_barrier_t
{ barrier_t local[NUM_NODES], global; };

void barrier_wait(hybrid_barrier_t *bar) {
    int thr_rem = atomic_sub(&bar->local.rem,1);
    if(thr_rem) {
        /* Spin a while before sleeping,
           return if all threads arrive */
        if(do_spin(bar)) return;
        /* Sleep until all threads arrive */
        else sleep(&bar->local.sleep_key);
    } else { /* Per-node leader */
        global_barrier(bar);
        bar->local.rem = bar->local.total;
        wake(&bar->local.sleep_key);
    }
}

void global_barrier(hybrid_barrier_t *bar) {
    int n_rem = atomic_sub(&bar->global.rem,1);
    /* Sleep until all leaders arrive */
    if(n_rem) sleep(&bar->sleep_key);
    else { /* Wake up other leaders */
        bar->rem = bar->total;
        wake(&bar->sleep_key);
    }
}
```

Listing 8.2: Hierarchical barrier pseudocode. Threads call into `barrier_wait()` and check to see if they are the last thread to arrive for the domain. If not, they wait at the local barrier, only ever touching data already mapped to the domain. If they are the last thread, they are elected the domain's leader and synchronize at the global barrier.

Hierarchical Barriers. OpenMP makes extensive use of barriers for synchronization at the end of many directives. For many OpenMP runtimes, barriers are implemented using a combination spin-sleep approach where threads spin until some condition becomes true, or sleep if it does not within some fixed interval. While suitable for shared memory systems where there are few threads and cache-line contention is relatively cheap, this form of synchronization causes enormous overheads in multi-domain systems as many threads executing in different domains spin-wait, causing the DSM layer to thrash. `libopenpop` avoids this by using hierarchical local/global barriers. A hierarchical barrier consists of a local spin-wait barrier for each domain and one top-level global barrier as shown in Listing 8.2. Threads use a synchronous selection process to pick a per-domain leader; all threads not selected wait at their respective local barriers. A synchronous selection process is required in order to estab-

lish a *happens-before* relationship between all threads arriving at the barrier on all domains and the global barrier release. Otherwise, threads could be released on individual domains before all threads executing on all domains had reached the barrier. The per-domain leaders wait at the global barrier for all domains to arrive. Threads entering the global barrier, as shown in Listing 8.2, do not spin but instead do a single atomic operation, which reduces cross-domain contention on the global barrier's state[1]. Once all leaders reach the global barrier, they are released and join the local barriers to release the rest of the threads. Note that all barriers, both local and global, are placed on separate pages to avoid cross-domain contention.

Hierarchical reductions. Similarly, reductions can also be broken down into local and global computation. OpenMP requires that reductions are both associative and commutative [38], meaning they can be performed in any order and thus do not require a happens-before relationship. `libopenpop` uses an optimistic leader selection process to pick per-domain leaders to reduce data for each domain. The leader waits for threads to produce data for reducing, allowing threads to execute in parallel while it performs the reduction operation. Once the leader has reduced all data from its domain, it makes the domain's data available for the global leader (which is also selected optimistically). The global leader pulls data from each domain for reduction, producing the final global result. The hierarchy again reduces cross-domain traffic as reduction data is only transferred once per domain.

Moving Shared Variables to Global Memory. OpenMP describes a number of *data-sharing attributes* which describe how threads executing parallel regions access variables in enclosing functions. Developers can specify variables as *private*, meaning all threads get their own copy of the variable, or *shared*, meaning all threads read and write the same instance of the variable. For shared variables, the compiler typically allocates stack space on the main thread's stack and passes a reference to this storage to all threads executing the parallel region. In a multi-domain setting this leads to false sharing as threads reading/writing the shared variables contend with the main thread as it uses its stack for normal execution. To avoid this situation we modified `clang` to copy shared variables to global memory for the duration of the parallel region so that threads accessing these variables do not access the main thread's stack pages. Copying shared variables to and from global memory could cause high overheads in situations with many and/or large shared variables. However, we did not find this situation in the benchmarks we evaluated.

Future optimizations. Similar to these primitives, other synchronization and work sharing primitives such as `critical` directives and `task` work-sharing regions can be refactored to use a leader selection process to reduce inter-domain traffic. We leave these engineering optimizations as future work.

[1] Global synchronization could be further optimized with new kernel-level multi-domain primitives

```
/* Sub-optimal - start many parallel regions */
for(j = 0; j < NUM_RUNS; j++) {
#pragma omp parallel for private(i, price, priceDelta)
    for(i = 0; i < numOptions; i++)
        ...(compute)...
}

/* Better - separate parallel and work-sharing directives */
#pragma omp parallel private(i, price, priceDelta)
for(j = 0; j < NUM_RUNS; j++) {
#pragma omp for
    for(i = 0; i < numOptions; i++)
        ...(compute)...
}
```

Listing 8.3: `blackscholes` parallel region optimization. Rather than starting many parallel regions, users should start fewer regions with multiple work sharing regions.

8.2.3 Using OpenMP Efficiently

When developing `libopenpop` we discovered several OpenMP usage patterns that cause sub-optimal behavior in multi-domain settings. Many of these sources of overhead can be rectified by small code modifications. Here we detail how developers can avoid these overheads.

Remove excessive parallel region begins. Each `parallel` directive causes the compiler to generate a new outlined function and the runtime to start a new thread team to execute the parallel region. While most OpenMP runtimes maintain a thread pool to avoid overheads of re-spawning threads, each encountered parallel region causes communication with the main thread, e.g., passing function and argument pointers to team threads in order to execute the region. Even with the previously described per-domain team start optimization, this can cause high overheads for applications that start large numbers of parallel regions. As shown in Listing 8.3 for the `blackscholes` benchmark, users should lift parallel directives out of loops to avoid these initialization overheads wherever possible.

Access memory consistently across parallel regions. Cross-domain execution overheads are dominated by the DSM layer, and thus data placement in the system. In order to minimize data movement, threads should use the same data access patterns when possible to avoid shuffling pages between domains. Listing 8.4 shows an example from `cfd` where a copy operation accesses memory in a different pattern from the compute kernel. The optimized version (`copy_distributed()`) instead copies data using the same access pattern.

Use `master` instead of `single` directives. OpenMP provides a number of easy-to-use synchronization primitives, include `master` directives which specify only the main thread should execute a block of code, and `single` directives, which specify only a single thread should execute a block of code (not necessarily the main thread). Users should substitute `single`

```
void compute_step_factor(...) {
#pragma omp parallel for
    for(int blk=0; blk < nelr/blength; ++blk) {
        int b_start = blk * blength,
            b_end = (blk + 1) * blength;
        for(int i = b_start; i < b_end; i++) {
            float density = variables[i+...];
            ...(compute)...
        }
    }
}

/* Sub-optimal - does not access arrays in same way as compute_step_factor() */
void copy(...) {
#pragma omp parallel for
    for(int i = 0; i < N; i++)
        old_var[i] = variables[i];
}

/* Better - use same memory access pattern */
void copy_distributed(...) {
#pragma omp parallel for
    for(int blk=0; blk < nelr/blength; ++blk) {
        int b_start = blk * blength,
            b_end = (blk + 1) * blength;
        for(int i = b_start; i < b_end; i++) {
            old_var[i+...] = variables[i+...];
            ...(other copying)...
        }
}
```

Listing 8.4: `cfd` memory access optimization. Threads should access memory consistently across all parallel regions where possible.

for `master` directives when possible. `single` directives require two levels of synchronization – the first thread to encounter the single block executes the contained code while other threads skip the block and wait at an implicit barrier. This functionality is implemented by atomically checking if a thread is the first to arrive. However this synchronization operation requires cross-domain traffic, leading to significant overheads. Users should utilize `master` and `barrier` directives together to implement the same semantics. The `master` directive specifies that only the main thread should execute a code block and requires no synchronization (threads maintain their own IDs). Thus, users get the same functionality with less overhead.

8.3 Implementation

`libopenpop` extends and optimizes GNU's `libgomp` [87] v7.2, an OpenMP implementation packaged with `gcc`. For the optimizations that require compiler-level code generation changes, we modified `clang/LLVM` v3.7.1 due to its cleaner implementation versus `gcc`. However, `clang` emits OpenMP runtime calls to LLVM's `libiomp` [190], a complex cross-OS and cross-architecture OpenMP implementation with 3 times more lines of code versus `libgomp`. We opted for simplicity and added a small translation layer (∼400 lines of code) to `libopenpop` to convert between the two. Note that `libopenpop`'s design is not tied to the choice of either compiler or OpenMP runtime – we chose this particular combination simply for ease of implementation.

In addition to the runtime changes, we added a small memory allocation wrapper around `malloc` inside of musl that organizes memory allocations into per-domain heaps, similar in spirit to arena or region-based memory allocators [82]. During a call to `malloc`, the wrapper identifies in which region the thread is currently executing and steers the memory allocation request to the corresponding heap. This allows `libopenpop` to remove another source of false sharing – when threads on separate domains allocate data on the same page they can cause a large amount of contention and unintentionally bottleneck execution. The wrapper also supports moving the allocation another domain-specific heap through the `realloc` call.

Chapter 9

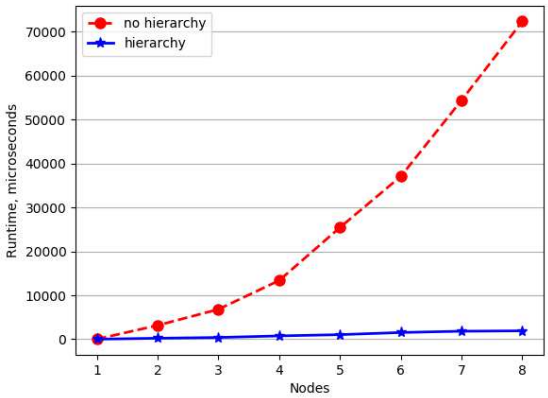
Scale-out OpenMP Evaluation

In this chapter we evaluate `libopenpop` as described in Chapter 8 on a small cluster as a representative multi-domain setup (the term "node" is used interchangeably with domain to indicate a server in the cluster). The evaluation investigates where `libopenpop` currently provides good performance and areas of improvement for future research. In particular the evaluation answers the following questions:

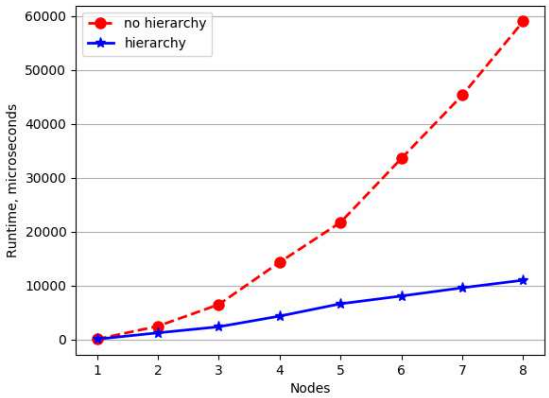
- How do the OpenMP runtime optimizations described in Sections 8.2.1 and 8.2.2 scale to multiple domains? (Section 9.1)
- How do applications perform both with and without the optimizations described in Section 8.2.3 when scaled to multiple domains? (Section 9.2)
- What types of applications currently run well when using `libopenpop` and what types could benefit from future optimizations? (Section 9.3)

Experimental Setup. `libopenpop` is evaluated on a cluster of 8 Xeon servers, each of which contains 2 Intel Xeon Silver 4110 processors (max 2.1GHz clock) and 96 GB of DDR4-2667 MHz RAM. Each Xeon processor has 8 cores with 2-way hyperthreading for a total of 16 threads per processor, or 32 threads per server. `libopenpop` was evaluated with up to 16 threads per server due to implementation limitations in Popcorn Linux. Each server is equipped with a Mellanox ConnextX-4 Infiniband adapter supporting bandwidth up to 56 Gbps.

Applications. `libopenpop` was evaluated using OpenMP benchmarks from PARSEC [33], Rodinia [50] and NASA Parallel Benchmark [23, 175] suites. We selected a subset of benchmarks that 1) had enough parallel work to scale across multiple domains and 2) had representative performance characteristics from which we could draw conclusions about `libopenpop`'s effectiveness. Of note, we were able to survey cross-domain performance for 26 benchmarks by only re-compiling and re-linking using the Popcorn Linux infrastructure. We performed



(a) Latency to execute a single barrier for different numbers of domains with and without hierarchical barriers, in microseconds.



(b) Latency to execute a global reduction for different numbers of domains with and without hierarchical reductions, in microseconds.

Figure 9.1: Evaluation of OpenMP primitives with and without the thread hierarchy

an initial evaluation of benchmarks to determine scalability and contention points, then optimized applications as described in Section 8.2.3. We consider execution on a single server as the baseline, as comparing to other cluster programming solutions would require either significant application refactoring or complex compiler/runtime extensions to support the applications (one of the major benefits of Popcorn Linux and `libopenpop`).

9.1 Microbenchmarks

First we evaluated the effectiveness of the hierarchy in scaling multi-domain synchronization for several OpenMP primitives. The first microbenchmark we ran spawns 16 threads on each domain from 1 to 8 domains (no cross-domain execution for 1 domain) and executes 5000 barriers in a loop. Figure 9.1a shows the average barrier latency with and without hierarchy optimizations. Clearly `libopenpop`'s hierarchy provides much better scalability – a naïve implementation where all threads on all domains wait at a single global barrier leads to tens of millisecond latencies that drastically increase with domain count (up to 72.4 milliseconds for 8 domains). Meanwhile the hierarchical barrier leads to much better scalability with up to a 1.9 millisecond latency for 8 domains, a 38x speedup.

We next ran a microbenchmark that sums all the elements in an array to stress cross-domain parallel reductions. We again spawned 16 threads per domain and allocated 50 pages of data for each thread to accumulate – each thread received the same amount of work to remove load imbalance effects on reduction latencies. Figure 9.1b shows the latency when performing a naïve reduction (all threads use atomic operations on a global counter) versus a hierarchical

reduction (leaders first reduce locally and then globally). Similarly to barriers, hierarchical reductions have much better scalability than normal global reductions, taking 11 and 58 milliseconds on 8 domains, respectively. Interestingly, the performance gap on 8 domains between the normal and hierarchical reductions is only 5.4x. This is due to how the compiler implements reductions – the compiler allocates a thread-local copy of data to be reduced on each thread's stack and passes a pointer to that data to the runtime. Each per-domain leader passes that pointer to the global leader for reduction, which causes contention on the per-domain leader's stack page (global leader reads reduction data, per-domain leader uses stack for normal execution). Nevertheless, the hierarchy provides large performance benefits.

9.2 Benchmark Performance

Next, we ran benchmarks to evaluate the effectiveness of optimizations listed in Section 8.2.3. All benchmarks were run with hierarchical barriers and reductions enabled. Figures 9.2-9.7 show application performance when run with varying numbers of domains (x-axis) and threads per domain (trend lines). For example, when using 8 threads per node and 8 nodes, the benchmark was run with a total of 64 threads. The y-axis shows runtime of each configuration in seconds; lower numbers mean better performance. Dotted red lines indicate application performance before optimization while solid blue lines indicate running time after optimization.

Application performance falls into three categories: applications that scale with more domains (`blackscholes`, `EP`, `kmeans` and `lavaMD`), applications that exhibit some scalability but have non-trivial cross-domain communication (`CG`) and applications that do not scale (`cfd`). For applications that scale, running with the highest thread count on 8 domains led to a geometric mean speedup of 3.27x versus the fastest time on a single machine, or 4.04x not including `blackscholes` which has a significant sequential region. For CG, the fastest multi-domain configuration achieved a slowdown of 5.4%, meaning there is plenty of room for performance optimization. The optimizations described in Sections 8.2.3 help for every multi-domain configuration in every single application, although its effects are limited in those that do not scale. The scalable applications experience further performance gains by lifting shared variables into global memory and applying lightweight code modifications. `lavaMD` and `kmeans` experienced the largest benefits from optimizations, in particular lifting shared variables into global memory and using per-domain memory allocations.

9.3 Performance Characterization

Here we describe application characteristics that have a significant impact on performance and future directions for further optimization in Popcorn Linux and `libopenpop`.

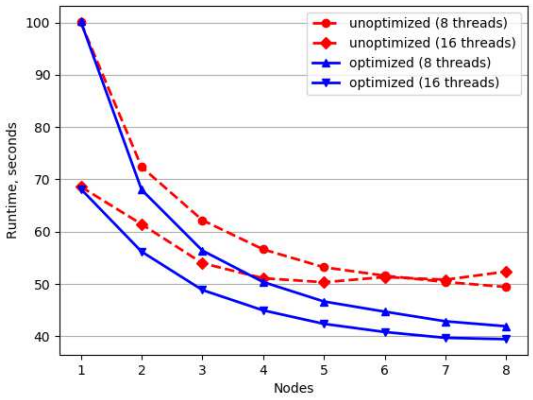


Figure 9.2: `blackscholes`

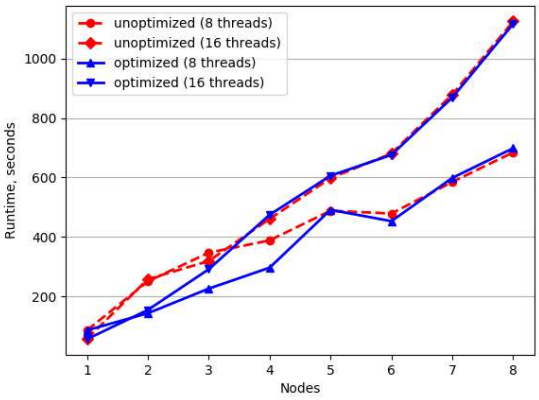


Figure 9.3: `cfd`

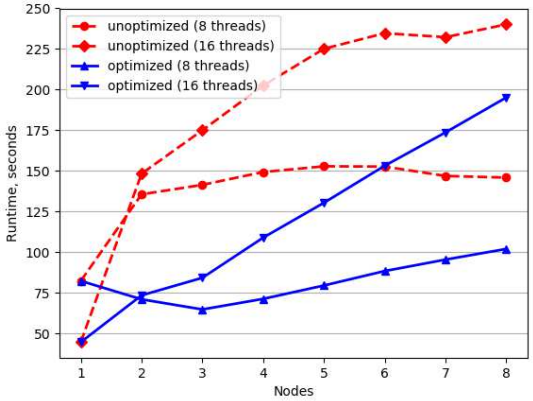


Figure 9.4: `CG` class C

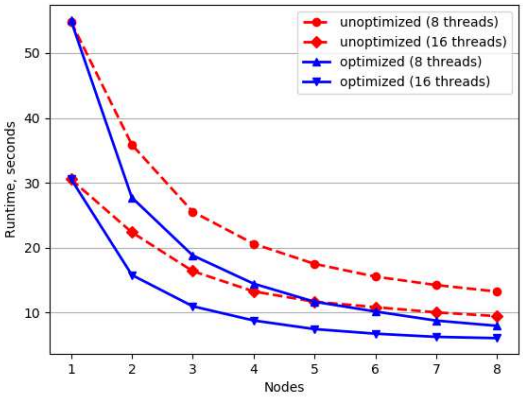


Figure 9.5: `EP` class C

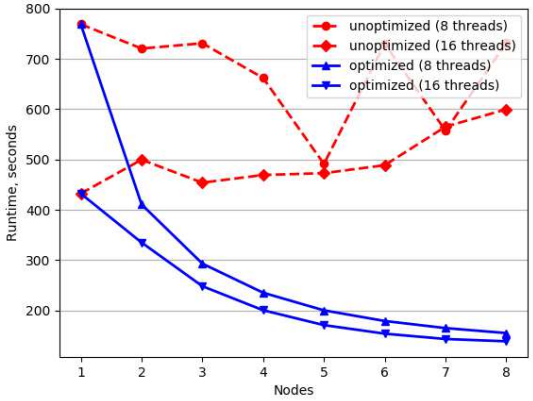


Figure 9.6: `kmeans`

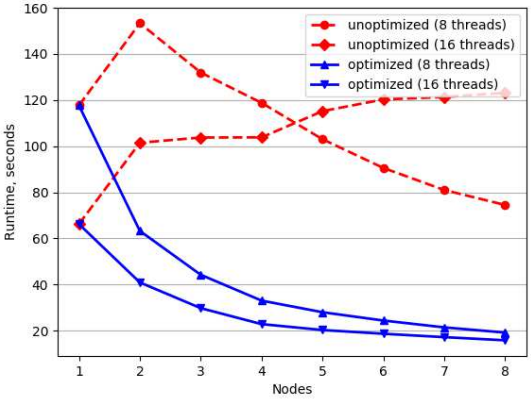


Figure 9.7: `lavaMD`

Scalable applications. These applications have little-to-no communication between threads on different domains and thus once the initial data exchange between domains has been completed, threads run at full speed without inter-domain communication. This is the ideal scaling scenario, but requires problems with large datasets that can be processed completely independently. Scalability is only limited by benchmark data size (`EP-C`, `kmeans`, `lavaMD`) or serial portions within the application (`blackscholes`).

