Breaking the Boundaries in Heterogeneous-ISA Datacenters

Antonio Barbalace Rob Lyerly Christopher Jelesnianski Anthony Carno
Ho-ren Chuang Binoy Ravindran
Bradley Department of Electrical and Computer Engineering, Virginia Tech
{antoniob,rlyerly,bielsk1,acarno,horenc,binoy}@vt.edu

Abstract

Energy efficiency is one of the most important design considerations in running modern datacenters. Datacenter operating systems rely on software techniques such as execution migration to achieve energy efficiency across pools of machines. Execution migration is possible in datacenters today because they consist mainly of homogeneous-ISA machines. However, recent market trends indicate that alternate ISAs such as ARM and PowerPC are pushing into the datacenter, meaning current execution migration techniques are no longer applicable. How can execution migration be applied in future heterogeneous-ISA datacenters?

In this work we present a compiler, runtime, and an operating system extension for enabling execution migration between heterogeneous-ISA servers. We present a new multi-ISA binary architecture and heterogeneous-OS containers for facilitating efficient migration of natively-compiled applications. We build and evaluate a prototype of our design and demonstrate energy savings of up to 66% for a workload running on an ARM and an x86 server interconnected by a high-speed network.

1. Introduction

The x86 instruction set architecture is the de-facto ISA of the datacenter today [48, 57, 61, 37]. However, a new generation of servers built with different ISAs are becoming increasingly common. Multiple chip vendors, including AMD, Qualcomm, APM, and Cavium, are already producing ARM processors for the datacenter [4, 56, 6, 21, 33]. The PowerPC ISA is also gaining traction, with IBM forming the OpenPower foundation by partnering with companies such as Google, NVIDIA, Mellanox and others [47]. These new servers promise to have higher energy proportionality [13], reduce costs, boost performance per dollar, and increase density per rack [64, 65]. Increasing interest in alternative server architectures is shown by a number of works that analyze the advantages of these new servers compared to x86 [8, 38, 3, 62, 47]. Interest is also driven by the increasing availability of ARM and PowerPC cloud offerings [49, 43, 58, 45] in addition to traditional x86 servers. It is therefore clear that the datacenter, now mostly built with single-ISA heterogeneous [48, 67] machines, will be increasingly populated by heterogeneous-ISA machines.

Cutting electricity costs has become one of the most important concerns for datacenter operators today [74]. Energy proportionality [13] has become an important design criterion, leading hardware and software architects to design more efficient solutions [67, 74, 71, 69, 70]. There are several software-based approaches that are effective for conserving energy, including load balancing and consolidation. Load balancing spreads the current workload evenly across nodes, while consolidation groups tasks on a minimal number of nodes and puts the rest in a low-power state. Both solutions migrate tasks between machines using techniques such as virtual machine migration [69, 46, 51], or more recently container migration [5]. Using these techniques allows datacenter operators to conserve energy and adjust the datacenter's computational capacity in response to changing workloads.

Increasing instruction set architecture diversity in the datacenter raises questions about the continued use of execution migration to achieve energy efficiency. Can applications be migrated across machines of different ISAs, and is there any energy advantage for migration?

In this work we introduce system software that prepares native applications (i.e., applications written in non-managed languages), to be deployable on multiple ISAs and to be migratable during execution. Execution migration is supported by an operating system extension, called heterogeneous OS-containers, that allows for a Linux container to migrate among Linux instances seamlessly, despite differences in ISA. We approach the problem as an application state transformation problem [7] in user-space, and present techniques to minimize the amount of state to be transformed to enable fast migration. Additionally, we leverage a replicated-kernel OS [12] in which OS services are distributed, and thus their state can be migrated between servers. We evaluate a prototype on two heterogeneous-ISA servers, an ARM and an x86 server, showing that there is up to a 30% energy savings on some workload mixes, with different projected energy costs for several scheduling policies. Due to these advantages, we predict greater benefits can be obtained at the rack or datacenter scale. Thus, in this work we present the following contributions:

- A formalization of software state for multi-threaded applications running on a process-model monolithic operating system and an analysis of its dependence on the ISA.
- A new software architecture which stretches applications and operating system sub-environments (containers) across heterogeneous-ISA servers, allowing applications to run natively and migrate between servers dynamically.
- A set of techniques and mechanisms at various levels of the system software stack that implement the proposed architecture, i.e., *multi-ISA binaries* and *heterogeneous OS-containers*.
- A prototype built around the Linux ecosystem using Popcorn Linux [12], LLVM, and musl, and evaluated on a dual-server setup equipped with ARM and x86 processors.

Section 2 discusses the background and motivation for redesigned system software, Section 3 introduces a formal model

of software for multi-threaded applications running on SMP OS, and Section 4 uses the model to describe the proposed software architecture. Section 5 describes our prototype's implementation details for both the OS and compiler/runtime. In Section 6 and Section 7, we describe the experimental setup and evaluate our implementation. Section 8 discusses related work and Section 9 concludes.

2. Background and Motivation

Datacenter operators, including cloud providers, manage their fleet of machines as pools of resources. Modern cluster management software, i.e., datacenter operating systems [73, 59], extend the concept of single machine operating systems to a pool of machines. This software abstracts away management of individual machines and allows developers to manage resource pools as a single entity, similarly to an operating system managing processing, memory, and storage resources in a single computer. Example datacenter OSs include OpenStack [20], Mesosphere/Mesos [50, 34], and Kubernetes [18].

One of the key characteristics of datacenter OSs is that multiple applications can be run on the same cluster. Concurrently executing applications share resources, maximizing cluster utilization and increasing energy efficiency. To achieve economic utilization of cluster resources, datacenter OSs both load balance across machines and consolidate jobs to fewer nodes. Load balancing [55, 39] spreads the current workload evenly across nodes, using equal resources on each machine for reduced power consumption. Although this solution may not maximize energy efficiency, it allows datacenter operators to react quickly to load spikes. Alternatively, consolidating workload onto fewer servers at runtime is one of the most effective approaches for conserving energy. The machines executing the workload are run at high capacity (expending significant amounts of power), while the remaining machines are either placed in a low-power state or are completely shut down. This has been shown to increase energy proportionality at the group-of-machines "ensemble" level [67], but reduces the ability of the datacenter to react quickly to workload spikes. Both techniques statically assign jobs to nodes. However, advanced versions of these techniques may also dynamically migrate jobs between nodes, which are today assumed to be homogeneous (or at least single-ISA heterogeneous [48]).

Heterogeneous-ISA Datacenters. As heterogeneous-ISA servers are introduced into the datacenter, resource managers are constrained to either splitting the datacenter into multiple per-ISA partitions or statically allocating jobs to machines. Splitting the datacenter into per-ISA partitions allows resource managers to load balance and consolidate tasks across a subset of servers. This is the current model, as ARM and x86 cloud providers [43, 58] offer separate ARM and x86 partitions (e.g., OpenStack zones) to customers. Partitioning resources has many disadvantages [34] – for example, one partition could be idle while another is overloaded, leading to wasted processing power and service disruption. The capability to move jobs

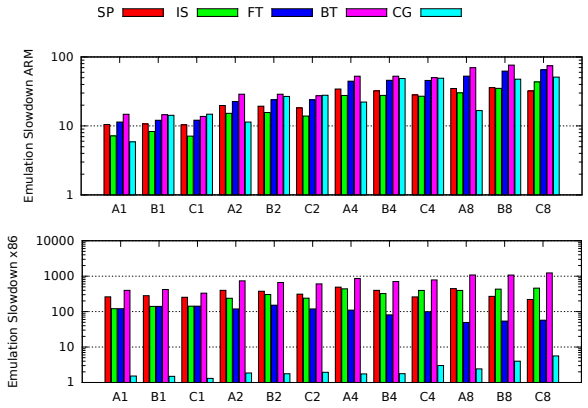

Figure 1: Slowdown when emulating ARM applications on x86 versus running natively on ARM (top graph) and the reverse for native x86 applications in the bottom graph.

across partitions is needed to cope with varying workloads.

