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

Robert F. Lyerly

Preliminary Examination Proposal 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
Robert P. Broadwater
Changhee Jung
Cameron D. Patterson
Haibo Zeng

October 16, 2016
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 hit fundamental limits in traditional processor scaling, they began 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 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, 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 has serious performance implications, as execution migration is a fundamental mechanism used by schedulers to reach performance or fairness goals.

This thesis describes system software for automatically migrating natively compiled applications across heterogeneous-ISA processors. This thesis describes implementation and evaluation of a complete software stack using real hardware emulating a datacenter. This thesis 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.

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 PCIe. This thesis shows that sub-millisecond state transformation is achievable. Additionally, it shows that for a datacenter-like workload using benchmarks from the NAS Parallel Benchmark suite, the system can trade performance for up to a 66% reduction in energy and up to an 11% reduction in energy-delay product.

This thesis also proposes post-preliminary examination work. The first proposed work is a set of techniques for reducing state transformation latencies further, and an extended study of state transformation using a wider variety of benchmarks. The second proposed work is a new OpenMP runtime which executes threads across heterogeneous-ISA processors to achieve performance and power benefits. The third proposed work is relaxing the constraints of the current prototype to support more diverse architectures, such as migration between

32-bit and 64-bit architectures. The final proposed work is developing a set of techniques which reduce the time to migration using checkpointing.

This work is supported in part by ONR under grant N00014-13-1-0317 and under grant N00014-16-1-2104, AFOSR under grant FA9550-14-1-0163, and NAVSEA/NEEC under grant 3003279297 and grant N00174-16-C-0018.

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 [96, 97]. As traditional uniprocessors hit the clock speed, power, instruction-level parallelism and complexity walls, chip designers have been forced to rethink computer architecture from the ground up. This has led to an explosion in new architectures such as graphics processing units (GPUs), digital signal processors (DSP) and field-programmable gate arrays (FPGA). Additionally, general-purpose CPUs have been re-architected in order to meet energy and performance goals for varying form factors [50, 26]. 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 [50, 20]. The high-performance computing (HPC) community has embraced heterogeneity, with the top two supercomputers in the Top500 list [100] mixing symmetric chip-multiprocessors (CMP) with general-purpose and OS-capable [70] 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 [31]. 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 [46, 68].

Due to the history of how these different CPUs were created, many utilize different instruction set architectures (ISA) [74]. The ISA defines the hardware-software interface, and provides a fundamental definition of how software can execute on a given processor. The ISA is fixed, 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. It is therefore impossible for applications compiled for one ISA to be run another ISA with today's compilers, operating systems and runtimes.

Application migration is necessary to achieve higher performance and improved energy efficiency [73, 56, 105, 102]. Application migration allows the system software to optimize how a given workload executes in the system. Without application migration across heterogeneous-ISA CPUs, the system has limited ability to adapt to application or workload characteristics and may leave significant benefits on the table. 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.

1.1.1 Heterogeneous Datacenters

The x86 instruction set architecture is the most widely processor in datacenters today [65, 82, 49]. Recently, however, there has been a push to introduce the ARM ISA into the server space. Multiple chip vendors including AMD, Qualcomm, APM and Cavium are producing ARM processors for datacenters and the cloud [2, 69, 4, 21]. Additionally, the POWER ISA is regaining relevance, with IBM forming the OpenPOWER foundation by partnering with companies such as Google, NVIDIA, Mellanox and others [34]. Interest in alternative processor architectures is driven by increasing availability of ARM and POWER cloud offerings [66, 59, 24] in addition to traditional x86 services. These new processor architectures promise higher energy proportionality [13], 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 [107]. Datacenter hardware and software designers have proposed many techniques for improving energy efficiency while maintaining acceptable computational capacity [99, 107, 105, 103]. 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 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

Recent works have demonstrated significant advantages for executing migration between tightly-coupled cores with identical ISAs but heterogeneous microarchitectures [46, 68, 95,

50, 56]. Existing mechanisms for execution migration in symmetric multiprocessors (SMP) work without modification for these new processors because all cores share an 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 [46].

More recent works by DeVuyst [29] and Venkat [102, 101] show that there are further performance and energy benefits obtained by migrating between heterogeneous-ISA cores versus ACMPs. Applications may exhibit ISA affinities based on characteristics of the source code, such as register pressure, memory addressing modes, floating-point and SIMD computation, etc. Additionally, migrating execution between heterogeneous-ISA cores provides a defense against return-oriented programming attacks [101]. However, these works simulate a cache-coherent shared memory processor with heterogeneous-ISA cores that does not run an operating system. How are applications built and migrated between heterogeneous-ISA processors in real 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 processors places a large burden on developers because they can no longer use a shared-memory programming model [1]. Instead, developers must reason about application structure and memory layout in order to obtain maximum performance [71, 44, 43]. 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 [73, 107].

One solution for heterogeneous-ISA execution is to use a language-level virtual machine, e.g., a Java virtual machine [60]. In these 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 [36, 37, 40, 23]. However, using these approaches requires applications be rewritten in the interpreted language. Many datacenter applications, e.g., Redis [81], are written using natively-compiled languages 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 of these techniques rely on language-level mechanisms, 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 native application execution across

heterogeneous-ISA processors.

1.2 Contributions

This thesis 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 PCIe point-to-point connection. 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 the OS's capabilities. This thesis makes the following contributions:

- The design and implementation of the Popcorn compiler toolchain is presented. The Popcorn compiler 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 processors.
- The design and implementation of the state transformation runtime is presented. The state transformation runtime efficiently 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.

Although previous works have explored heterogeneous-ISA execution migration [29, 102], none of these works explore it in a real system. In particular, they simulate a non-existent processor without an OS, and thus do not integrate their approach in a full working system. Instead, the Popcorn compiler toolchain and state transformation runtime are designed to enable execution migration in real heterogeneous-ISA systems. This involves cooperating with the OS to invoke the migration, and more importantly, bootstrapping execution post-migration. Additionally, the Popcorn compiler toolchain and state transformation provide a mechanism for attaching to individual threads for state transformation in a multi-threaded environment.

The Popcorn compiler toolchain and state transformation runtime are co-designed with the Popcorn Linux OS [12], a replicated-kernel operating system which provides thread migration and distributed shared virtual memory across heterogeneous-ISA processors. Unlike previous works which implement process migration in simulation, the full Popcorn Linux

software system is demonstrated on a datacenter-like setup. Using this setup, this thesis 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 [9]. Additionally, Popcorn Linux demonstrates up to a 66% reduction in energy and up to an 11% reduction in energy-delay product [54] for a multiprogrammed, datacenter-like workload.

1.2.1 Popcorn Compiler Toolchain

We present the Popcorn Compiler toolchain, which builds multi-ISA binaries suitable for migration across heterogeneous-ISA boundaries. The toolchain natively compiles applications written in 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, and a final post-processing step optimizes the application for efficient state transformation. The compiler is built using clang and LLVM [77] for compilation and GNU gold [39] for linking. Unlike previous works by DeVuyst [29] and Venkat [102], we build multi-ISA binaries with minimal changes to the core data layout mechanisms of the compiler. This allows our implementation to be more easily ported to new architectures.

1.2.2 State Transformation Runtime

This thesis 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. Unlike previous works by DeVuyst and Venkat, this thesis develops a methodology to invoked migration for multi-threaded applications in a real system. This thesis 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. Finally, the evaluation demonstrates that it is possible to eliminate much of the complexity of their compiler while achieving similar state transformation overheads.

1.3 Summary of Proposed Post-Preliminary Examination Work

After the preliminary examination, this thesis proposes to optimize and extend the state transformation runtime in order to reduce transformation costs. The runtime can be optimized by removing the use of DWARF debugging information and by adding on-demand state transformation. Reducing transformation overheads would allow migration at granularity finer than function call sites, giving the scheduler more freedom to adapt the system workload. This thesis also proposes studying an expanded set of benchmarks in order to more fully understand which types of applications benefit from cross-ISA execution.

This thesis proposes implementing a new OpenMP runtime which uses cross-ISA execution migration to distribute work across all available processors. In particular, it proposes implementing the OpenMP runtime so that it distributes work to balance performance and energy efficiency, allowing system administrators to adjust application execution to reduce latency or increase energy efficiency. To further optimize cross-ISA work distribution, this thesis proposes studying compiler and runtime techniques for enabling disjoint memory access parallelism across architectures. This is necessary due to the fact that Popcorn Linux provides shared memory across processors at a page-level granularity.

This thesis also proposes adding support for more diverse ISAs in the Popcorn compiler toolchain and state transformation library. Currently, the toolchain and runtime only support ISAs which have identical data sizes and layouts. Additionally, it only supports 64-bit architectures. This thesis proposes extending the toolchain and runtime to support architectures which have different data sizes and alignments using padding. It also proposes handling 32- and 64-bit architectures, which requires adjusting the layout of an application's virtual address space at migration time.

Finally, this thesis proposes reducing the time to migration with a set of new techniques. The current prototype only supports execution migration at function call sites. This thesis proposes inserting additional migration points in loop nests. It also proposes leveraging lightweight checkpointing to enable immediate execution migration by rolling back to a checkpoint at a migration site. This thesis proposes studying the benefits and tradeoffs of this technique versus the current prototype.

1.4 Thesis Organization

This thesis proposal is organized as follows. Chapter 2 summarizes related work in the area of execution migration in heterogeneous-ISA systems. Chapter 3 describes Popcorn Linux, the replicated kernel operating system used to provide execution migration across ISA boundaries. It also formalizes application state 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. Finally, Chapter 7 concludes and describes proposed post-preliminary examination work.

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 [71] 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 [44], OpenMP 4.0 [17] and OpenACC [72] 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. [87] 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) [43] 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 [35] 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 [32] uses a modified C/MPI compiler to instrument MPI applications for execution migration in a cluster and uses checkpointing to transferring 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 comes into and goes out of scope, adding significant runtime overhead for metadata collection in addition to checkpointing costs. Popcorn Linux allows flexible execution migration between processors and distributed shared virtual memory.

The Lime programming language [8] and the Liquid Metal runtime [7] 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 [85] and PTask runtime [84] 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 [30] and Shub [90] 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 [106] shows that state transformation can also be applied as pages are migrated between machines, rather than in bulk at migration time. Theimer and Hayes [98] 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 [92] or Infiniband [5], 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. [6] 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 [91] 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. [33] 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 [64] 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. [104] 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 utilizes and extends their definition of points of equivalence.