Mildly-scaling applications. These applications share non-trivial amounts of data between threads during execution. For example, `CG-C` uses arrays of pointers to access sparse matrices through indirection, meaning there is little data locality when accessing matrix elements. We believe that prefetching up-to-date copies of data across domains could significantly improve performance for these types of applications.

Non-scalable applications. These applications have many small parallel regions, low compute-to-memory access ratios and continually shuffle pages between domains. `cfd` iteratively scans one variables array and writes to a second node-local array. In the next iteration, these two array are swapped, leading to huge DSM layer overheads as writes must be propagated among domains and reads that were replicated across domains must be invalidated. There is not enough computation to amortize the cost of shuffling data. Instead, application developers would need to find alternate sources of parallelism, i.e., performing several of the computations in parallel. This could be achieved through nested OpenMP parallel regions; we leave implementing this functionality within the thread hierarchy as future work.

From the previously described characteristics, the main performance limitation was attributed to cross-domain data shuffling in and between work-sharing regions. In order to further investigate system bottlenecks, we evaluated how much network bandwidth the DSM layer was able to utilize by running `cfd` on 2 domains with 32 threads and capturing the number of pages transmitted in one second intervals to determine time varying bandwidth usage. Throughout the parallel portion of the application the messaging layer used on average 85.2 MB/s of bandwidth, close to two orders of magnitude less than 56 Gbps Infiniband can provide. This leads us to believe that future efforts should focus on how to use the ample available cross-domain network bandwidth in order to better hide cross-domain memory access latencies.

9.4 Future Work

There are numerous opportunities for future research with `libopenpop`. As previously mentioned, cross-domain memory access latencies cause severe overheads for applications that shuffle large amounts of data between domains. This can be attributed to two main factors: 1) distributed shared memory consistency overheads, and 2) on-demand data migration. In terms of DSM overheads, enforcing sequential consistency across domains can block memory operations even when there is no data transfer involved. For example, in order to write to

a page on a domain, Popcorn Linux's DSM protocol first must invalidate permissions on all other domains and then acquire write permissions (along with page data). Even when the domain has the most recent page data (for example, after reading the page), a domain must first invalidate permissions on other domains before allowing threads write access due to sequential consistency semantics. However, OpenMP uses a release consistency model [38], meaning that Popcorn Linux's protocol provides stricter guarantees than is necessary for correct OpenMP semantics. Relaxing the DSM layer's consistency would eliminate much of the memory consistency maintenance overheads.

The second source of latency is due to the use of on-demand data migration. The DSM implementation observes memory accesses through the page fault handler and migrates data at the last possible moment. While this avoids migrating unused data, it places the data transfer latency directly in the critical path of execution. As mentioned previously, the interconnect provides ample unused cross-domain bandwidth; techniques which leverage this bandwidth to preemptively place data can better hide cross-domain latencies. For example, the compiler could place data "push" hints that inform the DSM when a thread has finished writing a page so that it can be proactively pushed to other domains (similar in spirit to prefetching). Because OpenMP work sharing regions often structure memory accesses affine to loop iterations, the compiler could analyze memory access patterns in work sharing regions and inject data placement hints into the application.

Chapter 10

Heterogeneous OpenMP

Chapter 8 described how `libopenpop` optimizes OpenMP execution for non-cache-coherent systems, including how the OpenMP runtime is refactored to minimize cross-domain coherency traffic while maintaining a shared memory parallel programming model for developers. However, new mechanisms must be added to `libopenpop` in order to more efficiently utilize heterogeneous-ISA systems where CPUs exhibit different architectural designs and therefore diverse computational abilities. In this chapter we describe extensions to `libopenpop` that allow it to tailor parallel execution to the compute capabilities of heterogeneous-ISA systems.

In the context of such heterogeneous-ISA systems, `libopenpop`'s goal is to automatically determine where to place parallel computation across heterogeneous CPUs to maximize total system performance. `libopenpop` incorporates two new components into the OpenMP runtime to achieve this goal. First, it provides the mechanisms necessary to adjust how parallel work is distributed to CPUs in order to balance the workload across heterogeneous CPU cores; in OpenMP work-shared `for` loops, this involves adjusting how loop iterations are assigned to threads. Second, `libopenpop` automates work distribution decisions (i.e., assigning loop iterations to threads) by measuring communication and execution performance metrics and using those metrics to drive workload distribution decisions. In addition to extending existing schedulers for heterogeneous-ISA systems, `libopenpop` implements a new loop iteration scheduler, called the *HetProbe* scheduler, that uses performance metrics to decide whether to utilize cross-node execution or to only execute on a single node. By measuring and automating workload distribution decisions, `libopenpop` alleviates developers from having to manually configure applications for each new hardware setup, e.g., new CPUs or different types of interconnects.

10.1 Mechanisms for Heterogeneous-ISA Execution

Before deciding how much work should be given to each CPU in the system, `libopenpop` must first be able to configure how loop iterations are distributed to threads in work-shared `for` loops. In order to work with existing OpenMP applications, `libopenpop` must be compatible with existing semantics and therefore must abstract all mechanisms behind OpenMP runtime entry points. OpenMP already provides this capability by allowing developers to specify *loop iteration schedulers*, which define how loop iterations are assigned to threads executing a work-sharing region. The OpenMP standard defines several default schedulers [38], meaning the OpenMP compiler and runtime already provide the abstractions necessary to define alternate loop schedulers. `libopenpop` extends the default schedulers to provide new capabilities for heterogeneous-ISA systems and adds a new scheduler called the *HetProbe* scheduler for automating workload distribution decisions.

All of the features described in Chapter 8 are also used for heterogeneous-ISA systems. `libopenpop` migrates threads between domains when starting thread teams in order to execute OpenMP `parallel` regions across nodes. After thread migration, `libopenpop` internally separates runtime chores (work distribution, synchronization, performance monitoring) into per-node and global operations in order to minimize DSM traffic generated by the runtime itself. After team initialization, the runtime executes the `parallel` region. When encountering a work-shared `for`-loop, `libopenpop` must distribute loop iterations, including measuring application performance across and within nodes to adjust distribution decisions.

10.1.1 Cross-node Execution

When starting a `parallel` region, `libopenpop` organizes threads into the previously described thread hierarchy. At application startup in heterogeneous systems, however, `libopenpop` queries the system to determine each node's characteristics, such as type and number of CPUs available, and uses this information to initialize the thread hierarchy. For heterogeneous systems, the hierarchy may be unbalanced – for example, in a system containing a 16-core Xeon and a 96-core ThunderX, `libopenpop` spawns and places 16 and 96 threads, respectively, in each domain for a total of 112 threads. Internally, `libopenpop` initializes the same per-node data structures (barriers, reduction variables, etc.) but additionally creates per-domain loop iteration scheduler metadata structures. Because the runtime may wish to change workload distribution decisions after analyzing execution behavior, `libopenpop` allows re-configuring the thread hierarchy between parallel regions (but not within a parallel region). This allows the runtime to, for example, join the threads on an unused domain and only execute on a single domain if cross-node execution is determined to not be beneficial.

`libopenpop` uses the thread hierarchy for many types of synchronization, including barriers and reductions as previously described. Additionally, `libopenpop` uses the thread hierarchy for work distribution metadata. Figure 8.2 illustrates using the thread hierarchy for

synchronization, where green threads are elected as leaders to synchronize globally and non-leader threads synchronize locally. The leader/non-leader designation significantly reduces cross-node communication as most threads do not touch global data (in the Xeon/ThunderX setup, only 2 threads touch global data instead 112).

10.1.2 Workload Distribution

OpenMP defines several loop iteration schedulers that affect how iterations of a work-shared parallel loop are mapped to threads. The default loop iteration schedulers (e.g., `static`, `dynamic`) implement several strategies with the goal of evenly partitioning work to avoid overloaded straggler threads from harming performance. `libopenpop` provides the ability to distribute iterations across nodes by extending these schedulers to account for heterogeneity and to efficiently synchronize how threads grab iterations. Due to limitations in each (see below), `libopenpop` introduces the HetProbe scheduler for automatic iteration distribution in consideration of CPU and interconnect.

`libopenpop` assumes each node in the system contains a set of homogeneous CPU cores with identical micro-architecture and cache coherence. To quantify performance differences between nodes, `libopenpop` defines a *core speed ratio* (CSR) to rank the relative compute capabilities of individual CPU cores on one node versus another. For example, a Xeon core with a core speed ratio of 3:1 compared to a ThunderX core means the Xeon core is considered 3x faster than a ThunderX core and threads running on the Xeon will get 3x as many loop iterations as threads on the ThunderX. `libopenpop` assigns CSRs to each work sharing region, as applications may have multiple work sharing regions that each exhibit different performance characteristics on the same CPUs.

10.1.3 Cross-node static scheduler

OpenMP's `static` scheduler evenly partitions loop iterations among threads, assigning each thread the same number of iterations. The scheduler implicitly assumes all CPUs are equal and all loop iterations perform the same amount of work. Rather than considering all threads equal, `libopenpop` allows developers to specify per-node CSRs to skew the work distribution for threads on different nodes. The challenge, however, is that developers must manually discover the ideal CSR for each work sharing region and hardware configuration through extensive profiling and therefore limits its portability.

10.1.4 Cross-node dynamic scheduler

OpenMP's `dynamic` scheduler instructs threads to repeatedly grab user-defined batches of iterations from a global work pool, implemented by issuing atomic operations on a global

counter. This scheduler targets work sharing regions where individual loop iterations perform varying amounts of work. `libopenpop` optimizes grabbing batches using an optimistic leader selection through the thread hierarchy – threads first attempt to grab iterations from the node-local work pool instantiated during team setup. If the local pool is empty, the thread currently grabbing iterations is elected leader and transfers iterations from the global pool to the per-node pool. Because the leader represents the entire node, it grabs a batch of iterations for each thread executing on the node. This reduces the number of threads accessing the global pool and thus the amount of global synchronization required for work distribution. Note that while one thread is grabbing iterations from the global pool, other threads executing in the same node continue unblocked if they still have remaining work because no happens-before ordering is required when distributing loop iterations.

While not traditionally meant for load balancing on heterogeneous systems, the dynamic scheduler can load balance work distribution based on the compute capacity of CPUs in the system. However, continuous synchronization both at the local and global level to grab batches of work can negatively impact performance, especially with small batch sizes. Users must once again profile to determine the ideal per-region and per-hardware batch size. Non-deterministic mapping of loop iterations to threads can also cause "churn" in the DSM layer for applications that execute the same work sharing region multiple times. With a deterministic mapping of iterations to threads, data may settle after the first invocation of the work-sharing region as nodes acquire the appropriate pages and permissions. With the `dynamic` scheduler, however, loop iterations are assigned to different threads for each invocation and thus the data cannot settling on nodes.

The main problem with the default `static` and `dynamic` schedulers is that users must extensively profile to find the best workload distribution configuration in a large state space, i.e., determine CSRs or batch sizes for each individual work sharing region on every new heterogeneous platform. Additionally, if cross-node execution is not beneficial for a work sharing region due to large DSM overheads, users must profile to determine the best CPU for single-node execution and manually reconfigure the thread team (including the thread hierarchy) to only execute work-sharing regions on the selected CPU.

10.1.5 HetProbe scheduler

In order to avoid the tuning complexity of the previously mentioned schedulers, `libopenpop` introduces a new scheduler, called the heterogeneous probing or HetProbe scheduler, for automatically configuring execution of parallel computation. The HetProbe scheduler executes a small number of iterations across both nodes, called the *probing period*, during which it measures per-core execution time, cross-node page faults and performance counters to analyze a work sharing region's behavior (while still making forward progress on the computation). For work sharing regions with regular loops, i.e., all loop iterations perform constant amounts of work and have consistent memory access patterns, the scheduler uses

this information to distribute the remaining iterations as described in Section 10.2.

The HetProbe scheduler must be precise when distributing iterations for the probing period in order to accurately evaluate system performance. First, the scheduler issues a constant number of loop iterations to each thread, regardless of node, in order to compare the execution time of equal amounts of work on each CPU. Second, the scheduler must deterministically issue iterations, so that threads executing a work sharing region multiple times receive the same batch of iterations across invocations to account for the previously-mentioned data settling effect. If the HetProbe scheduler non-deterministically distributes probe iterations, data might unintentionally churn and cause falsely higher DSM overheads.

The probing period is configurable in order to tune its behavior to individual parallel regions. If too few iterations are used for probing, the probe measurements will be small and noisy (i.e., high variance). In particular, the network/DSM layer can add large and variable latencies, meaning the calculated core speed ratios will be inaccurate. Alternatively if too many loop iterations are used for probing, threads executing on faster cores will finish the probe early and waste time waiting for the slower cores to complete, leading to a load imbalance. Currently the HetProbe scheduler defaults to using 10% of the loop iteration range for probing, which we found to provide consistent measurement results while minimizing load imbalance effects.

`libopenpop` also implements a *probe cache* for applications which execute a work sharing region multiple times. This has two benefits – first, it allows the runtime to reuse results from previous probing periods to avoid probing overheads. The runtime simply reuses previously calculated statistics and workload distribution decisions. Second, `libopenpop` uses multiple probing results to smooth out measurement variations for shorter-running work sharing regions. `libopenpop` uses an exponential weighted moving average for measurement statistics, which favors more recent measurements and quickly converges on accurate values. `libopenpop` uses this type of average because initial probing values for regions may be inaccurate due to the DSM layer initially replicating data across nodes, whereas subsequent executions may incur fewer DSM costs.

10.2 Workload Distribution Decisions

The HetProbe scheduler uses the execution time, page faults and performance counters measured during the probing period to determine where to execute parallel work. Specifically, the HetProbe scheduler makes three decisions after executing the probing period:

1. Should the runtime leverage multiple nodes for parallel execution? While coupling together multiple CPUs provides more theoretical computational power, not all applications benefit from cross-node execution. As mentioned in Chapter 8 there is a significant cost for on-demand data marshaling and page coherency across nodes. To understand DSM overheads, we ran a microbenchmark that varies the number of compute operations

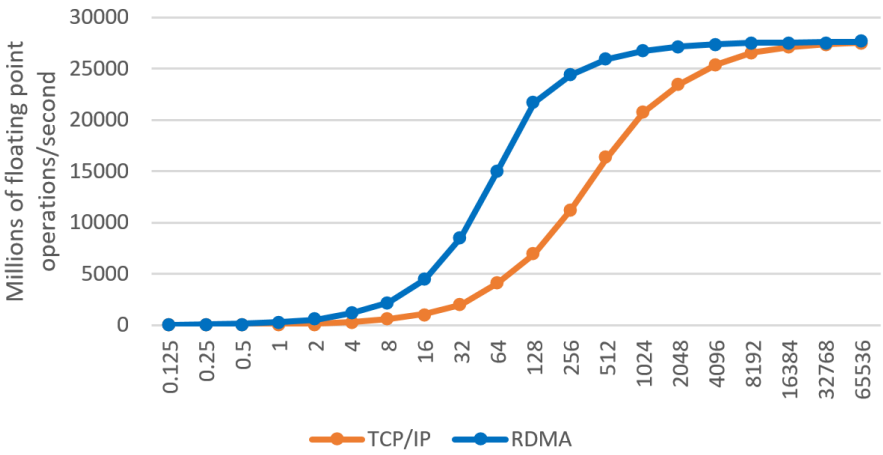
Description	Xeon 2620v4	ThunderX
Vendor	Intel	Cavium
Cores	8 (16 HT)	96 (2 x 48)
Clock (GHz)	2.1 (3.0 boost)	2.0
LLC Cache	L3 - 16MB	L2 - 32MB
RAM (Channels)	32 GB (2)	128 GB (4)
Interconnect	Mellanox ConnectX-4 56Gbps	

Table 10.1: Experimental setup.

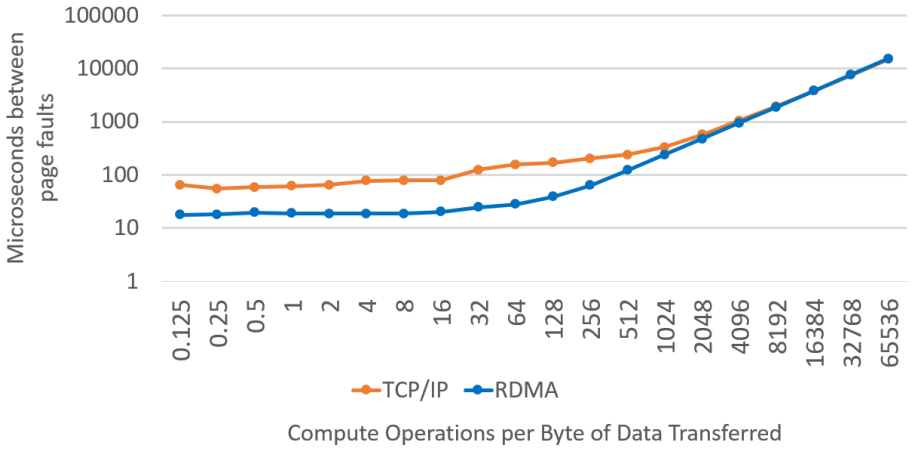
executed per byte of data transferred over the interconnect. The microbenchmark fills all cores in the system with threads and initializes data so that threads access separate pages and always cause a fault when first touching a page. Once the page has been transferred, threads perform differing numbers of floating point operations on the page. We use the experimental setup shown in Table 10.1 and evaluated the DSM layer using two different network protocols supported by Popcorn Linux's messaging layer, TCP/IP and RDMA.

Figure 10.1a shows the compute throughput in millions of floating point operations per second when varying the number of compute operations per byte of data transferred over the interconnect. Figure 10.1b shows the average page fault period, i.e., elapsed time between consecutive page faults. Intuitively, as threads perform more computation per byte transferred, the computation is able to amortize the DSM costs and reach peak compute throughput – the application spends more time performing useful computation versus waiting for the DSM to map data pages. Another takeaway from Figure 10.1a is that there are significant latency differences when using RDMA versus TCP/IP. Page faults using RDMA cost around 30 microseconds, whereas they cost 90 and 120 microseconds for the Xeon and Cavium servers, respectively, with TCP/IP. Thus, the amount of computation needed to amortize DSM costs when using TCP/IP is significantly higher than RDMA. This leads to another conclusion – as cross-node page fault latency decreases, the amount of computation needed to amortize data transfer costs decreases. Thus we can expect as heterogeneous-ISA CPUs become more tightly coupled, cross-node execution will become more beneficial for a wider variety of applications.

To determine if cross-node execution is beneficial, the HetProbe scheduler calculates the page fault period by measuring execution times and number of faults. The break-even point when cross-node execution becomes beneficial can be seen in Figure 10.1a when the microbenchmark is close to maximum throughput: above 512 operations/byte for RDMA, 32768 operations/byte for TCP/IP. Correlating these values to Figure 10.1b, the runtime uses a threshold of 100 μs/fault for RDMA and 7600 μs/fault for TCP/IP to determine whether there is enough computation to amortize DSM costs and benefit from executing across multiple CPUs. As faulting latency drops (e.g., if CPUs share physical memory), fewer compute operations are needed to amortize cross-node memory access latencies. When the interconnect between CPUs changes, this microbenchmark can be re-used as a tool to automatically determine the threshold value of when cross-node execution becomes beneficial.