Native applications can be compiled for heterogeneous-ISA servers, but cannot migrate between them at runtime. Applications written using retargetable or intermediate languages (e.g., Java, python, etc.) can run on heterogeneous-ISA servers, but are usually statically assigned to servers. Although there are tools that implement execution migration for these languages [29, 30], migrating stateful applications is costly due to the serialization/de-serialization process between ISA-specific formats. Additionally, many applications are written in lower-level languages like C for efficiency reasons (e.g., Redis). Moving jobs between machines increases energy proportionality [71], meaning inter-ISA migration is key.

Execution Migration. Execution migration at the hypervisor and application level is implemented by various open-source and commercial products (e.g., VMware, Xen, KVM/QEMU, Docker). Although it is not officially supported, it is possible to migrate an application between ARM and x86 machines with KVM and QEMU. In order to understand the costs of using KVM/QEMU to abstract the ISA, we measured the slowdown when migrating an application (including the operating system) between KVM on x86 and QEMU on ARM. Figure 1 shows the slowdowns experienced when running applications from the NPB benchmark suite [9] in emulation versus native execution. The top graph shows the slowdown experienced by applications (compiled for ARM) when emulated on x86 versus running natively on ARM. The bottom graph shows the slowdown experienced by applications (compiled for x86) when emulated on ARM versus running natively on x86. Additionally, the same experiment for Redis [2], a typical datacenter application, incurs 2.6x slowdown for ARM and a 34x for x86. Clearly, using emulation is not a suitable technique for hiding heterogeneity, as several applications experience slowdowns of several orders of magnitude. The cost of emulation, even when using Dynamic Binary Translation (DBT), is unacceptably high.

Software State and Code Mobility. Execution migration

in the traditional SMP programming model relies on the fact that both applications and the OS share data in a common format, as all processors are of the same ISA. Similarly, VM and container migration exploits the fact that the same software state can be migrated unmodified between homogeneous-ISA machines. In the latter case, the hypervisor (for VMs) or the operating system (for containers) provides a layer of abstraction to mimic the same hardware and software resources on different machines.

Today, when processors of different ISAs must communicate or transfer application execution, mechanisms that make the application distributed have been used to circumvent ISA differences. However, these same mechanisms prevent execution migration. The Internet provides a common format that stretches applications across multiple heterogeneous-ISA nodes – messages are serialized from an ISA-specific format into a pre-agreed format for all communication. Similarly, code offloading and message passing require the developer to manually partition and map the application to each processor in the system, with explicit communication between the different parts. Application state must be manually split, copied, and kept consistent amongst all pieces, and the boundaries between application components are fixed. Additionally, serialization and de-serialization is necessary to convert each piece of data between formats for each ISA.

We propose minimizing runtime conversion of state by transforming binaries compiled for different ISAs to use a common state format – i.e., memory can be migrated without any transformation. For state that must be transformed, the operating system and the runtime work together to transform state and to enable execution migration with minimal performance impact.

3. A Model of Software

We propose a formal model of software to describe execution migration. Software is composed of executable code and data (e.g., constants, variables). We consider a model in which executable code is compiled to native machine code (i.e., no intermediate representation) and does not mutate during program execution (i.e., no self modifying code). During execution the state of the software includes the state of the hardware – CPU registers, configuration and peripherals registers, etc.

We define a model of the state of the software for multi-threaded applications running on a multi-tasking process-model monolithic operating system. We consider operating system services to be atomic [27]. For application software running on such an operating system, the hardware-related state is minimal (essentially, CPU registers) due to the OS's role in managing and abstracting access to hardware resources. Hence, the hardware-related state is attributed to the OS state. In our model the OS completely mediates IO, such that an application's address space exclusively maps memory – this model does not support mapping devices into virtual memory, but can be easily extended to support it.

Application. The state of an application is a collection of variables (data) and executable code. Each multithreaded application includes a per-thread state for each thread i, T_i, and a per-process state, P. If the application is multiprocess, the model extends to sharing between multiple processes[1]. The per-thread state for thread i contains thread local storage data (L_i), user-space stack (S_i), and the user-space visible state of the CPU (R_i). L_i includes program- and library-declared per-thread variables (e.g., variables declared with `__thread` in GCC). Hence, $T_i =< L_i, S_i, R_i >$. The per-process state includes all other user-visible state that makes up the application's address space, such as global data structures allocated in the heap or in the program's data sections. P also includes the application's executable code (i.e., the `.text` section).

Operating System. The operating system state can be also defined in terms of thread-related data, however a formalization centered around the application is required to migrate an application container. From the point of view of an application thread executing in kernel-space, T_i^K includes the kernel stack (S_i^K), the kernel CPU registers (R_i^K), and the kernel per-thread local data (L_i^K, e.g., the thread control block). For a thread executing in user-space, T_i^K only includes the per-thread local data. Note that in message-passing kernels, the thread's receiver buffer state belongs to either T_i^K or T_i if the thread is executing in kernel- or user-space, respectively. Thus, $T_i^K =< L_i^K, S_i^K, R_i^K >$. P^K is composed of all T^K, interrupt state, and kernel thread state for the kernel services used by a process. It also includes hardware-related state, e.g., the CPU's page table. Kernel state can be divided by operating system service O_x, where x is a specific service. Because kernel services are atomic from an application point of view, each kernel service can be split into a per process state $P_{j,x}^K$ (for each user-process j using that service), a kernel wide state K_x and a hardware-related state W_x, if there is an hardware device or peripheral associated with that operating system service. Thus, each operating system service's state can be defined as $O_x =< K_x, W_x, P_{0,x}^K, .., P_{k,x}^K >$, where there are k processes using O (the model can be extended to support per-task state).

4. Architecture

We propose a redesign of system software in order to create native applications that can be deployed on and seamlessly migrated between heterogeneous-ISA machines. The datacenter OS already extends horizontally across multiple machines, independently of the ISA. Currently, however, native applications can only be deployed on the ISA for which they were compiled and cannot migrate among ISAs without paying a huge emulation overhead.

We introduce *multi-ISA binaries* and a runtime that enables an application, compiled with a new toolchain, to have a common address space layout on all ISAs for most application state. State that is not laid out in a common format is converted between per-ISA formats dynamically during migration,

[1] We do not consider this case in our formalization, although extending the model to support multiprocess applications is trivial.

with minimal overhead. We present a series of distributed services at the kernel level to enable the seamless migration of applications in an OS container between heterogeneous-ISAs machines. Both the user-space and kernel-space state of applications is automatically transferred between machines. Thus, *heterogeneous OS-containers* elastically span across ISAs during execution migration.

Application. Seamlessly migrating a multithreaded application between ISAs requires each application thread be able to access its code and data on all ISAs. Rather than attempting to dynamically transform and keep application state consistent in a per-ISA format, we propose to have multi-ISA binaries in which each ISA's machine code conforms to a single address space layout. The application's data and text, P, is kept in a common format across all ISAs. Additionally, per-thread state T_i is kept in a common format except where the layout is dictated by the underlying ISA (per-thread register state R_i) or where changing the layout has significant performance cost (a thread's stack, S_i). We advocate for a common format in order to avoid transformation costs.

