Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms

Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran

ECE, Virginia Tech

{antoniob,sadini,bmsaif86,bielsk1,akshay87,ckendir,alastair,binoy}@vt.edu

Abstract

The recent possibility of integrating multiple-OS-capable, high-core-count, heterogeneous-ISA processors in the same platform poses a question: given the tight integration between system components, can a shared memory programming model be adopted, enhancing programmability? If this can be done, an enormous amount of existing code written for shared memory architectures would not have to be rewritten to use a new programming paradigm (e.g., code offloading) that is often very expensive and error prone. We propose a new software architecture that is composed of an operating system and a compiler framework to run ordinary shared memory applications, written for homogeneous machines, on OS-capable heterogeneous-ISA machines. Applications run transparently amongst different ISA processors while exploiting the most optimized instruction set for each code block. We have implemented and tested our system, called Popcorn, on a multi-core Intel Xeon machine with a PCIe Intel Xeon Phi to demonstrate the viability of our approach. Application execution on Popcorn demonstrates to be up to 52% faster than the most performant native execution on Linux, on either Xeon or Xeon Phi, while removing the burden of the programmer having to adopt a different programming model than shared memory on a heterogeneous system. When compared to an offloading programming model, Popcorn is shown to be up to 6.2 times faster.

1. Introduction

The landscape of heterogeneous computing is changing. The main CPU on a platform is no longer exclusively paired with special-purpose computational units (e.g., GPUs); instead, general-purpose high-core count multicore processors are starting to be adopted [13, 32]. These emerging high-core count processors are general-purpose and are thus able to run a fully-fledged operating system, i.e. they are "OS-capable" [21]. As a result, the platform that can be assembled out of them will be running different OSes, each potentially utilizing a different instruction set architecture (ISA). Each application runs on a single OS/ISA, unless written within an exotic programming library (OpenCL, MYO, etc.) or within a distributed memory programming model (e.g., MPI) to exploit the heterogeneity. We foresee that such platforms will become increasingly popular. Hence, in order to improve their programmability, we propose a software architecture that enables applications written within the shared memory programming model and compliant to the POSIX interface to run transparently across the heterogeneous system. Furthermore, to enhance performance we propose to exploit the workload diversity that exists between and within applications by mapping each workload's code block onto the optimal processor island in a heterogeneous system, when predicted to be efficient.

Emerging heterogeneous platforms are characterized by tightly-coupled ISA-diverse processor islands. Every ISA is distinguished by a different performance profile depending on the workload. A processor island is defined as multiple processor cores sharing a single ISA and a single memory coherency domain. However, multiple processors islands may or may not support shared memory, and if they support shared memory the link between them may or may not implement cache coherency. Therefore, two main aspects are required to run applications transparently and efficiently across a heterogeneous system: a mechanism to provide shared memory amongst different processor islands when not present, and another mechanism to exploit ISA heterogeneity by mapping different code blocks onto the most suitable processor island at any time.

This paper targets a platform on which Intel Xeon processors and an Intel Xeon Phi board (Xeon-Xeon Phi hereafter) are connected via PCIe. Xeon and Xeon Phi are two processor islands with different architectures but overlapping-ISAs. Each one can remotely access the main memory of the other processor, but there are no memory coherency guarantees.

We show that with our software infrastructure, shared memory programs do not have to be rewritten to run on the Xeon-Xeon Phi. Depending on the number of cores available, our system always makes the best mapping decision of code to processor islands, delivering a performance im-

provement over the best native execution of up to 52% and 6.2 times faster execution than OpenCL and offload versions of the same benchmarks. Even though the current Popcorn prototype was built, deployed, and evaluated on two overlapping-ISA processors, we believe that the same ideas and similar results also apply to fully heterogeneous-ISA platforms, such as platforms that integrate Tilera TileGx [32] and x86, or ARM and x86. However, when the heterogeneity between architectures increases, more engineering will be needed to run applications amongst multiple processor islands. We will address the extra effort needed while describing the implementation.

1.1 Motivations

Programming paradigms other than the shared memory model can be cumbersome, limit the programmer's expressivity [30], and promote breaking of the one-OS-per-platform model. We advocate that the shared memory programming model should be used on heterogeneous OS-capable systems, and that even if the platform is heterogeneous, the programmer should see a single system/OS.

Application offloading has been used within multiple OS-capable processor islands. On the Xeon-Xeon Phi, OpenCL, the MYO and LEO runtimes can be exploited [33] for this purpose. The application is loaded on one processor island and every time a block of code should be offloaded, that block and the data on which it is working are transferred to the other. The offloaded computation is compiled for the island-specific ISA and integrated into the application by the runtime and/or the compiler. The programmer has to manually partition the application and decide which code blocks should and should not be offloaded. Offloading breaks the shared memory paradigm and adds an additional layer of complexity due to the difficulty of manually finding the best code partition (which is usually application- and accelerator model- dependent). Moreover, from an OS point of view, processors that are not part of the main CPU island are seen as external devices and not as processors themselves, breaking the execution-flow abstraction. When executing on a different processor, the offloaded code sees a different execution environment.

A distributed memory programming model or a partitioned global address space (PGAS) programming model can be also adopted on OS-capable heterogeneous-ISA platforms. We assume that nowadays, a large code base written using the shared memory programming model exists. Rewriting such a large code base would require a serious investment. Further, rewriting an application with a distributed memory programming paradigm is cumbersome and error prone [34]. In fact, some applications have to be almost totally rewritten in order to comply with a different programming paradigm. For example, comparing the distributed memory (MPI) and shared memory (OpenMP) versions of the NPB applications [1], we noticed that more than 44% of the code needed to be rewritten (see Table 1).

Benchmark	CG	EP	FT	IS	MG
Difference	98%	44%	98%	46%	97%
Total OMP LOC	1150	297	1106	1108	1481

Table 1. Code differences between the OpenMP and MPI versions of NPB applications [1] (version 3.3).

Benchmark	CG	EP	FT	IS	MG
OpenMP	21%	14%	4%	37%	6%
OpenCL	303%	164%	143%	177%	189%
Serial LOC	506	163	606	454	852

Table 2. Additional code required by the OpenCL and OpenMP versions compared to the serial version of NPB applications (SNU NPB [29] version 1.0.3).

Different programming models require the code to be modified from its serial version, which can demand up to 98% of the application's code to be rewritten in the case of MPI (Table 1) or up to 303% additional code in the case of OpenCL (Table 2). Finally, hybrid approaches exist, but they inherit the same problems of their constituent models.

1.2 Popcorn

Popcorn bridges the programmability gap in heterogeneous-ISA platforms for the application developer, increasing productivity by reducing the development and porting time. In fact, Popcorn enables multi-threaded shared memory programs written for homogeneous-ISA multiprocessors to run on heterogeneous-ISA platforms. This eliminates the necessity of learning a different programming model and removes the burden of code rewriting and/or code optimization. In contrast to offloading, the developer does not have to partition the code. Instead, Popcorn will prepare the code to run on the available processor islands. Based on the code block's performance profile, Popcorn will pick the best island on which to execute. In this way, the diversity and different degrees of parallelism in the same application can be exploited. In contrast to distributed memory programming, in which the programmer has to not only partition the memory but also instruct the runtime to transfer it, Popcorn provides distributed shared memory (DSM) transparently to the application. All these features come with no additional effort from the application developer.

1.3 Contributions

We contribute a design, a prototype, and an initial evaluation of an architecture, made up of an operating system and a compiler framework, that enables POSIX shared memory applications to run in a heterogeneous-ISA environment. Our prototype runs on the Xeon-Xeon Phi platform. In order to gain more traction, the software architecture is based on the Linux environment, hence many existing applications written for Linux can be tested on Popcorn without

any rewriting. Our operating system is the first Linux-based replicated-kernel OS [3] running on a heterogeneous-ISA platform. We also introduce an extended memory subsystem for Linux that allows consistent task-based address space replication and DSM amongst kernels. Our compiler framework transforms shared memory programs so that they can migrate and execute on different processor islands. The runtime selects for each thread, at specific points, on which processor island to execute. If execution is deemed more efficient on another processor island, the runtime triggers an inter-kernel migration. To the best of our knowledge, we are the first in contributing such an architecture and deploying it.

In the next section, we introduce Popcorn's software architecture on generic hardware, and in Section 3, we describe our prototype implementation. Section 4 presents the experiments and Section 5 describes the results. We present related work in Section 6 and conclude in Section 7.