(a) Floating point operations per second



(b) Page fault period, i.e., microseconds between faults

Figure 10.1: Performance metrics observed when varying the number of compute operations per byte of data transferred over the interconnect. For example, a 16 on the x-axis means 16 math operations were executed per transferred byte or 65536 operations per page.

2. If utilizing cross-node execution, how much work should be distributed to each node? As mentioned previously, during the probe period the runtime measures the execution time of a constant number of iterations on each core in the system. The HetProbe scheduler uses this information to directly calculate the core speed ratios of each node and skew the workload distribution of the remaining loop iterations.

3. If not utilizing cross-node execution, on which node should the work be run? Determining on which node an application executes best involves understanding how the application stresses the architectural properties of each CPU. Performance counters provide insights into how applications execute and what parts of the architecture bottleneck

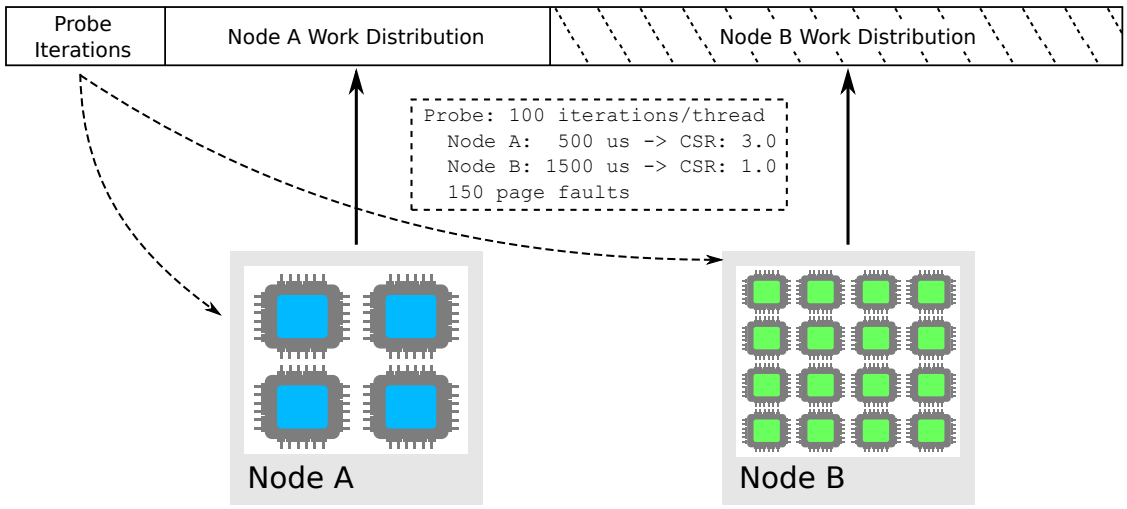


Figure 10.2: HetProbe scheduler. A small number of probe iterations are distributed at the beginning of the work-sharing region to determine core speed ratios of nodes in the system. Using the results, the runtime decides either to run all iterations on one of the nodes or distribute work across nodes according to the calculated core speed ratio (shown here).

performance. For our setup, the ThunderX has a much higher degree of parallelism versus the Xeon, meaning it has a much higher theoretical throughput for parallel computation. However, the biggest challenge in utilizing all 96 cores is being able to supply data from the memory hierarchy. Although the ThunderX uses quad-channel RAM (with twice the bandwidth of the Xeon), it only has a simple two level cache hierarchy versus the Xeon's much more advanced (and larger per-core) three level hierarchy. If an application exhibits many cache misses, it is unlikely to fully utilize the 96 available cores and would be better run on the Xeon. The HetProbe scheduler measures cache misses per thousand instructions during the probing period to determine how much the work-sharing region stresses the cache hierarchy (users can specify any performance counters prudent for their hardware). We experimentally determined a threshold value of three cache misses per thousand instructions – below the threshold and the application can take advantage of the ThunderX's parallelism, but above the threshold the ThunderX's CPUs will continuously stall waiting on the cache hierarchy and thus the computation should be run on the Xeon. Note that the HetProbe scheduler must use performance counters and cannot simply use execution times from the probing period to decide on a node; the probing period measures execution times with DSM overheads that are not present when executing only on a single node.

Once a node has been chosen, the HetProbe scheduler falls back to existing OpenMP schedulers for single-node work distribution. Currently it defaults to the static scheduler, but this is configurable by the user. Additionally, `libopenpop` joins threads on the unused node to avoid unnecessary cross-node synchronization overheads. For example, if not using the ThunderX there is no reason to keep 96 threads alive simply to join at end-of-region barriers.

Figure 10.2 shows an example of a work sharing region with 20000 loop iterations executing using the HetProbe scheduler. The first 2000 iterations are used for the probing period and each of the 20 cores across both nodes is given an equal share of 100 iterations. Importantly, the probing period is performing useful work, albeit in a potentially unbalanced way. After the probing period, `libopenpop` measures that Node A's cores executed 100 iterations in 500μs whereas Node B's cores executed 100 iterations in 1500μs. The HetProbe scheduler determines that Node A's cores are 3x faster than Node B's cores for this work sharing region, meaning threads on Node A should get 3x more iterations than threads on Node B to evenly distribute work (the CSR is set to 3:1 for Nodes A and B, respectively). In this example, the HetProbe scheduler determined that cross-node execution was beneficial (see Section 10.2). For the remaining 18000 iterations, each thread on Node A receives 1929 iterations and each thread on Node B receives 643. Thus the HetProbe scheduler automatically determines the relative performance of heterogeneous CPUs through online profiling and distributes the remaining work accordingly. Note that if cross-node communication was deemed too costly, the remaining 18000 iterations would all be distributed to either Node A or Node B depending on performance counters.

10.3 Implementation

`libopenpop` is built on top of GNU `libgomp`, the OpenMP runtime used by `gcc`. It adds 5,145 lines of code, primarily to implement the thread hierarchy (and all associated machinery), runtime measurement and dynamic work distribution. As previously mentioned, because Popcorn Linux's compiler is built on `clang` which emits API calls the `libiomp` runtime, `libopenpop` includes a small shim layer to forward `libiomp` function calls to `libgomp`. Because page faults are transparent to the application, `libopenpop` reads page fault counters from a `proc` file exposed by Popcorn Linux. Currently Popcorn Linux's compiler does not support runtime performance counter collection (although it could be supported through an interface such as PAPI [142]); to work around this, we collected performance counter data offline and fed it to `libopenpop` to make node selection decisions. For applications with multiple work-sharing regions, we currently manually specify which region should be probed to make distribution decisions by evaluating which region executes the longest. This could be automated by `libopenpop` by running the application for a small period of time and querying the probe cache to select region(s). We leave these engineering tasks as future work.

Chapter 11

Heterogeneous OpenMP Evaluation

When evaluating `libopenpop` and the HetProbe scheduler on heterogeneous-ISA systems, we asked the following questions:

1. Is `libopenpop` able to efficiently leverage the compute capabilities of asymmetric server-grade heterogeneous CPUs for OpenMP-parallelized applications?
2. Is `libopenpop`'s HetProbe scheduler able to accurately measure runtime behavior and make sound workload distribution decisions? Specifically, can the HetProbe scheduler accurately determine if cross-node execution is beneficial, distribute appropriate amounts of work to each node, and select the best CPU for single-node execution?
3. Which schedulers are best suited for which types of runtime behaviors and applications?

11.1 Experimental Setup

We evaluated `libopenpop` using the experimental setup shown in Table 10.1 which approximates our envisioned tightly-coupled platform; because no existing systems integrate heterogeneous-ISA CPUs via point-to-point connections, we approximate such a system by connecting two servers with high-speed networking. Our setup includes an Intel Xeon server with a modest number of high-powered cores and a Cavium ThunderX server with a large number of lower-performance cores. These servers represent two ends of the CPU design space and are therefore useful for targeting a variety of workloads. The machines are interconnected via 56Gbps InfiniBand adapters which provide low latency and high throughput. We use the RDMA protocol for all experiments except where explicitly mentioned due to its significantly lower latency. Both machines run the latest version of Popcorn Linux; the Xeon server uses Debian 8.9 while the ThunderX server uses Ubuntu 16.04. Popcorn Linux's compiler is built on `clang`/LLVM 3.7.1, and `libopenpop` is built on `libgomp` 7.2.0.

Benchmark	Core speed ratio – Xeon : ThunderX
blackscholes	3 : 1
EP-C	2.5 : 1
kmeans	1 : 1
lavaMD	3.666 : 1

Table 11.1: Core speed ratios calculated by HetProbe scheduler. Used by Ideal CSR and HetProbe configurations. Without the HetProbe scheduler, developers would have to manually determine these values via extensive profiling.

Benchmark	Time	Benchmark	Time
blackscholes	85.76	kmeans	989.77
BT-C	310.08	lavaMD	104.52
cfd	76.47	lud	258.75
CG-C	71.36	SP-C	210.57
EP-C	32.00	streamcluster	67.86

Table 11.2: Baseline execution times in seconds when run on Xeon with 16 threads using the static scheduler

Benchmarks. We selected 10 benchmarks from three popular benchmarking suites – The Seoul National University [175] C/OpenMP versions of the NAS Parallel Benchmarks [23], PARSEC [33] and Rodinia [50]. These benchmarks represent HPC and data mining benchmarks from a variety of areas and therefore represent a number of potential use cases for `libopenpop`. In addition to their applicability, these benchmarks exhibit a wide variety of computational patterns on which to evaluate `libopenpop`. All benchmark results are the average of 3 runs using each configuration (execution times were stable across runs).

Work Distribution Configurations. We evaluated running the benchmarks using the following workload configurations:

- **Xeon** – represents running the benchmark entirely on the Xeon. Serial phases run on a single Xeon core and work-sharing regions use the Xeon's 16 threads.
- **ThunderX** – similar to Xeon except on the ThunderX CPU. Serial phases run on a single ThunderX core and work-sharing regions use the ThunderX's 96 cores.
- **Ideal CSR** – executes work-sharing across both CPUs. Serial phases run on a Xeon core and work-sharing regions always split loop iterations across the Xeon and ThunderX (112 total threads) using the cross-node static scheduler and skew distribution using the CSRs in Table 11.1. The CSRs were gathered from runs with the HetProbe scheduler and manually supplied via environment variables.
- **Cross-Node Dynamic** – identical to Ideal CSR except that it uses the cross-node dynamic scheduler described in Section 10.1 for work-sharing regions. We experimen-

tally determined the best chunk size for each benchmark; most benchmarks performed better with smaller sizes, i.e., finer-grained load balancing.

- **HetProbe** – executes work-sharing regions using the HetProbe scheduler. HetProbe uses both CPUs during the probing period and then decides whether cross-node execution is beneficial. If so, it uses measured execution times on each CPU to calculate CSRs (Table 11.1) to skew loop iteration distribution for the remaining iterations similarly to Ideal CSR. If not, it selects the best CPU and falls back to OpenMP's original static scheduler on a single node; threads on the not-selected node are joined to avoid unnecessary synchronization. The probe period was configured to use 10% of available loop iterations. The HetProbe scheduler probed for up to 10 invocations of a given work-sharing region (using an exponential weighted moving average to smooth out measurements), after which it re-used existing measurements from the probe cache. For several benchmarks, the HetProbe scheduler chose single-node execution on the ThunderX. As a comparison point, "HetProbe (force Xeon)" shows the same results except forcing the HetProbe scheduler to use single-node execution on the Xeon; these results are explained below.

11.2 Results

Table 11.2 shows the total benchmark execution times, including both serial and parallel phases, on the Xeon; this is considered the baseline configuration. Figure 11.1 shows the speedup normalized to homogeneous Xeon execution for each of the aforementioned configurations. The benchmarks can broadly be classified into two categories: those that benefit from cross-node execution and those that do not. blackscholes, EP-C, kmeans and lavaMD fall into the former category whereas the others fall into the latter. Across benchmarks that benefit from multi-node execution, all but blackscholes achieve the highest speedup under Cross-Node Dynamic. This is due to the fact that with a granular chunk size, work is distributed across nodes in an almost perfect balance. Additionally, due to the thread hierarchy, there is significantly reduced global synchronization for loop iteration distribution as threads grab work from a local work pool the majority of the time. Across these four benchmarks, Cross-Node Dynamic yields a geometric mean speedup of 2.68x. Ideal CSR is 12.5% faster for blackscholes and close behind Cross-Node Dynamic for the other three, achieving a geometric mean speedup of 2.55x. Finally, HetProbe is slightly slower than the other two configurations for cross-node execution, achieving a geometric mean speedup of 2.4x. This is due to probing overheads – the probe period runs a constant number of iterations for all cores leading to an initial workload imbalance. Additionally, measurement machinery (timestamps, parsing the `proc` file for DSM counters) and probe cache synchronization add extra overheads. For these four benchmarks, probing overhead is equal to the difference between Ideal CSR and HetProbe, as they are functionally equivalent after probing. HetProbe adds 5.2%, 5.3%, 11.5% and 2.8% overhead for blackscholes, EP-C, kmeans and lavaMD, respec-

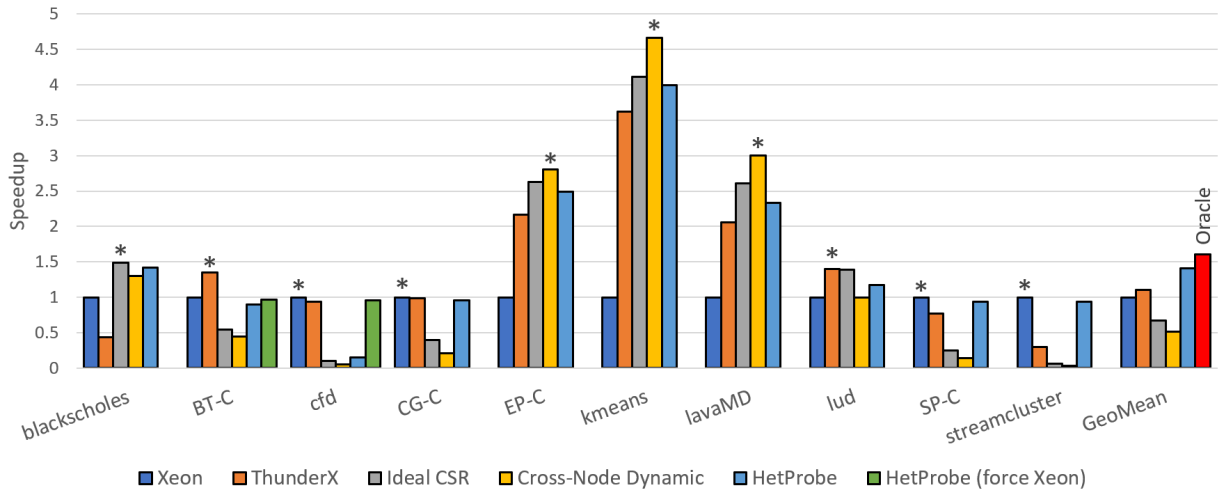


Figure 11.1: Speedup of benchmarks versus running homogeneously on Xeon (values less than one indicate slowdowns). Asterisks mark the best workload distribution configuration for each benchmark. "Cross-Node Dynamic" provides the best performance across applications that benefit from leveraging both CPUs (blackscholes, EP-C, kmeans, lavaMD), but causes significant slowdowns for those that do not. "HetProbe" achieves similar performance to Ideal CSR and Cross-Node Dynamic for the four scalable applications but falls back to a single CPU for applications that cause significant DSM communication and hence have worse cross-node performance. For geometric mean, "Oracle" is the average of the configurations marked by asterisks, i.e., what a developer who had explored all such possible workload distribution configurations through extensive profiling would choose.

tively, for a geometric mean overhead of 5.5%. This demonstrates the HetProbe scheduler provides competitive performance with minimal overheads for benchmarks that benefit from cross-node execution.

For benchmarks that do not scale across nodes, however, the Ideal CSR and Cross-Node Dynamic configurations significantly degrade performance with geometric mean slowdowns of 3.63x and 5.89x, respectively. This is due to DSM – threads spend significant time waiting for pages from other nodes, which also forces application threads on other nodes to be time-multiplexed with DSM workers. There is not enough computation to amortize DSM page fault costs over the network. The Cross-Node Dynamic scheduler is exclusively worse than the Ideal CSR scheduler due to additional work distribution synchronization caused by threads repeatedly grabbing batches of iterations. The HetProbe scheduler, however, successfully avoids cross-node execution for these benchmarks by measuring the page fault period and determining cross-node execution to not be beneficial (geometric mean slowdown of 39%, or 2.4% without cfd). Figure 11.2 shows measured page fault periods for each application; applications with a period below 100μs were considered not profitable for cross-node execution.

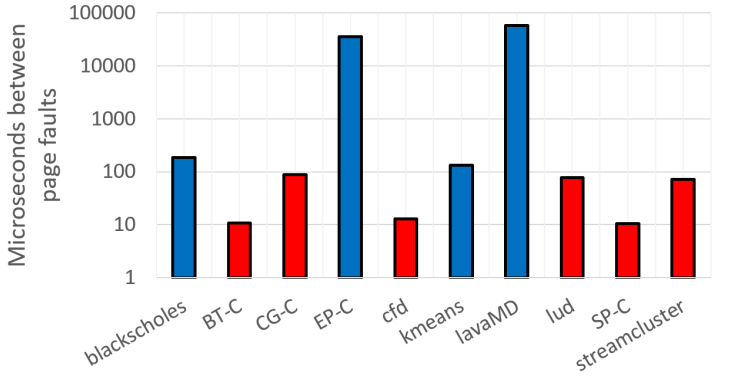


Figure 11.2: Page fault periods used to determine whether cross-node execution is beneficial. Red bars (cross-node execution not profitable) are below the RDMA threshold indicated in Section 10.2, blue are above.

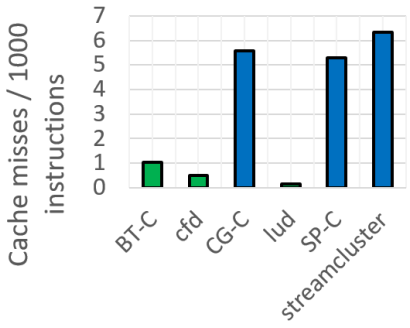


Figure 11.3: Cache misses for applications not executed across nodes. Green bars (including lud) indicate the application was run on the ThunderX, blue were run on the Xeon.

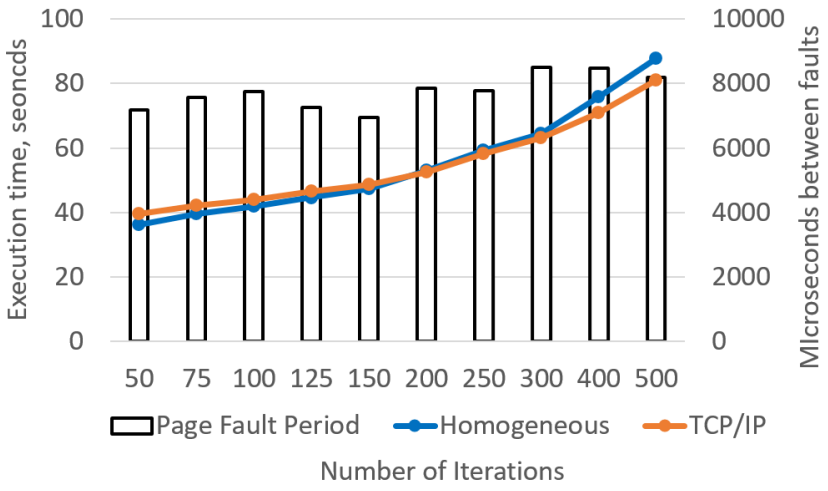


Figure 11.4: Execution time (lines, left axis) and page fault period (bars, right axis) when varying the number of iterations of blackscholes. "Homogeneous" refers to Xeon configuration, "TCP/IP" refers to using HetProbe over TCP/IP.