To enforce a common state for an application P that will run on ISA A (IA) and ISA B (IB), all symbols in the application must have the same virtual address. This allows the identity function to be used to map all state between ISA-specific versions of the process, $P^{IA} = P^{IB}$ (note that the application binary will contain a `.text` section compiled for IA and for IB, but function symbols will be mapped to the same virtual addresses). For each application thread, the thread local data has the same format on each ISA, $L_i^{IA} = L_i^{IB}$. However, to allow the compiler to optimize stack frame layout for each ISA, the stack is not kept in a common format and a separate mapping function is used to convert each stack frame from one ISA to the other, $f^{AB}() : S_i^{IA} \rightarrow S_i^{IB}$ and $f^{BA}() : S_i^{IB} \rightarrow S_i^{IA}$. Moreover, we define a state transformation function $r^{AB}() : R_i^{IA} \rightarrow R_i^{IB}$ and $r^{BA}() : R_i^{IB} \rightarrow R_i^{IA}$ that maps the program counter, the stack pointer and the frame pointer between ISA-specific versions of the program. However, $f^{AB}()$, $f^{BA}()$, $r^{AB}()$, and $r^{BA}()$ are only valid at certain points in the application's execution, known as *equivalence points*. Equivalence points exist at function boundaries, among other locations in the program.

Operating System. In the datacenter, each server runs a natively compiled operating system kernel. The datacenter operating system manages all servers somewhat similarly to a multiple kernel OS [14, 11] but at a different scale. Our architecture merges these two designs – our design introduces distributed operating system services (similarly to a replicated-kernel OS) that presents a (containerized) single working environment to the application when migrating between servers.

The operating system is able to provide a single execution environment due to the fact that applications interact with the operating system via a narrow interface: the syscall, and in *NIX operating systems, the filesystem. Because OS services are distributed, kernels can reproduce the same OS interface and resource availability regardless of the architecture on which the application is executing, providing a single elastic operating environment. This single operating environment is maintained among kernels for the duration of the application. Moreover, it supports applications running among servers. After migration, the process's data is kept on the source kernel until there are residual dependencies, i.e., it has all been migrated.

For each operating system service O_x, the service on ISA A (IA) and on ISA B (IB), O_x^{IA} and O_x^{IB}, keeps the per-process state consistent among kernels. Thus, an identity mapping applies to $p^{AB}() : P_{x,j}^{K,IA} \rightarrow P_{x,j}^{K,IB}$ or $p^{BA}() : P_{x,j}^{K,IB} \rightarrow P_{x,j}^{K,IA}$. Every time the state of a service is updated on one kernel, it must be updated on all other kernels (different services require different consistency levels). This per-process state is the only part of the state that must be kept consistent for kernel services running among kernels.

4.1. System Software Redesign

In addition to a redesigned operating system and compiler toolchain, a runtime must provide state transformation where necessary. Thus, we advocate for a compiler toolchain that produces multi-ISA binaries, a heterogeneous OS-container that allows execution migration between heterogeneous-ISA machines, and a runtime that provides state transformation for application state not laid out in a common format.

Multi-ISA binaries and runtime. We propose a compiler toolchain that creates a binary per ISA. In addition to creating a common virtual address space, the compiler inserts call-outs at equivalence points, called migration points, that allow the application to migrate between architectures. The compiler also generates metadata that describes the functions to transform stack frames ($f^{AB}()$ and $f^{BA}()$) and register state ($r^{AB}()$ and $r^{BA}()$) between ABIs at the inserted call-outs.

Heterogeneous Containers. The proposed software infrastructure allows the developer to write an application targeting an SMP machine, and migrate it amongst multiple diverse-ISA machines at runtime. The proposed software architecture provides a single operating system sub-environment across multiple kernels – i.e., operating-system based virtual machines (containers or namespaces [17]) on different ISA machines, and migration amongst them. We call these virtual machines *heterogeneous OS-containers*.

5. Implementation

We implemented a prototype of the proposed architecture on two heterogeneous-ISA servers, with ARM and x86 processors (both 64-bit), interconnected through a low-latency network via the PCIe bus. This is representative of future datacenters due to the current dominance of x86 and the push for ARM in the cloud. The prototype is based on the Linux system software ecosystem to take advantage of its support for many hardware architectures and the vast availability of applications. However, we believe that the proposed architecture applies to other software ecosystems, including any multiple-kernel operating

system design (e.g., Barrelfish). The multiple-kernel operating system which provides the heterogeneous-OS container functionality is based on the Linux kernel. The heterogeneous compiler toolchain is built using clang/LLVM and GNU binutils. The runtime library uses compiler-generated metadata and DWARF debugging information for state transformation. The prototype currently only targets applications written in C.

5.1. The Operating System

We extended the Popcorn Linux replicated-kernel OS [10, 12] to support heterogeneous-ISA machines. Popcorn is based on the Linux kernel and re-implements several of Linux's operating system services in a distributed fashion. We ported the original code to support ARMv8 (APM X-Gene 1 platform [6]) as well as x86, 64-bit. Moreover, we implemented a new messaging layer to support communication between the two servers. We both introduced new operating system services and redesigned previous ones to support migratable heterogeneous containers, including a heterogeneous-binary loader, heterogeneous distributed shared memory (hDSM), and heterogeneous continuations.

The replicated-kernel OS consists of different kernels, each compiled for and running on a different-ISA processor. Kernels do not share any data structures, but interact via messages to provide applications with the illusion of a single operating environment amongst different processors. The OS state is broken down into OS services, whose state is replicated amongst kernels. The replicated state provides the illusion of a single operating environment, thus enabling thread and process migration and resource sharing among kernels. Popcorn Linux introduces a thread migration operating system service that provides the foundation for migrating a program between kernels during execution. Heterogeneous-OS containers are resource-constrained operating system environments that migrate among kernels. Thus even if the kernel is running on another ISA, the application accesses the same file system, the same abstract hardware resources, the same syscalls, etc. This is built using Linux's namespaces and Popcorn Linux's distributed services.

Heterogeneous distributed shared memory (hDSM). The memory state of each migrating application is replicated and is kept consistent amongst kernels until all threads of the same application migrate to the same kernel. DSM enables on-demand migration of memory pages without forcing all threads to migrate at once (i.e., no "stop-the-world").We extended the software DSM implemented in Popcorn Linux [12] to support heterogeneous platforms (hDSM). We added memory region aliasing, specifically for `.text` sections and vDSO sections. Moreover, we disabled `vsyscalls` in order to force all syscalls to enter the OS. Even if the specific interconnect we used between servers as well as recent network technologies (e.g., RDMA) offer a form of shared memory through PCIe, due to the higher latencies for each single operation, we opted for a full DSM protocol between ARM and x86 servers. In other words, the hDSM service migrates pages in order to make subsequent memory accesses local rather than repeatedly accessing remote memory.

Heterogeneous binary loader. We implemented heterogeneous binaries as one executable file per ISA (see Section 5.2). Binaries contain an identical address space layout but each has its own `.text` section natively compiled for that ISA. Thus, the compiler provides a per-ISA version of an application's machine code. Each kernel loads the address space of the application and executes that ISA's native code. When execution migrates between kernels, the machine code mappings are switched to those of the destination ISA. This is implemented in Linux's ELF binary loader and integrated within the hDSM kernel service, which aliases the `.text` section of each ISA within the same virtual address range.

Thread migration and heterogeneous continuations. This work extends a process model OS. Each application thread has a user-space stack as well as a kernel-space stack. The proposed software architecture manages each stack differently. To facilitate user-level process and thread migration, threads use the same user-space stack regardless of the ISA on which they are running. This design requires transforming the user-space stack during migration (see Section 5.2). Conversely, each thread has a per-ISA kernel-space stack. This is handled similarly to a continuation [26]. An application thread that is executing code in kernel space cannot migrate during execution of a kernel service; otherwise, service atomicity is lost. Moreover, kernel threads do not migrate. When a user thread migrates amongst different-ISA processors, the kernel provides a service that maps the program counter, frame pointer, and stack pointer registers from one ISA to the other.