2. Popcorn Architecture

Popcorn aims to create the illusion of a single execution environment amongst tightly-coupled diverse processor islands, allowing applications to exploit the most suitable processor island for each code block at runtime.

Design Principles Popcorn has its foundations in some of the design principles typical of just-in-time (JIT) high level languages, SMP OSes, and cluster OSes:

- *Transparency.* The user should not see processor island boundaries, but a single system on which applications can run everywhere and use all the available resources on the platform. Developers should not have to slice up their programs in order to make them faster; instead, they should be able to focus on the application's logic while assuming an SMP system.
- *Load Sharing.* The produced application executable should be able to run on any processor island by potentially containing the code for each architecture. The OS should be able to handle the binary format and the ISA switch. Tasks should migrate intra- and inter- processor islands without being limited to a processor island.

In addition, Popcorn incorporates one more principle in order to take advantage of heterogeneity:

- *Exploiting Asymmetries.* Although sometimes it may be desirable to mask the asymmetries, a new programming interface should expose the architectural distinctions and allow for their exploitation. Architectural asymmetries, which may be due to the CPU microarchitecture or the hardware resources local to each processor island on which the task can execute, should be exploited.

These design principles lead to the Popcorn architecture in Figure 1 and Figure 2.

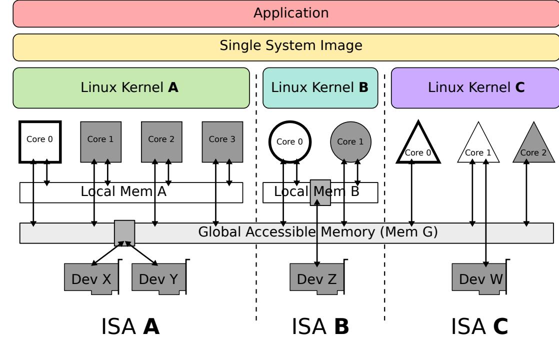

Figure 1. Heterogeneous-ISA generic hardware model, and Popcorn operating system software layout.

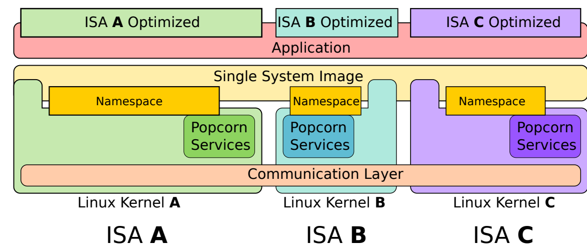

Figure 2. Popcorn kernel- and user-level software layout.

2.1 Hardware Model

Our software architecture is designed to work with a generic hardware platform, depicted in Figure 1, which attempts to abstract current and emerging hardware. Popcorn assumes a hardware model where processors of the same ISA are grouped in islands, and different islands share access to a global, eventually consistent, memory (*Mem G* in Figure 1). Computational units of a single island may have exclusive access to a memory area (*Mem A* and *Mem B*) and across islands, the same memory area can be mapped to different physical address ranges. A similar model holds for accessing devices and peripherals that are mainly memory-mapped. Some devices, like *Dev X* and *Dev Y*, can be directly accessed by any processor. Others, like *Dev Z* or *Dev W*, cannot.

2.2 Software Layout

Figure 1 shows Popcorn's software architecture and how it is layered on top of a generic hardware model. The schema illustrates a single application compiled with the Popcorn compiler framework that is running on the Popcorn operating system. The application is multi-threaded, and different threads potentially run on different kernels (and therefore on different-ISA processors). The application's code is compiled for each ISA available on the system, and in each case with the highest optimization. The user-space runtime, interacting with the operating system, picks the optimal processor island on which to execute a given thread.

2.3 Operating System Architecture

The operating system consists of different kernels, each compiled for, and running on, a different processor island, as shown in Figure 1. Therefore the kernel code must be portable to any ISA of the heterogeneous platform. Kernels interact to provide applications with the illusion of a single OS amongst different processor islands. The OS state is partially replicated on all kernels in order to account for thread migrations across them, and resource sharing (e.g., memory and devices). When no hardware cache coherent shared memory is available between kernels, Popcorn additionally provides software DSM, so that multi-threaded applications, written for shared memory architectures can continue to work.

A communication layer glues kernels together and provides basic data conversion between ISAs, as in Figure 2. The communication layer is a key component: all replicated-kernel OS services rely on it (e.g., thread migration, page coherence, thread synchronization, etc.). Kernels communicate through it to maintain a single (partially replicated) OS state.

Popcorn's services and namespaces layer, depicted in Figure 2, strive to create a single environment for applications running amongst kernels, and make applications assume that they are executing on a traditional SMP OS. Popcorn's namespaces layer on each kernel provides a unified processes and resources view, similar to Plan 9 [23].

Amongst kernels, the state of each migrating application (e.g., address space) is replicated and is kept consistent. Because we are targeting shared memory applications that can potentially run different threads on different processor islands, and two tightly connected processors islands may or may not be cache coherent, software DSM should be provided. Finally, strategies to migrate threads across-ISA should be introduced too: because the whole system is heterogeneous, a scheduler must consider the asymmetries existent between processor islands.

2.4 Compiler Support

Popcorn's compiler framework takes an application's source code as input, and after a series of offline analysis and profiling combined with a pre-built knowledge of the hardware platform, emits a multi-ISA binary that can run on the Popcorn operating system.

The operating system provides a single cohesive view of the multiple kernels running on heterogeneous hardware. However, it cannot hide the functional incompatibility of different processors, and it should not hide the asymmetric performance that they provide. Compiler support is required to produce programs that are both capable of running on a heterogeneous OS, and to exploit the best possible application workload division on the replicated-kernel OS.

Some existing approaches to heterogeneous systems make use of intermediate languages [21] or embedded target-independent intermediate formats [10], but the dominant way of programming in Linux still involves compiling to a single architecture-specific binary. Our approach continues to provide this single binary solution, but packed with support for multiple architectures and allowing switching architectures at specific migration points. While it would be possible to pack multiple versions of a program into a single binary and choose an architecture when the program starts, this would not allow for efficient use of the hardware. Many programs contain both serial and parallel code and would benefit from using different hardware at those different points of execution.

Arbitrary scheduling of a heterogeneous program across different architectures is not possible with this approach as incompatible instructions and application binary interfaces (ABI) will cause immediate problems. This is why our compiler framework chooses code blocks where a migration is likely to give a boost in performance and inserts code to interact with the kernel to optionally perform a migration across architectures at that point. The inserted code chooses whether to perform a migration, and if a migration is to be performed, it packs up the data required for a transition in an ABI-independent manner.

3. Implementation

We deployed Popcorn on a PCIe interconnected Intel Xeon and Intel Xeon Phi platform, which is an overlapping-ISA heterogeneous hardware configuration. Both multiprocessors implement a common set of the x86 instructions but Xeon and Xeon Phi implement different x86 ISA extensions (e.g., using different sets of FPU registers) that makes them ISA heterogeneous. Moreover, the clock frequency is 2.2 times higher on the Xeon processors and other microarchitectural differences exist. Our implementation consists of $\sim 37k$ lines of code amongst the Linux kernel and the patches to the Intel MPSS to boot and communicate with the Xeon Phi. The compiler implementation requires $\sim 5k$ lines of code amongst compilers passes, scripts and modifications to the libraries to make them work across-ISAs. The implementation details are discussed in the following section. The full source codes for Popcorn Linux and associated tools can be found at `http://www.popcornlinux.org`.

3.1 Operating System

Popcorn implements a replicated-kernel OS design using the Linux kernel as the basic building block [3]. We deploy one kernel on the Xeon and another on the Xeon Phi (respectively compiled with the `x86-64` and the `k1om` compiler target architecture). Different software components have been introduced in the Linux kernel and are described hereafter. To support fully heterogeneous-ISA platforms such components shouldn't be redesigned but just ported to the target architectures because diversity is handled by the upper software layers. Our changes to the architecture-dependent sub-

directory of the Linux kernel consists only of $\sim 1.5k$ lines, easing portability.

3.1.1 Messaging Framework

We designed a kernel-level, pluggable, low latency inter-kernel messaging layer able to sustain high data throughput. A pluggable interface was chosen to ease portability and to support multiple communication channels between kernels.

There are different types of messages and for each message type a message handler is registered. When a message arrives at a receiving end, it is stored in a queue first, and then picked up by one of a set of kernel threads that executes the handling function corresponding to the message's type.