For applications deemed not beneficial to execute across nodes due to high DSM overheads, the HetProbe scheduler utilized cache misses per 1000 instructions to determine whether to execute work-sharing regions on the Xeon or ThunderX. As shown in Figure 11.3, there is a clear separation between applications that benefit from the ThunderX's high parallelism (BT-C, cfd, lud) and those that are bottlenecked by memory accesses (CG-C, SP-C, streamcluster). It is worth noting that although cfd's parallel region runs faster on the ThunderX, it has a long serial file I/O phase that runs slowly on ThunderX, leading the benchmark's overall execution time to be faster on the Xeon (hence why homogeneous ThunderX is actually slower than Xeon). When selecting a node, the HetProbe scheduler used a threshold value of three misses per thousand instructions, placing BT-C, cfd and lud on the ThunderX and the others on Xeon. For the three benchmarks placed on Xeon, probing overhead is equivalent to the difference between Xeon and HetProbe since HetProbe degrades to Xeon after probing. The probing period adds 4.8%, 6.6% and 7.1% for CG-C, SP-C and streamcluster, respectively, for a geometric mean overhead of 6.1%. This shows performance close to single-node execution on the Xeon, meaning the probing period has minimal impact on performance.

Interestingly for BT-C, cfd and lud, executing parallel regions on the ThunderX achieved worse than expected performance due OS limitations. Popcorn Linux's kernel currently only supports spawning threads on the node on which the application started, meaning one thread must remain on the Xeon even when work-sharing regions execute on the ThunderX. Each of these benchmarks executes hundreds to thousands of work-sharing regions (and their associated implicit barriers), causing significant cross-node synchronization. As a comparison point for BT-C and cfd, we ran an additional experiment to force the HetProbe scheduler to select the Xeon for single-node execution; it added 3.2% and 4.2% probing overhead, respectively. lud is an interesting case – the HetProbe scheduler decides cross-node execution is not profitable and runs work sharing regions on the ThunderX. The aforementioned OS limitation impacts HetProbe's performance enough that Ideal CSR actually achieves 20% better performance than HetProbe (although still worse than running solely on the ThunderX). We expect that when Popcorn Linux allows spawning threads on remote nodes, `libopenpop` will be able to more efficiently leverage both machines.

It is important to note that none of Xeon, ThunderX, Ideal CSR or Cross-Node Dynamic perform best in all situations, clearly illustrating the need for the HetProbe scheduler. As shown in Figure 11.1, HetProbe provides the best performance out of all evaluated configurations across all benchmarks with a geometric mean performance improvement of 41% (ThunderX provides an 11% improvement). In contrast, Ideal CSR causes a slowdown of 49% and Cross-Node Dynamic causes a 96% slowdown, highlighting the importance of communication traffic when deciding whether to distribute computation across multiple nodes. As a comparison point, "Oracle" shows that developers could obtain a geometric mean speedup of 60% if they had extensively profiled all configurations and selected the best for all benchmarks. As Popcorn Linux matures, HetProbe will be able to more closely match the Oracle, as the aforementioned thread spawning limitation has a significant impact on HetProbe's

performance.

What types of applications benefit from cross-node execution? The four applications that benefit from cross-node execution have a high enough compute to cross-node communication ratio to leverage the compute resources of multiple CPUs. blackscholes has an initial data transfer period but repeats computation on the same data, allowing it to settle on nodes (blackscholes also has a lengthy file I/O phase that benefits from the Xeon's strong single-threaded performance). EP-C performs completely local computation (including heavy use of thread-local storage) with a single final reduction stage. lavaMD computes particle potentials through interactions of neighbors within a radius, meaning multiple threads re-use the same data brought across the interconnect. Similarly, kmeans alternatively updates cluster centers and cluster members – all threads on a node alternate between scanning the cluster member and cluster center arrays, re-using pages brought over the interconnect.

Benchmarks that do not benefit cannot amortize data transfer costs. For example, BT-C and SP-C access multidimensional arrays along different dimensions in consecutive work sharing regions, causing the DSM to shuffle large amounts of data between nodes. Other benchmarks have little data locality – CG-C and streamcluster calculate a set of results and then access them in irregular patterns using an indirection array. This behavior causes extensive latencies for local cache hierarchies, let alone DSM. lud's work-sharing region sequentially accesses an array, but does not perform enough computation per byte to amortize DSM costs. Additionally, there is a large amount of "false sharing" where threads on different nodes write to independent parts of the same page. False sharing can be avoided by the use of a multiple-writer protocol such as lazy-release consistency [12].

What applications benefit from Ideal CSR versus Cross-Node Dynamic? Three of the four benchmarks that benefit from cross-node execution achieve the best performance with Cross-Node Dynamic due to fine-grained load balancing. For blackscholes, however, Ideal CSR achieves better performance. This is due to pages settling into a steady state after the initial work sharing region. Threads receiving the same loop iterations across multiple invocations of the work sharing region access the same data, thus all data pages required by threads are already mapped to the appropriate node. With Cross-Node Dynamic, however, threads receive different loop iterations across separate executions, meaning pages containing results must be continually shuffled across nodes. This settling behavior is why the HetProbe scheduler deterministically distributes iterations for the probing period.

Case Study: TCP/IP. In order to evaluate whether our approach for determining the profitability of cross-node execution is valid for different types of interconnects, we ran blackscholes with varying number of iterations (more iterations means more compute operations per byte since blackscholes' data settles after the first iteration) using the TCP/IP protocol described in Section 10.2. Figure 11.4 shows the execution time when running homogeneously on the Xeon versus cross-node execution (lines) and the page fault period of each cross-node run (bars). As mentioned in Section 10.2, we use a page fault period of

7600μs to determine whether cross-node execution will be beneficial when using TCP/IP. The results are somewhat noisy (TCP/IP tends to have more variable latencies) but consistent with expectations – only after the page fault period climbs above 8000μs does cross-node execution pay off. Thus we conclude using page fault periods as the determining factor for cross-node execution is applicable for different types of interconnects.

Optimization opportunities. As mentioned previously there are several limitations in Popcorn Linux's current implementation which hinder performance. Popcorn Linux does not currently support spawning threads on remote nodes, meaning the thread which starts the parallel must reside on the origin node (the Xeon in our setup). Many libc-specific functions are not well optimized for Popcorn Linux – common operations like file I/O are protected via global locks, which incur large overheads when threads perform synchronization across nodes (threads parse the `proc` file containing page fault information using standard file I/O).

11.3 Discussion

There are number of ways in which `libopenpop` can be extended and evaluated. First, the HetProbe scheduler is designed for work sharing regions with loops where each loop iteration performs a constant and equal amount of work. This assumption allows the HetProbe scheduler to make workload distribution decisions by monitoring the behavior of a small number of probe iterations. For irregular applications such as graph traversal algorithms [117], however, this assumption is no longer valid. For these applications, predicting cross-node DSM traffic becomes significantly more difficult and potentially requires co-designing DSM and parallel programming runtimes to minimize communication.

Due to limitations in Popcorn Linux we are currently only able to evaluate `libopenpop` on Xeon and first-generation ThunderX CPUs. Evaluation using other CPU architectures such as the next generation ThunderX2 or POWER9 would allow us to better evaluate `libopenpop`'s ability to determine the optimal node using performance counters. We believe that as architectures become more diverse in terms of cache hierarchy, hardware threading and microarchitecture, `libopenpop` will need to incorporate new analyses to more accurately determine node affinity.

`libopenpop` also currently focuses on achieving maximum performance but not energy efficiency. The first-generation ThunderX CPUs consume large amounts of power, meaning that even though cross-node execution may provide the best performance oftentimes the heterogeneous setup consumes more energy than running solely on one node. Optimizing OpenMP execution for different efficiency metrics may yield different workload distributions, especially as the system architecture (CPUs, interconnect) changes.

`libopenpop` currently assumes exclusive access to the machine for runtime measurement. Some of the performance metrics measured by `libopenpop` (in particular execution time)

may be affected in multiprogramming scenarios when applications must time-share resources. Taking into account not only how the current application but all applications in the system utilize system resources (interconnect, cache hierarchy, hardware threads) will affect workload distribution decisions made by `libopenpop`.

Chapter 12

Chameleon – Runtime Re-randomization

The Popcorn Compiler generates stack layout information in order to reconstruct a thread's execution state at equivalence points within an application. While this process was designed for execution migration between heterogeneous-ISA CPUs, it can be leveraged in other contexts such as hardening applications against security vulnerabilities. In particular, we leverage the Popcorn compiler's abilities in order to continuously randomize the layout of a thread's execution state (stack, registers) in order to thwart attacks that target buffers allocated on the stack [63] and return-oriented programming (ROP) attacks which string together existing code into arbitrary functionality [177]. Chameleon is a re-randomization runtime that utilizes the state transformation information produced by the Popcorn compiler to periodically and transparently randomize the code and stack layout of target applications.

12.1 Background

Buffer overflow attacks are one of most common forms of memory errors plaguing C and C++ applications. Because there is no bounds checking in these languages, logic errors present in applications may unknowingly allow malicious attackers to read or write outside of the buffer's bounds, leaking sensitive information or overwriting execution state. Stack overflow attacks exploit buffer overflows for stack-allocated variables to modify the thread's runtime stack [152]. These exploits, called "stack smashing" attacks, overwrite the stack with machine code for an exploit and modify the return address on the stack so that upon returning from the vulnerable function, the application jumps to the exploit. However, these types of exploits and other code injection attacks have fallen by the wayside as new techniques such as avoiding mapping the stack or any data region as executable (e.g., data execution prevention in Windows [6]) and the use of stack canaries (StackGuard [63]) have become widespread.

Additionally, modern compilers and operating systems implement address space layout randomization [178] to randomize the base addresses of various application components such as code, data and heap regions, preventing attackers from bootstrapping exploits using known code or data locations.

With the ability to inject code into vulnerable applications becoming more difficult, attackers have instead turned to reusing existing code in applications to construct and execute exploits. One form of attack, dubbed "return-to-libc" [69], overwrites the return address on the stack and populates registers with arguments to jump into functions in the standard C library (which is linked into almost every application). These attacks can be used to, for example, spawn a shell – the `system` standard C library API allows executing a shell command. A more general form of attack known as return-oriented programming (ROP) [177], strings together small snippets of existing code in the target application to execute arbitrary functionality. The attacker must discover "gadgets" in the machine code of the application that implement primitive operations such populating a register with a value, performing some math operation, or storing data onto the stack. The attacker then constructs a "gadget chain" by stringing together gadgets through careful construction of stack data. Using these gadget chains, ROP attacks have been shown to provide Turing-complete functionality and can be constructed using widely available ROP compilers [173]. While initially targeting x86 due to the prevalence of unaligned `ret` instructions (the return instruction, opcode `0xc3`, appears frequently inside other instructions), ROP attacks have been extended to work in the absence of return instructions [51] and attack other architectures such as SPARC [43], ARM [116] and PowerPC [129]. Additionally, many new variants of ROP attacks such as jump-oriented programming (JOP) [36], Blind-ROP [35] and JIT-ROP [182] extend the same basic idea with new forms of gadgets, allowing attackers to bypass defenses built into applications hardened against unintentional `ret` instructions, web servers with pathological behavior that can be abused to de-anonymize layouts and dynamic gadget discovery in the presence of fine-grained load-time ASLR.

Chameleon is designed to disrupt these kinds of attacks by continuously randomizing the stack layout and code. By changing the stack layout, Chameleon makes it more difficult for attackers to corrupt specific stack elements such as return addresses. Additionally, the locations of payloads injected by attackers changes, making it more difficult to bootstrap exploit execution. Chameleon forces attackers to guess the locations of stack elements, risking causing a crash or triggering intrusion detection systems. Chameleon must also rewrite the application's code in order to match the randomized stack layout. Rewriting the code has the secondary benefit of disrupting gadgets used by the attacker. Gadgets are used to provide a very specific functionality (e.g., load a value from a specific stack slot into register `rax`); any deviation from this expected behavior breaks gadget chain functionality. By rewriting the code section, Chameleon changes the instructions inside of gadgets and alters their semantics, disrupting these carefully constructed ROP chains. Because exploits such as JIT-ROP discover gadgets and construct payloads at runtime by following dynamically-discovered code pages, Chameleon must continually re-randomize the target to prevent attackers from

discovering gadgets after an initial randomization.

12.2 Threat Model

In Chameleon's threat model, the attacker can interact with the target application through typical I/O interfaces such as sockets or files (including standard in/out/err). The attacker does not have the ability to directly read or write memory, but does have the ability to invoke a memory error (e.g., buffer overflow or memory disclosure) to both read and write application memory indirectly. Thus the attacker can discover the layout of the application and overwrite code pointers (e.g., return addresses) to alter the application's control flow. The application is running using standard memory protection mechanisms such that no page has both write and execute permissions; this means the attacker cannot directly inject code but must instead rely on constructing gadget chains. However, the gadget chains crafted by the attacker can invoke system APIs such as `mprotect` to create such regions if needed. The attacker knows that the target is running under Chameleon's control and therefore knows of its randomization capabilities. We assume the system software infrastructure (compiler, kernel) is trusted and therefore the capabilities provided by these systems are correct and sound. However, the attacker may be running a malicious process on the same machine as the target, giving the attacker the ability to launch side-channel attacks such as cache timing attacks [205, 130, 113]. Using side-channel attacks gives the attacker the ability to leak information such as Chameleon's randomization metadata, albeit at a much slower pace than normal memory reads and writes.

12.3 System Architecture

Chameleon's goal is to continuously re-randomize the code section and stack layout of an application (named the *target* or *child*) in order to harden it against memory disclosures, stack smashing/stack buffer overflows and ROP attacks. As a result of running under Chameleon, gadget addresses or stack buffer locations that are leaked by memory disclosures and that help facilitate other attacks (buffer overflows, construction of gadgets) are only useful until the next randomization, after which the attacker must re-discover the new layout and locations of sensitive data. Thus Chameleon aims to continuously randomize the application quickly enough so it becomes probabilistically impossible for attackers to construct and execute attacks against the target. Currently Chameleon supports x86-64 but can be easily extended to ARM64 or other architectures, unlike some previous works which rely on x86-specific features to hide performance overheads [203].

Chameleon has several design goals. First, Chameleon is designed to re-randomize the target transparently – the target application has no knowledge of running under Chameleon's control. In addition to transparency, Chameleon is designed to provide strong isolation

guarantees between itself and the target by executing in a separate process and therefore separate virtual address space. This provides security benefits by minimizing the interface between Chameleon and the target. Chameleon is also designed to have limited contact with the outside world. After initializing the target, Chameleon only ever interacts with the kernel and with the target through kernel-mediated interfaces. Separating Chameleon into a separate process exposes optimization opportunities by allowing Chameleon to asynchronously perform any chores in the background (e.g., generate the next set of randomized code) while the application executes. This separation of duties into separate processes allows for a cleaner implementation versus previous DBI-based frameworks [66, 196, 203], which require complex self-hosting and bootstrapping mechanisms because they operate within the same process as the target. Additionally, this allows code randomization to be performed completely asynchronously versus DBI-based frameworks, which not only place generating re-randomized code in the critical path of applications, but also add unavoidable DBI overheads (e.g., virtualizing indirect branches via trapping into an interpreter [41]).

In order to implement re-randomization, Chameleon needs the following capabilities:

1. **Disassemble, randomize and re-assemble code** (Section 12.4). For all functions in the target application, Chameleon must be able to randomize the locations of stack elements. Therefore, Chameleon must know how the compiler has laid out the stack and must be able to find references to stack elements. After randomizing the layout, Chameleon must rewrite code emitted by the compiler in order to update references to stack elements to instead refer to their randomized locations.

2. **Served randomized code pages to the target** (Section 12.5). After rewriting the code, Chameleon must be able to transparently serve the randomized code to the target in place of the on-disk code emitted by the compiler. Chameleon accomplishes this through Linux's `userfaultfd` mechanism [110], which allows a user-level process to handle page faults. Chameleon must prepare the target's code region for attaching via `userfaultfd` and then serve requests on-demand for randomized code pages.

3. **Re-randomize the target** (Section 12.6). After an initial randomization, Chameleon must be able to stop a running application (executing using the previous randomization) and atomically switch it to another randomization. In addition to exercising the previously mentioned capabilities, Chameleon must also transform the stack from its current layout to the newly randomized layout.

Chameleon uses two kernel interfaces, `ptrace` and `userfaultfd`, to monitor and transform the target application. `ptrace` [4], or *process trace*, is an interface widely used by debuggers to inspect and control the execution of *tracees*. In particular, `ptrace` allows *tracers* (e.g., Chameleon) to read and modify per-thread register sets, signal masks and virtual memory in the tracee. `ptrace` also allows intercepting events in the tracee such as signals and system calls, depending on how the tracer configures tracee execution. Finally, `ptrace`

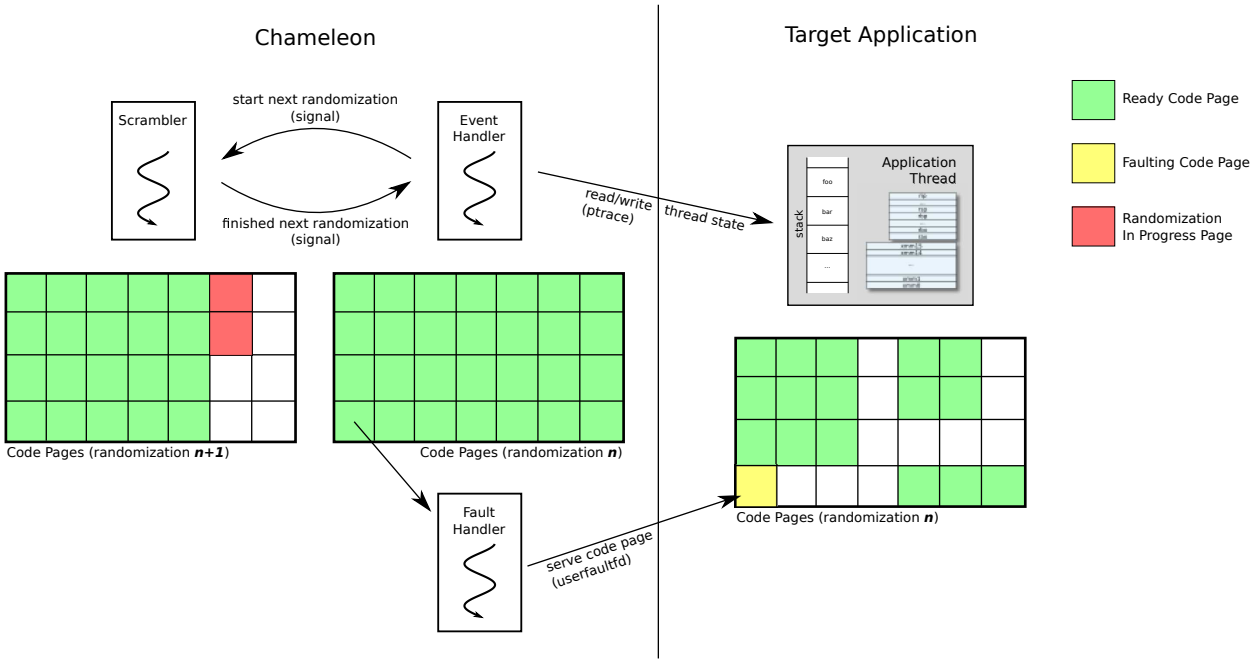


Figure 12.1: Chameleon runtime system. An *event handler* thread waits for events in a target application thread (e.g., signals), interrupts the target thread, and reads/writes the target thread's execution state (registers, stack) using `ptrace`. A *scrambler* thread concurrently prepares the next set of randomized code for the next re-randomization. A *fault handler* thread responds to page faults in the target by passing pages from the current code randomization to the kernel through `userfaultfd`.

allows tracers to forcibly interrupt tracee threads in order to gain control of their execution. `userfaultfd` [110] is a Linux kernel mechanism which allows delegating handling of page faults to user-space. When accesses to a region of memory attached to the `userfaultfd` file descriptor cause a page fault, the kernel sends a request to a process reading from the descriptor. The process can then respond with the data for the page by writing to the file descriptor. Using these two mechanisms together, Chameleon can transparently and continuously re-randomize target applications with little overhead.

Figure 12.1 shows the runtime system architecture for Chameleon. Users launch the target application by passing the command line arguments to Chameleon. After reading the code and state transformation metadata from the target application binary, Chameleon forks the target application and attaches to it via `ptrace` and `userfaultfd`. From this point on, Chameleon starts target execution and enters the *re-randomization loop*. At the start of a new randomization cycle, a *scrambler* thread iterates through every function in the target's code, randomizing the stack layout as described in Section 12.4. At some point, a *re-randomization* event is triggered within Chameleon according to a user-defined policy; currently Chameleon uses a timer trigger which periodically initiates a re-randomization.