5.2. The Compiler

The compiler is based on the LLVM framework and ensures that data and executable code are placed in the appropriate locations in virtual memory so that the OS's services just presented can transparently migrate applications between ISAs. The toolchain must also provide these guarantees with minimal impact on performance. Hence, for application state which cannot have common layout without a large impact on performance (e.g., a thread's runtime stack S_i), state transformation is provided by the runtime (Section 5.3).

There were two design goals for the compiler toolchain. The first was to prepare applications to be migratable among architectures without developer intervention. Hence, the compiler needed to support traditional SMP semantics and application interfaces, such as the standard C library, POSIX threads library, etc. The second was to limit changes to the core compiler itself. This allowed compiled applications to benefit from existing compiler analyzes and optimizations to generate highly tuned machine code. Additionally, by limiting changes to the generated code (e.g., no changes to stack frame layout as required in [24, 68]), it makes the job of porting the toolchain to new architectures simpler.

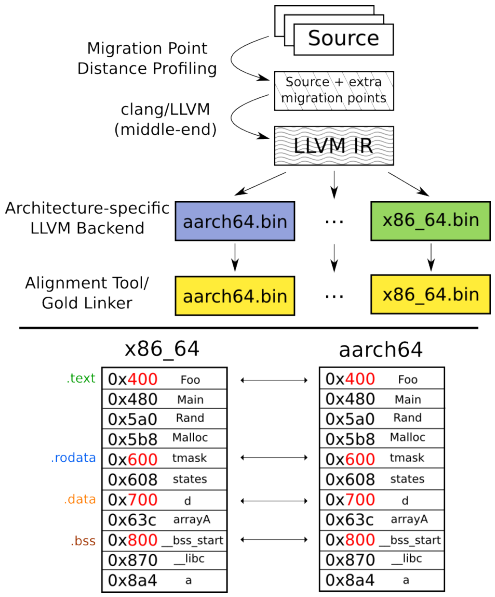

Figure 2: The compilation process and resulting cross-binary virtual memory layout.

Compiler Architecture. We modified clang/LLVM [40] as the compiler for ARM64 and x86-64. We also modified the GNU gold [66] linker to change the application layout to enforce a common address space among ISAs. The compilation process is shown in Figure 2. After an initial profiling phase, the compiler inserts migration points into the application source so that the application has the chance to migrate between architectures more frequently. Next, the toolchain runs standard compiler optimizations and several custom passes over LLVM's intermediate representation (LLVM bitcode) to enable symbol alignment. Then, the architecture-specific backends generate binaries for each available architecture in the system. Finally, all application symbols are aligned so that global data are laid out in a common format and code memory pages can be aliased by the OS heterogeneous binary loader. We describe each component in the following sections.

5.2.1. Migration Points Because the kernel cannot interrupt and migrate threads between architectures at arbitrary locations, application threads check if the scheduler has requested a migration at known-good locations. These *migration points* are implemented entirely in user-space. The kernel scheduler interacts with the application through a shared memory page between user- and kernel-space (vDSO). When the scheduler wants threads to migrate, it sets a flag on the page requesting the migration. At migration points threads check if the flag has been set, and if so, they initiate the state transformation and migration mechanisms detailed below.

Inserting Migration Points. Migration points can only be inserted at equivalence points in the application source. Function boundaries are naturally occurring equivalence points, so the compiler automatically inserts migration points at function entry and exit. Additionally, the compiler can insert migration points into other locations in the source in order to adjust the *migration response time*, i.e., the time between when the scheduler requests a migration and when the thread reaches a migration point. More migration points means a lower migration response time, but higher overhead due to more frequent migration request checks.

Optimizing Migration Point Frequency. The number of migration points inserted into the code dictates the frequency at which an application can be migrated between different architectures. We developed a tool based on Valgrind [52] to analyze the number of instructions between migration points during an application's lifetime. This analysis gives insight into where additional migration points should be inserted to minimize overhead from checking for migration requests while maximizing migration flexibility. We used this analysis to place additional migration points to enable the application to migrate approximately once per scheduling quantum (roughly 50 million instructions).

5.2.2. Symbol Alignment After migration points have been inserted, the toolchain generates optimized LLVM bitcode and compiles a binary for each target ISA. With the traditional compilation process each binary has a different virtual memory layout due to differences in symbol size, symbol padding, etc. The binaries for each architecture must have aligned symbols so that accesses to global data can be kept consistent by the hDSM service, and calls to functions can be aliased to the correct per-architecture version of the function by the heterogeneous binary loader. A per-architecture linker script places data and function symbols at the same virtual addresses for each binaries.

Alignment Tool. We developed a Java tool that reads symbol size and alignment information generated by the linker, and generates a per-ISA linker script that aligns symbols at identical virtual memory addresses. The tool aligns symbols in loadable ELF sections (e.g., `.text`, `.data`, `.rodata`, `.bss`, etc.) by progressively calculating their addresses in virtual memory. Aligning data symbols is simple, as the primitive data types have the same sizes and alignments for ARM64 and x86-64[2]. However, aligning function symbols requires adding padding so that function sizes are equivalent across binaries for all target architectures.

Thread-Local Storage (TLS). We also modified the gold linker in order to ensure that TLS (and its associated relocations) was laid out according to a common format across all binaries. Thus, the TLS layout for all binaries was changed to map symbols identically to the x86-64 TLS symbol mapping.

5.3. The Runtime

Enabling migration between architectures requires additional runtime support to transform per-thread state so that a migrating thread can resume execution on the destination architecture. The runtime must transform all state that is not laid out in a common format – in particular, the stack (S_i) must be rewritten

[2] Architectures that have different primitive data sizes or alignments would require more careful handling

to conform to the destination architecture's ABI, and the destination architecture register state (R_i) must be initialized to a known-good state. The runtime state transformation mechanisms are activated at migration points, before migration occurs. Once the scheduler has requested a thread migration, the runtime re-writes the stack and patches up architecture-specific register state (e.g., the stack pointer, link register, etc.). After state transformation is completed, the thread makes a system call to the thread migration service to migrate execution to the destination processor.

Stack Transformation. The stack transformation runtime is responsible for converting each thread's stack from the current ABI to the destination ISA's ABI. It does this without restrictions on stack frame layout, meaning there are no limitations preventing the compiler from doing aggressive register allocation and optimizing the stack frame layout for each architecture. The runtime attaches to a thread's stack at migration points and rewrites the stack frame-by-frame in a single pass.

The runtime utilizes metadata generated by the compiler for transformation. The compiler records the locations of live variables at function call sites and generates DWARF frame unwinding information so the runtime is able to traverse the stack. Note that the runtime only needs live value information at function call sites, as they are the only points at which transformation can occur – the stack is by definition a series of frames corresponding to live function invocations (a chain of function calls), and the most recent function invocation makes a call-out to a migration library, where special handling begins the transformation process.

Stack transformation is performed in user-space, but is hidden inside of the migration runtime. The runtime divides a thread's stack into two halves. When preparing for migration, the runtime rewrites from one half of the stack to the other, and switches stacks right before invoking the thread migration service. The stack transformation library begins by analyzing the thread's current stack to find live stack frames and to calculate the size of the transformed stack. It then transforms a frame at a time starting at the outer-most frame (i.e., the frame of the most recently called function), from the source to the destination stack until all frames have been re-written.