Many works use language-level virtual machines to perform heterogeneous-ISA migration. Heterogeneous Emerald [93] implements a TUI-like heterogeneous migration system for the Emerald language. PadMig [36] and JnJVM [37] migrate threads of execution between Java virtual machines (JVM), using Java's reflection capabilities to serialize/de-serialize objects between architecture-specific formats. COMET [40] and CloneCloud [23] 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 no 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 [15] 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. Barbalace et al. [12] 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 [29] and Venkat [102, 101] implement process migration in simulated heterogeneous-ISA CMPs. 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. Their prototype uses a simulated heterogeneous-ISA CMP with cache-coherent shared

memory. Furthermore, their prototype does not incorporate an operating system. Popcorn Linux demonstrates execution migration on real hardware using an ARM and an x86 processor are interconnected via PCIe and using a complete software stack.

2. Their prototype does not support multi-threaded applications. Their compiler does not support aligning thread local storage, and they do 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 for multi-threaded applications.

3. In order to perform stack transformation between ISA-specific formats, their 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. Their compiler generates a mostly-identical call frame layout across different compilations of the application. This adds tremendous complexity to the compiler, and makes porting their toolchain to new architectures difficult. The Popcorn compiler toolchain demonstrates that this additional complexity is unnecessary. While the design of the Popcorn compiler toolchain forces the state transformation runtime to fix up pointers to the stack, the evaluation demonstrates that our runtime provides similar performance.

4. Their 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. They 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. Their 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 must cooperate to migrate threads. Although this hinders the scheduler's flexibility, it significantly reduces migration costs. Their results show that DBT can cause up to a several millisecond delay when migrating.

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 [29, 102, 101, 12]. 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 [94]. Current monolithic kernel OSs like Linux provide thread migration in SMP systems through hardware and OS mechanisms [73]. 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 [14] 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 [57] 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 [12] 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, 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.

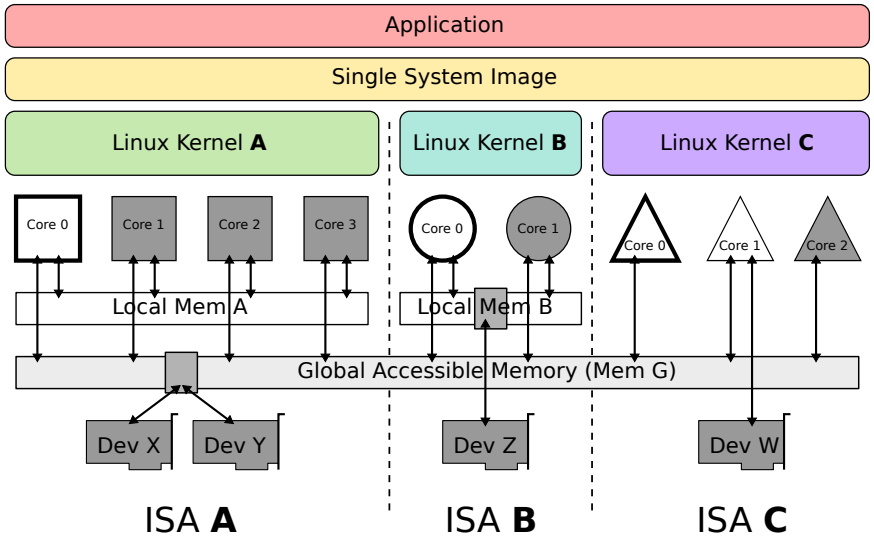


Figure 3.1: Replicated-kernel OS architecture and application interface

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 [53].

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 [104], 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. This requirement is relaxed to allow the compiler to further optimize generated machine code while retaining semantic equivalence with the stricter version of this property.
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 shared memory for heterogeneous processors [29, 102, 47], no commercially available heterogeneous-ISA CMPs currently exist that support shared memory. In order to sidestep this issue, the replicated-kernel OS provides distributed shared virtual memory (DSVM). 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, or by migrating memory objects between memory regions to increase data access locality. The DSVM system provides the illusion of a single shared memory region overlaid across a set of nodes, allowing applications to be developed using a shared-memory programming model [80].

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 [86] across kernels that acts like a multiple-reader, single-writer lock on pages. When application threads executing on a single kernel access a page, there is no coherency required. When threads executing on different kernels access a page with read-only permissions, the page is replicated across both kernels. This allows both scalability across kernels and data access locality. However, when the page has both read and write permissions, only one kernel may own the page at a time. When a migrated thread accesses the writable 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. 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 memory, and supports ISA-specific locking mechanisms across architectures. However it can lead to pathological behavior and poor performance when threads spread across multiple processor islands access the same pages [86].

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 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 [19, 67, 38]. 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,

[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.

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 [29, 102].

3.2.2 Laying Out Application State

Attardi et al. [6] and Smith and Hutchison [91] 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. [29] and Venkat and Tullsen [102] 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, the compiler and linker lay out symbols in P_C and P_D at common addresses across all compilations of the application, meaning 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 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, symbols in P_C, P_D and P_H are **aligned** across all compilations.

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.

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 [25], 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.

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

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 [58, 48]:

- **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 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].

[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`.

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.

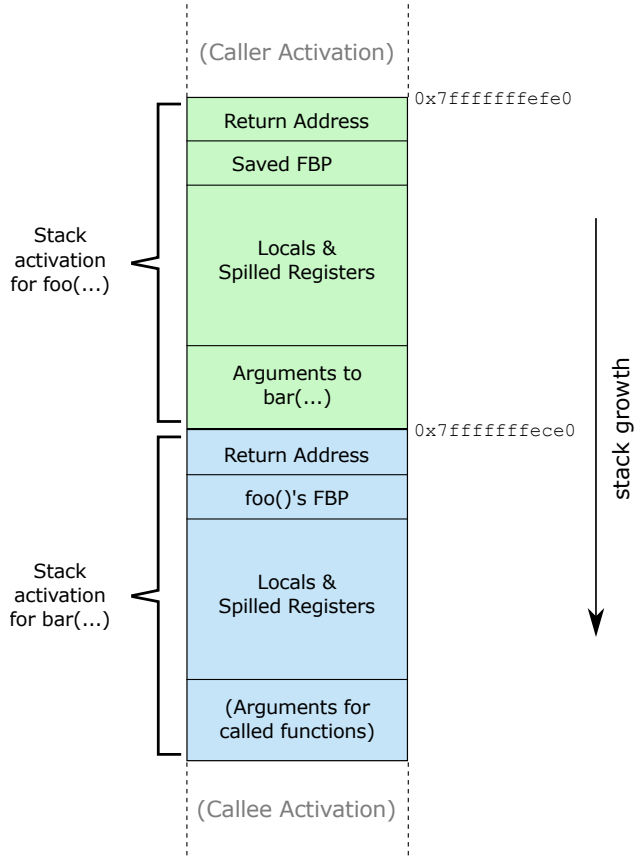


Figure 3.2: Stack frame layout. The stack includes call frames for function `foo(..)`, which calls function `bar(...)`.

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 3.2 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 3.2, `bar`'s call frame saves the instruction address at which execution will resume when returning to `foo`.

- **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 3.2, `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 3.2, `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 and 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 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, and 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 [77] as the compiler and a modified GNU gold [39] as the linker. The toolchain also uses several custom-built tools for post-processing binaries in preparation for migration. Figure 4.1 shows an overview of how application source 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). 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 aligning function and data symbols at identical

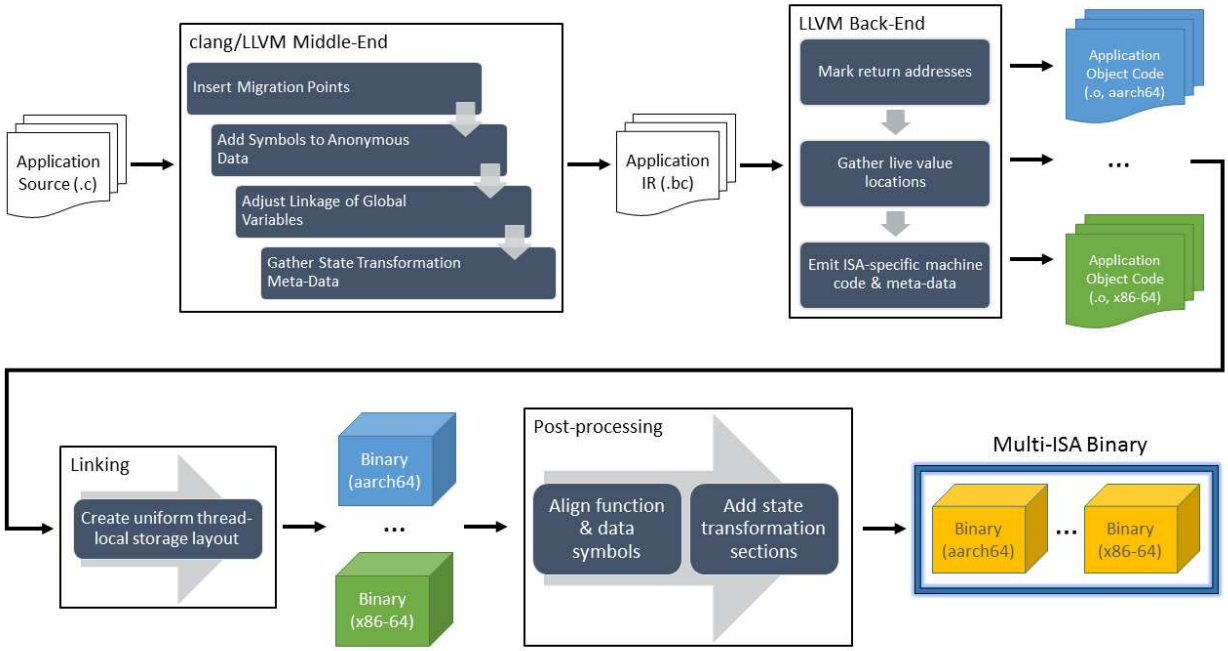


Figure 4.1: Popcorn compiler toolchain

virtual addresses across all binaries. Additionally, post-processing adds state transformation metadata. At this point the multi-ISA binary has been built and is ready for execution migration across kernesl 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. 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 various locations throughout the IR in preparation for runtime state transformation (Section 4.3).