When the re-randomization event fires, the *event handler* thread interrupts the target and atomically switches the target to the newly randomized code as described in Section 12.6. The event handler reads the target thread's current execution state (stack, registers) and transforms it to adhere to the newly randomized layout produced by the scrambler thread. After transformation, the event handler writes the transformed state back into the context of the target and drops the stale code pages. The event handler then resumes the child and waits for the next re-randomization event. As the child begins executing, it causes code page faults by fetching instructions from dropped code pages. A *fault handler* thread handles these page faults and serves the newly randomized code on demand as described in Section 12.5. In this way the entire re-randomization procedure is transparent to the target and Chameleon can atomically switch to new randomized code.

Chameleon creates a one-to-one mapping of event handler to target application threads, which allows more easily handling events in each child thread. Events are passed to Chameleon through the blocking `wait` system call, which makes multiplexing target threads onto event handlers complicated. Additionally, this simplifies executing a re-randomization, i.e., interrupting the target thread and transforming the thread's stack. However, there is only one scrambler and fault handler thread per target process as all threads share the same randomized code. Chameleon can detect when the target application spawns new threads, as `wait` will return information indicating the target thread issued a `clone` system call (the API used by standard C libraries to create new threads).

Chameleon also supports more complicated applications that fork entirely separate processes such as web servers. When the target forks a new child, `wait` informs Chameleon through a fork event. At this point, Chameleon instantiates a new scrambler, fault handler and event handler for the newly forked child. However, `ptrace` only allows one process to be the tracer of any given tracee. By default, the tracer of a given process is also the tracer of any forked child processes. Chameleon must therefore hand-off tracer privileges between event handler threads. In order to do this, Chameleon first redirects the new child to a blocking read on a socket through code installed via a parasite (see Section 12.5). The original event handler thread then detaches from the new child, allowing the new event handler thread to become the tracer for the new child while it is blocked on the socket read. After attaching, the new event handler performs some remaining initialization tasks and restores the new child to the fork location. In this way, Chameleon always maintains complete control of applications even when they fork new processes.

12.4 Code Randomization

Chameleon's first task is to rewrite code emitted by the compiler to randomize the locations of stack elements. This hardens applications against exploits which use logic errors in conjunction with locations of known stack elements to corrupt the stack [152] and ROP-like exploits which string together gadgets to execute arbitrary functionality [177]. Chameleon's

randomization moves the location of stack elements and rewrites code to update references to those stack elements, thus disrupting both of the aforementioned attacks. Chameleon uses DynamoRIO [40], a dynamic binary instrumentation framework, as its machine code disassembler and re-assembler[1].

When rewriting code, re-randomization frameworks must handle limitations of the code emitted by the compiler. When emitting machine code, the compiler directly encodes target addresses (or offsets from the program counter in the case of position-independent code [32]) in order to transfer control between different code blocks. If the re-randomization framework wishes to change the size of code, either by adding/removing instructions or changing the encodings of existing instructions, the framework must be able to find and update all code references to reflect the changed size. This problem is known to not be statically solvable [201]. Re-randomization must not only handle directly encoded code targets, but must reify all code pointers within the application (which can be stored in arbitrary locations) that were created and stored during a previous randomization cycle.

Previous works leverage dynamic binary instrumentation (DBI) frameworks like Pin [132] and DynamoRIO to continuously re-randomize code. DBI frameworks rewrite code on-demand by maintaining a *code cache* of dynamically discovered *basic blocks*, or sequences of instructions that end in a control flow instruction. At application startup, the DBI framework loads the application's first basic block, rewrites the ending control flow instruction to instead return control to the DBI framework and adds the block to the DBI's code cache. Upon executing the basic block and returning to the DBI framework, the DBI interpreter loads the next basic block starting at the target of the control flow instruction, again changing the final instruction to return control and placing the basic block in the code cache. This process occurs repeatedly, allowing the DBI framework to discover and translate all code targets, i.e., all instructions to which execution can transfer via control flow. Because of this capability, DBI frameworks can dynamically change the sizes of an application's code to add instrumentation or change the layout of stack elements for re-randomization. However this abstraction and instrumentation has performance costs such as added branch mispredictions and new mechanisms for handling indirect branches – for example, DynamoRIO adds on average 13% overhead for SPEC CPU 2006 on x86-64 as the minimal DBI cost [41].

However, executing under a DBI framework is not enough to handle changing code sizes during re-randomization. Previous works [203] use a layer of indirection similarly to the procedure linkage table [32] to force all code references to flow through a known location, adding even more overhead to applications. Chameleon, in contrast, does not attempt to find and rewrite control flow targets to enable arbitrary instrumentation, but instead randomizes by rewriting existing code in place. All code references as emitted by the compiler and linker remain valid when executing under Chameleon, as the size of code blocks is not changed. However by not allowing changing code sizes, Chameleon trades off some randomization

[1]DynamoRIO provides a full runtime instrumentation framework but Chameleon only uses it as a standalone disassembler/re-assembler.

flexibility for improved performance.

Chameleon targets randomizing on-stack elements or *stack slots* and thus must transform stack memory references to point to the slots' randomized locations. Chameleon uses the stack slot metadata emitted by the compiler, i.e., offset, size and alignment of slots, to detect and rewrite references to stack slots to instead reference their randomized location (see Section 4.5 for a description of the metadata). As shown in Figure 3.3 Chapter 3, there are a number of stack regions in a function activation and many can be randomized. Due to the previously-mentioned in-place requirement, however, there are restrictions on how each region can be randomized, which are specific for each ISA. We denote a region as *immutable* if the locations of slots in the region cannot be randomized, *permutable* if the ordering of elements in the region can be randomized but no extra padding can be added between elements, and *fully-randomizable* if slots can be placed at arbitrary locations. Stack frames for x86-64 have the following restrictions:

- Return address – immovable. The return address is pushed implicitly by a call instruction and thus cannot be moved without extra compiler instrumentation.
- Callee-saved registers – permutable. Compilers usually save and restore registers through a series of `push` and `pop` instructions, respectively. In order to save and restore registers from arbitrary locations, `push` and `pop` instructions would have to be replaced with `mov` instructions. However, this increases the size of the function prologue and epilogue – `push`/`pop` instructions are encoded in 1-2 bytes whereas `mov` instructions with a memory operand require at least 4 bytes. Therefore, Chameleon is restricted to solely randomizing the order in which callee-save registers are pushed and popped.
- Local variables – permutable or fully randomizable depending on offset. Compilers emit references to local stack-allocated variables as offsets from either the frame base pointer (FBP) or stack pointer (SP). For x86-64, these offsets are encoded in different numbers of bytes depending on the required range. For example, in the stack slot reference `-0x10(%rbp)`, i.e., the value at the memory address created by subtracting 16 from the pointer stored in register `%rbp`, the offset `-0x10` can be encoded in a single byte, which has a range of $-128 \leftrightarrow 127$. In contrast, the offset for memory reference `-0x100(%rbp)` (offset of -256) falls outside this range and must instead use a 4-byte encoding (range of -2GiB $\leftrightarrow$ 2GiB). Therefore, slots whose offsets from the FBP or SP are within the 1-byte range are permutable – Chameleon can randomize their ordering but cannot add any padding as it may force a larger offset encoding and therefore change the code size. Slots whose offsets use a 4-byte encoding, however, are fully randomizable and can be placed at arbitrary locations with extra padding added between slots[2].

[2]Fully-randomizable slots must not be moved into the 1-byte offset range either, as this will decrease the code size. `nop` instructions could be added to pad to the required length, however.

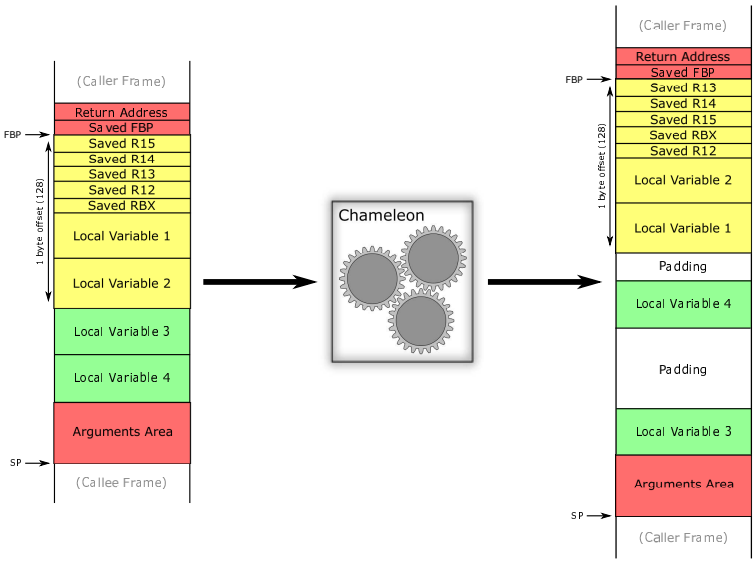


Figure 12.2: Frame randomization. Chameleon cannot randomize the locations of the return address, saved frame base pointer and call arguments. Chameleon can permute the ordering of callee-save register locations and stack slots whose offsets can be encoded in a single byte. Chameleon can place the remaining slots at arbitrary locations, including adding padding between slots.

- Argument region – immutable. Randomizing the layout of the call region means rewriting all call sites to adhere to the new argument passing convention. As previously mentioned, finding all such call sites is not solvable through static analysis.

Chameleon analyzes code emitted by the compiler and organizes detected slots into the aforementioned regions. During target application startup, Chameleon loads the code section of the binary and the state transformation metadata emitted from the Popcorn Compiler. The metadata describes the functions in the code section, including starting and ending address, frame size, callee-save register locations and stack slots (offsets, sizes and alignments). Chameleon disassembles the code and performs a one-time analysis by iterating through all instructions, detecting call slot references and additional randomization restrictions (for example, slots which cross the fully-randomizable and permutable region). After analysis, Chameleon performs an initial code randomization before the application begins execution, similar in spirit to load-time ASLR [178]. Because the application has not yet begun execution, there is no state that must be transformed. Chameleon walks through the function's instructions and rewrites stack slot references from the original to their randomized locations.

For each function, Chameleon randomizes each stack region and creates a mapping between the original and randomized offset of the stack slot. This mapping is required both for rewriting the code for the next re-randomization cycle and for state transformation from the current randomized code to the next. For permutable regions, Chameleon simply randomizes

the ordering of slots. For fully-randomizable regions, Chameleon randomizes the ordering of slots and additionally places a (user-definable) random amount of padding between the slots. To generate random numbers, Chameleon seeds a per-function pseudo-random number generator with a true random number in order to get non-determinism with high performance. The randomized code is stored in an in-memory buffer in Chameleon and is used to serve page faults for the target application.

It is worth mentioning that with extra compiler implementation, many of the previously mentioned randomization restrictions can be lifted. For example, the compiler could pad the function prologue and epilogue with `nop` instructions, giving Chameleon extra space to substitute `push`/`pop` instructions with longer `mov` instructions for callee-saved registeers. Additionally, the compiler could relax offset encodings to always use 4 bytes, allowing Chameleon more flexibility in adding padding between local variable stack slots. We leave this engineering effort as future work.

12.5 Serving Code Pages

After randomizing the code, Chameleon needs a method to serve code pages to the child. While Chameleon could use `ptrace` to write the randomized code into the address space of the child application, this could cause large re-randomization delays (i.e., switching from the current to the next set of randomized code) for applications with large code sections as Chameleon would have to bulk write the entire code section at once. Instead Chameleon uses an on-demand mechanism which allows quicker atomic switches between code sections upon re-randomization and avoids writing unused code pages into the target application's address space. In older versions of Linux, users could emulate handling page faults in user-space by mapping a region of memory with no permissions (`PROT_NONE`) and updating the permissions and memory contents on demand by catching segmentation faults when the application tried to access the region. This rudimentary mechanism causes large overheads as applications both create and catch `SIGSEGV` signals and constantly change memory permissions; with re-randomization, these faults become more burdensome as the randomization framework continually forces new faults. Newer versions of Linux instead implement `userfaultfd`, a mechanism to delegate page fault handling to user-space. Users create a file descriptor and register regions of memory with the descriptor. Page faults are handled by reading page fault events from the descriptor and writing responses containing the data to be served. This avoids the signaling and permissions maintenance overheads of the `PROT_NONE+SIGSEGV` approach.

`userfaultfd` descriptors must be opened in the context of the application in which they will be attached. Thus, Chameleon cannot directly open a `userfaultfd` descriptor attached to the target. However, open descriptors can be passed between processes via sockets. Therefore, Chameleon induces the target application to create a `userfaultfd` descriptor and pass it to Chameleon before the target application begins normal execution. Chameleon uses

`compel` [1], a support library built for CRIU [2], which facilitates implanting *parasites* into applications controlled via `ptrace`. Parasites are small binary blobs which execute arbitrary functionality in the context of the target application – for Chameleon, the parasite opens a `userfaultfd` descriptor and passes it to Chameleon through a socket. To execute the parasite, `compel` takes a snapshot of the target process' main thread (registers, signal masks). Then, it finds an executable region of memory in the target and writes the parasite code into the target. Because `ptrace` allows writing a thread's registers, `compel` redirects the target thread's program counter to the binary blob and begins execution. The parasite opens a control socket, initializes a `userfaultfd` descriptor, passes the descriptor to `compel`/Chameleon, and finishes at a well-known exit point. After pausing the thread at the exit point, `compel` restores the thread's registers and signal mask and returns the `userfaultfd` file descriptor to Chameleon.

After receiving the `userfaultfd` descriptor, Chameleon must prepare the target's code region for attaching (`userfaultfd` descriptors can only attach to virtual memory areas not backed by files on disk). Chameleon again uses `compel` to execute a system call in the context of the target – `compel` takes a snapshot of the execution state, writes the system call instruction into the target, executes the system call, restores the snapshot and returns to Chameleon. Chameleon uses `mmap` to remap the code section as anonymous (i.e., not file-backed) and then registers the code section with the `userfaultfd` descriptor. After registering the code section with `userfaultfd`, Chameleon starts the fault handler thread which is solely responsible for serving page faults to the target application. The fault handler performs blocking reads on the descriptor until the target fetches an instruction from an unmapped page. The kernel sends a page fault message to the fault handler, which replies with a pointer to the relevant page in the randomized code buffer (Section 12.4). The kernel copies the randomized code to the target application's address space, thus allowing Chameleon to transparently serve code pages to the target.

12.6 Re-randomizing the Target

At startup, Chameleon sets a periodic alarm which triggers re-randomizations in the target. The alarm signal wakes a dedicated alarm thread, which in turn interrupts the event handler threads blocked waiting for target application events. When the event handlers are interrupted, they issue a `ptrace` interrupt request to the kernel to forcibly grab control of threads in the target. At this point Chameleon begins the process of atomically switching the target to the new set of randomized code.

The first challenge in re-randomizing target threads is advancing target threads to transformation points. As described in Chapter 4, Section 4.2, the Popcorn Compiler generates state transformation metadata at migration points. Because Chameleon must transform each target thread's current stack to match the newly randomized code, Chameleon must advance the target application thread to a transformation point (i.e., function boundary) in

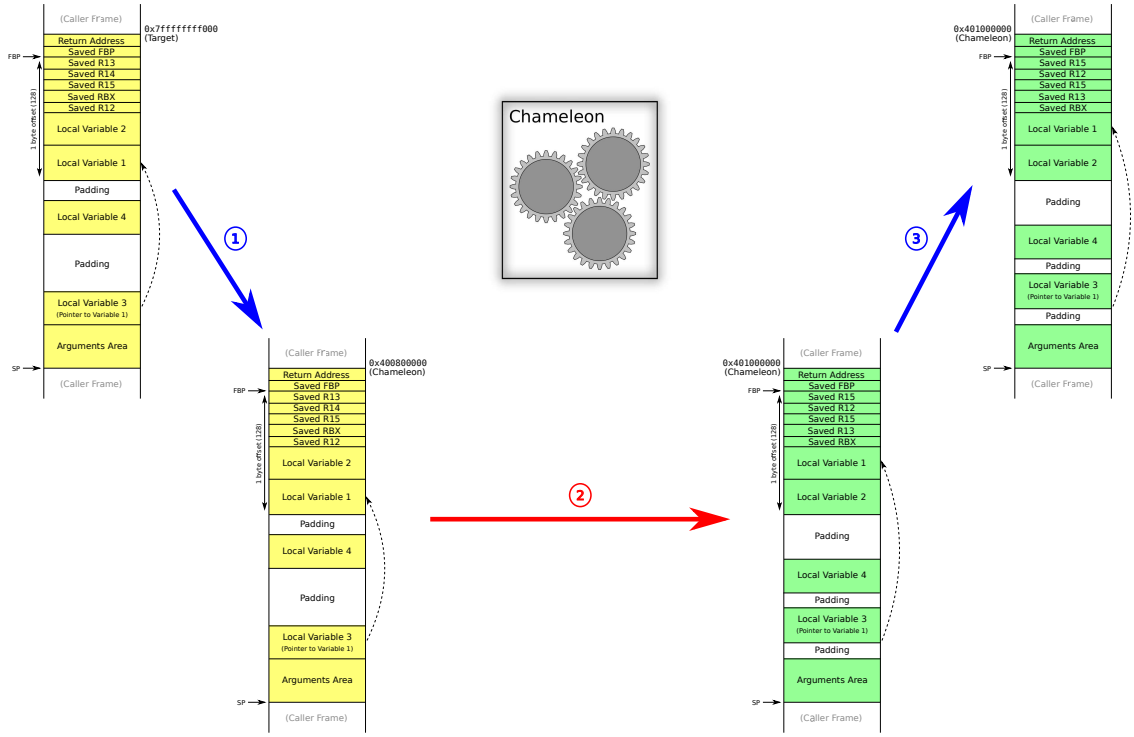


Figure 12.3: Re-randomizing the stack layout. Chameleon (1) reads the target's current execution state, (2) transforms the state from the previous randomization to the new randomization, including reifying pointers to the stack in the target's address space to reference the newly randomized locations, and (3) write the execution state back into the target's context and drop the previous code pages.

order to bootstrap the transformation.

During the initial code analysis, Chameleon records all suitable transformation locations in each function, including function entry, function return and all call instructions. After interrupting a thread, Chameleon looks up the analysis information for the function in which the thread was interrupted. At each transformation point, Chameleon uses `ptrace` to write trap instructions into the code – for x86-64, Chameleon uses the `int3` instruction which is also used by debuggers to insert breakpoints. After spraying all transformation points in the function with trap instructions, Chameleon resumes the thread and waits for the next event. When the thread reaches a transformation point, it executes the trap instruction and the kernel passes the event back to Chameleon's event handler through `wait`, allowing Chameleon to advance the thread to a program location suitable for bootstrapping state transformation. After reaching a transformation point, Chameleon restores the original instructions and begins state transformation.

Because Chameleon executes in a separate virtual address space from the target application, Chameleon cannot access a thread's stack data through regular loads and stores. The state

transformation runtime could instead use `ptrace` to read and write stack data, but would require significant modification (the runtime was written assuming access to stack data through normal memory loads and stores). This would also incur significant overhead as `ptrace` only allows reading/writing eight bytes per `ptrace` system call. Instead, Chameleon bulk reads a thread's stack from the `mem` file exposed by Linux in the `proc` filesystem. This file allows Chameleon to seek to arbitrary addresses in the target's address space and bulk read/write data. After reaching a transformation point, Chameleon reads the thread's entire stack into a buffer from the target's address space using the thread's stack pointer. Chameleon passes the stack pointer, register set and buffer containing stack data to the state transformation runtime.