We use the MPSS's SCIF [12] library provided by Intel to communicate between Xeon and Xeon Phi. SCIF was designed to be a per-process library. Therefore, we modified it to have a set of communication channels which are usable from every thread in the system. These channels are created at boot time and are managed by a set of kernel threads.

Communication channels are implemented on top of two mechanisms for message delivery: PIO (`scif_send()`) or DMA (`scif_writeto()`). We developed a hybrid messaging layer that switches between PIO and DMA, dynamically, depending on the message size, but at different thresholds on Xeon and on Xeon Phi. From our experiments, as shown in Figure 3, we found that for small messages, PIO transfers have lower latencies; however DMA transfers are faster for large messages. In fact we observed that the platform has asymmetric communication times; Figure 3 shows the cost of sending a message using PIO (*scifSend*), DMA without notification (*dmaSend*), DMA with a short message notification (*dma*), and Popcorn varying the message size in both directions. From Xeon to Xeon Phi PIO transfers are faster than DMA until $64kB$, however from Xeon Phi to Xeon PIO transfers are faster than DMA only up to $256B$. The fact that our messaging layer is the fastest in Figure 3 is due to our enhancements to SCIF. Amongst other changes we increase the size of the SCIF PIO buffers to $2MB$. This also impacts the DMA transfers that rely on PIO for message delivery notification. Finally in Figure 4 we report the cost of a message ping-pong between Xeon and Xeon Phi varying the message content size. This represents another asymmetry of the adopted platform: the cost is more than 4 times higher when the sending is initiated on the Xeon Phi. This likely reflects the difference in clock frequency and core topology between the two processor islands.

We implemented a DMA buffering scheme with maximum message size of $8kB$ that provides enough room to send every message type without fragmentation. Each transmission channel has 256 pre-allocated $8kB$ DMA mapped buffers. Moreover, we introduced a transfer mechanism for huge messages up to 512 pages long ($2MB$). We experimentally found that to ensure maximum utilization of the PCIe bus we must use 8 parallel channels. This is equivalent to the number of the DMA engines available on the Xeon Phi.

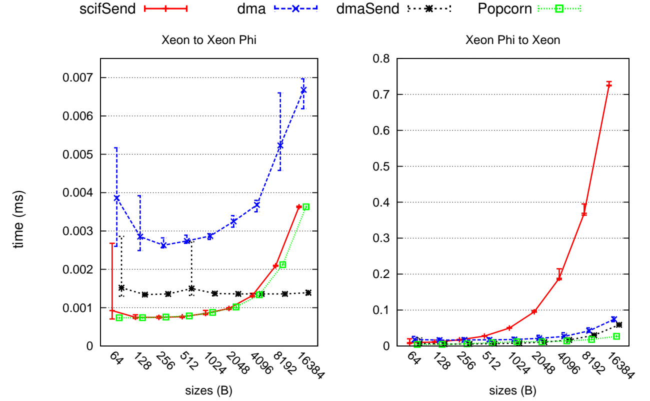

Figure 3. The cost to send a message from Xeon to Xeon Phi and the reverse. The cost is asymmetric.

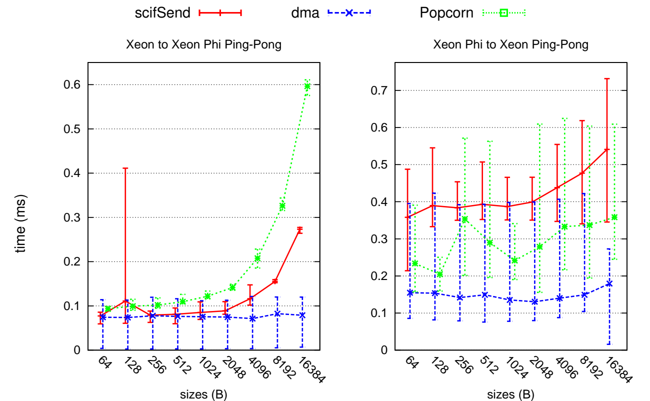

Figure 4. Ping-pong message cost from Xeon to Xeon Phi and the reverse. The cost is asymmetric.

Each channel has one receiver kernel thread at each end. This is reflected in an application's execution time and shown in Figure 5, where 8 channels always represent the best configuration for performance. We evaluated a buffered, i.e., asynchronous, and non-buffered, i.e., synchronous, version of the messaging layer. It was found that with the non-buffered version of the messaging layer, increasing the number of threads caused a significant slowdown in the application execution. Figure 6 shows this impact in conjunction with the memory footprint of the applications, where small, medium, and large correspond to class A, B, and C of the NPB benchmarks [1]. Note that the buffered version is up to 2.5 times faster when 228 threads are running. In our experiments in Section 4, we use 8 communication channels and the buffered version of the messaging layer.

3.1.2 Namespaces

Linux provides namespace mechanisms to be used as a form of lightweight virtualization [20]. Popcorn re-engineered Linux's namespaces to provide the opposite mechanism: an aggregate view of all local resources available on each kernel (users, file system, IPC, PID, network, and CPU).

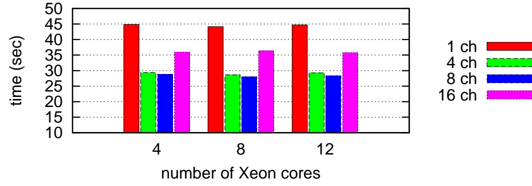

Figure 5. Application performance varying the number of Xeon cores and messaging layer channels.

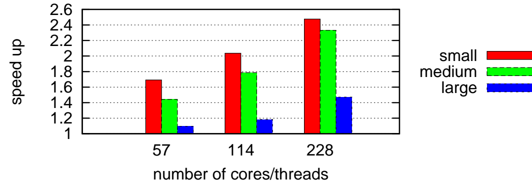

Figure 6. Application speed up of buffered vs non-buffered messaging layer varying the number of application's threads and memory footprint (small, medium, large).

By default the Linux kernel initializes a set of namespaces; however, the user must switch to Popcorn's namespaces to exploit Popcorn's capabilities. Popcorn initializes its namespace objects and updates them every time a new kernel joins Popcorn Linux (joining happens via the messaging layer).

We added the CPU namespace to Linux. The CPU namespace enables an application running on a kernel to list all the CPUs available on all kernels and eventually migrate to any of them. The CPU namespace affects */proc/cpuinfo* entry but also the behavior of `sched_setaffinity()` and `sched_getaffinity()`. These functions manage all the CPUs on which Popcorn OS is running. Because of the pthread library's assumption that the number of CPUs in a system does not vary dynamically outside a certain static range, after an application joins the Popcorn CPU namespace, the number of CPUs seen by the application is fixed. In other words, if another kernel joins Popcorn CPU namespace after an application has been associated, the application will not see the new kernel's resources.

Linux's PID namespace is based on the integer id management library (IDR). IDR allocates PIDs from a set of available integers in an efficient way. In Popcorn, instead of creating a central server for integer allocation, which could slow down the system, we subdivided the set of allocatable integers amongst the kernels. In the Xeon-Xeon Phi setup, we allocate the lower half of integers to Xeon and the upper half to Xeon Phi.

The file system namespace is currently running by relying on NFS to share the same filesystem on both kernels. We describe how file descriptors are migrated in Section 3.1.4. Section 3.1.4 explains the Futex IPC mechanism.

3.1.3 Task Migration

Popcorn introduces inter-kernel user-space task (thread and process) migration in Linux. A task migration consists of copying the task state from one kernel, where the task currently executes, to another kernel that potentially runs on a different ISA processor. A kernel-level migration service runs on each kernel. To handle migrating tasks quickly, a pool of dummy user-space tasks is maintained on each kernel. Dummy tasks are kept in a sleeping state until they are activated due to an incoming inter-kernel task migration request. Therefore, the pool adds a minimal resource overhead to the system. When a new task arrives from a remote kernel, the state of the migrating task is installed in a dummy task and the execution is resumed in the new kernel.

Migrations are triggered by the application/user calling the system call `sys_sched_setaffinity()`. According to a CPU bitmask provided to those functions, a destination kernel is selected by evaluating which CPU ranges are associated to which kernel. When a migration is triggered a message is sent to the designated kernel and the local task is put to sleep. If the task migrates back to this kernel at a later time, then this sleeping task will be resumed. The first migration has a higher OS cost than successive migrations. This is because the destination kernel needs to select threads from the dummy pool and attaching each of them to the incoming process. From the second time onward a thread that is migrated to that kernel already knows its hosting thread and therefore successive migrations are faster. Figure 7 shows the initial migration OS cost varying the number of concurrent migrating threads. The blue line shows that successive migrations are up to 35 times faster, but this reduces with the number of threads to only 2 times faster.