During compilation, an analysis pass is run over the LLVM bitcode to collect live values at function call sites. Another pass inserts an intrinsic into the IR, which informs the various LLVM backends to generate variable location information after register allocation. This metadata serves two purposes – it maps function call return addresses across architectures (allowing the runtime to populate return addresses up the call chain) and it tells the runtime how to locate all the live values needed to resume the function invocation as the thread unwinds back through the call chain on the destination architecture. The compiler also generates DWARF frame unwinding metadata, detailing the per-architecture, per-function register save procedure for the runtime.

To transform an individual frame, the runtime reads the live value location metadata and copies live values between stack frames. Additionally, the runtime saves a return address and previous frame pointer, i.e., the saved frame pointer from the caller's frame. The runtime must ensure the stack adheres to the destination architecture's ABI, meaning that it must follow the register-save procedure for callee-saved registers on the destination ISA. If the runtime finds a live value in a callee-saved register, it walks down the function call chain until it finds the frame where the register has been saved, and places the value in the correct stack slot (some registers may still be live in the outermost function invocation, however).

The runtime must also fix up pointers to data on the source stack to point to the appropriate location on the destination stack (pointers to global data and the heap are already valid due to symbol alignment and the hDSM service). When the stack transformation runtime finds a pointer in a frame that points to an address within the source stack's bounds, it makes a note that a pointer on the destination stack needs to be resolved. When the runtime finds the pointed-to data on the source stack during transformation, it first copies the pointed-to data to the destination stack (as part of normal frame re-writing) and fixes up the pointer with the address of the newly copied to data on the destination stack.

5.4. Limitations

The prototype is limited to 64-bit architectures, as migrating applications between 32-bit and 64-bit address spaces would require dynamically changing the address space layout and may be impossible in the general case. Currently, the toolchain does not support applications that use inline assembly, as live variable analysis in the middle-end is not compatible with assembly. Additionally, architecture-specific features such as SIMD extensions and setjmp/longjmp are not supported, although we plan to study these in future work. Finally, applications cannot migrate during library code execution (e.g., during calls to the standard C library).

6. Evaluation

We evaluated the mechanisms described for container migration among heterogeneous-ISA servers on our prototype. We wanted to answer the following questions:

- Is it possible to migrate an application container between server machines with different ISAs at runtime?
- What are the costs for this migration?
- Does migration enable effective load balancing and consolidation for obtaining energy proportionality among heterogeneous-ISA servers in the datacenter?
- What types of scheduling policies can better exploit the heterogeneity among machines in the datacenter?

Hardware. We built our prototype with an x86 machine and an ARM development board. The x86 is a server-class Intel Xeon E5-1650 v2 (6 cores, 2-way hyper-threaded at 3.5GHz, 12MB of cache), with 16GB of RAM. We disabled hyperthreading in the experiments. The ARM development

board is an Applied Micro (APM) X-Gene 1 Pro based on the ARMv8 APM883208 processor (8 cores at 2.4GHz, 8MB of cache), with 32GB of RAM. The two motherboards were connected via a Dolphin ICS PXH810 [25], which was the fastest interconnect on the market at the time we designed the experiment (up to 64Gb/s). However, our prototype supports any other network interface card.

Power measurements. We recorded power consumption via both on-board sensors and external power measurement equipment. On the x86 processor, we used Intel's RAPL [32] to measure power for the core and uncore, while on the ARM board we queried the off-socket power-regulator chips via I2C. Power was measured externally by inserting .1Ω shunt resistors on each ATX power supply line. A data acquisition system was built using a National Instruments 6251 PCIe DAQ in a separate system. We acquired readings at 100Hz on both systems in order to have readings at high resolution (which would be low-pass filtered if recorded at the wall).

Software. Our prototype extends Popcorn Linux (based on Linux version 3.2.14) to Linux version 3.12 (APM X-Gene 1 Pro baseline). Where not indicated, vanilla Linux version 3.12 was used in the evaluations. We leveraged LLVM 3.7.1 [40] along with the clang front-end, Java 1.8.0-25, GNU Binutils 2.27, and gold 1.11 [66] to create multi-ISA binaries. We modified musl-libc, version 1.1.10, to create a common TLS layout and to provide additional support needed for pthreads across different ISAs. To run OpenMP applications, we exploited the POMP library provided with Popcorn [12].

Benchmarks. We selected multiple applications in order to create a mix of short- and long-running workloads as well as memory-, compute-, and branch-intensive workloads, similarly to the analysis in [8, 38]. We used applications from the NAS Parallel Benchmarks (NPB) [9] because they can be both short and long running by varying the problem size (classes A, B, and C). In addition, we ran the Verus Model Checking tool version 0.9 [19] (which we ported to 64-bit) and bzip2smp version 1.0 [1] (extended to support the same command line arguments as GNU bzip). These add branch-intensive workloads with variable input sizes to the benchmarks. This mix of benchmarks covers execution times ranging from milliseconds to hundreds of seconds, which is what it is usually expected in datacenters [57] (including low-latency jobs). We focus on a worst case utilization scenario for the ARM machine which is currently not as powerful as the x86 server [8, 38]. We scope out network applications in order to highlight the network costs of our architecture, including performance and power, due to execution migration.

Job Scheduling. We evaluated how to take advantage of heterogeneous migration via scheduling. Without heterogeneous migration, the scheduler must partition jobs between different architectures and jobs cannot move between machines. There exists a large body of work on scheduling for heterogeneous processors and servers; most of this work focuses on single-ISA heterogeneity (e.g., [48]). We developed scheduling heuristics that assign and migrate jobs while using a minimal amount of information from each machine (CPU load), leaving the exploration of further policies on a larger scale (heterogeneous-ISA clusters) as future work. One important observation is that in heterogeneous multi-core processors, unbalanced thread scheduling can provide significant energy savings [23]. With that in mind, we designed two dynamic policies which assign and dynamically migrate applications between servers. The first policy balances the number of threads on the x86 and on the ARM machine; the second keeps the number of threads unbalanced on the x86 and on the ARM machines, such that the x86 machine runs more threads than the ARM machine. We compare these two dynamic policies to the following static policies which cannot migrate applications, and thus cannot change scheduling decisions after assigning threads to servers: balancing the number of threads on two identical x86 processors; balancing the number of threads on an x86 and an ARM processor; and unbalancing the number of threads on an x86 and an ARM processor, such that the x86 processor runs more threads than the ARM processor.

Migrating Competitors. There are few projects that support heterogeneous-ISA migration and that have source code available [31, 30, 29]. At the time of writing we were able to use PadMig [29] on our architecture. PadMig is based on Java and is written in Java. It exploits Java reflection to serialize and de-serialize an application's objects during migration. Thus, we compared migration using a managed language versus our prototype which migrates at the native code level.

7. Results

We evaluated the individual migration mechanisms and the energy advantages achieved using migration in our heterogeneous-ISA prototype. Due to space limitations, only a subset of results are presented.

Inserting Migration Points. We wanted to understand whether we could insert enough migration points into applications to reach the granularity of a migration point every 50 million instructions. Figures 3, 4 and 5 show a distribution of the number of instructions between migration points for CG, IS, and FT (class A). We ran each benchmark using the Valgrind tool (described in Section 5.2.1) to count the number of instructions between function calls ("Pre"). Using this information, we then inserted migration points to break up regions containing larger numbers of instructions between migration points ("Post"). As the graphs show, using the analysis we were able to insert enough migration points to reach our goal.

Migration Point Overhead. Next, we wanted to evaluate the cost of inserting migration points into the code. Figures 6, 7, 8 and 9 show the overhead for inserting migration points for CG and IS versus uninstrumented versions of the application with various class sizes and numbers of threads. As shown in these graphs, the overheads for instrumentation are small compared to total application execution time. Most overheads are less than 5%, and in general decrease as class

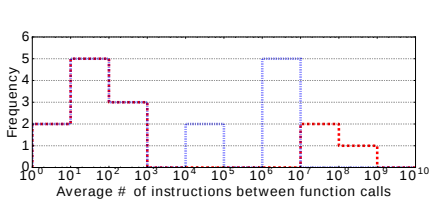

Figure 3: NPB CG number of instructions between migration points.