- **Backend Analysis** (LLVM back-end) – Several backend 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 x86-64 layout [10].

- **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 (while 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 [10].
- **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. 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 a few 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 and `setjmp/longjmp` 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 [12, 10, 63, 11]. This thesis 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[1].

2. **Live values must be in memory** – this requirement is relaxed so that live values may be in memory or in registers at function call sites. This is semantically equivalent due to 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.

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 library. At application startup, the main thread maps a shared page between the kernel and the application. When the scheduler requests that a thread migrate, it sets a flag on this page. At migration points, threads check to see if this flag has been set, and if so, begin 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

[1]The current implementation of the compiler does not support architecture-specific live values. This is planned for future work as a requirement for supporting new benchmarks.

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. [18], 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 [79]. 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.
- **Program Location** – The stack map record includes information about the function which contains the stack map. It 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.
- **Call Frame Size** – In addition to encoding the instruction address, the record stores the size of the call frame for the containing function. This allows the transformation runtime to construct call frames for the transformed 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.

It is important to note that the stack map intrinsic does not add any overhead to the generated code – it does not cause any additional machine instructions to be generated and it does not change where live values are allocated. Stack maps prevent the `-fomit-frame-pointer` optimization because 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 able to be captured in the middle end. LLVM implements IR lowering to machine code using a set of target-independent analyses and transforms, meaning our modifications are available for all targets supported by LLVM. Unlike previous works [101, 102, 29], 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 were 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 is more prevalent on RISC architectures, because live values must be loaded from and stored to memory in order to do computation on them. However, these problems also arise on CISC architectures, depending on the results of register allocation.

A liveness range checking was implemented 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 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[2] are checked. 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.

[2]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. Problems only arise when promoting values from stack slots to registers or when copying values between registers.

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 transformation runtime does not need to worry about constants, as the machine code will create them as needed.

The following fields were added to location records using IR-level information in order to handle several cases:

- **Pointer** – flag indicating if the value is a pointer. The state transformation runtime requires special handling for pointers to the stack, 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. This flag indicates that the live value is allocated to the stack.
- **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).

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 in a suitable format for state transformation.

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 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, gathers all data into a single monolithic format and appends three extra sections to the multi-ISA binary.

The first two sections provide stack map records which contain the following fields:

1. **ID** – the stack map or rewriting site ID.
2. **Program location** – return address of the function call defining the rewriting site.
3. **Call frame size** – size of the call frame.
4. **Number of live values** – number of live values at the rewriting site.
5. **Live value records offset** – offset into the live value location record section, as described below.

The first two sections provide stack map record tables sorted by ID and program location, respectively, as shown in Figure 4.4. These sections provide a dictionary 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 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. Because all records have a constant size, the transformation runtime can use a binary search to quickly locate records, given a program location or stack map ID.

ID	Program Location	Call Frame Size	# Live Values	Offset in Live Value Section
0	`0x410126`	112	19	0
1	`0x41013b`	32	5	19
2	`0x410264`	16	2	24
3	`0x4104ec`	32	4	26
4	`0x410210`	16	3	30
...	...			

(a) Stack map records, sorted by ID

ID	Program Location	Call Frame Size	# Live Values	Offset in Live Value Section
0	`0x410126`	112	19	0
1	`0x41013b`	32	5	19
4	`0x410210`	16	3	30
2	`0x410264`	16	2	24
3	`0x4104ec`	32	4	26
...	...			

(b) Stack map records, sorted by program location

Figure 4.4: Stack map record sections

The third section added to the multi-ISA binary contains live value location records for all stack maps. The transformation runtime finds live variables in a function activation by reading the offset and the number of location records from the stack map record and pulling the records from this third section. Figure 4.5 shows example location records.

These records are also constant size, meaning the transformation runtime can directly jump to a record given an offset.

At this point, the multi-ISA binary has finished compilation and is ready for execution on Popcorn Linux.

Type	Register	Offset	Is Pointer?	Is Alloca?	Stack Size
Register	12	(n/a)	No	No	(n/a)
Register	14	(n/a)	Yes	No	(n/a)
Stack Variable	(n/a)	16	Yes	Yes	32
Stack Variable	(n/a)	24	Yes	No	8
...					

Figure 4.5: Live value location records

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.

The runtime operates in user-space for several reasons. First, it allows the runtime to use many interfaces that are not as well supported in kernel space, such as DWARF debugging information. Second, it provides a cleaner separation of responsibilities. Pushing state transformation into the kernel requires integrating application-specific logic into kernel space, 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, our prototype performs state transformation in user-space due to the aforementioned benefits. Our state transformation runtime differs from that of DeVuyst [29] and Venkat [102] in that our runtime reconstructs the destination stack from the source stack while their implementation does in-place modification. This is an artifact of our choice not to unify call frame layout in the compiler.

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.

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:

- **ISA ID** – a numeric ID uniquely identifying the architecture, as defined by ELF.
- **Pointer Size** – size of pointers as defined by the ISA's ABI. This is always 8 bytes (64-bit) in our prototype.

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

- **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 (i.e., register size, which registers are callee-saved) and stack properties (i.e., stack pointer alignment).
- **DWARF Frame Unwinding Metadata** – DWARF data structures which describe the call frame unwinding procedure for all functions in the application.
- **Stack Map/Rewriting Site Records** – the metadata generated by the compiler (described in Section 4.5) to locate rewriting sites in the machine code and to locate live variables at those sites.
- **Frame Unwinding Information for Starting Functions** – frame unwinding information for thread start functions, e.g., `__libc_start_main` from the C standard library. Rather than querying this information for each state transformation, the runtime caches it for quick use because the last activation on any thread's stack will correspond to one of these functions.

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 instantly available for threads to perform transformation when the scheduler requests migrations. Unfortunately the current runtime prototype uses a DWARF library which is not thread safe, meaning that threads cannot concurrently access handles. This is an implementation issue, however, and is not an inherent design issue – none of the metadata contained in the handles is ever changed, meaning multiple threads would be able to concurrently access the rewriting metadata in a more complete implementation.

The runtime must also prepare the main thread's stack for transformation due to a quirk in 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. When stack growth causes a page fault, Linux checks to see if the access is on a page adjacent to a previously allocated stack page. If so, Linux maps a 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 on a page adjacent to a currently allocated page Linux traps it as 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 touches all pages in the main thread's stack

[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.

area so that they are ready for rewriting. This is not a problem for threads forked by the threading library, as the library allocates stack memory by using `mmap` or `malloc`, both of which sidestep this issue.

Finally, the runtime maps in a shared page between the kernel and the application. The scheduler sets flags on this page in order to request migrations, as requested in Section 4.2.

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, it copies all of its register state into memory so that the thread can continue execution (to transform its stack) while operating on a snapshot of the thread at the migration point. The thread then calls into the runtime to begin transformation.

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). Because the runtime does not know for which ISA the context will be used, it does not which registers require unwinding per activation. Rather than dynamically allocating a buffer for this

information as activations are being 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 stack map 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 a 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 DWARF 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 the pseudocode 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 called 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 (Section 4.5). The runtime uses the ID of the record to find the corresponding rewriting site record for the destination. The rewriting site records are cached in the source and destination rewriting contexts, respectively. The destination stack size is updated using the destination record, which contains the size of the call frame for the

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

function in which it is contained. Finally, the source activation is unwound from the source stack, which sets the source activation to its caller.

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 shell 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 frame base pointer 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.
- **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 must only 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 as call 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. If the runtime finds an activation that saves the register, it duplicates 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 outermost activation and the runtime duplicates 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 of the IR, and therefore the register allocator is responsible for allocating storage for the same

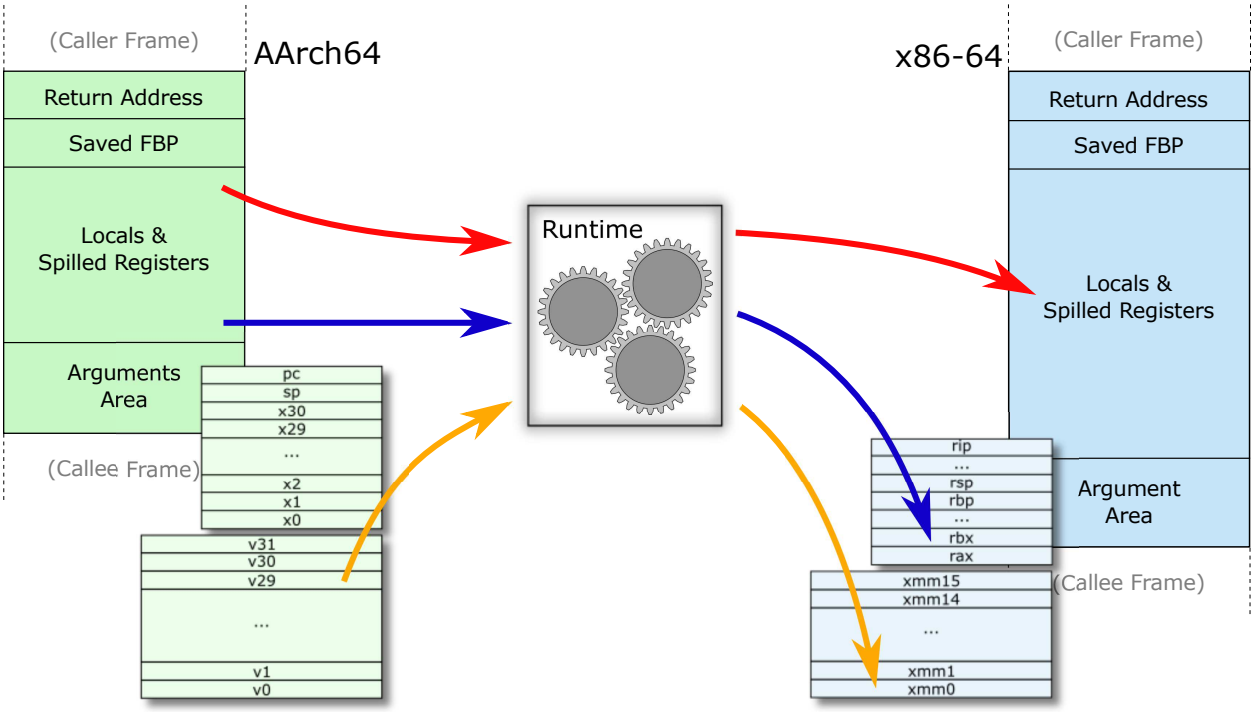


Figure 5.1: An example of the state transformation runtime copying live values between source (AArch64) and destination (x86-64) activations.