State transformation proceeds largely the same as described in Chapter 5 – the runtime unwinds the current stack to find live function activations and transforms the stack to adhere to the new code (in Chameleon's case, the randomized layout created by the scrambler thread). There are several changes to the procedure, however. While unwinding the current stack to find live activations, the runtime also uses a callback provided by Chameleon to get randomization metadata for functions encapsulating each of the discovered call sites. This information contains slot remapping information and randomized frame sizes for both the current and newly randomized code. This allows the runtime to correctly unwind the stack in accordance to the current randomized layout and to calculate the size of the transformed stack for the next randomization. The other change to the runtime involves remapping offsets from the transformation metadata generated by the Popcorn compiler. Stack slots, including callee-save register locations, are encoded in the metadata as offsets from either the FBP or SP. For Chameleon, the runtime must use the slot remapping information, generated as described in Section 12.4, to find the randomized location of a given stack slot. Additionally, if storing or loading a value from the stack slot, a final level of translation is applied to convert the stack address in the target's virtual address space to the buffer in Chameleon's virtual address space into which the stack was read. This complicates handling fixups for pointers to the stack, as the runtime must check pointers to randomized slot locations in the target's address space and then write reified pointers into the buffer in Chameleon's address space. Nevertheless, this conversion is fairly inexpensive and only involves a few math operations.

After state transformation, Chameleon writes the transformed stack back into the target application's address space and updates the target thread's register set to point to the new stack. Here, Chameleon can provide enhanced security with some performance overhead. While the Popcorn compiler's runtime rewrites from one half of the currently mapped stack to the other, Chameleon can map in a completely new region of memory and unmap the old stack region. In this way, attackers trying to manipulate or smash the stack will not know a-priori where the stack is located or will lose any exploit data currently placed on the stack.

The final step in re-randomization is to force the child process to bring in the newly randomized code. Chameleon uses `compel` to execute an `madvise` system call with the `MADV_DONTNEED` flag for the code section in the context of the target, which instructs the ker-

nel to drop all existing code pages and cause fresh page faults upon subsequent instruction fetches. The fault handler thread is switched to pull pages from the next set of randomized code and the target is released to continue normal execution; the target is now executing under a new set of randomized code. At this point, the event handler thread signals the scrambler thread to begin generating a new set of randomized code and the event handler blocks until the next event. In this way, Chameleon blocks the target only so that it can transform the target thread's stack and drop the existing code pages; the most expensive work of generating newly randomized code happens concurrently with the target application's execution. This highlights one of the benefits of cleanly separating re-randomization machinery into a separate process from the target application.

Chapter 13

Evaluation – Chameleon

In this chapter we evaluate Chameleon's capabilities both in terms of security benefits and overheads. In particular, we analyze the following:

- What kinds of security benefits does Chameleon provide? In particular, how many ROP-style gadgets in application binaries does Chameleon disrupt? How much randomization does Chameleon inject into frame layouts? Finally, how can users know that Chameleon itself is secure? (Section 13.1)
- How much overhead does Chameleon impose for these security guarantees, including how expensive are the individual components of Chameleon and how much overhead does it add to the total execution time? (Section 13.2)
- A real-world case study of how Chameleon disrupts a real ROP attack on nginx, an open-source and widely used web server. (Section 13.3)

Experimental Setup. Chameleon was evaluated on an x86-64 server containing an Intel Xeon 2620v4 with a clock speed of 2.1GHz (max boost clock of 3.0GHz). The Xeon 2620v4 contains 8 physical cores and has 2 hardware threads per core for a total of 16 hardware threads. The server contains 32GB of DDR4 RAM. Chameleon is run on Debian 8.11 "Jessie" using Linux kernel 4.9. Chameleon was configured to add a maximum padding of 128 bytes between stack slots in the fully randomizable region.

Benchmarks. Chameleon was evaluated using benchmarks from the SNU C version of the NASA parallel benchmarks (NPB) [23, 175] and SPEC CPU 2017 [42]. All NPB benchmarks were run using the single-threaded version of the benchmarks and were compiled using the Popcorn Linux compiler, built on clang/LLVM v3.7.1.

13.1 Security Analysis

The most important aspect of Chameleon to be evaluated is the security benefits provided to target applications. Because Chameleon, like other approaches (e.g., [66, 203, 196]), relies on layout randomization to disrupt attackers, it cannot make any guarantees that attacks will not succeed. There is always the possibility that the attacker is lucky and guesses the exact randomization (both stack layout and randomized code) and is able to construct a payload to exploit the application and force it into a malicious execution. However, with sufficient randomization, the probability that such an attack will succeed is so low as to be probabilistically impossible. Thus Chameleon must dynamically and repeatedly change the target application so that attackers cannot discover the target application's layout and construct attacks. In this section we evaluate both the quality of Chameleon's runtime re-randomization in the target and describe the security of the Chameleon framework itself.

13.1.1 Target Randomization

First, we evaluated Chameleon's randomization quality for target applications. Chameleon randomizes the target in two dimensions: randomizing the layout of stack elements in a function's frame and rewriting the code to match the randomized layout.

We first evaluated how rewriting the code affects gadgets in the binary. Recall that gadgets are a small chain of instructions that end in a control flow instruction. Gadgets usually implement some small functionality such as populating a register or writing memory. Attackers construct malicious executions by chaining together gadgets, and therefore rely on the fact that gadgets only perform a very basic and low-level operation (e.g., writing register `rax` to the memory location pointed to by `rdi`). Additionally, gadgets may have unintended side effects that the attacker must handle – for example, a "stray" instruction in the gadget may clobber the `rax` register, meaning that the attacker cannot place sensitive information in `rax` when using that gadget. Thus gadgets, and therefore gadget chains, are very frail. Slight disruptions to the gadget's behavior can disrupt the entire intended functionality of the chain.

As part of the re-randomization process, Chameleon rewrites the application's code to match the randomized stack layout. A side effect of this process is that gadgets may be disrupted – Chameleon may overwrite part or all of a gadget, changing its functionality and therefore disrupting the gadget chain. To analyze Chameleon's impact on disrupting gadgets, we searched for gadgets in the benchmark binaries and cross-referenced gadget addresses with instructions rewritten by Chameleon. We used Ropper [171], a python-based gadget finder tool, to find all ROP gadgets (those that end in a return) and JOP gadgets (those that end in a call or jump) in the application binaries. We configured Ropper to only find gadgets of 6 instructions or less, as longer gadgets become increasingly hard to use due to the aforementioned stray instructions. We then dumped the addresses of all instructions

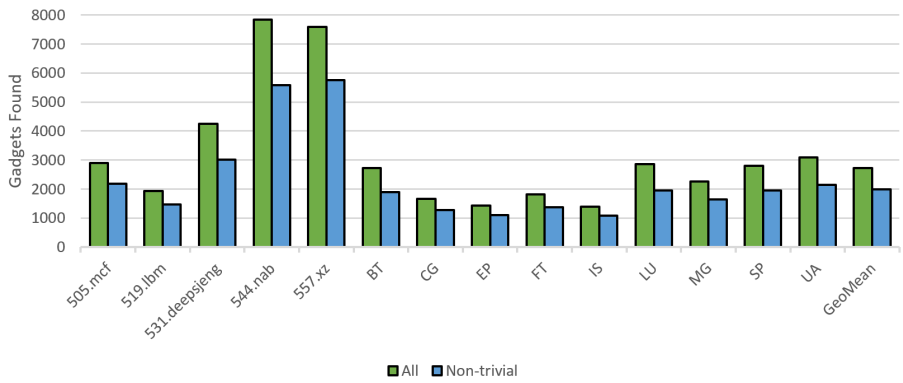


Figure 13.1: Number of gadgets found. Non-trivial gadgets are gadgets that include more than just one instruction, i.e., more than just the control flow instruction.

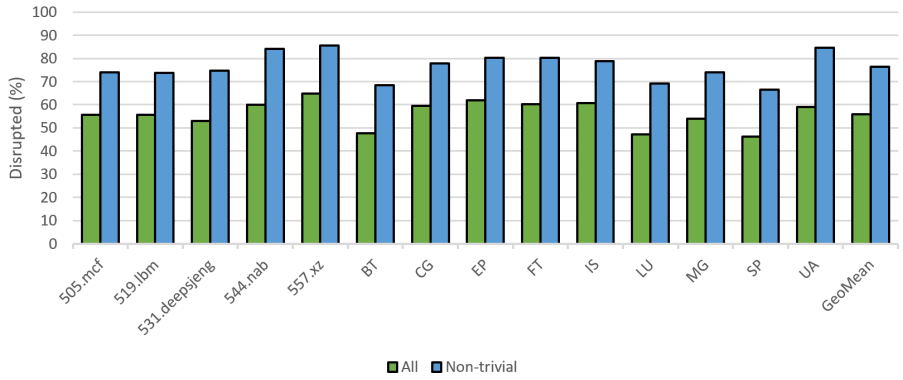


Figure 13.2: Percent of gadgets disrupted by Chameleon's code randomization

rewritten by Chameleon as part of the randomization process in order to determine which gadgets were rewritten by Chameleon and therefore disrupted.

Figure 13.1 shows the number of gadgets found by Ropper in each of the 14 benchmarks evaluated. The x-axis lists the benchmark name (benchmarks prefixed with numbers are from SPEC CPU 2017, the others are from NPB) and the y-axis lists the number of gadgets found. There are two bars for each benchmark – total gadgets found and non-trivial gadgets found. A trivial gadget is a single-instruction gadget, i.e., one that only contains control flow. While these may potentially be useful for attackers, they do not perform any useful computation and thus we expect them to be used sparingly. Additionally, every trivial gadget is double-counted as they are contained as part of a non-trivial gadget.

The number of gadgets found directly correlates with the application's code size – nab (code size of 375KB) and xz (code size of 201KB)) both have the largest code sections of all evaluated benchmarks. However, even benchmarks with small amounts of code contain a significant number of gadgets – for example, EP only has 11KB of generated machine code

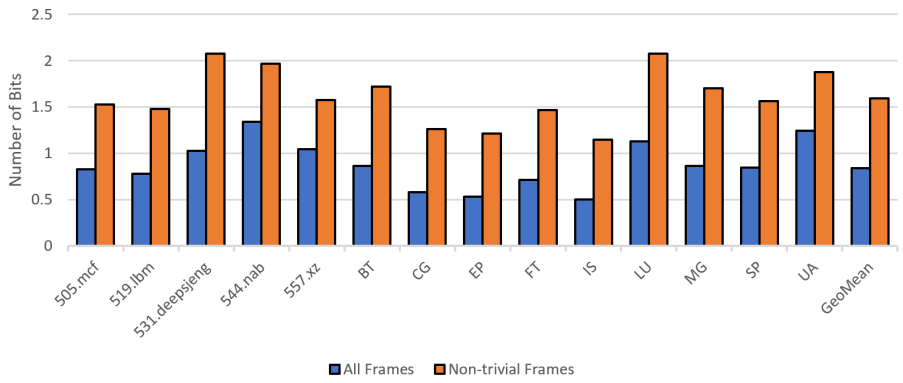


Figure 13.3: Average number of bits of entropy for a stack element across all functions within each binary. Bits of entropy quantify in how many possible locations a stack element may be post-randomization – for example, 2 bits of entropy mean the stack element could be in $2^2 = 4$ possible locations with a $\frac{1}{4} = 25\%$ chance of guessing the location.

but Ropper managed to 1429 gadgets (1102 non-trivial). However, the C standard library adds another 26KB of machine code to every application, giving attackers a larger code base from which to draw gadgets. Ropper was able to find a geometric mean average of 2731.91 gadgets from each application. Some of these may be redundant or slight variations of others, but all are available for attackers to use.

Figure 13.2 shows the percent of gadgets disrupted as part of Chameleon's stack randomization process. Chameleon disrupted a geometric mean of 55.81% or 76.32% of non-trivial gadgets. While Chameleon did not disrupt all gadgets, it disrupted enough that attackers will have a hard time chaining together functionality without having to use one of the gadgets altered by Chameleon. To better understand the attacker's dilemma, previous work by Cheng et al. [53] mentions that the shortest useful ROP attack produced by the Q ROP compiler [173] consisted of 13 gadgets. Assuming gadgets are chosen with a uniform random possible from the set of all available gadgets, attackers would have a probability of $(1 - 0.5582)^{13} = 2.44 \times 10^{-5}$ of being able to construct an unaltered gadget chain, or a $(1 - 0.7632)^{13} = 7.36 \times 10^{-9}$ probability if using non-trivial gadgets. Therefore, probabilistically speaking it is very likely that the attacker will be forced to contend with Chameleon's re-randomizations interfering with gadgets in the binary when constructing gadget chains.

Next, we evaluated how well randomizing the stack disrupts attacks that utilize known locations of stack elements. When quantifying the randomization quality of a given system, many works use *entropy* or the number of randomizable states as a measure of randomness. For Chameleon, entropy refers to the number of potential locations a stack element could be placed, i.e., the number of randomizable locations. As previously mentioned in Chapter 12, Section 12.4, x86-64 machine code emitted by the compiler is subject to a number of limitations. Nevertheless, Chameleon still has some freedom to re-arrange stack elements to disrupt attackers from abusing known locations of stack data.

Figure 13.3 shows the average entropy created by Chameleon for each benchmark. Entropy is quantified in terms of bits, i.e., the number of bits required to encode all possible randomization states for a given stack element. For example, if Chameleon could place a given slot in one of 4 locations, the stack slot would have 2 bits of entropy ($\log_2$(# locations)). For each application, the y-axis indicates the average number of bits of entropy across all stack slots in all functions. In addition to "All Frames" bar which illustrate average entropy for all functions, we additionally break out "Non-trivial Frames" as a separate bar. Functions with trivial frames are defined as functions that do not use the stack except for saving the return address. The standard C library contains many small functions that do not set up a stack and therefore have zero entropy. Because Chameleon cannot currently randomize these functions, we removed their skew and plotted the results in the second set of bars.

Across all benchmarks, Chameleon provides a geometric mean 0.84 bits of entropy or 1.59 bits of entropy for non-trivial frames. While the amount of entropy is low for guessing the location of a single stack element, again consider that the attacker must chain together knowledge of multiple stack locations to make a successful attack. For example, when overflowing a buffer on the stack, the attacker must know the offsets of the buffer relative to the frame and/or stack pointer and any other data the attacker wishes to corrupt, e.g., callee-saved registers. Not only will the buffer be placed at a randomized position, the other stack elements' locations are randomized. Consider an attack that must corrupt 3 stack elements in a given frame. With 1.59 bits of entropy, a given stack element can be placed in on average $2^{1.59} = 3.01$ locations, meaning the probability of guessing the correct location is $\frac{1}{3.01} = 0.33$. For corrupting three such locations, the attacker has a $0.33^3 = 0.037$ probability of correctly guessing the stack locations, therefore making successful attacks hard to achieve. It is also important to note that these numbers are heavily impacted by the non-randomizable and permutable stack regions. For elements in the fully randomizable region, Chameleon can randomize their location with on average 5.87 bits of entropy, making guessing even a single stack element's location difficult (this number can be increased arbitrarily by increasing the padding added between slots). Nevertheless, this highlights one of the trade-offs of Chameleon – because it cannot change the size of code, there are limitations to the amount of entropy it can inject into target applications because it must work within the constraints of what is emitted by the compiler. However, with some compiler enhancements this number could be boosted significantly – see Section 13.4.

13.1.2 Chameleon

As part of the evaluation, we also analyzed how secure Chameleon is itself from attackers. Because Chameleon has a large amount of control over the target, it is imperative that attackers not gain control, otherwise they would be able to inject arbitrary functionality into the target. However, one of Chameleon's design principles is limited interaction with the outside world – Chameleon's interface with other applications is intentionally limited as much as possible so as to avoid the possibility of other attackers exploiting flaws in

accept*	access	arch_prctl	bind*	brk	clock_gettime
clone	close	execve	exit_group	fcntl	fstat
ftruncate	futex	getpid	getsockname*	ioctl†	listen*
lseek	mmap	mprotect	munmap	openat	pread
prlimit64	ptrace	read	readlink	recvfrom*	recvmsg*
rt_sigaction	rt_sigprocmask	rt_sigreturn	sendmsg*	sendto*	setns*
set_robust_list	set_tid_address	socket*	socketpair*	tgkill	timer_create
timer_delete	timer_settime	uname	wait4	write	

Table 13.1: Complete list of system calls invoked by Chameleon. Calls marked with * are only used by `compel` only to communicate with the target when initializing. Calls marked with † are only used for `userfaultfd`.

Chameleon to gain control of the target.

Table 13.1 lists the complete set of system calls invoked by Chameleon when running and re-randomizing the target application. Chameleon only uses 47 system calls and only 9 (`ioctl`, `mmap`, `pread`, `ptrace`, `read`, `recvfrom`, `recvmsg`, `sendmsg`, `sendto`) can potentially interface with other applications. We discuss how Chameleon uses each.

- `ioctl`: only used to register the target's code region with `userfaultfd`
- `mmap`: used for large memory allocations and by libelf/state transformation runtime to map the target binary in for reading the code and state transformation metadata sections. `compel` briefly maps in shared memory with the target during target initialization, but unmaps the region after cleaning up the parasite.
- `pread`: used by libelf to read ELF metadata, i.e., magic start bytes, section table, etc.
- `ptrace`: used by Chameleon to control the target process, including reading and writing target registers and memory.
- `read`: used by Chameleon to read information from the `proc` filesystem, i.e., reading target's memory map and reading target's stack
- `recvfrom, recvmsg, sendmsg, sendto`: used by `compel` to initialize, communicate with and clean up the parasite. Not used after target application is initialized.

There is very little interaction with the outside world which is directly controllable by would-be attackers. The only avenue that attackers could potentially use to hijack Chameleon would be through corrupting state in the target application which is then subsequently read during one of the re-randomization periods. Although it is conceivable that attackers could corrupt memory in such a way in the triggers a flaw in Chameleon, it is unlikely that they would be able to gain enough control to perform useful functionality; the most likely outcomes of such an attack are null pointers exceptions caused by Chameleon following erroneous pointers

505.mcf	461.79	EP	75.63
519.lbm	270.3	FT	57.93
531.deepsjeng	482.85	IS	22.24
544.nab	635.84	LU	162.69
557.xz	211.2	MG	8.06
BT	228.29	SP	144.59
CG	105.14	UA	337.71

Table 13.2: Original execution times of benchmarks, in seconds, without Chameleon

when transforming the target's stack. Additionally, because Chameleon is small (5526 lines of code), it would be easy to instrument with safeguards potentially even formally verify its behavior. This is a large benefit of placing the re-randomization machinery outside the context of the target application itself – verifying correctness becomes a much simpler task as the randomization machinery itself is small. Thus, we argue that Chameleon provides a compelling system architecture for enhancing the security of target applications.

13.2 Performance

We next evaluated the performance of target applications executing under Chameleon's control. As mentioned in Chapter 12, Sections 12.5 and 12.6, Chameleon must perform a number of duties to continuously re-randomize applications. In particular, Chameleon runs a scrambler thread to generate a new set of randomized code, runs a fault handler to respond to code page faults with the current set of randomized code, and periodically switches the target application between randomizations.

Table 13.2 shows the execution times of benchmarks without Chameleon, i.e., normal non-randomized performance. Figure 13.4 shows the slowdown of each benchmark versus the times listed in Table 13.2 with Chameleon re-randomizing the application with different periods. Chameleon was run with randomization periods of 100ms, 50ms and 10ms in order to understand how increasingly frequent re-randomizations impacts performance; lower periods mean more frequent re-randomizations and thus less of a chance for attackers to discover and exploit the current target application's layout.

Figure 13.4 highlights the benefits of lifting the re-randomization machinery out of the critical path of the application. Chameleon re-randomizes target applications with a geometric mean 0.7% overhead with a 100ms period and 1.1% overhead with a 50ms period. Re-randomizing with a 10ms period raises the overhead to a geometric mean of 9.1% due to reasons explained below. However, this is still significantly better than related works – for example, Shuffler [203], which uses a DBI framework to instrument the code inside the address space of the application, demonstrates an average of 14.9% overhead when re-randomizing

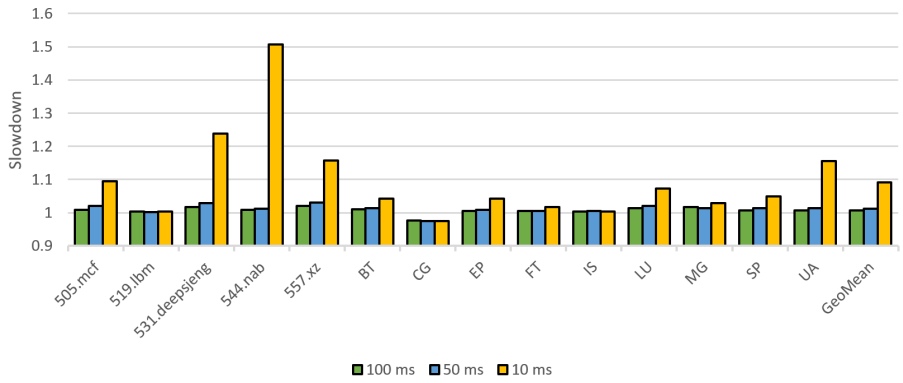


Figure 13.4: Overhead when running applications under Chameleon compared to unsupervised execution. Overheads rise with smaller re-randomization periods, but are negligible in most cases. With 10 millisecond re-randomization period, some benchmarks exhibit high overheads due to waiting for the scrambler to finish generating the next set of randomized code.