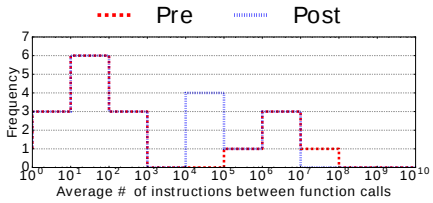

Figure 4: NPB IS number of instructions between migration points.

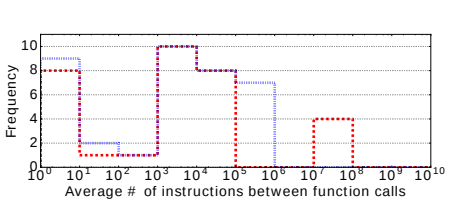

Figure 5: NPB FT number of instructions between migration points.

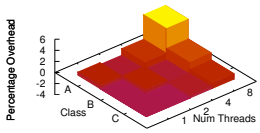

Figure 6: NPB CG ARM wrapper code overhead.

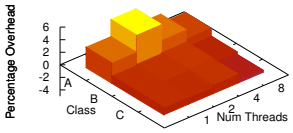

Figure 7: NPB CG wrapper x86 code overhead.

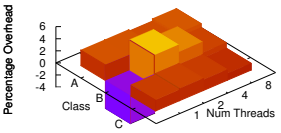

Figure 8: NPB IS wrapper ARM code overhead.

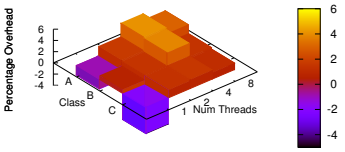

Figure 9: NPB IS wrapper x86 code overhead.

size and number of threads increase (several configurations show speedups due to cache effects). These results indicate that inserting migration points does not significantly impact performance, as migration points consist only of a function call and a memory read.

Unified Layout. We next evaluated the cost of the final stage of the modified compiler – imposing a unified layout by aligning symbols across multi-ISA binaries. Table 1 shows the execution time and the L1 instruction cache miss ratios of IS and CG (classes A, B and C) versus the unaligned version of the binary. As shown in the table, execution time changes up to 1% in these configurations, meaning that symbol alignment has a negligible impact on performance for applications. L1 instruction cache miss ratios are strongly correlated with application speedup/slowdown. We observed less than a 0.001% difference in L1 data cache misses. This demonstrates that data alignment has a small impact on performance.

	IS A	*CG A*	*IS B*	*CG B*	*IS C*	*CG C*
$x86_{Exec}$	0.984	1.018	1.009	1.036	0.999	1.014
$x86_{L1IMiss}$	0.843	1.005	1.000	1.091	0.942	1.040
ARM_{Exec}	0.994	1.0177	1.006	1.003	1.0074	1.004
$ARM_{L1IMiss}$	0.870	2.096	2.8254	1.005	1.175	1.129

Table 1: ARM and x86 execution time and L1 cache miss ratios compiling w and w/o alignment. NPB IS and CG, class A, B, and C compiled with `-O3`. *Exec* values higher than 1 indicate a slowdown due to alignment, lower values a speedup.

Stack Transformation. After evaluating overheads imposed by the new compiler toolchain, we then evaluated the runtime costs of migration. Figure 10 shows the stack transformation latency, in microseconds, for the CG, EP, FT, and IS benchmarks. The plots show the range, 1st and 3rd quartiles and median latencies for transforming the stack at all migration points in the binary. The x86 processor is able to transform the stack in under 400μs for the majority of cases, while the ARM processor requires 2x as much latency. Regardless, transformation latencies are small enough that they

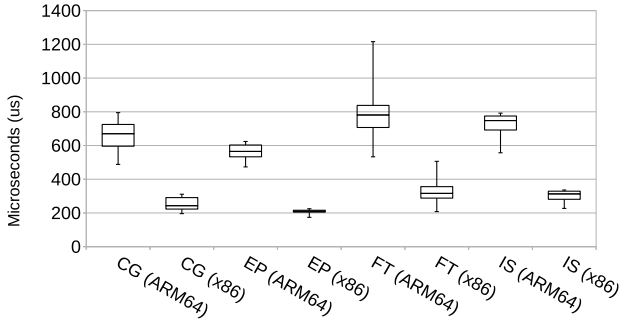

Figure 10: Stack transformation latencies. Each plot shows the minimum, 1st quartile, median, 3rd quartile and maximum transformation latencies experienced across all migration points for each benchmark.

do not become a bottleneck for frequent thread migrations.

In general, stack transformation latencies rise proportionally with the number of stack frames and variables in each stack frame. This is due to both parsing the compiler-generated metadata to analyze stack frames and for copying live values from the source to destination stack. For example, the migration point for the function `fftz2` in FT requires re-writing 7 frames and a total 31 live values, leading to heavier lookup and re-writing costs. This migration point caused the longest transformation latency for x86-64 and ARM.

Migration. We evaluated the instantaneous power (both processor and external readings) and CPU load when migrating an application between x86 and ARM. We compared against PadMig (Java) which serializes application objects and sends them over the network. We migrated one function of the NPB IS B serial benchmark (`full_verify()`) to ARM, with the remainder of the application on x86. We used NPB version 3.0 which includes IS in both Java and C. Results are depicted in Figure 11, with PadMig on the left and our prototype on the right. The first row shows ARM power and load, while the second shows the same for x86. The total execution time is 23s for Java and 11s for native. The results show how serializing data (from seconds 5 to 7 of the bottom left

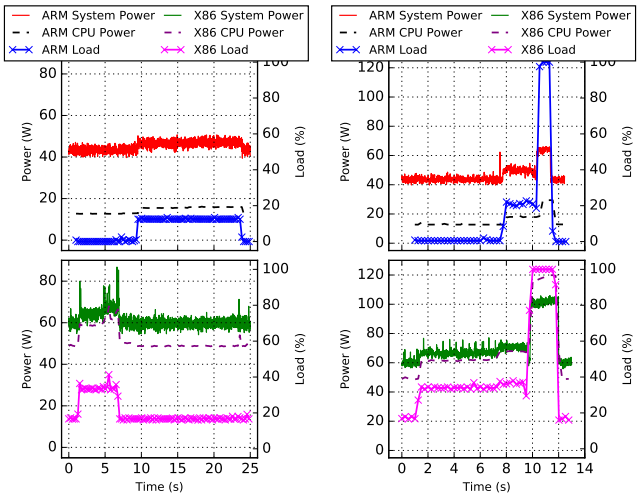

Figure 11: PadMig (Java) vs Multi-ISA binary migration (native). Power and load traces for NPB IS B serial execution.

graph) and de-serialization (from seconds 9 to 13) requires up to 8s of execution time. Migration in our solution starts at second 8, and the application resumes execution immediately on ARM. Power and load of our solution spikes towards the end of execution because the system is transfering lots of pages (for a period of only 2s). This is because the hDSM service is multithreaded, even though the application is serial. The graphs also show how the external power consumption for this benchmark (similarly to all our benchmarks) is proportional to the internal power readings, and thus we only report internal power readings for the rest of the section.