Thread and Process States. The task state of a thread can be divided into process related and thread related state. The process state is shared between threads of the same process, whereas the thread state is private.

The thread state includes the content of all CPU registers used by the task before entering kernel space. All information regarding the task state is migrated each time a task moves to another kernel. On the Xeon-Xeon Phi platform, whose processors are both x86, overlapping-ISA, we are currently migrating the integer registers as-is (in Linux,

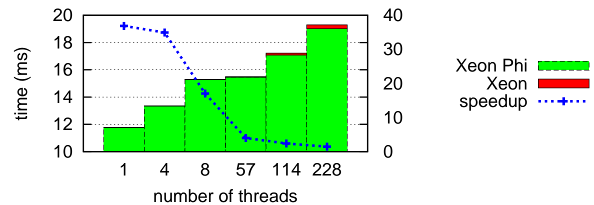

Figure 7. The bar graph shows average per-thread first migration OS cost, on Xeon and on Xeon Phi. The blue line shows the speed up for subsequent migrations.

saved in `struct pt_regs` object). The floating point registers (Linux's `union thread_xstate`) are not migrated because of differences between the two architectures. Popcorn's compiler framework is responsible for FPU register handling during migration. Therefore floating point code cannot migrate at arbitrary points.

The process state includes all information regarding the memory layout (e.g., virtual memory areas), file descriptors, futex, process-shared signals, IPC, and user and group credentials. Because of the shared nature of such state, this information cannot be simply copied during thread migration. Specific services have been introduced to keep this information updated during the lifetime of the process. The services that are used to keep file descriptors consistent and maintain futex states amongst kernels are described in Section 3.1.4. The memory management is also described in 3.1.4. IPCs and credentials are managed via namespaces (see Section 3.1.2).

3.1.4 Consistent Services

A process's memory state is comprised of virtual memory areas (VMA) and a virtual to physical addresses map. In Linux, `struct vm_area_struct` and the page directory, accessible from `pgd_t` in the `struct mm_struct`, contain these information, respectively. VMAs divide the address space in non-overlapping ranges of addresses, each one with specific memory access privileges and properties. Underneath the VMA layer, the memory is further divided and seen as organized in pages of fixed granularity (only $4kB$ page size is currently supported in our prototype). Each virtual page of a process is paired with a physical memory page that contains the actual data. Virtual to physical memory mappings are stored in a hierarchical page directory and are used by the hardware memory management unit (MMU) when virtual addresses must be resolved upon load/store operations by the CPU.

The first thread of a process that migrates to another kernel resumes its execution into a dummy task that has an empty memory state, without VMAs and without (user-space) virtual to physical address mapping. Anytime this task tries to access a new page, an exception is raised by the hardware. The kernel will resolve the VMAs and virtual to physical mappings on demand. In Popcorn, Linux's memory exception handler (`page_fault()`) has been modified to notify Popcorn's memory management service of the migrated task's memory faults. Intercepting memory exceptions allowed us to implement a user-level transparent DSM between threads of the same processes running among different kernels. Such transparency makes code instrumentation unnecessary.

The memory management service implements different protocols according to the physical memory architecture that exists among kernels. For Xeon-Xeon Phi we decided to use a page replication protocol of exploiting the non cache-coherent shared memory available on the heterogeneous platform. In this protocol, each page accessed by a process is replicated on demand and kept consistent by a page-coherency algorithm.

Page Replication Algorithm. The page replication algorithm is the core mechanism of Popcorn's page-granularity user-level DSM. For transparency, we provide the same memory consistency the user expects on SMP platforms. We exploit the protection mechanisms provided by the MMU to monitor application's accesses to memory pages, and the messaging layer is used to transfer updates. A generic implementation of this protocol for an arbitrary number of kernels is proposed in [26]. For Xeon-Xeon Phi, we optimized the algorithm to run with two kernels. The goal of this protocol is to maximize the number of accesses that can be performed directly on the local address space copy, without triggering updates to the other copies, and therefore, to minimize the messages exchanged between kernels to keep replicas consistent. In the protocol, a replicated page can be in one of the MSI [22] states: *Modified*, *Shared*, or *Invalid*. A page in *Modified* can be accessed in read/write without triggering any page fault, a page in *Shared* can only be accessed for read, and a page in *Invalid* cannot be accessed. We implemented the same protocol transitions as in MSI and introduced various optimizations.

To quickly resolve concurrent write conflicts (i.e. two kernels wanting to write on the same page in *Shared* state at the same time), we introduced the page owner, as previously proposed by Ivy [16]. At any time, only one kernel has the ownership of a replicated page. The replica that owns the page can be upgraded to *Modified* state, whereas the other copy is set to *Invalid*. The ownership of a page is dynamic and it is transferred when a write (transition to *Modified*) is acknowledged.

To reduce message traffic, the page replication algorithm is not always active on all pages of the address space. The page replication algorithm works on demand: a local page copy will be created only if a task accesses such a page from that kernel. To model this, we introduced two new states in the protocol: *Not Replicated* and *Not Mapped*. If a page has never been accessed by a kernel, the page is in the *Not Mapped* state in that kernel. If it has been accessed by only one of the two kernels, the page is in the *Not Replicated* state and read/write accessible by that kernel. Pages will transition from *Not Mapped* or *Not Replicated* states to MSI states when a process' task migrates to another kernel and asks for those pages.

Guided Pre-fetching. We implemented guided pre-fetching of pages to speed up process execution. A new system call has been introduced to inform the memory management service about the virtual addresses that will be needed by the application after migrating to a different kernel. The corresponding pages are then moved to the selected kernel with the highest access privilege granted by their VMA (*Not Replicated* if read only, *Modified* if read-write accessible).

The list of addresses needed is inserted at compile time and passed to the memory service just before a task migration is invoked.

File Descriptors Algorithm. To support task migration we extended the Linux virtual file system layer (VFS) while providing the same POSIX semantics offered by Linux. Tasks are initially migrated between kernels without their file descriptors. To minimize migration cost file descriptors are sent on demand. When a task invokes a file system operation (e.g., read, write) the extended VFS layer checks if that particular file descriptor is present in the task's file descriptor table. If it is not present, and the task has been migrated, the layer sends a message to the kernel where this process was first created (home kernel). The home kernel returns the file descriptor information such as file name, current offset, mode and the kernel where it was opened (owner kernel). This information is used to open the same file in the current kernel (via NFS and mount namespaces) and the file descriptor table is updated accordingly. If a task wants to open a new file while on a remote kernel it has to query its home kernel to get the next available file descriptor.

To guarantee the same file system consistency level provided by Linux while reducing the message traffic, we augmented the file descriptor object with a mode flag. The file descriptor can be in one of the two modes: *exclusive* or *shared*. If a file descriptor is used by only one thread of a process all changes to the file offset are kept in the kernel where the thread resides (*exclusive* mode). When multiple threads query for the same file descriptor the *exclusive* mode is changed to *shared* mode. In *shared* mode all file offset updates are sent by threads to the owner kernel.

Futex Algorithm. Fast user-space mutex [19] (Futex) is a central mechanism used in libc. Because almost all POSIX applications are based on libc we decided to support it.

Futex performs inter-thread synchronization using the OS `futex_wait` and `futex_wake` syscalls. SMP Linux implementation handles concurrent requests by means of a table of global queues (Linux's `futex_queues` object) protected by a spinlock. Serialization is ensured by this mechanism for a monolithic kernel. However, because Popcorn is a replicated-kernel OS, we extend this functionality within Popcorn by adopting a client/server model. The kernel on which the user-space futex variable has been initialized becomes the server for that futex. The futex queue in the server kernel is considered as the global queue for that futex, maintaining global state of all the requests. All the others maintain a local queue. Messages are exchanged between the futex server and its clients to ensure synchronization.

3.2 Compiler Framework

Section 2.4 described the need for compiler support to be able to produce a single binary that can effectively run on a heterogeneous OS. The implementation of this support takes place in three phases: code analysis and profiling, code transformation, and the additional work required to make external libraries work with the resulting heterogeneous binary.

We constrain our compiler framework to allow migrations to only occur at function boundaries (i.e., the call site must be in a function that the compiler can modify and not at the call site of an immutable external library function, such as `printf`). This means that the compiler can modify the program such that instead of the function `foo` calling `bar`, it can now insert the code necessary to migrate between architectures so that `bar` will be, optionally, executed on a different processor island. This introduces the restriction that when `bar` returns, the execution migrates back to the original processor island, so that a single execution of `foo` always runs on a single architecture. These limitations reduce the amount of work required to support multiple ABIs. If the program execution did not return to the original architecture, then the entire stack would need to be converted to the new architecture's ABI.