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.

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 applies the DWARF call frame unwinding procedure to the destination activation (it has already been applied to 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, and sets the current activation to the caller activation.

The runtime repeatedly transforms activations until it gets to 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 DeVuyst's [29] and Venkat's [102] work, because they align pointed-to data in call frames across all ISAs, they do not have to reify pointers to the stack. However, our runtime is able to efficiently handle fixing up pointers.

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 would calculate 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`.

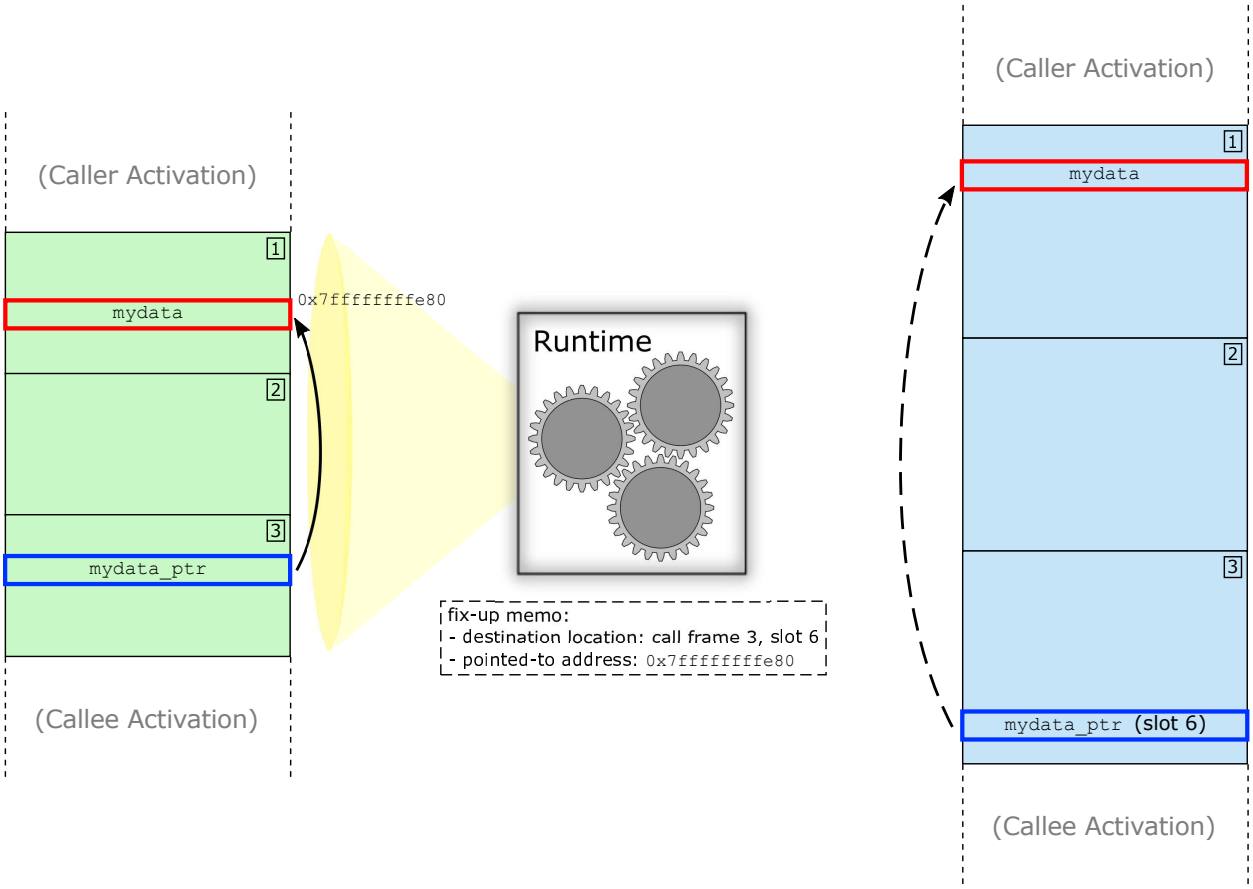


Figure 5.2: Example of the runtime observing and transforming a pointer for the destination activation.

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 thread returns to the migration point and initiates migration. The thread saves a pointer to the transformed register set, then invokes a Popcorn Linux-specific system call to migrate to another kernel. The thread passes to the kernel a CPU set describing the destination CPUs, the program counter at which to resume execution, and a pointer to the transformed set of CPUs. 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 (e.g., floating point registers) and cleans up the migration data.

The thread then returns to application code.

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 [9]. 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 communica-

tion 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 1.11.

	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 I2C. This sensor provides instantaneous power for the motherboard's power regulator chips. The Xeon implements Intel's Running Average Power Limit (RAPL) [27], 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.

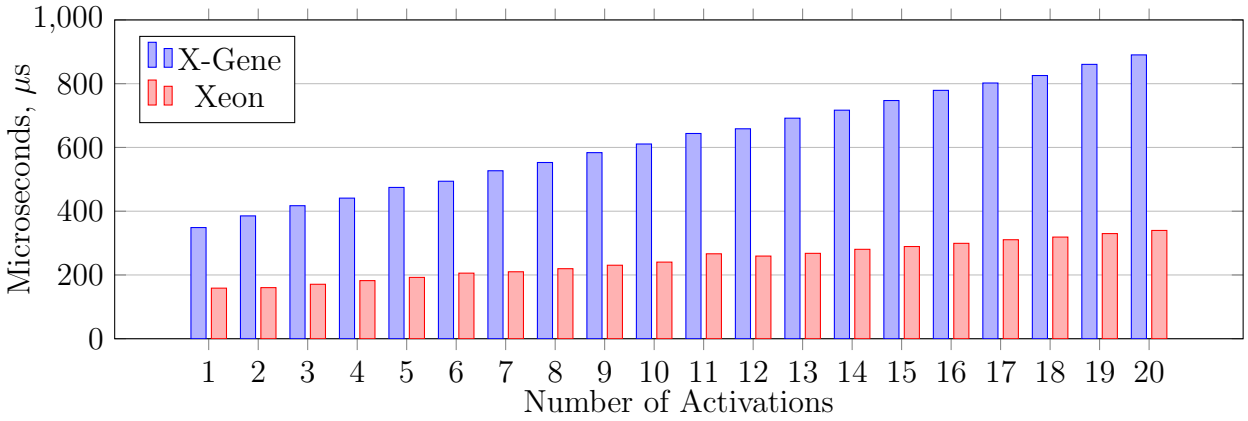
4. **Cleanup** – time required to tear-down and free the rewriting contexts for both the source and destination contexts.

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.

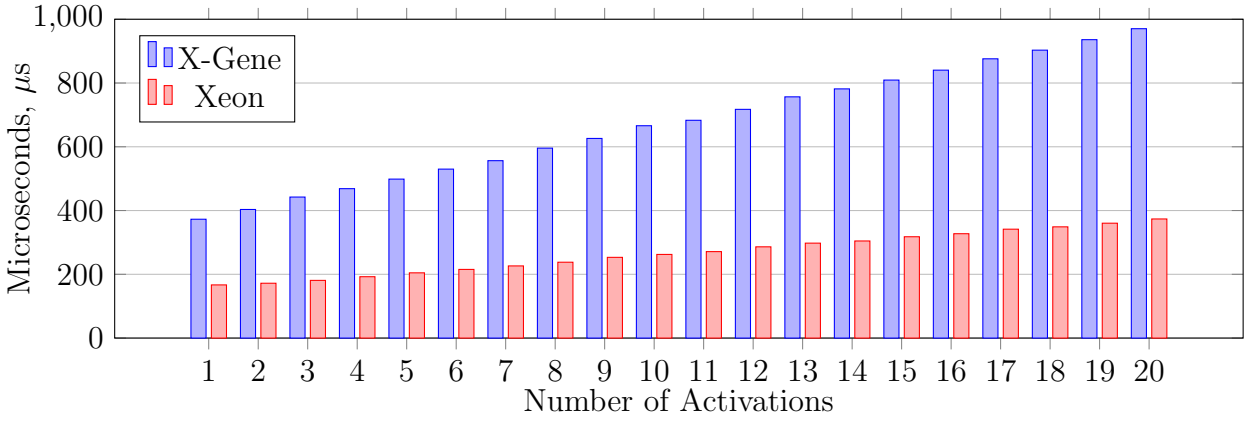
The transformation timing was broken into different actions required per activation. Figure 6.3 shows the percentage of the total transformation latency spent performing the following actions:

- **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.