Job Arrivals and Scheduling. We evaluated how dynamic scheduling using migration compares to static load balancing. For our comparison we generated sets of jobs from our benchmarks using a uniform distribution, evaluating both a sustained workload and periodic arrivals. Because the X-gene 1 is a first-generation development board with sub-optimal power consumption, we used McPAT [42] to project that on FinFET technology, future ARM processors will consume 1/10th of the measured power while running at the same clock frequency. We first did compare static versus dynamic policies among ARM and x86 but the results shown a net win of dynamic scheduling (independently on the scheduling policy), which is minimum consuming twice more energy and taking double more time to execute (on multiple workloads). Therefore, here we compare static policies on two (identical) x86 machines with dynamic load balancing on the ARM and x86.

Sustained workload. Figure 12 shows the total energy and the makespan ratio between different policies on 10 sustained workloads. Each workload consists of 40 jobs that arrive sequentially without overloading any of the machines. Once a job finishes, another job is immediately scheduled in its place. As shown in Figure 12, job migration increases the flexibility of the system and reduces energy consumption at the expense of execution time (49% on average with the balance policy as the slowest). Despite the slowdown, the unbalanced policy achieves up to a 22.48% reduction in energy compared to the static policy (unbalanced provides on average a 11.61% energy reduction, while balanced is 7.88% more energy efficient).

Periodic workload. Figure 13 shows the total energy and the Energy Delay Product (EDP) of the static and the dynamic policies of 10 periodic workloads. Each workload consists of 5 waves of arrivals of up to 14 jobs each (in order to not overload the two machines). Each group of arrivals is spaced in time between 60 and 240 seconds. We omitted the dynamic unbalanced results because the results differ from the dynamic balanced policy by less than 1%. As shown in Figure 13, migration improves both energy and EDP. Our system provides on average a 30% energy reduction and an 11% reduction in EDP. The ARM and x86 setup with heterogeneous-ISA migration provides an energy reduction for all sets (up to 66% for set-3), although EDP reduction is variable between sets.

8. Related Work

Heterogeneous Migration, State Transformation. Seminal work from Attardi *et al.* [7] advocated for user-space process migration among heterogeneous-ISA servers, and was implemented by Smith and Hutchinson in the TUI System [60]. TUI implements execution migration in distributed systems with full state conversion when applications migrate across heterogeneous-ISA servers. Yalamanchili and Hy-

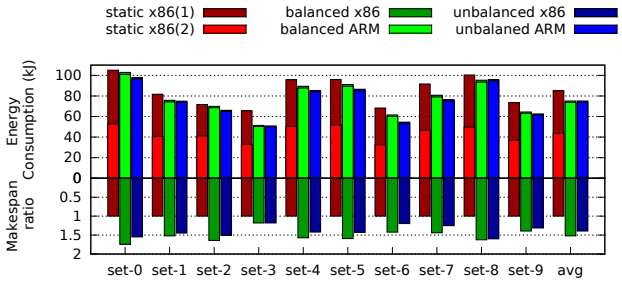

Figure 12: Sustained workload. Energy consumption breakdown by machine for each scheduling policy and total makespan ratio of the heterogeneous scheduling policies to the static policy for different workload mixes.

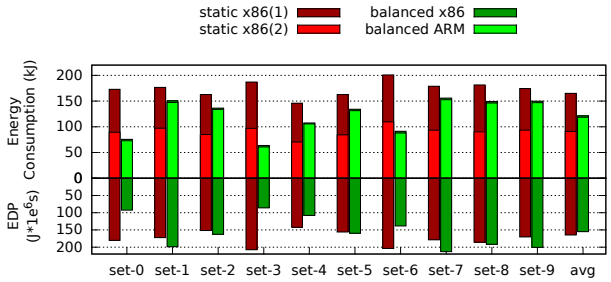

Figure 13: Periodic workload. Energy consumption breakdown by machine for each scheduling policy and Energy Delay Product (EDP) for the static and the dynamic policies for different workload mixes.

att [72] enumerated the differences between migrating among homogeneous and heterogeneous machines and proposed a transformation-based approach. Similarly, our work implements execution migration targeting native compiled applications. However instead of relying on state transformation, we modify the compiler so that binaries conform to a common address space format to the extent that is possible, i.e., when performance is not affected.

More recently, DeVuyst *et al.* [24], Venkat and Tullsen [68], and Barbalace *et al.* [16] introduce application migration among heterogeneous-ISA processors that share memory, enforcing a (partially) common address space for threads on each ISA. DeVuyst explores process migration by performing program state transformation melded together with binary translation to migrate on a heterogeneous-ISA CMP. This paper focuses instead on distributed systems that enable ensemble-level energy advantages. Differently from these works, we provide a formalization, a new (multi-ISA) binary architecture, operating system extensions, and a real prototype.

Multiple works exist on migration among heterogeneous-ISA machines with object-oriented languages. Heterogeneous Emerald [63], implemented in the Emerald language compiler and runtime (without OS support), passes objects between machines using serialization/de-serialization. PadMig [29] and JnJVM [30] use reflection in the Java language to also serialize and de-serialize objects. More recent works such as COMET [31] and CloneCloud [22] propose migrating Java applications between portable devices running on ARM processors and x86 servers in the cloud. Alternatively, our design does not require object semantics or managed languages for migrating applications between heterogeneous-ISA machines.

Heterogeneous DSM. Zhou *et al.* [75] introduced Mermaid, a heterogeneous distributed shared memory system similar to our hDSM service. Instead of taking an abstract approach, however, we built a prototype in order to study its performance. Our hDSM service was also inspired by IVY [41], although hDSM was implemented in kernel space and not in user space. IVY uses a modified malloc, and thus only provides DSM for heap-allocated objects. During allocation, the developer must specify a data type so that during memory page transfers each element on the page can be converted between formats. Our design does not require converting page content: it is in a common format across binaries. IVY also does not facilitate thread migration, although it supports multithreaded applications. A similar approach to Mermaid was implemented with a more rigorous formalism in Mach [28]. Mach tags data objects in memory (typed malloc, similarly to Lisp) so that at runtime a converter can translate object contents. Differently from other approaches, our design requires no code transformation and minimal runtime conversion, reducing migration execution overheads.

Operating System Heterogeneity Support. Operating system designs to support heterogeneous-ISA processors have been proposed in the context of a single platform [36, 15, 12, 44]. None of these designs have been shown to work for fully heterogeneous-ISA processors. Moreover, they are similar to distributed OSs and thus do not provide a generic OS extension to migrate OS containers. Helios [53], implemented on top of Singularity [36], provides primitives to migrate a managed application between ARM and x86 in a single platform.

Sprite, a network OS proposed by Ousterhout *et al.* [54], aimed to hide the distributed aspect of networked machines. Popcorn Linux [12] mimics this, though for heterogeneous CPUs instead. In this paper we extended the Popcorn Linux OS to migrate Linux containers between heterogeneous-ISA servers. Thus only interactions among processes in the container must be propagated among machines creating the containerized environment.

Linux applications can be migrated among homogeneous machines using checkpoint/restore functionality [5]. Other operating system provide homogeneous-ISA migration capabilities, e.g., Dragonfly BSD [35]. Our work contributes seamless thread migration among heterogeneous-ISA machines without the overheads of checkpoint/restore mechanisms.

9. Conclusion

Datacenters are already built with heterogeneous-ISA machines, but the fundamental software mechanisms that currently enable energy-efficiency among homogeneous machines are hindered by this heterogeneity.