To support fully heterogeneous platforms the runtime must be extended as proposed in previous works (e.g., DeVuyst *et al.* [6]), and the binaries must be generated accordingly by the compiler framework.

3.2.1 Finding Optimal Partitionings

The first step to produce a heterogeneous binary is to find an ideal partitioning that would most benefit the performance. The benefit in migrating a program from one architecture to another is that some functions would get better performance being mapped to one architecture, while other functions could be optimally mapped to another architecture. The cost of migrating a program is primarily the data-transfer cost that will be incurred by switching architectures at a given point. Often this cost can outweigh the benefits of performing a migration, so an effective analysis is required to determine when it is worth migrating. For example, in the implementation presented herein the Xeon processor is faster at executing serial code, but the Xeon-Phi is faster at executing parallel code. Simply running all serial code on one and all parallel code on the other is unlikely to provide the best mapping. Since the page faults incurred to handle data-transfers within the heterogeneous OS must be handled over PCIe, they are much more expensive than cache misses. Therefore it is often best to run a function on a sub-optimal architecture if it means that it will be accessing local data.

The Cost Model. Figure 8 demonstrates the cost model used to find an optimal partitioning. Note that this analysis phase is a one-time offline cost per program per platform. The result of the analysis phase can vary according to the number of cores available on each processor island. Each vertex in a graph represents a function (A, B, C, D in Figure 8), each edge between two functions represents a cost that would be incurred by inserting a migration at that call boundary. The raw data used to create these cost graphs for each benchmark is obtained by using a LLVM [15] pass to

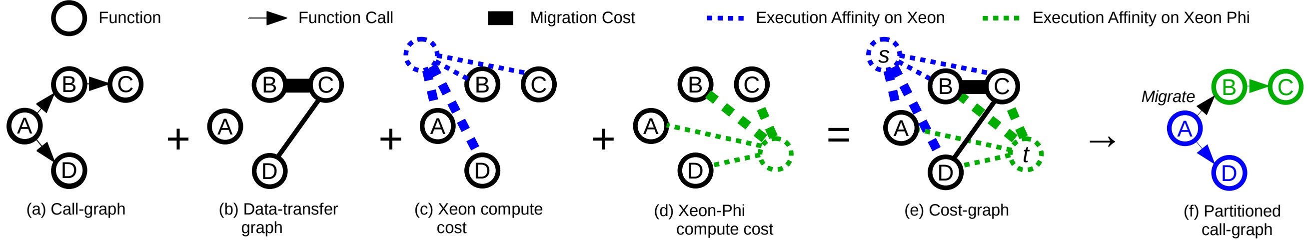

Figure 8. The different steps of the offline analysis to partition the code between Xeon (in blue) and Xeon Phi (in green).

annotate every function call and memory access with a call to a tracking library we developed. In turn, this tracking library then generates the necessary data during a profiling run of the binary. It is necessary to track every address to know which pages are being accessed, providing a foundation for a precise runtime analysis of the application memory access pattern. The analysis library builds up a call-graph, shown in Figure 8(a), and then finds a partitioning of functions between architectures[1].

The first cost considered, is the raw cost of migrating a thread across architectures as there is a time involved in pausing a task and migrating it to another kernel. The second cost, shown in Figure 8(b), represents the data-transfer requirements between functions. This is where the runtime analysis is essential, so that it is able to know precisely which function most recently read or wrote to a particular page and thus owns it (i.e. that page currently resides on the architecture that executed that function). The thicker lines in Figure 8(b) represent pairs of highly coupled functions that access many of the same pages, and thus it is desirable for those two functions to reside on the same architecture to avoid data-transfers. Finally, Figures 8(c) and (d) represent how well each function maps to each architecture, again a thicker line means that it is desirable for that function to be mapped to that architecture.

The four graphs are combined into a single cost graph, shown in Figure 8(e), by assigning a weight to each type of event. These weights are the number of nanoseconds required to handle a single event of that type. The edge between two functions represents the number of nanoseconds that will be added to the program's run-time if a migration happens at that function call boundary. The edge between a function and a virtual compute cost node is the estimated cost in nanoseconds of *not* running on that architecture, i.e. how many nanoseconds will be added to the total run-time by choosing a different architecture.

The migration and page fault costs are stable enough to be considered constant for this analysis and thus their weights are measured directly. However, the differing compute cost of executing a single function on different architectures varies greatly and can only be approximated. An approximate nanosecond cost of executing a single memory access is found by dividing the runtime of a set of benchmarks by the number of tracked memory accesses. The number of tracked memory accesses per-function is then weighted by this measured value. Finally, the compute cost is divided by the number of processors available for functions that will execute in parallel (i.e. those that contain parallel OpenMP loops, or are called from inside a parallel loop), but is left unmodified for functions that are not parallel.

To find the optimal partitioning we need to assign each vertex to a partition such that the sum of the weights of the edges crossing between the two partitions is minimized. For example, in Figure 8(e) the best partitioning is {*s*, A, D}, and {B, C, *t*} as the edges crossing between these sets have small weights. This is known as the "min-cut", and although there are many algorithms for solving that globally, we have the additional constraint that *s* and *t* reside on opposite sides of the cut. We find an *s*-*t* min-cut by exploiting the max-flow/min-cut duality theorem [7]. We map Figure 8(e) to a flow network, find the maximum-flow from *s* to *t*, and then map that back to a *s*-*t* min-cut by exploiting the property that any vertex reachable from s using residual flow in the network must belong to the same partition as *s*. This lets us find the partitioning shown in Figure 8(f), and shows us that we should migrate between architectures on the edge between vertices A and B.

Finding the min-cut of a graph results in a binary partitioning. For example, in the final reachability stage of the above algorithm, each node is either reachable from *s*, or not. This means that the approach as presented does not generalize to N architectures, which is not required for the Xeon-Xeon Phi, but should be supported to follow the proposed design principles. Less precise graph partitioning approaches such as clustering algorithms could be used to split the cost graph into N partitions, but are out of scope for this paper. Finally, the partitioning analysis only considers computational capability. It could be extended to consider how other costs, such as network or disk I/O, differ between processor islands when determining optimal partitionings.

3.2.2 Transforming Programs

Given the resulting partitioning sets, e.g., $\{A, D\}$ and $\{B, C\}$, the set mapped to the Xeon Phi is used as input in the transformation phase. This phase has been implemented with the

[1] If a function is not executed during the profiling run it will not considered for migration.

source-to-source Rose [24] compiler, targeting the C/C++ language. The program transformation locates in the original source files all the functions that belongs to the $\{B, C\}$ set, that have been marked with a `pragma` keyword above the function definition during the partitioning phase. These functions, hereafter called *compute functions*, are copied in a new source file that will be exclusively compiled for Xeon Phi. For distinguishability, all *compute functions* are appended with a postfix to show whether they are compiled to run on host or target architecture. To resolve any dependencies of sub-functions that might be called from the *compute functions*, the program transformation recursively finds any function called within the *compute functions* and creates a static version in the newly created file. Each call to a *compute function* is a possible migration site.

For each migration site the program transformation tool packs any data required for a migration in a data structure, allowing ABI independence. Specifically, this refers to the argument list that is passed to the called *compute function*.

We use this data structure to guarantee that the necessary arguments are saved in the heap instead of in the stack or registers. Then to facilitate the task returning to the host architecture, when the function completes, the return value is copied into the same data structure.

Any instance of a function call to a *compute function* in the original source code is replaced with a call to the `migration_hint()` function. This function takes as arguments `struct_fun`, `fun_Xeon`, `fun_XeonPhi`, where `struct_fun` is the ABI independent packaged data structure. `fun_Xeon` is the *compute* function that, if selected, will run on Xeon; `fun_XeonPhi` is the *compute* function that, if selected, will run on Xeon Phi (additional arguments can be added to accommodate more ISA kernels within the same system). The `migration_hint()` function initiates the transition from one ISA architecture to another. Finally, this phase does minor clean-up tasks, adding required headers, function declarations, and substituting standard library calls with per-architecture ones where necessary.

3.2.3 Library Support

When external libraries are needed, such as *libm*, then within a traditional linking approach that library will be compiled for a single architecture. In a heterogeneous binary it is essential that a function compiled for a particular architecture is only linked to functions compiled for the same architecture. To compile and execute applications in a heterogeneous environment we rebuilt system libraries in order to provide both Xeon and Xeon Phi implementations of their functions.