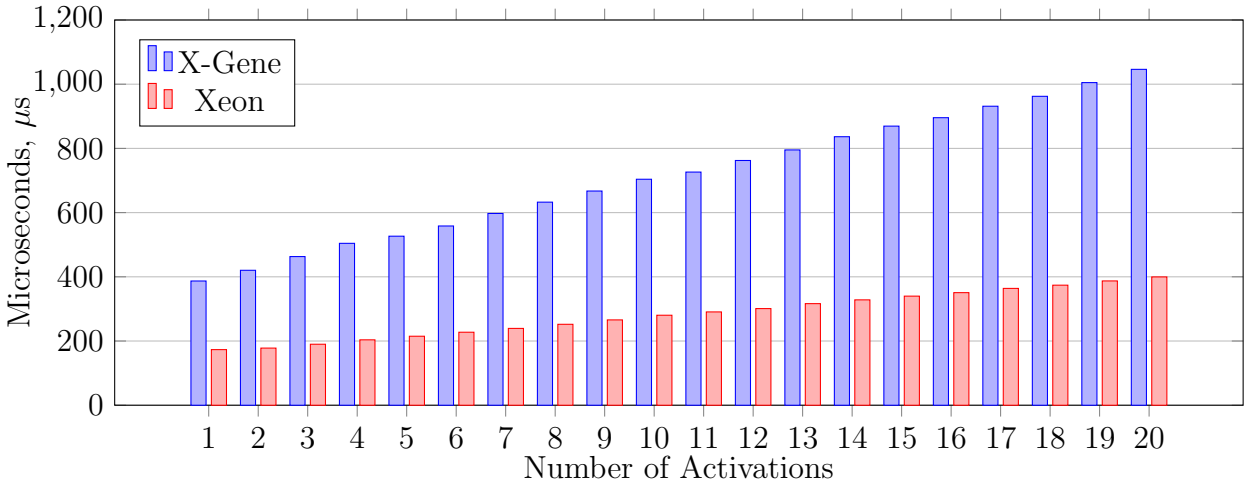
Figure 6.3 shows the timing breakdown in percentage of total transformation time for the aforementioned actions. The breakdowns for both the X-Gene and Xeon are virtually identical, demonstrating that although the total time required is different between the two processors, the costs of the different 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



(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

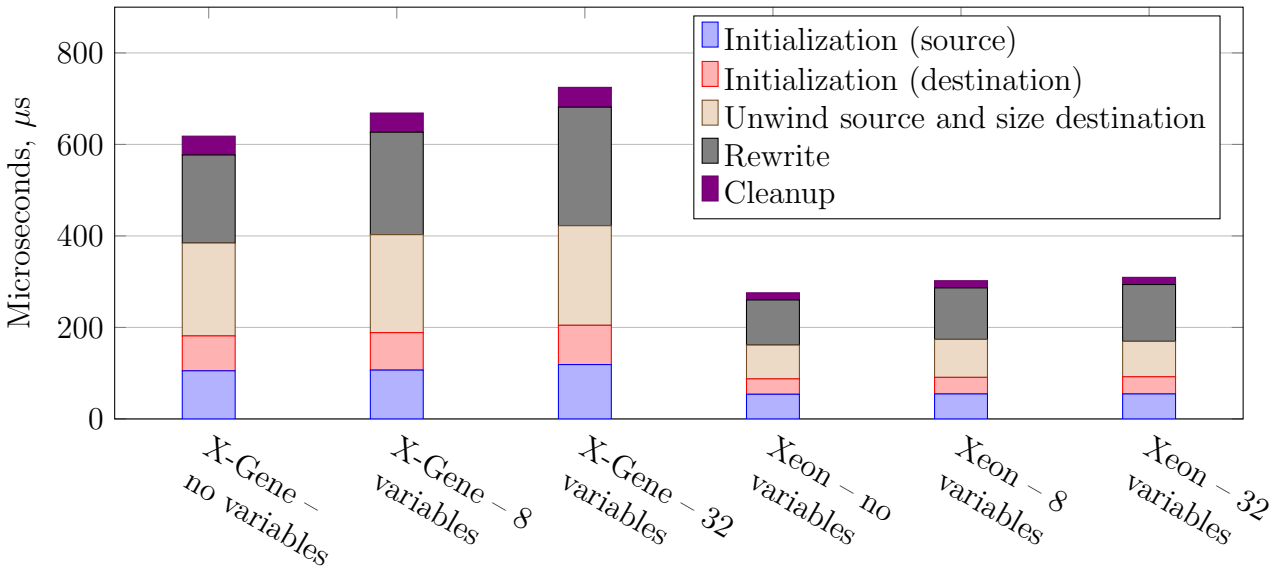


Figure 6.2: Time spent in each phase of state transformation

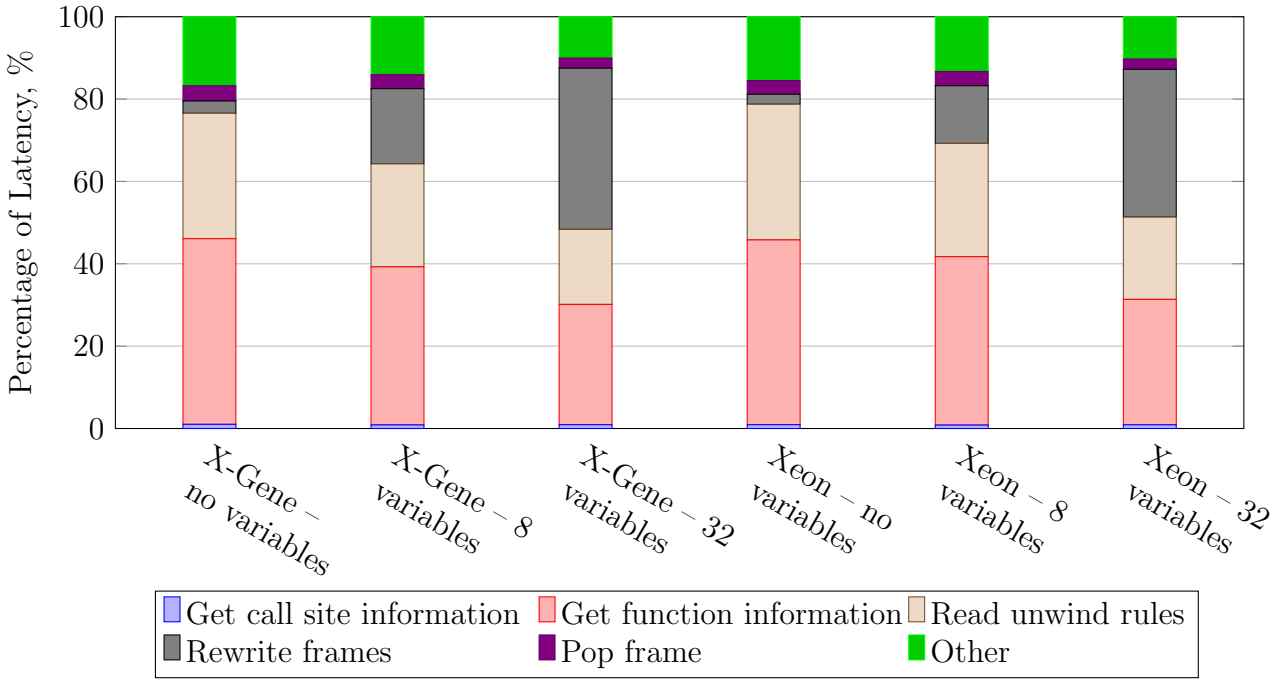


Figure 6.3: Percentage of time spent executing different actions during state transformation

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. Both of these sources of overhead

could be reduced by encoding frame unwinding metadata in a different format.

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.

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 [9, 89] 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 6.4 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 in the main function of each application, which call a few helper routines to do the heavy computation.

Figure 6.5 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.

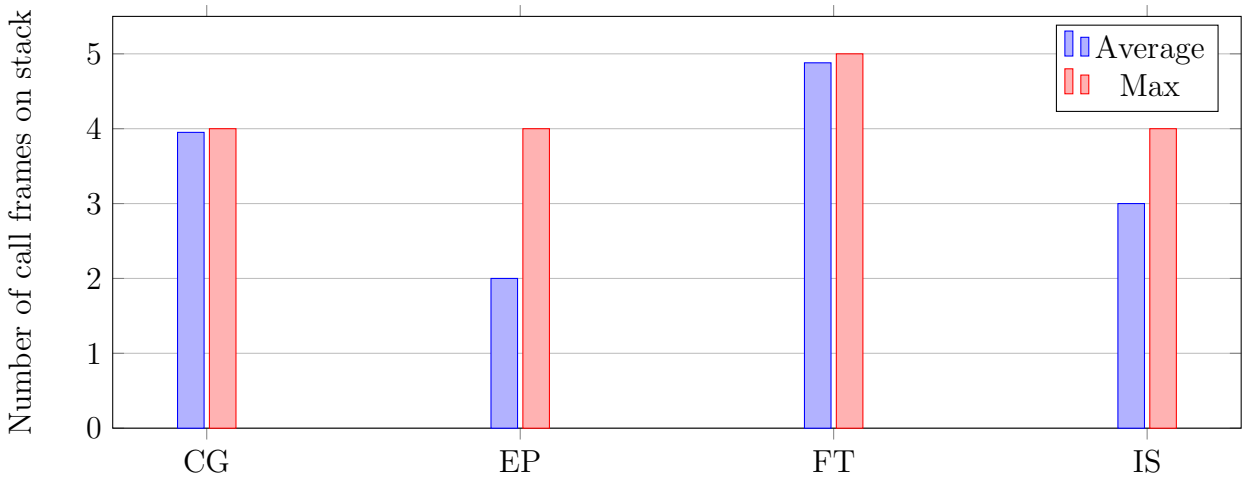


Figure 6.4: Average and maximum stack depths for benchmarks from the NPB benchmark suite

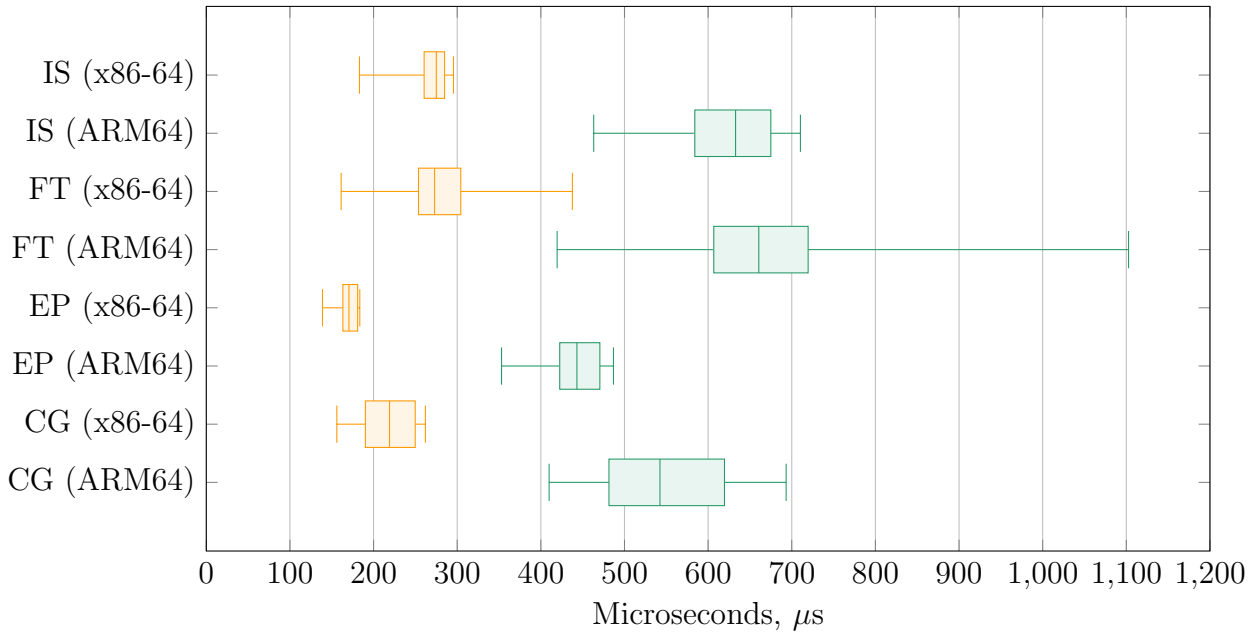


Figure 6.5: State transformation latency distribution for all migration points in real applications. Box-and-whisker plots show the minimum, 1st quartile, median, 3rd quartile, and maximum observed transformation latencies.