In this work, we propose a redesign of the traditional software stack in order to enable natively-compiled applications to migrate between ISA-diverse machines in the datacenter. Specifically, we introduce a compiler that builds multi-ISA binaries, which conform to a single layout, and a runtime that transforms application state that cannot be kept in a common format (mostly due to performance reasons). Additionally, we present an operating system that enables elastic containers that can migrate between kernels, based on a multi-kernel design. We built a prototype based on Linux and demonstrated that applications do migrate between ARM and x86 faster than Java serialization/deserialization. Applications compiled with our toolchain experienced no more than a 1% impact on performance. Stack transformation, the only state transformation needed in our approach during migration, took on average less than one-half millisecond on x86 and less than a millisecond on ARM. We show with different arrival patterns that migration on heterogeneous-ISA machines can improve energy efficiency up to 22% for sustained loads and up to 66% for bursty arrivals, as compared to static assignment among two homogeneous x86 machines; moreover, the EDP is on average reduced by 11%.

References

[1] BZIP2SMP. http://bzip2smp.sourceforge.net, 2005.

[2] Redis. http://redis.io/, 2016.

[3] David Abdurachmanov, Brian Bockelman, Peter Elmer, Giulio Eulisse, Robert Knight, and Shahzad Muzaffar. Heterogeneous high throughput

scientific computing with apm x-gene and intel xeon phi. *Journal of Physics: Conference Series*, 608(1):012033, 2015.

[4] AMD. AMD Opteron A-Series Processors. http://www.amd.com/en-us/products/server/opteron-a-series.

[5] Tycho Andersen. Lxd live migration of linux containers. Linux Conference Australia, 2016.

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

[7] G. Attardi, I. Filotti, and J. Marks. Techniques for Dynamic Software Migration. In *In ESPRIT '88: Proceedings of the 5th Annual ESPRIT Conference*, pages 475–491. NorthHolland, 1988.

[8] R. Azimi, X. Zhan, and S. Reda. How good are low-power 64-bit socs for server-class workloads? In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 116–117, Oct 2015.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing '91*, 1991.

[10] 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*, 2014.

[11] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Ottawa Linux Symposium*, 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*, EuroSys '15, pages 29:1–29:16. ACM, 2015.

[13] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 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*, SOSP '09, pages 29–44. ACM, 2009.

[15] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your Computer is Already a Distributed System. Why Isn'T Your OS? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 12–12. USENIX Association, 2009.

[16] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. Harnessing Energy Efficiency of heterogeneous-ISA Platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '15, pages 6–10. ACM, 2015.

[17] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, July 2008.

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, January 2016.

[19] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. *SIGPLAN Not.*, 30(11):70–78, November 1995.

[20] Juan Angel Lorenzo del Castillo, Kate Mallichan, and Yahya Al-Hazmi. Openstack federation in experimentation multi-cloud testbeds. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 02*, CLOUDCOM '13, pages 51–56, Washington, DC, USA, 2013. IEEE Computer Society.

[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] 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.

[23] Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 140–140, Washington, DC, USA, 2006. IEEE Computer Society.

[24] 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. ACM, 2012.

[25] Dolphin Interconnect Solutions. Express IX. http://www.dolphinics.com/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf.

[26] Richard P. Draves. Control Transfer in Operating System Kernels. Technical Report MSR-TR-94-06, Microsoft Research, May 1994.

[27] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 101–115. USENIX Association, 1999.

[28] Alessandro Forin, Ro Forin, Joseph Barrera, Michael Young, and Richard Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for mach. Technical report, In 1988 Winter USENIX Conference, 1988.

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

[30] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. *Live and Heterogeneous Migration of Execution Environments*, pages 1254–1263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[31] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[32] Part Guide. Intel® 64 and ia-32 architectures software developer's manual, 2011.

[33] Hewlett Packard Enterprise. Hpe moonshot system. https://www.hpe.com/us/en/servers/moonshot.html.

[34] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[35] Jeffrey M Hsu. The dragonflybsd operating system. *Proceedings USENIX AsiaBSDCon, Taipei, Taiwan*, 2004.

[36] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[37] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. 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, Nov 2010.

[38] Adam Jundt, Allyson Cauble-Chantrenne, Ananta Tiwari, Joshua Peraza, Michael A. Laurenzano, and Laura Carrington. Compute bottlenecks on the new 64-bit arm. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, E2SC '15, pages 6:1–6:7, New York, NY, USA, 2015. ACM.

[39] Willis Lang, Jignesh M. Patel, and Jeffrey F. Naughton. On energy management, load balancing and replication. *SIGMOD Rec.*, 38(4):35–42, June 2010.

[40] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[41] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, New Haven, CT, USA, 1986. AAI8728365.

[42] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[43] Boston Limited. Boston viridis; presenting the world's first hyperscale server – based on arm processors, July 2016.

[44] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. *ACM Trans. Comput. Syst.*, 33(2):4:1–4:27, June 2015.

[45] Linaro. Developer cloud; the developer cloud for the arm64 ecosystem, July 2016.

[46] Peng Lu, Antonio Barbalace, and Binoy Ravindran. Hsg-lm: Hybrid-copy speculative guest os live migration without hypervisor. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 2:1–2:11, New York, NY, USA, 2013. ACM.

[47] R. Luijten, D. Pham, R. Clauberg, M. Cossale, H. N. Nguyen, and M. Pandya. 4.4 energy-efficient microserver based on a 12-core 1.8ghz 188k-coremark 28nm bulk cmos 64b soc for big-data applications with 159gb/s/l memory bandwidth system density. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, pages 1–3, Feb 2015.

[48] 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. ACM, 2013.

[49] Marvell. Chinese internet giant baidu rolls out world's first commercial deployment of marvell's arm processor-base server, February 2013.

[50] Mesosphere Inc. Introducing the mesosphere datacenter operating system. http://mesosphere.com/.

[51] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[52] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, June 2007.

[53] 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*, SOSP '09, pages 221–234. ACM, 2009.

[54] John K Ousterhout, Andrew R Cherenson, Frederick Douglis, Michael N Nelson, and Brent B Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[55] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Compilers and operating systems for low power. chapter Dynamic Cluster Reconfiguration for Power and Performance, pages 75–93. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[56] Qualcomm. Qualcomm makes significant advancements with its server ecosystem. https://www.qualcomm.com/news/releases/2015/10/08/qualcomm-makes-significant-advancements-its-server-ecosystem, October 2015.

[57] 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*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[58] Scaleway. Cloud computing features for your infrastructure, July 2016.

[59] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[60] Peter Smith and Norman C. Hutchinson. Heterogeneous Process Migration: The Tui System. Technical report, 1996.

[61] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[62] P. Stanley-Marbell and V. C. Cabezas. Performance, power, and thermal analysis of low-power processors for scale-out systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 863–870, May 2011.

[63] 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.

[64] MOOR Insight & Strategy. The first enterprise class 64-bit armv8 server: Hp moonshot system's hp proliant m400 server cartridge, 2014.

[65] MOOR Insight & Strategy. Building the ecosystem for arm servers, November 2015.

[66] Ian Lance Taylor. A New ELF Linker. In *Proceedings of the GCC Developers' Summit*, 2008.

[67] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering energy proportionality with non energy-proportional systems: Optimizing the ensemble. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, pages 2–2, Berkeley, CA, USA, 2008. USENIX Association.

[68] Ashish Venkat and Dean M. Tullsen. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132. IEEE Press, 2014.

[69] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[70] 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*, USENIX'09, pages 28–28, Berkeley, CA, USA, 2009. USENIX Association.

[71] Daniel Wong and Murali Annavaram. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 119–130, Washington, DC, USA, 2012. IEEE Computer Society.

[72] Mallik V. Yalamanchili and Robert M. Hyatt. Heterogeneous process migration: Issues and an approach. In *Proceedings of the 35th Annual Southeast Regional Conference*, ACM-SE 35, pages 275–281, New York, NY, USA, 1997. ACM.

[73] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[74] 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*, ICAC '12, pages 145–154, New York, NY, USA, 2012. ACM.

[75] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):540–554, September 1992.