In addition, an optimization was made to the *glibc* library in order to improve the performance of the page fault coherence algorithm described in Section 3.1.4. As supporting concurrent writes to a single page across multiple kernels is expensive, false sharing should be avoided. Often multiple mutexes are created together. To avoid multiple mutexes on a single page, the `pthread_mutex` data structure is padded to $4kB$ to ensure that only a single instantiation can exist on a page. The current prototype is restricted to static linked executables only, dynamic loading is out of the scope of this work.

4. Experimental Evaluation

Hardware. We ran all experiments on a system containing two Intel Xeon E5-2695 (12 cores, 2-way hyper-threaded at 2.4GHz per socket in a dual-socket configuration), 64GB of RAM, and an Intel Xeon Phi 3120A (57 cores, 4-way hyper-threaded at 1.1GHz, 6GB of RAM) connected via PCIe.

Following the results of the analysis in Section 3.1.1, our data was collected with a configuration of 8 Xeon cores and 228 Xeon Phi cores. We limit the experiments on the Xeon to 8 cores because the majority of the NPB applications do not see any performance gain by running on the Xeon Phi, when more than 8 Xeon cores are used. Table 3 demonstrates this for the CG application. Thus, the partitioner tool never decides to migrate from Xeon to Xeon Phi. We kept this configuration for all experiments.

Software. The Popcorn prototype is based on Linux 3.2.14 and Intel MPSS 3.2.3, which was ported from kernel 2.6.38.8 to 3.2.14. Since the namespace code on Linux 3.2.14 was not complete, we backported part of the namespace code from Linux 3.8. The Linux distribution used in the experimental system was CentOS 6.5.

In order to compile applications for Popcorn Linux, we used a combination of LLVM 3.4, ICC 14.0.3, gcc 4.4.7, gcc 4.7 (k1om), and Rose 0.9.5a. We partially rewrote and recompiled GNU libc 2.13 (shipped with Intel MPSS 3.2.3) and Intel OpenMP 5.0 (libiomp) to make them work across ISAs and to enable a medium compiler memory model[2] for statically-compiled NPB applications.

Compute/Memory Intensive Benchmarks. The main difference between the processor islands in our test platform is that the Xeon Phi can run highly parallel code more efficiently than the Xeon processor. On the other hand, serial and I/O bound workloads run faster on the Xeon processor. To exploit this heterogeneity, we evaluated our prototype by using compute/memory intensive workloads written in OpenMP.

We ran the OpenMP C version of the NAS Parallel Benchmarks from the SNU NPB benchmark suite [29], ver-

[2] NPB requires a medium compiler memory model, the `-mcmodel` option.

Cores	Xeon 4	Xeon 8	Xeon 12	Xeon Phi 228
CG.A	1.04s	0.53s	0.09s	0.24s
CG.B	59.56s	31.46s	10.87s	15.41s
CG.C	162.13s	86.28s	29.93s	60.71s

Table 3. Xeon core counts vs Xeon Phi core counts. Executing on 12 Xeon cores is faster than executing on any number of Xeon Phi cores.

sion 1.0.3. We compared the execution time on Popcorn against native Linux execution on only the Xeon cores and only on the Xeon Phi cores. For a fair comparison, *x86-64* and *k1om* binaries were compiled with the same optimizations (`-O3`) and compiler (`icc`). Moreover, the same executable was used to run homogeneously (intra-ISA) and heterogeneously (inter-ISA). This was made possible by modifying the ELF header.

We also compared with the OpenCL version of SNU NPB, and with the offloaded version of OpenMP. We wrote the offloaded version starting from the original SNU NPB OpenMP versions, offloading all available parallel sections. We wrote the offloaded versions using Intel's Language Extension for Offload (LEO [33]), an extension available for the Xeon-Xeon Phi platform. Both the OpenCL and the offloaded versions rely on the *coi_daemon* distributed with MPSS 3.2.3. The collected OpenCL numbers refer only to the compute kernel execution time.

Single System Benchmarks. To demonstrate that Popcorn works without the need to rewrite an application, we ran POSIX applications "as is", i.e. without refactoring them with the partitioner. Moreover, we inserted explicit scheduler migration calls because the prototype currently does not support in-kernel scheduler migration decisions.

We ran the POSIX-compliant compression application Parallel BZIP version 1.1.2 (pbzip2) [8]. We generated the input files using the system random device `/dev/urandom` and used default settings, including $900kB$ buffer slot size. Parallel BZIP creates one thread per processor core available on the platform and two I/O threads. In Popcorn, the number of available cores is returned as the sum of the total cores available on Xeon and the total cores available on Xeon Phi. Therefore, pbzip2 places its threads on each available core regardless of the CPU architecture and considers them as part of an SMP architecture. pbzip2 is a stream-oriented application that stresses all of Popcorn's consistent services implemented in the prototype and is therefore representative of most POSIX applications.

5. Results

The goal of our experimental evaluation was to demonstrate that our software infrastructure is effective in providing an operating system for heterogeneous platforms that supports transparent programmability. Additionally, we wanted to show that the compiler framework produces an effective code block placement on the available cores on the Xeon Phi. We assume that all applications start on the Xeon CPU.

Running with Code Analysis. Figures 9, 10, 11, and 12 show the execution times (in seconds) of the CG, EP, IS, and SP benchmarks of the NPB suite, respectively. For each benchmark, we report the execution time for *Class A*, *Class B*, and *Class C* categories (i.e. different input data sizes) using different numbers of available cores on the Xeon Phi processor (4, 8, 57, 114, and 228). Each data point was averaged over 10 executions, and the maximum deviation per sample never exceeded $11ms$.

In the figures, an asterisk is used to mark whenever Popcorn migrates the execution at least once between the Xeon and Xeon Phi. The other benchmarks in the NPB suite either never migrate between Xeon and Xeon Phi (i.e., UA), or have a similar migration and performance pattern as the ones shown (e.g., BT has the same pattern as SP).

When an application runs on Popcorn, a migration occurs when the cost model (Section 3.2.1) determines that the performance benefits outweigh the communication overheads due to the migration. From the figures, it is clear that due to the large difference in the clock frequency between the Xeon and Xeon Phi processors, Popcorn never migrates threads when a small number of cores are allocated on the Xeon Phi, running on only the Xeon cores. When 57, 114, or 228 cores are available on the Xeon Phi, the behavior changes for each benchmark.

For the SP benchmark, presented in Figure 12, the partitioner decides to always migrate for 57, 114, and 228 threads, providing improvements over native execution. With 57 threads, Popcorn is up to 53% and 46% faster than native execution on the Xeon Phi and Xeon (Class C), respectively. When the number of threads increases to 114, Popcorn is up to 33% and 61% faster than native execution on the Xeon Phi and Xeon (Class B), respectively. With 228 threads, Popcorn is still able to outperform native execution, although its performance advantage is reduced. This performance gain demonstrates the benefits of the partitioner's decision to migrate execution to the Xeon Phi.

The SP benchmark allows for a direct comparison of Popcorn and the offload implementations because the migration points are the same in both versions. However, Popcorn is always faster than its competitors. In particular, Popcorn is up to 3.5 times faster on 57 cores on Class C. From this superior performance, we conclude that Popcorn's kernel-level shared memory model amongst processor islands allows for better performance than an offloading software stack implemented in user-space between the Xeon and Xeon Phi. The same performance trends and conclusions hold for both EP (Class B and Class C) and IS (Class C) benchmarks.

We observe that for all other benchmarks, the Class A versions have the shortest execution time by more than one order of magnitude. Because of this short execution time, the partitioner never migrates any of the benchmarks, as the advantage of greater parallelism in the Xeon Phi would be offset by the migration overhead. Thus, Popcorn executes the Class A versions of CG, EP, and IS on 8 Xeon cores.