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

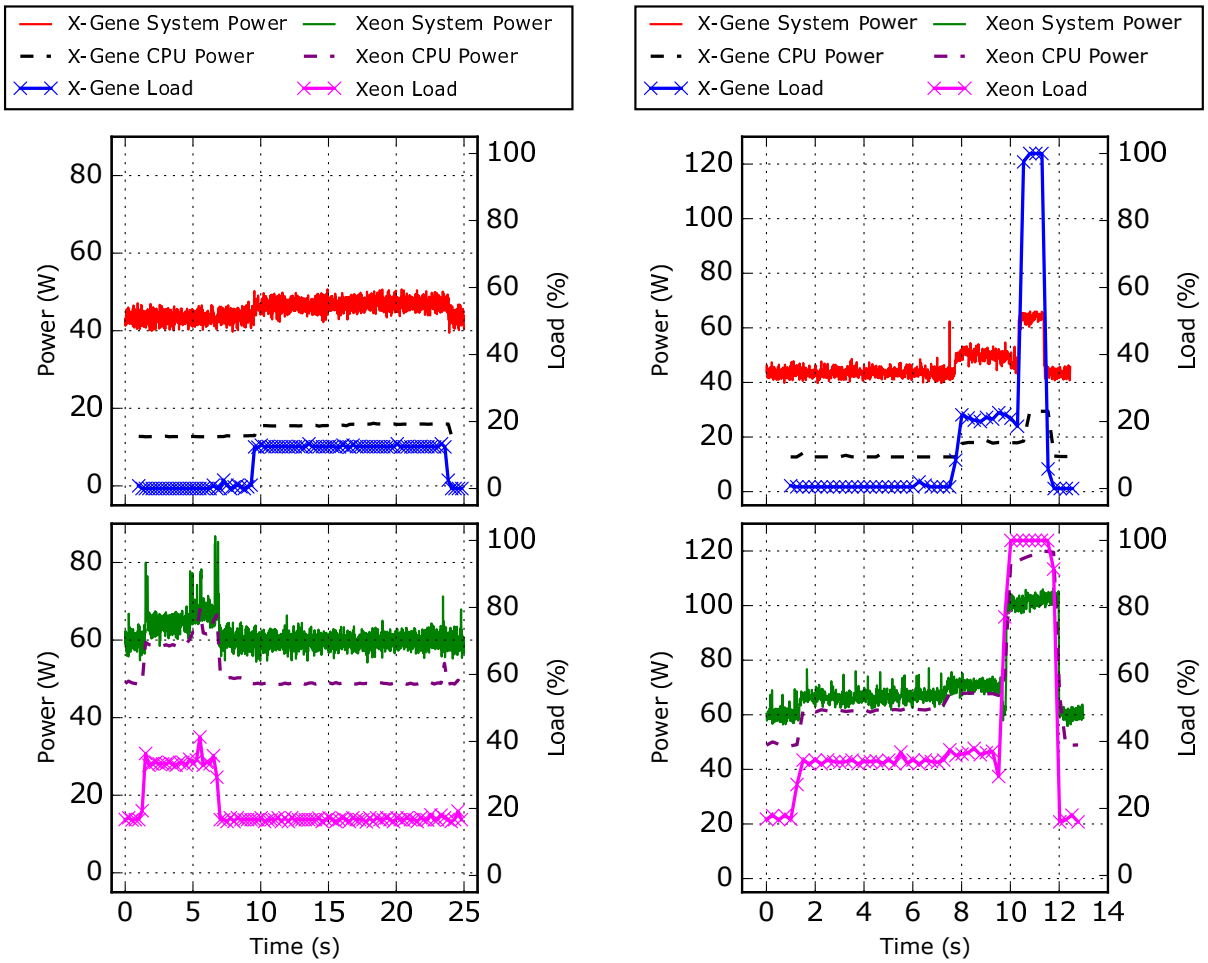
Table 6.2: Time required for executing individual actions on the Xeon

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) [36] 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 with 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 6.6 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.



(a) PadMig execution time and power consumption

(b) Popcorn Linux execution time and power consumption

Figure 6.6: 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.

Figure 6.6 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 6.6a) 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 6.6b. 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 [65] and DeVuyst et al. [28] 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 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.

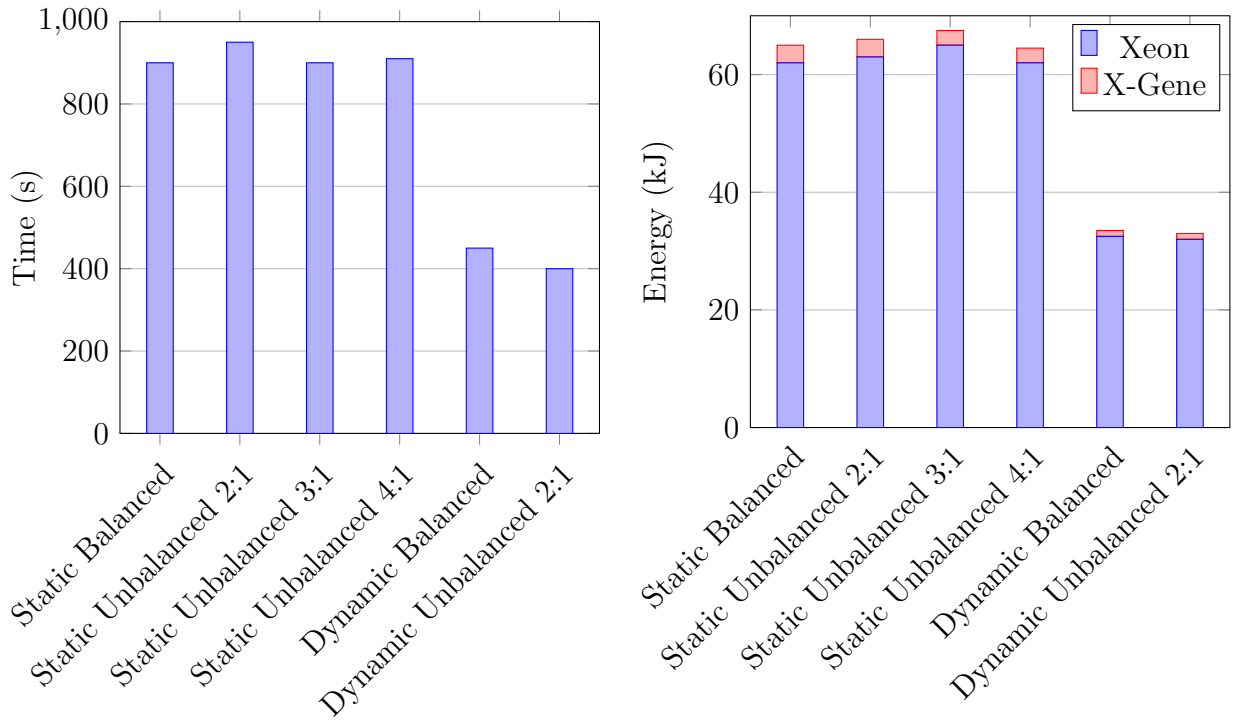
Popcorn Linux provides unique execution capabilities versus vanilla Linux. Without Popcorn Linux's thread migration and DSM support coupled with runtime state transformation, applications cannot migrate between heterogeneous-ISA architectures. This means that jobs can be scheduled onto the X-Gene or 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 [55]. 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 6.7 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).

As seen in Figure 6.7, 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.



(a) Total makespan ratio

(b) Energy consumption

Figure 6.7: Static vs. Dynamic scheduling policies in heterogeneous setup

Figure 6.8 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 6.8, execution migration allows the system to trade off performance versus 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 con-

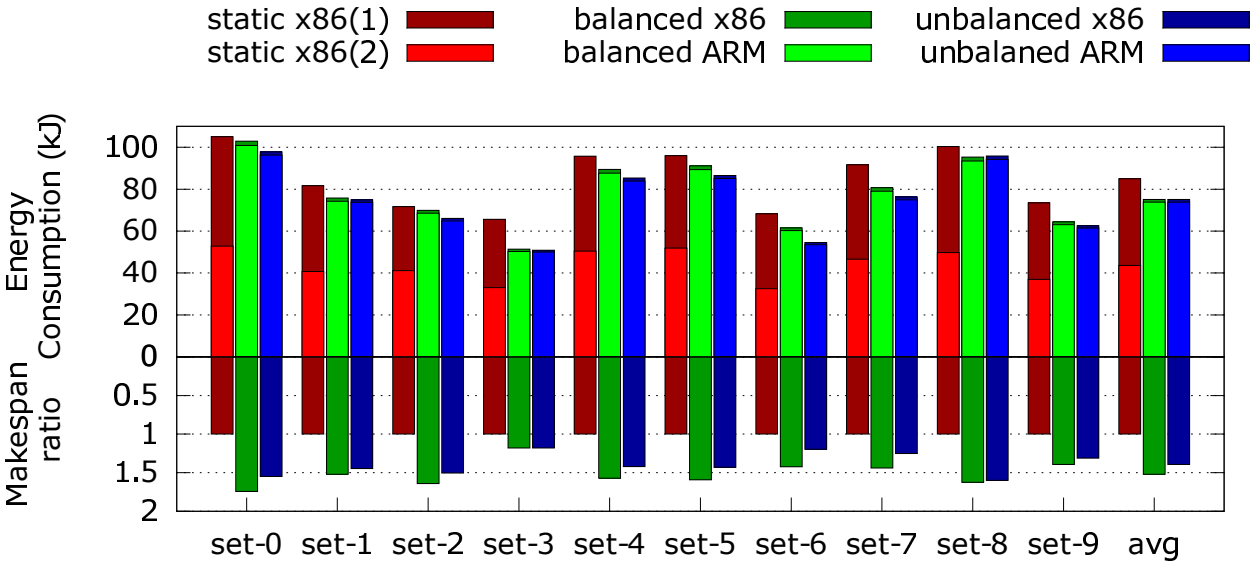


Figure 6.8: Energy consumption and makespan ratio for several single-application arrival patterns

serve 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 6.9 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 6.9 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, 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