every 50ms. This demonstrates how asynchronous code re-randomization provides significant performance boosts compared to DBI or other in-application solutions.

In order to better evaluate the performance costs of Chameleon's mechanisms, we added timers throughout the code to break out the cost of individual components. Figure 13.5 shows how long it takes the scrambler to generate a complete set of randomized code for each application for each of the re-randomization periods. This latency directly correlates to the size of the code section – as previously mentioned, nab and xz have the biggest code sections and thus the longest code rewriting time. Interestingly, as the re-randomization period drops the code randomization time drops as well. This is due to DVFS effects – as the re-randomization period drops, the scrambler is awoken more frequently to generate new code and thus keeps the processor clocked at a higher frequency. Conversely, with longer re-randomization periods, the scrambler spends more time sleeping and thus the processor clocks down. In general, however, on the Xeon 2620v4, the scrambler randomizes code at about 10KB of instructions per millisecond. Rewriting the code section is fairly efficient, but there are a number of optimizations that could be applied. First, Chameleon internally maintains a function's instructions as a linked list of DynamoRIO `instr_t` structs. This could be refactored to instead use a vector to improve cache locality. Additionally, Chameleon currently maintains all of the instructions contained inside the function, even those that are not randomized (e.g., moving values between registers). During randomization Chameleon must traverse this list, leading to parsing unchanged instructions. Chameleon could instead be changed to only maintain those instructions that must be rewritten during re-randomization. Finally, randomizing code could be trivially parallelized, as all randomization and rewriting metadata is maintained per-function and thus can be randomized independently.

Next we analyzed how long it took Chameleon to switch from the current randomization to

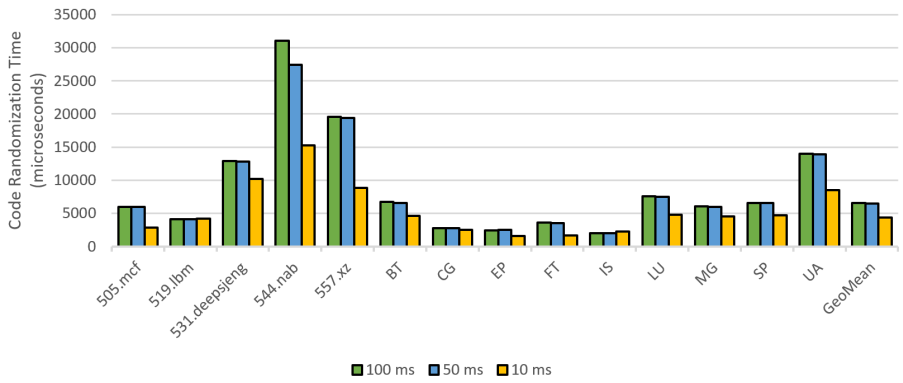


Figure 13.5: Average time to complete a single code randomization. Randomization time is correlated with the code size of each application. As the re-randomization period gets shorter, the scrambler thread spends less time sleeping and therefore maintains a higher clock frequency, hence a smaller time to randomize code.

the next randomization for each of the benchmarks. Figure 13.6 shows the individual cost to switch randomizations for each benchmark, including:

1. Interrupting the target, spraying the current function with transformation points and restoring the original instructions,
2. Waiting for the scrambler to finish randomizing code,
3. Reading the target's stack, transforming the stack from the current randomization to the next randomization and writing the transformed stack back into the target,
4. Setting the newly randomized code for the fault handler,
5. Dropping the existing code pages

As shown in Figure 13.6, atomically switching between randomizations is an inexpensive process. The benchmarks take a geometric mean of 280μs to perform the entire procedure for a 100ms period and 276μs for a 50ms period. For these two randomization periods, only deepsjeng and LU take longer than 600μs. This is due to large stack buffers that must be copied to their newly randomized location – for example, LU allocates a 400KB buffer on the stack that must be copied to its newly randomized location. Nevertheless, as a percentage of the re-randomization period, transformations are inexpensive: 0.2% of the 100ms re-randomization period and 0.5% of the 50ms period.

There are several performance outliers when analyzing transformation overhead for the 10ms period. deepsjeng, nab and UA's overheads increase drastically. This is directly correlated to the code randomization overhead. When the event handler thread receives a signal to start

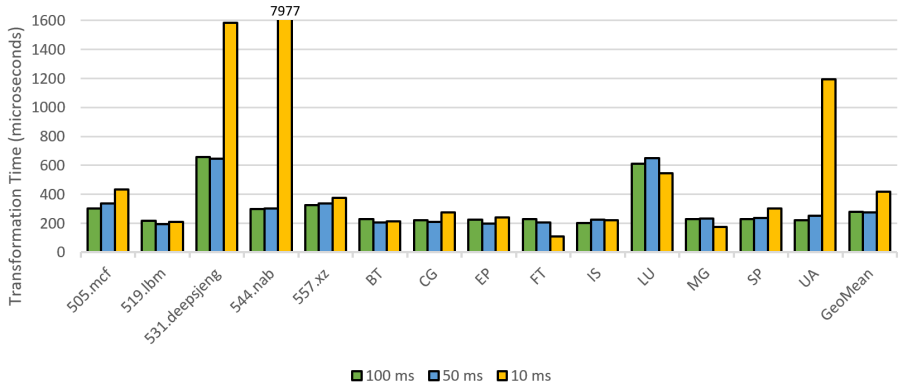


Figure 13.6: Time to atomically switch the target from the previous randomization to the next randomization, including transforming the stack and dropp the existing code pages. As the randomization period gets smaller, transformation balloon in several case (deepsjeng, nab, UA) as the transformation thread blocks waiting for the scrambler to finish randomizing the code and generating slot remapping information.

a re-randomization, it advances the target to a transformation point and blocks until the scrambler thread signals it has finished re-randomizing the code. Because these applications have higher code randomization costs, the event handler thread is blocked waiting for a significant amount of time. Interestingly, one would expect the same phenomena to occur for xz, which has a long code re-randomization period. However, recall from Section 12.6 that Chameleon can only transform the stack at function boundaries. xz contains long stretches of pure compute without function boundaries, meaning that the excessive code transformation latency is hidden by the time it takes to advance the target to the transformation point. With the aforementioned code randomization optimizations, however, we expect these overheads to drop further.

Finally, the last potential source of overhead from Chameleon is the continuous page faults. On the Xeon 2620v4, it took on average 5.06μs to serve a single page fault. This value fluctuated by several microseconds depending on how close in time page faults occurred and thus their impact on the fault handler's ability to sleep and clock down the CPU. However, serving code page faults added negligible overhead in all applications, usually less than 0.1%.

13.3 Case Study: nginx

In order to better understand how Chameleon's runtime re-randomization can help protect target applications from attackers, we evaluated how it can be used to disrupt a flaw found in a real application. nginx [165] is a lightweight and fast open-source webserver used by a large number of popular websites such as Buzzfeed, Adobe and Cloudflare [5]. nginx uses a multi-process architecture to aid in stability and leverage the available compute capacity

in multicore CPUs. Unlike traditional webservers which dedicate a process or thread to handling the entire lifetime of a given connection [78], nginx instead spawns a set of worker processes which are decoupled from connections. Worker processes continually iterate over open connections, servicing any outstanding requests asynchronously. Thus when interacting with nginx, would-be attackers must exploit flaws in the connection handling code in worker processes.

CVE-2013-2028 [72] is a security vulnerability affecting nginx v1.3.9/1.4.0 in which a carefully crafted set of requests can lead to a stack buffer overflow. When parsing an HTTP request, the worker process allocates a page-sized buffer on the stack and calls `recv` to read the request body. By using a "chunked" transfer encoding [146] and triggering a certain sequence of HTTP parse operations through specifically-sized messages, the attacker can underflow the size variable used in the `recv` operation on the stack buffer and allow the attacker to send an arbitrarily large payload. Because the attacker subsequently controls the stack, they can, for example, clobber the return address and inject payload data for the attack itself. Additionally, the attacker can exploit nginx's error handling behavior to even avoid other stack protection techniques such as stack canaries [63]. The attacker must not clobber the stack canary, otherwise the worker process will detect a stack corruption and safely exit. However, because nginx respawns worker process upon errors or crashes, the attacker can guess the canary bytes and reconstruct the canary value in the payload to avoid triggering the error (this is also the behavior exploited by Blind-ROP [35]).

As a demonstration of the severity of the exploit, VNSecurity [199] published a proof-of-concept attack that uses the overflowed buffer to inject an attack payload onto the stack. The attack builds a ROP gadget chain that begins by remapping a piece of memory with read, write and execute permissions. After creating a buffer for injecting code, the ROP chain copies instruction bytes from the payload to the buffer and "returns" to the payload by placing the address of the buffer as the final return address on the stack. The instructions copied into the buffer set up arguments and use the `system` standard C library API to spawn a shell on the server. The attacker can then remotely connect to the shell and gain privileged access to the machine.

The stack buffer targeted by the attack is randomizable by Chameleon and is located in the fully-randomizable region of the stack frame (i.e., references to the slot emitted by the compiler use four bytes). For this particular function, there are four slots in the fully-randomizable region meaning this particular stack slot can be in one of four locations in the final ordering of slots. Using a maximum slot padding size of 512, Chameleon will insert anywhere between 0 and 512 bytes of padding between slots. The slot has an alignment restriction of 16, meaning there are $512/16 = 32$ possible amounts of padding that can be added between the vulnerable buffer and the preceding stack slot. Therefore, Chameleon can place the buffer at $4*32 = 128$ possible locations within the frame for 7 bits of entropy. Thus, an attacker has a probability of $\frac{1}{2^7} = 0.0078$ of guessing the correct buffer location. System administrators can further diminish this probability to extremely small levels by increasing the maximum padding size, which is configurable within Chameleon. For example, using

a maximum padding size of 4096 bytes (1 page) creates 10 bits of entropy and leads to a probability of $\frac{1}{2^{10}} = 0.00098$. Because the attacker must know the location of the buffer relative to the start of the stack frame in order to overwrite the return address and start the ROP chain, the attacker will have a difficult time starting the attack.

This is not the only way Chameleon disrupts the attack. As part of the randomization process, Chameleon rewrites the code to match the randomized stack layout, thus potentially disrupting ROP chains constructed by attackers. As an example, as part of the function epilogue in one of nginx's functions, there is an addition instruction used to clean up the stack frame:

```
0x438b75:  48 83 c4 58    add $0x58,%rsp
0x438b79:  5b             pop %rbx
0x438b7a:  5d             pop %rbp
0x438b7b:  c3             retq
```

Listing 13.1: nginx assembly generated by compiler

However, the ROP chain constructed by the attacker uses an unaligned instruction at code address `0x438b78` to instead issue several stack pop instructions to populate registers with values from the stack:

```
0x438b78:  58             pop %rax
0x438b79:  5b             pop %rbx
0x438b7a:  5d             pop %rbp
0x438b7b:  c3             retq
```

Listing 13.2: Gadget assembly found by Ropper

The attack only uses the first instruction to pop a value off the stack into register `rax` (`rbx` and `rbp` are unused). However, as part of the randomization process, Chameleon overwrites the `0x58` byte (which contains the size of the stack frame) with a new value, therefore removing the `pop %rax` instruction and disrupting the gadget. There is also another gadget in the chain disrupted in a similar fashion, showing the difficulty faced by attackers in constructing exploits composed of several gadgets – tshere is a high likelihood of Chameleon disrupting a carefully constructed gadget chain and thwarting attackers.

13.4 Discussion

As previously mentioned, many of Chameleon's randomization limitations arise from its inability to change the code size and therefore it must randomize within the limitations of the generated code. This has an impact most notably on Chameleon's inability to randomize the return address location and the inability to add padding between slots in offset-limited

stack regions. However, both of these restrictions arise due to the x86-64 ISA and are non-existent in RISC-style architectures with fixed-width instructions. For example, a branch-and-link instruction on AArch64 (the equivalent of a call instruction) places the return address in the `x30` general-purpose register. During the function prologue, the compiler emits instructions to both allocate callee-saved register space and store the return address on the stack. Therefore, an AArch64 version of Chameleon would not be restricted by implicitly pushed return addresses. Similarly, when referencing stack slots on AArch64, the compiler has a fixed offset range which is common across all memory references – there is no variable length encoding, meaning different slots within the same stack frame are all subject to the same randomization range.

However, with several compiler modifications these x86-64-specific limitations could be removed. For the return address location, the compiler could be modified to pad the function prologue and epilogue with `nop` instructions that do not affect execution in normal runs of the application and unlikely have a significant effect on performance. However Chameleon could recognize these `nop` "buffers" and replace them with data movement instructions to place the return address at a different location in the stack frame. Applied equally to the saved frame-base pointer, this would remove two of the last fixed stack elements, drastically complicating the attacker's ability to construct ROP attacks. Additionally, in order to add extra entropy to stack element locations the compiler could be modified to always *relax* immediate encodings, i.e., always use four bytes to encode displacement operands in memory references even if the displacement can fit within one byte. This would dramatically improve frame randomization flexibility and further reduce the probability that an attacker could correctly guess the locations of stack elements. Finally, Chameleon currently makes no attempt to randomize register usage. Chameleon could be extended to randomize which live values are placed in which registers (similarly to [156, 115]), further disrupting gadgets in the binary.

Chapter 14

Conclusions and Future Work

In this dissertation, a full software stack was presented for execution migration across heterogeneous-ISA processors in real systems. This dissertation presented a compiler which builds multi-ISA binaries capable of execution migration across ISA boundaries. It additionally presented a runtime which dynamically translates thread execution state between ISA-specific formats in under a millisecond.

The Popcorn compiler toolchain builds multi-ISA binaries containing a code section for every ISA in the system. The middle-end automatically inserts migration points into the application at function call sites and performs a liveness analysis to gather the sets of live values at those call sites. The back-end generates metadata describing where those live values are placed in each ISA-specific version of function activations. The linker aligns code and data symbols across all binaries and a final post-processing tool optimizes the multi-ISA binary for efficient state transformation.

The state transformation runtime works with the operating system to transform a thread's register set and stack between ISA-specific formats. When the OS requests a migration, the runtime attaches to the thread's stack and reconstructs it in the destination ISA's format in a separate region of stack memory. After transformation, the runtime invokes the OS's thread migration service and passes it the reconstructed destination register set. After migration, the runtime bootstraps the thread on the destination architecture and then resumes normal execution transparently to the application.

The entire Popcorn Linux system software stack was evaluated on an APM X-Gene 1 processor interconnected to an Intel Xeon E5-1650v2 processor using a Dolphin PXH810 point-to-point connection over PCIe. State transformation costs were evaluated for a microbenchmark and several real applications from the NAS Parallel Benchmark suite and showed that sub-millisecond translation overheads are achievable on both processors. Additionally, the evaluation showed that for a datacenter-like workload, Popcorn Linux is able to achieve up to a 66% reduction in energy and up to an 11% reduction in energy-delay product.

This dissertation also described how the Popcorn Linux infrastructure could be used to accelerate applications parallelized using OpenMP. It described the design and implementation of `libopenpop` and how it refactors OpenMP execution for systems composed of non-cache-coherent domains. `libopenpop` organizes application threads into a hierarchy in order to separate local and global computation. Using the thread hierarchy, leaders are elected to perform synchronization at the global level on behalf of each domain, minimizing the amount of cross-domain page traffic caused by synchronization. Using this hierarchy, `libopenpop` demonstrated a 38x decrease in multi-domain barrier latency and a 5.4x decrease in data reduction latency. This dissertation also described compiler and C library modifications that further reduce unnecessary page transfers for OpenMP-parallelized applications. Finally, this dissertation identified OpenMP anti-patterns that cause excessive page faults and that can be remedied by developers. After applying all of these optimizations, `libopenpop` demonstrated a geometric mean speedup of 4.04x for scalable applications on an 8-node rack.

This dissertation next described how `libopenpop` distributes parallel work across heterogeneous CPU systems in consideration of CPU capabilities and application execution characteristics. `libopenpop` uses the thread hierarchy to optimize existing OpenMP loop iteration schedulers for multi-domain systems. Additionally, `libopenpop` utilizes a new scheduler, called the *HetProbe* scheduler, which measures execution behavior and makes workload distribution decisions. If deciding to use multiple CPUs for an OpenMP work-sharing region, the HetProbe scheduler automatically determines the workload partition to minimize execution time. If only using a single CPU due to the computation's communication overheads, the HetProbe scheduler picks the best CPU using performance counters. `libopenpop` was demonstrated to achieve up to a 4.7x speedup and a geometric mean speedup of 41% across 10 benchmarks versus the best homogeneous OpenMP execution.

This dissertation finally described Chameleon, a runtime re-randomization framework which builds on top of the Popcorn compiler and state transformation runtime to harden applications against attackers. Chameleon uses existing operating system interfaces to attach to applications and serve randomized code in place of the code emitted by the compiler. At runtime, Chameleon continuously generates new stack layouts and sets of randomized code. Periodically, Chameleon interrupts the target application and switches from the previous to the next randomization, using the state transformation runtime to remap application execution state. Because Chameleon executes in a completely separate context and performs re-randomization chores asynchronously, the target application executes with minimal overhead – for example, Chameleon adds 1.1% overhead when re-randomizing the target every 50 milliseconds. Additionally, Chameleon disrupts a geometric mean 76.32% of gadgets found by a gadget finder tool. Finally, Chameleon provides a geometric mean 1.59 bits of entropy across all stack elements in the evaluated benchmarks, or 5.87 bits of entropy for stack elements not subject to randomization limitations from the compiler.

This dissertation described the design and implementation of a full software system for migrating threads of execution between heterogeneous-ISA architectures on real hardware. This dissertation has shown that not only is it possible to migrate threads between these

architectures, but that migration between architectures can be achieved with low overheads and allow system software to adapt application execution to varying goals. The Popcorn Linux compiler and state transformation runtime can help achieve higher energy efficiency versus a system without execution migration, accelerate multithreaded applications across non-cache-coherent domains and harden applications against would-be attackers through continuous re-randomization.

14.1 Future Work

There are many future research directions for each of the aforementioned components. Here we describe several future research directions.

14.1.1 Heterogeneous-ISA State Transformation and Execution Migration

There are currently a number of language-level features that are not supported by the compiler and state transformation runtime, e.g., `setjmp/longjmp`, variable-argument data structures, etc. Additionally, the compiler and runtime do not support ISA-specific features such as x86 encryption extensions [103] or ARM scalable vector extensions [185]. The compiler and runtime could be extended to fully utilize such extensions and use software fallbacks when not available. The compiler and runtime could even be extended to proactively migrate applications between architectures when they detect that an application uses such features and would benefit from accelerated functionality built into the ISA.

The current Popcorn Linux compiler and state transformation prototype is limited to ARMv8 and x86-64 processors. However, there are numerous other processors that use different ISAs such as POWER, SPARC, RISCV, etc., which target different form factors. The compiler and state transformation runtime could be extended to each of these ISAs, allowing exploration of execution migration in new contexts (e.g., internet-of-things devices). Additionally, the compiler and runtime could be re-designed to be completely future proof, such that they could be fed an ISA description (registers, data layout, ABI, etc.) and automatically produce the required machinery for state transformation and execution migration without manual intervention.