In contrast, the OpenCL and offload versions of these benchmarks always migrate because these versions do not have any migration policy mechanism that selects a processor island on which to execute. Therefore, the OpenCL and offloading versions are (in most cases) slower than Popcorn,

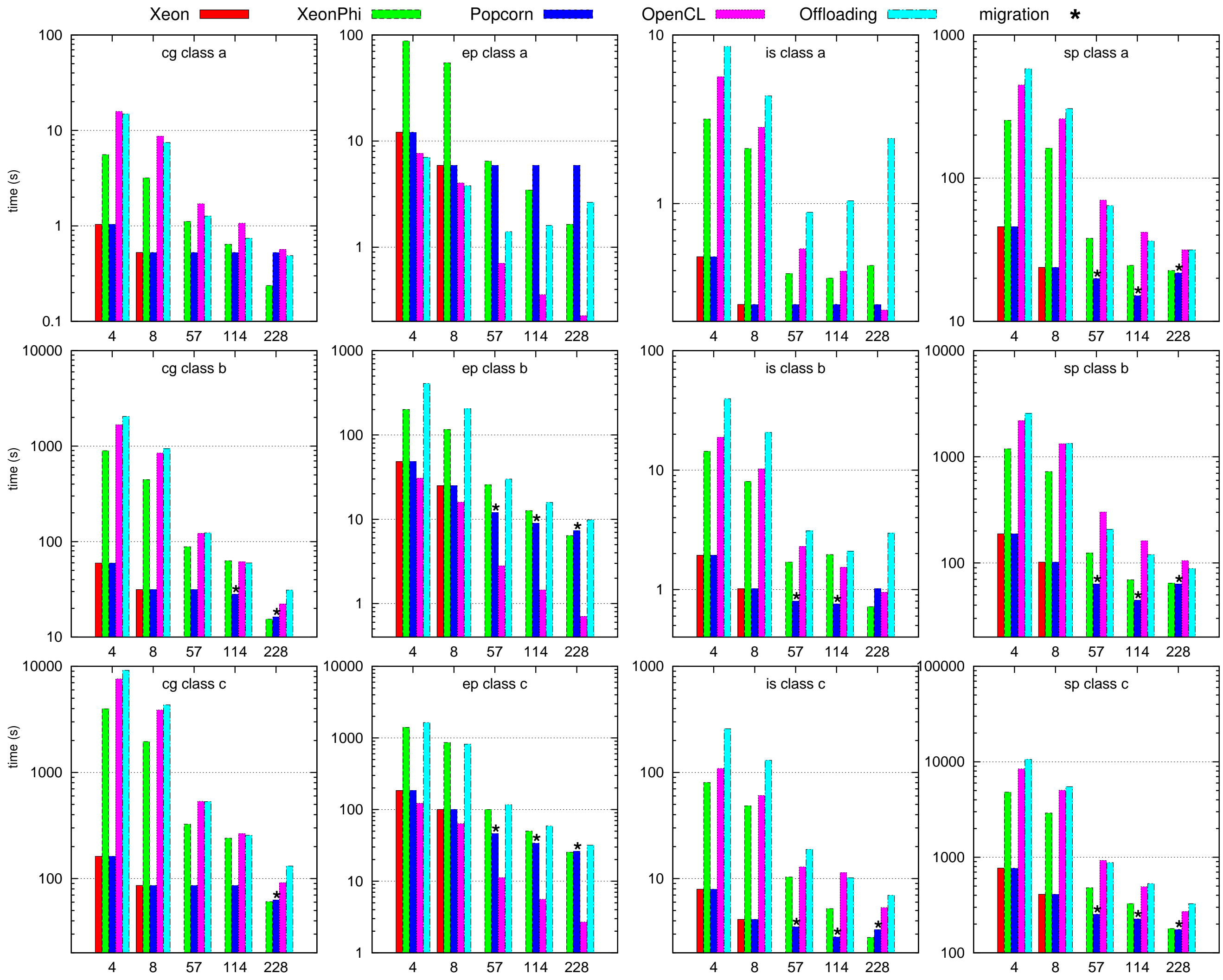

Figure 9. CG **Figure 10.** EP **Figure 11.** IS **Figure 12.** SP

with up to a 12 times slowdown for IS Class A using 228 cores.

The EP Class B and Class C benchmarks (Figure 10) have the same trend as the SP Class B and Class C benchmarks (Figure 12). Popcorn is up to 52% and 53% faster than native execution on the Xeon Phi for EP Class B and Class C, respectively, for 57 threads. With 114 threads, Popcorn's speedup decreases to 25% and 32%, respectively. For 228 threads, Popcorn is slower than native execution on the Xeon Phi (33% in Class B and 8% in Class C), but faster than native execution on the Xeon (65% and 72%) due to higher overhead in thread migration. EP represents a special case for OpenCL in which Intel's analysis tool allowed us to discover its exploitation of extremely efficient mathematical functions that could not be used for the other versions of the benchmark, hence Popcorn's performance is slower.

Uniquely to CG, the partitioner migrates for higher number of threads (114 or 228). In this case, Popcorn is up to 27% and 43% better than native execution on the Xeon in Class B and Class C, respectively. From Figure 9, we observe that for both Class B and Class C, native execution on the Xeon Phi has a clear advantage over native execution on the Xeon for only 228 threads. This is detected by Popcorn's partitioner tool. However, the OpenCL and offload versions are not able to detect this and experience up to 6.2 times slowdown for 57 cores, Class C.

IS is the fastest benchmark. It turns out that Popcorn's partitioned IS migrates to Xeon Phi only for 57 and 114 threads in Class B and Class C, respectively. In both cases, execution using Popcorn is faster than native execution on the Xeon Phi (for Class C, Popcorn is also always faster than native execution on the Xeon). We hypothesize that this behavior is due to the fact that the Popcorn partitioner has a cost model that better fits FPU computations than integer computations on which the Xeon Phi appears to be slower (IS only has integer computations).

Our evaluation illustrates that compute/memory intensive benchmarks written for SMP and running on a replicated-

File Size	128MB	256MB	512M	1G
Xeon 8 cores	4.3s	8.3s	17.3s	36s
Xeon Phi 228 cores	9s	15s	24s	74s

Table 4. pbzip2 execution time on 8 Xeon cores and 228 Xeon Phi cores. pbzip2 is always faster on the Xeon cores.

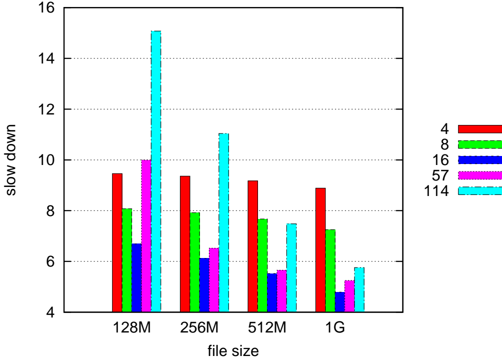

Figure 13. Execution time slowdown of the pbzip2 application on the Xeon Phi with respect to 8 Xeon cores using Popcorn. The slowdown is shown for different input file sizes and number of Xeon Phi cores.

kernel OS with DSM can take significant advantage of a heterogeneous platform when compiled with our partitioner. Additionally, Popcorn's system software infrastructure is more performant than a user-space offloading software infrastructure. However, for short-running benchmarks there is no benefit in using Popcorn due to the high communication overhead of migrating execution between processor islands. Moreover, Popcorn does not always have low migration overhead for high thread counts. This performance profile is application-dependent.

Running without Code Analysis. In Table 4, we report the native execution time of pbzip2 on the Xeon and the Xeon Phi for different input file sizes. pbzip2 is always faster on the Xeon than the Xeon Phi by a factor of 2, meaning there is no benefit in migrating to the Xeon Phi. Exploiting 57, 114, and 228 Xeon Phi cores changes the execution time up to 10%.

Regardless, we ran pbzip2 on Popcorn on 8 Xeon cores varying the number of available cores on the Xeon Phi from 4 up to 114 for different input files. We compared this setup against executing pbzip2 natively on the Xeon, which was always faster. Figure 13 plots the slowdown in execution time. The slowdown in execution time reflects the overhead in all the stressed subsystems of the replicated-kernel OS. In particular, it stresses the DSM protocol and the Futex algorithm. The measurements show that the overhead decreases when the file size increases. This is because larger file sizes increase the per-thread computational time, which diminishes the interaction between threads, thereby amortizing the added OS overhead.

We also observe a different behavior with changes in core counts: the overhead decreases with a different trend relative to the number of cores involved. For a core count up to the number of communication channels, the trend is mostly linearly decreasing. However, once the number of cores used exceeds the number of available channels, the overhead decreases exponentially. This shows how the latency of the messaging framework negatively impacts performance.

This evaluation reveals that even though Popcorn provides a single OS image and DSM, which fosters transparent programmability on architectures with heterogeneous processor islands (a particularly important fact for legacy POSIX applications), the cost of distributed services when used without an analysis tool can significantly degrade performance (e.g., 5 to 16 times slower for pbzip2). An application can indeed run transparently amongst processor islands, but will have a performance advantage only if threads on different islands do not share most of their working sets. Alternatively, the application can be explicitly written to exploit the heterogeneous platform similarly to OpenCL applications.