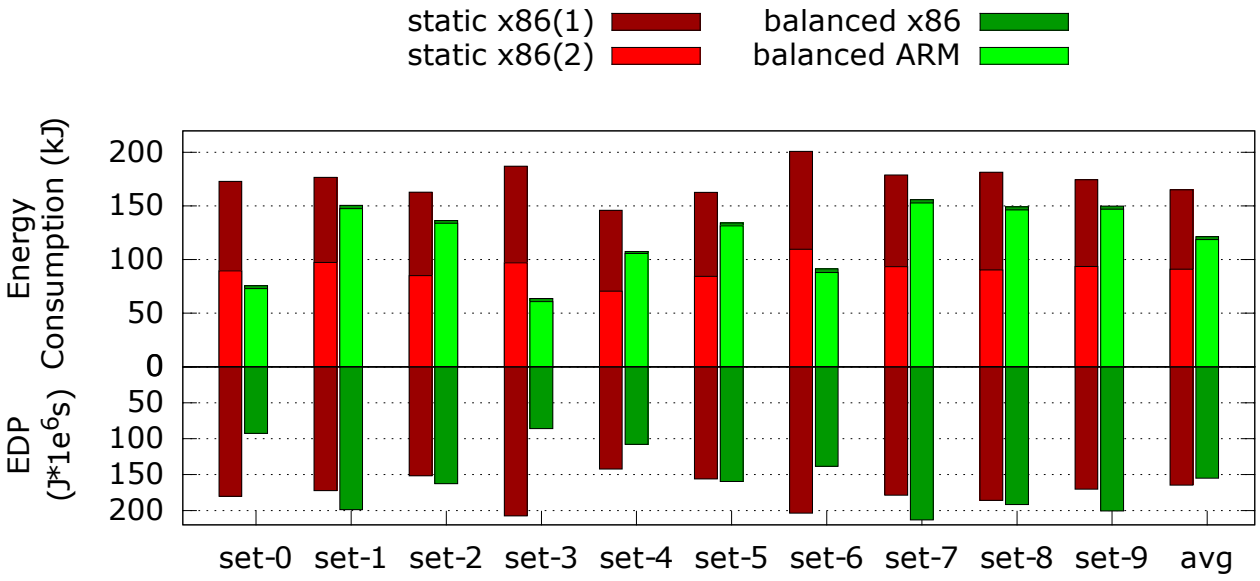


Figure 6.9: Energy consumption and makespan ratio for several clustered-application arrival patterns. Results for Dynamic Unbalanced policy are not shown as they differ by less than 1% from the Dynamic Balanced policy.

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

Conclusion

In this thesis proposal, a full software stack was presented for execution migration across heterogeneous-ISA processors in real systems. This thesis 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 front-end automatically inserts migration points into the source code at function call sites. The middle-end performs a liveness analysis to gather the sets of live values at each call site, and the back-end generates metadata describing where those live values are placed in each ISA-specific version 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.

The system software 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 thesis described the design and implementation of a full software system for migrating threads of execution between heterogeneous-ISA architectures on real hardware. This thesis 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. Additionally, Popcorn Linux can achieve higher energy efficiency versus a system without execution migration.

7.1 Post-Preliminary Exam Proposed Work

In this this chapter, several possible post-preliminary exam research directions are presented. Section 7.1.1 describes several optimizations to the state transformation runtime and further evaluation on a wider variety of benchmarks. Section 7.1.2 proposes applying execution migration to scale applications across heterogeneous-ISA nodes rather than running only on one at a time. Section 7.1.3 proposes extending the current state transformation runtime to more diverse architectures, including applying state transformation to migrate applications between 32-bit and 64-bit architectures. Finally, Section 7.1.4 proposes studying alternative techniques to reduce the time between when the scheduler requests a thread migrate and when the thread can migrate.

7.1.1 Extended Study and Optimization of the State Transformation Runtime

There are several straightforward optimizations for the runtime that can be implemented in order to reduce state transformation costs. Additionally, there are many computational behaviors not captured by NPB which should be studied.

Reducing Per-Activation Lookup Costs

As Figure 6.3 in Section 6.2 showed, there are significant costs associated with finding enclosing function information and call frame unwinding rules for each activation. Both of these actions require accessing DWARF debugging information through libDWARF. This library is not optimized for performance:

- Finding function information for a given program location is implemented as a linear search through a list of address ranges generated by the compiler. This overhead rises as the size of the application grows, as described in Section 6.3.
- The libDWARF library dynamically allocates memory for each function descriptor returned by the library. This descriptor must later be freed, increasing cleanup costs.

This dynamic memory allocation is the sole reason that state transformation requires mutually exclusion between threads.

- The libDWARF library copies out call frame unwinding information to a user-provided buffer rather than returning a pointer to metadata describing the procedure.

All of these operations are inefficient and lead to a much higher state transformation latency than is required. No state transformation metadata changes at run-time, and thus there is no need to do any memory allocation or copying of metadata – all metadata is generated by the compiler offline and simply read by the runtime to direct the transformation process. Using libDWARF also has a secondary impact on performance due to the page coherency protocol. By using writable memory for these data structures, the page coherency protocol must lock and transfer libDWARF data pages between kernels. If the call frame unwinding information was implemented using read-only memory, the page coherency protocol could replicate the pages rather than ping-ponging them between kernels.

We propose extending the Popcorn compiler toolchain to completely remove the use of libDWARF for call frame unwinding information. This would greatly reduce state transformation latency because querying DWARF debugging metadata accounts for a large portion of state transformation costs (see Figure 6.3). An alternative compiler mechanism would encode the call frame unwind procedure, which describes how to restore callee-saved registers and unwind to the previous call frame, in an additional read-only section for each ISA in the multi-ISA binary. Call site records would be augmented to include an offset into this section. Therefore, finding all transformation metadata for a given call site would only require a single binary search through call site records. The runtime would query this information rather than the DWARF metadata, which would eliminate the need for the compiler to generate DWARF debugging information.

On-Demand State Transformation

Another possible optimization to the state transformation runtime is to transform function activations on-demand. Currently, the runtime eagerly transforms all live function activations to the destination ISA's format. Rather than transforming all activations, the runtime would prepare the rewriting context to only transform frames as needed. When an on-demand transformation is requested, the runtime would unwind the source stack to calculate the size of the destination stack as normal. Then, the runtime would only transform the outermost activation, which is all that would be required to resume execution after a migration. The runtime would make one minor modification to the activation – rather than populating the stack frame with a return address into the calling function, the runtime would insert the address of a special handler in the runtime which would transform frames as the thread returns back through the call chain. After transforming the outermost frame, the thread would be migrated and execute as normal to the destination architecture. When the

thread returned from the outermost function, the handler in the runtime would be invoked to transform the next frame on the stack and return to the calling function. This process would continue until either the thread exits or another migration occurs.

Using on-demand state transformation, rewriting costs would be amortized during normal execution. A secondary benefit is that some frames would never need to be transformed between formats at all. If a thread migrates to a new architecture, then migrates back to the original architecture before returning to the innermost activation, some of the frames on the stack will remain in the original ISA's format throughout execution. One downside of on-demand state transformation is that the runtime repeatedly intercepts execution as the thread returns back through the call chain, which may lead to churn in the instruction and data caches. We propose studying the benefits of such an approach, and for what types of application behavior it proves beneficial.

Evaluation with More Benchmarks

Currently, state transformation and migration are evaluated using the NPB benchmark suite which focuses on HPC applications. We propose studying other benchmarks which focus on other areas of computation:

- **SPEC CPU 2006** [45] – the SPEC benchmark suite is the standard benchmark set for computer architecture research. SPEC focuses on a variety of real-world computations, including scripting, compression, compilation, numerical modeling, AI, etc. SPEC CPU 2006 includes both integer and floating-point computations.
- **PARSEC** [16] – the PARSEC benchmark suite focuses on emerging workloads for chip multiprocessors. PARSEC includes multithreaded applications which exhibit a variety of runtime behaviors. It includes applications from computer vision, media processing, computational finance, animation physics, etc.

These benchmark suites are widely used in system software and computer architecture research. However, many of these applications are written in either C++ or Fortran, meaning the Popcorn compiler toolchain must be extended to support those languages. Because LLVM was designed with modularity in mind, supporting new languages ultimately requires having a front-end which generates LLVM bitcode from source code. Because the majority of our compiler modifications occur in the middle-end and back-end, supporting new languages should require minimal compiler effort. We only need to understand how to insert migration points using each of the new front-ends.

Supporting C++ applications should be straightforward, as clang is a production-quality C and C++ compiler. We will study the impact of C++ language features, such as exceptions, on the ability to transform state and migrate applications between architectures. Fortran is

a procedural language similar to C that is popular for high-performance computing. Currently, there is no LLVM-supported front-end for Fortran. Instead, Fortran developers use DragonEgg [76] to compile Fortran programs using LLVM. DragonEgg uses GCC's front-ends to generate LLVM bitcode, which can then be optimized and used to generate machine code using any of LLVM's back-ends. Thus we will use the DragonEgg project to generate LLVM bitcode from Fortran applications.

7.1.2 Scaling Applications Across Heterogeneous-ISA Processors

Currently, Popcorn Linux's scheduler migrates application threads en masse between architectures. All of an application's threads are migrated when the scheduler requests a migration. However, this approach limits the scalability of the application and the ability of the application to take advantage of architecture heterogeneity.