14.1.2 OpenMP for Non-Cache-Coherent Domains

This dissertation described how OpenMP execution can be refactored to reduce page faults and how `libopenpop` determines if cross-node execution will be beneficial by observing page fault behavior. These system properties were evaluated on a specific set of hardware,

meaning that as the interconnect or CPUs comprising the system change these properties may need to be re-analyzed. One future direction would be to mathematically model how performance properties of the system change when varying the interconnect and CPUs in the system. For example, in Section 10.2 a threshold for deciding whether cross-node execution is profitable was experimentally determined through a microbenchmark varying the amount of compute operations per byte of transferred data. Instead, a model would be able to take as input interconnect properties (e.g., latency per page fault, bandwidth) and CPU properties (number of cores, frequency, page fault handling latency) and automatically determine that threshold. Similarly, a model could be developed that takes as input program execution behavior, e.g., page fault period, parallel work, etc., and determine how well the application would scale as the number of cache coherence domains increases. Modeling performance and bottlenecks in the system as a function of Popcorn Linux's DSM behavior would allow developers more insights as to what types of applications scale and how they scale across different types of architectures.

One of the biggest current limitations of OpenMP on Popcorn Linux is on-demand data movement triggered by intercepting page faults in the kernel. Although this mechanism provides great flexibility and allows applications to execute transparently across multiple machines, it places data movement overheads directly in the critical path of application execution. As previously mentioned, network bandwidth continues to rise with future network interface cards promising hundreds of gigabits of bandwidth. Unfortunately network latencies are not improving nearly as fast, necessitating the need to better utilize the available bandwidth to hide latencies. For OpenMP applications, this means prefetching the results of one thread's computation across all the nodes in the system so that as threads access the produced data, they do not have to wait for network communication latencies. OpenMP's `for` work sharing regions often structure computation around loop iterations; iterations are assigned to threads, which use the iteration number to access the relevant information in buffers. With more sophisticated compiler analyses, OpenMP applications could be instrumented with *push* operations to push updated data other nodes in the system, obviating the need for an on-demand data transfer at some point during the application's future execution. Because of the ample available network bandwidth and increasing core counts in CPUs, nodes should be able to aggressively push out updates. Recent hardware trends in programmable NICs [147] also point to new possibilities in accelerating Popcorn Linux's messaging layer. It is possible that much of the page coherency protocol's logic could be offloaded to the network infrastructure, accelerating page fault overheads to wire-speed latencies. Thus, in order to better scale multithreaded applications across rack-scale systems using distributed shared memory, the system software must be optimized to better take advantage of emerging networking trends.

14.1.3 Runtime Re-randomization

Chameleon could be extended with extra compile-time enhancements to allow more flexible re-randomization (e.g., adding `nops` and forcing relaxed displacement encodings as previously mentioned). Currently Chameleon only supports x86-64. However, Chameleon could be extended to support other ISAs, especially ARM as it is already supported by the state transformation runtime. Chameleon could be even further extended to utilize Popcorn Linux's migration mechanisms to transparently migrate applications between heterogeneous-ISA CPUs, even to the point where the migration is completely hidden from the target application. The system software would have to be extended to instantiate Chameleon instances on remote nodes and hand off target processes between them. This would add an extra level of randomization that would be difficult for attackers to overcome, even with emerging side-channel attacks that exploit microarchitectural implementation details to leak information. By transparently migrating target applications between different nodes, Chameleon would physically remove all forms of sharing with would-be attackers.

Chameleon can also be extended with new defenses. For example, Chameleon already has the ability to reconstruct the stack, which requires unwinding the current stack to determine live function activations. Chameleon could be extended to perform on-demand control flow integrity checks to detect when the target application's control flow has been hijacked (e.g., ROP chains). Additionally because Chameleon can serve page faults for the target application, it can easily implement page table-based protection mechanisms. For example, a secure region of memory could be registered with Chameleon by the target. When accessing the memory, Chameleon would transparently populate the memory for the application. After a certain delay, Chameleon would then wipe the memory from RAM and store it on disk, preventing leakage via side-channel attacks. These sorts of protections would be especially useful for systems which may not have hardware-based secure enclaves [60], such as legacy servers, internet-of-things devices or even microcontrollers.

Bibliography

[1] Compel - CRIU. https://criu.org/Compel. Accessed on 1-29-2019.

[2] CRIU. http://criu.org/. Accessed on 1-29-2019.

[3] Ftrace. https://elinux.org/Ftrace. Accessed on 3-11-2019.

[4] ptrace(2) - Linux manual page. http://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed on 1-29-2019.

[5] Success Stories - NGINX. https://www.nginx.com/success-stories/. Accessed on 3-15-2019.

[6] Data Execution Prevention. https://docs.microsoft.com/en-us/windows/desktop/Memory/data-execution-prevention, May 2018.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[8] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[9] Misiker Tadesse Aga and Todd Austin. Smokestack: Thwarting dop attacks with runtime stack layout randomization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 26–36, Piscataway, NJ, USA, 2019. IEEE Press.

[10] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM.

[11] AMD. AMD a-series processors. http://www.amd.com/en-us/products/server/opteron-a-series.

[12] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb 1996.

[13] Anandtech. The arm diaries, part 1: How arm's business model works, June 2013. https://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works/2.

[14] Anandtech. Ibm offers power technology for licensing, forms openpower consortium, August 2013. https://www.anandtech.com/show/7204/ibm-offers-power-technology-for-licensing-forms-openpower-consortium.

[15] Anandtech. Intel's architecture day 2018: The future of core, intel gpus, 10nm, and hybrid x86, December 2018. https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86/7.

[16] Anandtech. Western digital reveals swerv risc-v core, cache coherency over ethernet initiative, December 2018. https://www.anandtech.com/show/13678/western-digital-reveals-swerv-risc-v-core-and-omnixtend-coherency-tech.

[17] Anandtech. Arm announces neoverse n1 & e1 platforms & cpus: Enabling a huge jump in infrastructure performance, February 2019. https://www.anandtech.com/show/13959/arm-announces-neoverse-n1-platform.

[18] InfiniBand Trade Association et al. Infiniband architecture specification: Release 1.0, 2000.

[19] Giuseppe Attardi, A Baldi, U Boni, F Carignani, G Cozzi, A Pelligrini, E Durocher, I Filotti, Wang Qing, M Hunter, et al. Techniques for dynamic software migration. In *Proceedings of the 5th Annual ESPRIT Conference (ESPRIT'88)*, volume 1. Citeseer, 1988.

[20] Joshua Auerbach, David F Bacon, Ioana Burcea, Perry Cheng, Stephen J Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, pages 271–276. ACM, 2012.

[21] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[22] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, 2014. USENIX Association.

[23] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[24] Antonio Barbalace, Rob Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-ren Chuang, and Binoy Ravindran. Breaking the bounadaries in heterogeneous-ISA datacenters. In *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (under submission)*, 2017.

[25] 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, New York, NY, USA, 2017. ACM.

[26] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Binoy Ravindran. Towards operating system support for heterogeneous-isa platforms. In *Proceedings of the 4th Workshop on Systems for Future Multicore Architectures (4th SFMA)*, 2014.

[27] 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 Tenth European Conference on Computer Systems*, page 29. ACM, 2015.

[28] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec 2007.

[29] A. Basumallik, S. J. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[30] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[31] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[32] Eli Bendersky. Position Independent Code (PIC) in shared libraries, November 2011. https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries.

[33] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[34] 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*, CCS '15, pages 268–279, New York, NY, USA, 2015. ACM.

[35] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242, May 2014.

[36] 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*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[37] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[38] The OpenMP Architecture Review Board. OpenMP application program interface, November 2015. http://openmp.org/wp/openmp-specifications/.

[39] Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont De Dinechin, and Fabrice Rastello. *Computing Liveness Sets for SSA-Form Programs*. PhD thesis, INRIA, 2011.

[40] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275, March 2003.

[41] D. Bruening and Q. Zhao. Tutorial: Building Dynamic Tools with DynamoRIO on x86 and ARM, March 2016. http://dynamorio.org/tutorial-cgo16.html.

[42] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. ACM.

[43] 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*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[44] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568, May 2012.

[45] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[46] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 139–149, New York, NY, USA, 1998. ACM.

[47] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, volume 14. Boston, MA, 2010.

[48] Cavium. ThunderX ARM Processor; 64-bit ARMv8 Data Center & Cloud Processors for Next Generation Cloud Data Center, HPC and Cloud Workloads. http://www.cavium.com/ThunderX_ARM_Processors.html.

[49] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[50] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[51] 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*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[52] X. Chen, H. Bos, and C. Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 514–529, April 2017.

[53] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[54] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the*

Sixth Conference on Computer Systems, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[55] IBM Cloud. Power 8 — softlayer, October 2016. http://www.softlayer.com/info/power8.

[56] CNBC. Amazon's cloud unit launches arm-based server chips, November 2018. https://www.cnbc.com/2018/11/26/aws-launches-arm-based-server-chips.html.

[57] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[58] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[59] Intel Corporation. Accelerate performance with 7th gen intel core processor family, 2016. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/7th-gen-core-family-brief.pdf.

[60] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[61] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 210–220, Piscataway, NJ, USA, 2017. IEEE Press.

[62] Tracy Counts. Running average power limit – RAPL, June 2014. https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl.

[63] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[64] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, May 2015.

[65] 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 Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[66] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[67] Matthew De Vuyst, Rakesh Kumar, and Dean M Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[68] Jeff Dean. Latency numbers every programmer should know, 2019. https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html.

[69] Solar Designer. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63, August 1997.

[70] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM.

[71] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM.

[72] Maxim Dounin. nginx security advisory (cve-2013-2028), 2013 2013. mailman.nginx.org/pipermail/nginx-announce/2013/000112.html.

[73] F Brent Dubach and CM Shub. Process-originated migration in a heterogeneous environment. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 98–102. ACM, 1989.

[74] Extremetech. Ampere emag 64-bit arm server platform targets intel data centers, September 2018. https://www.extremetech.com/computing/277242-ampere-emag-64-bit-arm-server-platform-targets-intel-data-centers.

[75] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[76] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile mpi programs in computational grids. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 22–31, New York, NY, USA, 2006. ACM.

[77] Adam Ferrari, Steve J Chapin, and Andrew Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, 2000.

[78] R. T. Fielding and G. Kaiser. The apache http server project. *IEEE Internet Computing*, 1(4):88–90, July 1997.

[79] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, May 2017.

[80] The OpenPOWER Foundation. OpenPOWER, October 2016. http://openpowerfoundation.org/.

[81] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[82] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 313–323, New York, NY, USA, 1998. ACM.

[83] Joachim Gehweiler and Michael Thies. Thread migration and checkpointing in java. *Heinz Nixdorf Institute, Tech. Rep. tr-ri-10-315*, 2010.

[84] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. Live and heterogeneous migration of execution environments. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1254–1263. Springer, 2006.

[85] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, September 1999.

[86] GNU. GNU binutils, September 2016. http://sourceware.org/binutils/.

[87] GNU. GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical report, 2017. https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[88] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, 2012.

[89] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 17, 2011.

[90] Dominik Grewe and Michael FP O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *International Conference on Compiler Construction*, pages 286–305. Springer, 2011.

[91] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[92] Khronos Group. The OpenCL specification, March 2016. https://www.khronos.org/registry/cl/.

[93] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the capacity limitations of hardware transactional memory. In *7th Workshop on the Theory of Transactional Memory (WTTM)*, 2015.

[94] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[95] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41:33–38, 2008.

[96] Jay P Hoeflinger. Extending openmp to clusters. *White Paper, Intel Corporation*, 2006.

[97] Serve The Home. Intel xeon phi knights mill for machine learning, August 2017. https://www.servethehome.com/intel-knights-mill-for-machine-learning/.

[98] HSA Foundation. HSA Platform System Architecture Specification v1.1, January 2016. http://www.hsafoundation.com.

[99] Jan Hubicka, Andreas Jaeger, Michael Matz, and Mark Mitchell. System V application binary interface, July 2013. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

[100] IBM. POWER9 Servers Overview. https://www.ibm.com/downloads/cas/KDQRVQRR. Accessed on 3-15-2019.

[101] IBM. Variable length arrays. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_71/rzarg/variable_length_arrays.htm. Accessed on 3-18-2019.

[102] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. Acccessed on 3-21-2019.

[103] Intel. Intel advanced encryption standard instructions (aes-ni), February 2012. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni.

[104] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2017.

[105] Intel. Intel® 64 and ia-32 architectures software developer manuals, February 2019. https://software.intel.com/en-us/articles/intel-sdm.

[106] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[107] Brian Jeff. Big.little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1143–1146. ACM, 2012.

[108] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 24–35, New York, NY, USA, 2016. ACM.

[109] David Katz, Antonio Barbalace, Saif Ansary, Akshay Ravichandran, and Binoy Ravindran. Thread migration in a replicated-kernel os. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 278–287. IEEE, 2015.

[110] The kernel development community. Userfaultfd - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html. Accessed on 1-29-2019.

[111] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348, Dec 2006.

[112] Junghyun Kim, Gangwon Jo, Jaehoon Jung, Jungwon Kim, and Jaejin Lee. A distributed opencl framework using redundant computation and data replication. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 553–569, New York, NY, USA, 2016. ACM.

[113] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[114] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[115] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477, May 2018.

[116] Tim Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[117] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[118] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. A hybrid approach of openmp for clusters. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 75–84, New York, NY, USA, 2012. ACM.

[119] James H Laros III, Kevin Pedretti, Suzanne M Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. Energy delay product. In *Energy-Efficient High Performance Computing*, pages 51–55. Springer, 2013.

[120] Per Larsen and Ahmad-Reza Sadeghi, editors. *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2018.

[121] G. Lee, H. Park, S. Heo, K. Chang, H. Lee, and H. Kim. Architecture-aware automatic computation offload for native applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 521–532, Dec 2015.

[122] 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*, pages 469–480. ACM, 2009.

[123] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[124] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[125] ARM Limited. Procedure call standard for the arm 64-bit architecture (aarch64), May 2013. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf.

[126] Boston Limited. Boston viridis; presenting the world's first hyperscale server – based on arm processors, July 2016. https://www.boston.co.uk/solutions/viridis/default.aspx.

[127] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, New York, NY, USA, 2014. ACM.

[128] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[129] Felix Lindner. Cisco ios router exploitation. *Black Hat USA*, 2009.

[130] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

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

[132] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[133] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE, 2009.

[134] Rob Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. Operating system process and thread migration in heterogeneous platforms. In *Proceedings of the 2016 Workshop on Multicore and Rack-scale Systems (MaRS'16)*, 2016.

[135] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, volume 2009, 2009.

[136] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[137] Marvell. Chinese internet giant baidu rolls out world's first commercial deployment of marvell's arm processor-base server, February 2013. http://www.marvell.com/company/news/pressDetail.do?releaseID=3576.

[138] Nathaniel McIntosh, Sandya Mannarswamy, and Robert Hundt. Whole-program optimization of global variable layout. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 164–172. ACM, 2006.

[139] D. A. Menasce. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, Nov 2002.

[140] Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing openmp programs on software distributed shared memory systems. *Int. J. Parallel Program.*, 31(3):225–249, June 2003.

[141] T. Y. Morad, U. C. Weiser, A. Kolodnyt, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, Jan 2006.

[142] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[143] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 144–157, June 2015.

[144] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 174–185, New York, NY, USA, 2007. ACM.

[145] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, 2015. USENIX Association.

[146] Mozilla Developer Network. Transfer-encoding - http. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding. Accessed on 3-15-2019.

[147] Barefoot Networks. Tofino: The world's fastest & most programmable networks. https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/. Accessed on 3-15-2019.

[148] James Niccolai. Qualcomm enters server CPU market with 24-core arm chip, October 2015.

[149] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[150] NVIDIA. Compute unified device architecture programming guide, October 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[151] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. Os support for thread migration and distribution in the fully heterogeneous datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 174–179, New York, NY, USA, 2017. ACM.

[152] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[153] OpenACC-Standard.org. The OpenACC application programming interface, October 2015. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[154] L. Opyrchal and A. Prakash. Efficient object serialization in java. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*, pages 96–101, June 1999.

[155] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[156] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, May 2012.

[157] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[158] Vinicius Petrucci, Orlando Loques, and Daniel Mossé. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[159] The Next Platform. Next-generation thunderx2 arm targets skylake xeons, June 2016. https://www.nextplatform.com/2016/06/03/next-generation-thunderx2-arm-targets-skylake-xeons/.

[160] LLVM Project. The LLVM compiler infrastructure, September 2016. http://www.llvm.org.

[161] LLVM Project. Stack maps and patch points in LLVM, September 2016. http://llvm.org/docs/StackMaps.html.

[162] Jelica Protic, Milo Tomasevic, and Veljko Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[163] R. Rakvic, Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González. Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors. *ACM Trans. Archit. Code Optim.*, 7(2):9:1–9:25, October 2010.

[164] RedisLabs. Redis, October 2016. http://redis.io/.

[165] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[166] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[167] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[168] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[169] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. A page coherency protocol for popcorn replicated-kernel operating system. In *Proceedings of the ManyCore Architecture Research Community Symposium (MARC)*, 2013.

[170] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 431–440, New York, NY, USA, 2009. ACM.

[171] Sascha Schirra. Ropper-rop gadget finder and binary information tool. https://github.com/sashs/ropper, 2014.

[172] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015.

[173] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.

[174] Thomas R. W. Scogland, Wu-chun Feng, Barry Rountree, and Bronis R. de Supinski. CoreTSAR: Adaptive Worksharing for Heterogeneous Systems. In *International Supercomputing Conference*, Leipzig, Germany, June 2014.

[175] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148. IEEE, 2011.

[176] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[177] 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*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[178] 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*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[179] Rahul Shrivastava and V. Krishna Nandivada. Energy-efficient compilation of irregular task-parallel loops. *ACM Trans. Archit. Code Optim.*, 14(4):35:1–35:29, November 2017.

[180] Charles M Shub. Native code process-originated migration in a heterogeneous environment. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 266–270. ACM, 1990.

[181] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The tui system. Technical report, Software Practice and Experience, 1996.

[182] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013.

[183] Dolphin Interconnect Solutions. PXH810 NTB host adapter, October 2016. http://dolphinics.com/products/PXH810.html.

[184] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers (includes sources). In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 68–77, New York, NY, USA, 1995. ACM.

[185] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[186] Richard Strong, Jayaram Mudigonda, Jeffrey C Mogul, Nathan Binkert, and Dean Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 43(2):35–45, 2009.

[187] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[188] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[189] Herb Sutter. Welcome to the jungle, August 2012. https://herbsutter.com/welcome-to-the-jungle/.

[190] The LLVM Project. LLVM OpenMP Runtime Library. Technical report, September 2015. http://openmp.llvm.org/Reference.pdf.

[191] Marvin M Theimer and Barry Hayes. Heterogeneous process migration by recompilation. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 18–25. IEEE, 1991.

[192] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering energy proportionality with non energy-proportional systems-optimizing the ensemble. *HotPower*, 8:2–2, 2008.

[193] Top500.org. Top500 list - november 2018, June 2016. https://www.top500.org/list/2018/11/.

[194] J. Torrellas, H. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[195] Ashish Venkat, Harsha Basavaraj, and DM Tullsen. Composite-isa cores: Enabling multi-isa heterogeneity using a single isa. In *25th IEEE International Symposium on High Performance Computer Architecture. IEEE*, 2019.

[196] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. *SIGPLAN Not.*, 51(4):727–741, March 2016.

[197] Ashish Venkat and Dean M Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 121–132. IEEE, 2014.

[198] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 28–28. USENIX Association, 2009.

[199] VNSecurity. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (cve-2013-2028), May 2013. https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[200] 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 (TOPLAS)*, 16(6):1842–1874, 1994.

[201] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[202] 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*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[203] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 367–382, 2016.

[204] 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.

[205] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[206] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[207] Hervé Yviquel, Lauro Cruz, and Guido Araujo. Cluster programming using the openmp accelerator model. *ACM Trans. Archit. Code Optim.*, 15(3):35:1–35:23, August 2018.

[208] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 13–24, New York, NY, USA, 1987. ACM.

[209] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th international conference on Autonomic computing*, pages 145–154. ACM, 2012.