6. Related Work

To the best of our knowledge, we are the first to present a redesign of Linux to accommodate heterogeneous-ISA hardware together with a compiler framework to support multi-packed binaries. There are, however, previous works regarding both heterogeneous and replicated-kernel operating systems and compilers for those architectures. In a previous paper [2], we present our vision for a replicated-kernel OS for heterogeneous-ISA platforms. In this paper, we follow and extend that vision by providing an actual OS implementation, with additional user-space system software.

Li et al. [17] proposed a Linux design for overlapping-ISA heterogeneous architectures. In their model, cores share a large set of common instructions and registers with identical encoding and semantics. We target a much broader hardware model. Our prototype runs on similar hardware configurations but without cache coherent memory amongst processor islands, that require the deployment of a kernel per coherency domain. K2 OS [18] adapts Linux to run on non-strictly cache coherent overlapping-ISA heterogeneous architectures by introducing the shared-most OS design. The shared-most OS is a middle ground between the shared-memory and the shared-nothing OS designs. Differently Popcorn implements the shared-nothing OS design, without relying on any special hardware supported for locking as K2 does. Similarly to Popcorn, K2 also requires the deployment of a kernel per coherency domain, local and global OS services.

Most research towards OSes for heterogeneous platforms involves the creation of new kernels. Barrelfish OS [4] in-

troduces the multikernel design that runs a microkernel per-core. A heterogeneous version of Barrelfish has also been proposed [28]. Similarly to Barrelfish, Popcorn uses message passing to keep global system state consistent. Although we use a similar message passing design, we do not use remote procedure calls; instead, we use inter-kernel task messaging. COSH [5], prototyped on Barrelfish, introduces a new OS level abstraction which exposes different kinds of memories available in a platform to the programmer at the cost of code rewriting. In contrast Popcorn provides a transparent DSM so code modification is not required. The Helios OS [21] also uses a multikernel-like design but is based on Singularity [11]. In contrast to our design the application programming model is message passing and the application is shipped in an intermediate format not in a multipacked binary.

Work presented by DeVuyst *et al.* [6] examines a MIPS-ARM platform that leverages binary translation for instantaneous migration whereas we avoid binary translation completely. We do however share the concept of migrating to natively compiled code at function call sites. Moreover in [6] the authors envision the availability of a heterogeneous OS that we actually implemented. Unlike adopting an intermediate language or using JIT, Popcorn builds on the idea of FatELF [9], i.e., multiarchitecture binary, or universal binary; code is compiled and optimized for each architecture.

In Saez *et al.* [27] the concept of an "architecture signature" is used to partition the code among processors of a heterogeneous platform. Our analysis phase is similar to their static approach which also creates an application profile offline based on the memory access pattern.

The Intel Compiler Collection shipped with the Xeon-Xeon Phi platform creates a single binary out of two per-architecture optimized assembly code for Xeon and Xeon-Phi. The compiler, together with a user-space daemon, provides the base to execute applications written within the Intel Offloading, MYO, and OpenCL programming paradigms on the Xeon-Xeon Phi platform. However, this compiler is completely driven by the decisions of the developer. Furthermore, there is no single system abstraction. MYO is a shared memory model for Xeon-Xeon Phi. This model requires the programmer to follow semantics similar to offloading to take advantage of shared memory within a heterogeneous system. Consequently, we created our own shared memory model in order to uphold our goal of user transparency.

A similar effort to produce a single system view, or central resource administration, while splitting the system load across processor islands, has been followed by other researchers. However, their efforts are focused on CPU-GPU configurations, as Ptask [25], GPUNet [14], and GPUfs [31], instead of all OS-capable processors, as Popcorn.

7. Conclusions

We demonstrated that it is possible for shared memory applications written for homogeneous-ISA multiprocessors to transparently execute on heterogeneous-ISA multiprocessors with high performance. Popcorn enables this through a replicated-kernel OS and a compiler framework that together creates a single execution environment across heterogeneous processor islands. The compiler framework builds multi-architecture binaries that exploit the most optimized instruction set for each code block, while minimizing the communication overheads of the replicated-kernel OS's distributed services.

We show that compute/memory-intensive applications that run on a Xeon-Xeon Phi platform and utilize Popcorn's capabilities can be up to 52% faster than the most performant native execution. In addition, Popcorn's cost model can adapt the migration decisions to a variable number of processor cores, and yields up to 6.2 times faster execution than the OpenCL and offload models. In particular, for applications with the exact same migration points, Popcorn is faster than an offloading software stack that requires user-level daemons.

Our work also shows that without using a compiler framework, a replicated-kernel OS can indeed be used to run multithreaded POSIX SMP applications as-is. However, such an execution may use hardware and OS services inefficiently, significantly degrading application performance. Popcorn's cost model can determine when the overhead of the replicated-kernel OS offsets the performance benefits of execution migration in heterogeneous platforms. Such models are important for effective resource management.

In future work, we plan to extend the evaluation of Popcorn beyond the Xeon-Xeon Phi platform to fully heterogeneous-ISA platforms, which will involve design and engineering across the system software stack. In general, finding the level of the software stack that must handle architecture diversity to obtain better application performance remains an open question. We also plan to explore how the cost model can be made dynamic and be used to make resource management decisions at run-time for improved performance.

Acknowledgments

The authors thank Mona Attariyan and all the anonymous reviewers and shadow reviewers for their important comments. This work is supported by the US Office of Naval Research under Contract N00014-12-1-0880.

References

[1] 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. .

[2] A. Barbalace, A. Murray, R. Lyerly, and B. Ravindran. Towards Operating System Support for Heterogeneous-ISA Platforms. In *Proceedings of The 4th Workshop on Systems for Future Multicore Architectures (4th SFMA)*, 2014.

[3] A. Barbalace, B. Ravindran, and D. Katz. Popcorn: a replicated-kernel os based on linux. In *Proceedings of Ottawa Linux Symposium (OLS '14)*, 2014.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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, 2009.

[5] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, and T. Roscoe. Cosh: Clear os data sharing in an incoherent world. In *Proceedings of 2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, 2014. USENIX Association.

[6] M. DeVuyst, A. Venkat, and D. 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.

[7] P. Elias, A. Feinstein, and C. E. Shannon. A Note on the Maximum Flow Through a Network. *IRE Transactions on Information Theory*, 2(4):117–119, Dec. 1956. ISSN 0018-9448. .

[8] J. Gilchrist. Parallel bzip2 (pbzip2) data compression software. URL `http://compression.ca/pbzip2/`.

[9] R. Gordon. Fatelf: Universal binaries for linux. URL `http://icculus.org/fatelf`.

[10] K. Group. The standard portable intermediate representation for device programs. URL `https://www.khronos.org/spir`.

[11] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[12] Intel. Intel manycore platform software stack (mpss). URL `https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss`.

[13] Intel Corporation. Xeon Phi product family. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[14] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. Gpunet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216. USENIX Association, 2014.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transaction on Computer System*, 7 (4):321–359, Nov. 1989.

[17] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[18] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300. ACM, 2014.

[19] LWN. A futex overview and update. URL `http://lwn.net/Articles/360699/`.

[20] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. Towards an efficient single system image cluster operating system. In *Proceedings of Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002.*, pages 370–377, Oct 2002.

[21] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[22] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[23] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[24] D. J. Quinlan. ROSE: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[25] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248. ACM, 2011.

[26] M. Sadini, A. Barbalace, B. Ravindran, and F. Quaglia. A page coherency protocol for popcorn replicated-kernel operating system. In *Proceedings of 2013 Many-Core Architecture Research Community (MARC) Symposium*, 2013.

[27] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, 71(1):114–131, Jan. 2011. ISSN 0743-7315.

[28] A. Schpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[29] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, Nov 2011. .

[30] J. Shen, J. Fang, A. L. Varbanescu, and H. Sips. Opencl vs. openmp: A programmability debate. In *Proceedings of the 16th Workshop on Compilers for Parallel Computing (CPC'12)*, January 2012.

[31] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498. ACM, 2013.

[32] Tilera Corporation. TILEncore platforms. `http://www.tilera.com/products/platforms`.

[33] E. Wang. *High-Performance Computing on the Intel® Xeon Phi(tm): How to Fully Exploit MIC Architectures*. Springer, 2014.

[34] M. Zelkowitz. *Advances in Computers: High Performance Computing*. Advances in computers. Elsevier Science, 2009.