We propose implementing a new OpenMP runtime which can be used to distribute computation across multiple processors in a system to take advantage of unique architectural characteristics. The runtime would distribute threads across processors in the system based on system-tuned performance metrics. For example, in the ARM64/x86-64 setup described in Section 6.1 the runtime would tune the numbers of threads executing on each architecture to reach a certain power and performance ratio. If an application seeks higher performance, the runtime could migrate threads to the Xeon; alternatively, the runtime could migrate threads to the X-Gene to conserve energy. The runtime would automatically load balance threads to meet some metric, e.g., a service-level agreement or a power threshold. This is similar in nature to NRG-Loops [52], although it applies to splitting multithreaded work across heterogeneous-ISA processors rather than migrating a single thread between single-ISA heterogeneous cores. Work-splitting is a well-studied problem for CPU-GPU systems [62, 42, 88, 41], although we are aware of no works that split work based on power and performance.

The page coherency protocol provides a unique obstacle when distributing threads across heterogeneous-ISA processors. Unlike true shared memory systems, the page coherency protocol acts as a multiple-reader/single writer lock for pages (see Section 3.1.2). This poses a problem for threads of an application executing on different kernels but writing to the same page. For application code P_C, read-only parts of static global data P_D and thread-local state $T_i = R_i, S_i, L_i$, threads will never write to shared pages and thus there is no performance issue. However for writable static global state P_D and heap data P_H, threads executing on separate kernels experience false sharing of pages similarly to how threads executing on different cores of a CMP may experience false sharing of cache lines [51]. We therefore propose studying compiler and memory allocation techniques for laying out P_D and P_H to enable disjoint access parallelism on writable pages. There exists a large body of work describing compiler transformations to reduce false sharing of cache lines [75, 19, 22, 67, 51, 61], and we propose applying the lessons learned to optimize for Popcorn Linux's page coherency

protocol.

7.1.3 Migrating Between Highly Diverse Architectures

One of the current limitations for the Popcorn compiler toolchain and state transformation runtime is that it only supports architectures which have identical primitive data types. In other words, the Popcorn compiler toolchain and state transformation runtime currently only supports architectures whose primitive data types have the same sizes and alignments, whose pointers are of the same size, and whose data representation is of the same endianness. We propose relaxing several of these constraints in order to support state transformation and execution migration between a wider variety of architectures.

There is a significant body of work in dealing with state transformation between ISA-specific formats of primitive data types [6, 91, 98, 30, 90]. Several works describe mechanisms for transforming between representations [6, 91, 98], while others use padding to ensure compatibility across architectures without incurring state transformation costs [30, 90]. We propose using the latter approach in order to minimize transformation costs. Developers can specify primitive data type sizes and alignments using LLVM's target data layout [78]. This may be sub-optimal for architectures which do not natively support the specified data layouts, as the compiler must emulate operations on those data types in software. Instead, we propose modifying the back-ends to add padding to ensure a correct alignment across all ISAs. Note that it is the developer's responsibility to ensure that using primitives of different sizes does not interfere with correct functioning of the application.

The more important issue is in regards to transforming pointers between 32-bit and 64-bit architectures. Applications executing on a 32-bit architecture have a 4GB limit on their virtual address space. This limit is obviously much higher for 64-bit architectures. Thus, the problem becomes how to dynamically modify the layout of an application's address space when moving between architectures with different address space sizes. We propose studying techniques for implementing the following capabilities:

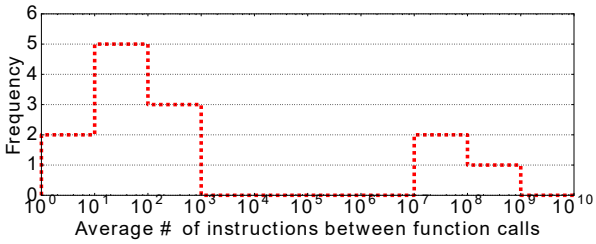
1. Detect when an application's total address space size is greater than 4GB. If so, disallow migration to a 32-bit architecture. This can be implemented in cooperation with the OS, which exposes the memory maps of processes using the `proc` filesystem.

2. If an application's total address space size is less than 4GB but the application has data laid out above the 4GB limit, the runtime must compact it to fit within 4GB, similarly to a compacting garbage collector [3]. For example, if an application migrates to a 64-bit architecture the memory allocator may allocate data above the 4GB threshold, or the application may map data into the address space above the 4GB limit. The runtime must be able to translate (i.e., relocate) all data in a process's virtual memory space. Additionally, the runtime must be able to transform all pointers in the application to point to the data's new location.

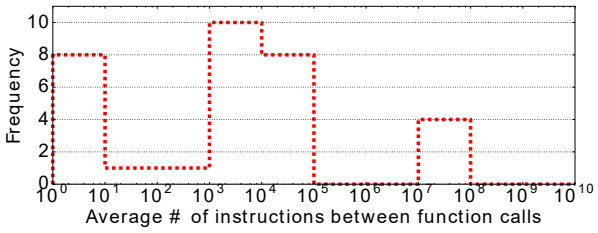
Similarly to other primitive data types, pointers must be padded to a common size across all architectures.

7.1.4 Techniques for Reducing Migration Response Time

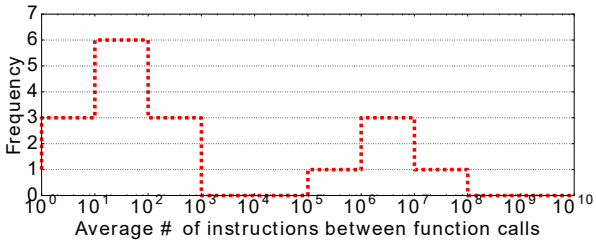
As described in Section 4.2, the Popcorn compiler inserts migration points at function call sites. However as shown in Figure 7.1, for several NPB applications this can lead to an extended **migration response time**, or the time between when the scheduler requests a migration and when a thread reaches a migration point. Figure 7.1 shows the instruction distance between function calls, i.e., migration points. There are multiple instances where there are hundreds of millions to billions of instructions between migration points. This behavior is often manifested in applications where nested loops perform a large amount of computation without calling a function. We propose several mechanisms to reduce the migration response time. Because there is a tradeoff in performance impact and the number of migration points inserted (each migration point incurs a function call and a memory read), these mechanisms must balance overhead and responsiveness.



(a) CG, class A



(b) FT, class A



(c) IS, class A

Figure 7.1: Distribution of number of instructions between migration points

The first step is to study the effect of inserting migration points at more equivalence points in the application. In particular, we propose modifying the front-end to insert migration points at the beginning of loops, another program location that satisfies the requirements of equivalence points. This includes studying the migration response time and performance impact of inserting migration points at varying levels in a loop-nest.

We also propose implementing immediate migration via checkpointing [83]. DeVuyst [29] and

Venkat [102] enable immediate migration using dynamic binary translation. When a thread migrates, a DBT framework emulates execution up until the next equivalence point, at which time state transformation is performed and the thread resumes native execution. However, they demonstrate that in some cases this leads to overheads of multiple milliseconds. We instead propose adding checkpointing mechanisms into the application to take snapshots of the program's execution (i.e., registers R_i and call frames in S_i) at migration points. When the scheduler triggers a migration, the thread rolls back to the checkpoint, transforms its state, and migrates to the new architecture. Thus, the thread loses some progress on its computation but can be immediately migrated to a new architecture. Checkpointing at migration points increases the per-migration point overhead, meaning further analysis is needed to understand where they should be inserted.

Thus, we propose implementing low-latency checkpointing at migration points and a new mechanism for attaching to a thread's execution state for asynchronous migration. We also propose a hybrid mechanism where the thread can choose to either roll back to a checkpoint and immediately migrate or delay migration and continue to the next equivalence point based on distance to the surrounding equivalence points.

Bibliography

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

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

[3] Andrew W Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[4] Applied Micro Circuits Corporation. X-gene product family. `https://www.apm.com/products/data-center/x-gene-family/x-gene/`.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

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

[19] 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.

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

[21] 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.

[22] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[23] 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.

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

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

[26] 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`

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

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

[35] 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.

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

[37] 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.

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

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

[40] 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.

[41] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 281–282, New York, NY, USA, 2013. ACM.

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

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

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

[45] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

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

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

[48] 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`.

[49] 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.

[50] 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.

[51] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 179–188, New York, NY, USA, 1995. ACM.

[52] Melanie Kambadur and Martha A Kim. Nrg-loops: adjusting power from within applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 206–215. ACM, 2016.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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`.

[59] 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`.

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

[61] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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`.

[67] 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.

[68] 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.

[69] James Niccolai. Qualcomm enters server CPU market with 24-core arm chip, October 2015.

[70] 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.

[71] NVIDIA. Compute unified device architecture programming guide, October 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[72] OpenACC-Standard.org. The OpenACC application programming interface, October 2015. `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf`.

[73] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[74] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[75] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 101–112, New York, NY, USA, 2002. ACM.

[76] LLVM Project. DragonEgg – using LLVM as a GCC backend, September 2016. `http://dragonegg.llvm.org`.

[77] LLVM Project. The LLVM compiler infrastructure, September 2016. `http://www.llvm.org`.

[78] LLVM Project. LLVM language reference manual, October 2016. `http://llvm.org/docs/LangRef.html`.

[79] LLVM Project. Stack maps and patch points in LLVM, September 2016. `http://llvm.org/docs/StackMaps.html`.

[80] Jelica Protic, Milo Tomasevic, and Veljko Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[81] RedisLabs. Redis, October 2016. `http://redis.io/`.

[82] 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.

[83] Gabriel Rodríguez, María J Martín, Patricia González, Juan Touriño, and Ramón Doallo. Compiler-assisted checkpointing of parallel codes: The cetus and llvm experience. *International Journal of Parallel Programming*, 41(6):782–805, 2013.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The tui system. Technical report, Software Practice and Experience, 1996.

[92] Dolphin Interconnect Solutions. PXH810 NTB host adapter, October 2016. `http://dolphinics.com/products/PXH810.html`.

[93] 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.

[94] 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.

[95] 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.

[96] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[97] Herb Sutter. Welcome to the jungle, August 2012. `https://herbsutter.com/welcome-to-the-jungle/`.

[98] 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.

[99] 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.

[100] Top500.org. June 2016 — TOP500 supercomputer sites, June 2016. `https://www.top500.org/lists/2016/06/`.

[101] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 727–741. ACM, 2016.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.