A Compiler Framework to Support and Exploit Heterogeneous Overlapping-ISA Multiprocessor Platforms

Christopher S. Jelesnianski

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

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Antonio Barbalace
Changhee Jung

October 28, 2015
Blacksburg, Virginia

Keywords: Compilers, Heterogeneous Architecture, Performance Profiling, Runtime, Code Transformation, System Software

A Compiler Framework to Support and Exploit Heterogeneous Overlapping-ISA Multiprocessor Platforms

Christopher S. Jelesnianski

(ABSTRACT)

As the demand for ever increasingly powerful machines continues, new architectures are sought to be the next route of breaking past the brick wall that currently stagnates the performance growth of modern multi-core CPUs. Due to physical limitations, scaling single-core performance any further is no longer possible, giving rise to modern multi-cores. However, the brick wall is now limiting the scaling of general-purpose multi-cores. Heterogeneous-core CPUs have the potential to continue scaling by reducing power consumption through exploitation of specialized and simple cores within the same chip.

Heterogeneous-core CPUs join fundamentally different processors each which their own peculiar features, i.e., fast execution time, improved power efficiency, etc; enabling the building of versatile computing systems. To make heterogeneous platforms permeate the computer market, the next hurdle to overcome is the ability to provide a familiar programming model and environment such that developers do not have to focus on platform details. Nevertheless, heterogeneous platforms integrate processors with diverse characteristics and potentially a different Instruction Set Architecture (ISA), which exacerbate the complexity of the software. A brave few have begun to tread down the heterogeneous-ISA path, hoping to prove that this avenue will yield the next generation of super computers. However, many unforeseen obstacles have yet to be discovered.

With this new challenge comes the clear need for efficient, developer-friendly, adaptable system software to support the efforts of making heterogeneous-ISA the golden standard for future high-performance and general-purpose computing. To foster rapid development of this technology, it is imperative to put the proper tools into the hands of developers, such as application and architecture profiling engines, in order to realize the best heterogeneous-ISA platform possible with available technology. In addition, it would be in the best interest to create tools to be as "timeless" as possible to expose fundamental concepts industry could benefit from and adopt in future designs.

We demonstrate the feasibility of a compiler framework and runtime for an existing heterogeneous-ISA operating system (Popcorn Linux) for automatically scheduling compute blocks within an application on a given heterogeneous-ISA high-performance platform (in our case a platform built with Intel Xeon - Xeon Phi). With the introduced Profiler, Partitioner, and Runtime support, we prove to be able to automatically exploit the heterogeneity in an overlapping-ISA platform, being faster than native execution and other parallelism programming models.

Empirically evaluating our compiler framework, we show that application execution on Popcorn Linux can be up to 52% faster than the most performant native execution for Xeon or Xeon Phi. Using our compiler framework relieves the developer from manual scheduling and porting of applications, requiring only a single profiling run per application.

Acknowledgments

There are many people who helped me with this work right up until the defense itself, and for whom I am forever grateful. I would like to thank the following people specifically for helping me obtain my degree:

Dr. Binoy Ravindran, for opening the door to my academic research career by introducing me to the Popcorn Linux Project, and providing guidance in all my research along the way.

Dr. Changhee Jung, for graciously taking time out if his busy schedule to help me with this work and being on my committee.

Dr. Antonio Barbalace, for his positive encouragement throughout setbacks, his incredible knowledge of Linux forward, backward and insideout, his passion to share knowledge and inspire new ideas, and for the countless brainstorming sessions. I would like to thank him for taking time out of his busy schedule to help me with this work and being on my committee.

Dr. Alastair Murray, for his patience and spending time with me to begin my journey in the compiler domain.

The members of the System Software Research Group for their support, skill, and friendship, including Rob Lyerly, Dr. Sachin Hirve, Saif Ansary, Anthony Carno and Dr. Roberto Palmieri.

Finally, my parents Stanisław and Elzbieta, and my sister Joanna, for all of their love and support throughout the entire journey to get this far.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A future towards Heterogeneous Computing

Because of the so called "brick wall" dilemma [26], the computing industry is experiencing a paradigm shift. There exist three elements in this barrier, and in order to continue to produce increasing through-put gains we must overcome them in some shape or form: the power wall, the instruction level parallelism (ILP) wall, and the memory wall. Although trends are following Moore's Law, the following question remains: Will this trend continue? For now, the answer seems to be "yes". But Herb Sutter argues that like all exponential progressions, Moore's Law must end someday [33]. As a result, future performance gains are now expected to be achieved through non-traditional designs compared to the past. One of those non-traditional designs are heterogeneous systems.

This concept, though complex, is continuing to gain momentum in the hope of salvation to break through this "brick wall." But with this approach, researchers and developers alike will face the challenge of managing ever-increasing complexity of systems as additional architectures and devices are thrown into the mix.

While many paradigms in the form of language extensions, various programming models, and frameworks have been created to aid developers writing software for a given heterogeneous system, few are able to truly give the developer an upper hand without also coming with an unavoidable trade-off. Some are platform specific (e.g., CUDA for Nvidia GPUs), while others are portable but at a low level of abstraction, as with OpenCL and Message Passing Interface (MPI). Most, if not all of these paradigms require explicit data-transfer, memory management, kernel launch to be coded as well as re-optimization and adaptation to achieve decent performance when migrating to a different device. Given the dramatic difference in complexity and additional effort needed writing applications for a heterogeneous platform versus writing for a traditional SMP homogeneous-ISA machine, mastering heterogeneity for the next generation of high performance computing (HPC), embedded, and all the other computing domains will be no small task.

As far back as 1994, Von Bank et al. [36] is one of the first publications to breathe life into heterogeneous computing as the next step forward in high-performance computing, putting forth a theoretical model as well as including a formal definition of heterogeneous process migration. They assert that the language system (i.e., compiler, assembler, and linker, etc.) can be designed to ensure equivalence points are at a user-determined granularity. The end goal of the work presented herein goes along this path, implementing a solution as a compiler framework providing developers with a means of transforming their applications to be compatible with an experimental heterogeneous platform by taking advantage of equivalence points, while, at the same time requiring minimal user intervention.

1.2 Motivations

The problem of running the same application on different platforms has a long history of investigation, dating back to the issue of code portability between different architectures. With the advent of high-level languages, like C, the main issue was to compile the same source code on different platforms. Once this was solved, the compiler community focused on the problem of running the same executable on different architectures, which interpreted languages (IR, or byte code) like Java or C# solved (static or dynamic binary translation can also achieve the same result). These approaches solve the problem of writing the code once and running the application on different architectures. However, these solutions assume that the application will always be executed on a given architecture, and will not be migrated to, and executed on another architecture with a different Instruction Set Architecture (ISA).

It should be noted that thus far, the term "heterogeneous computing" has generally been the way to describe machines that contain specialized hardware, dedicated accelerators, or even cores with inherently different characteristics, in addition to traditional general-purpose CPUs within the same machine. Up to this point, a large assumption has been made that all cores in the heterogeneous machine share the same ISA (a homogeneous-ISA platform). Developers can offload specific code blocks onto the specialized device thus allowing the computation to complete faster compared to letting it execute on the general-purpose CPU. This definition has quickly gone stale with today's advances in research and available commodity hardware. Technology is now at a point where a daring few are pushing the envelope of heterogeneity; specialized devices such as FPGAs and NICs (like the Tilera) have evolved to become fully general purpose devices and thus even OS-capable! We define "OS-capable" as being able to run a fully-fledged operating system and have the ability to control scheduling and memory compared to traditional, non-programmable I/O devices [24].

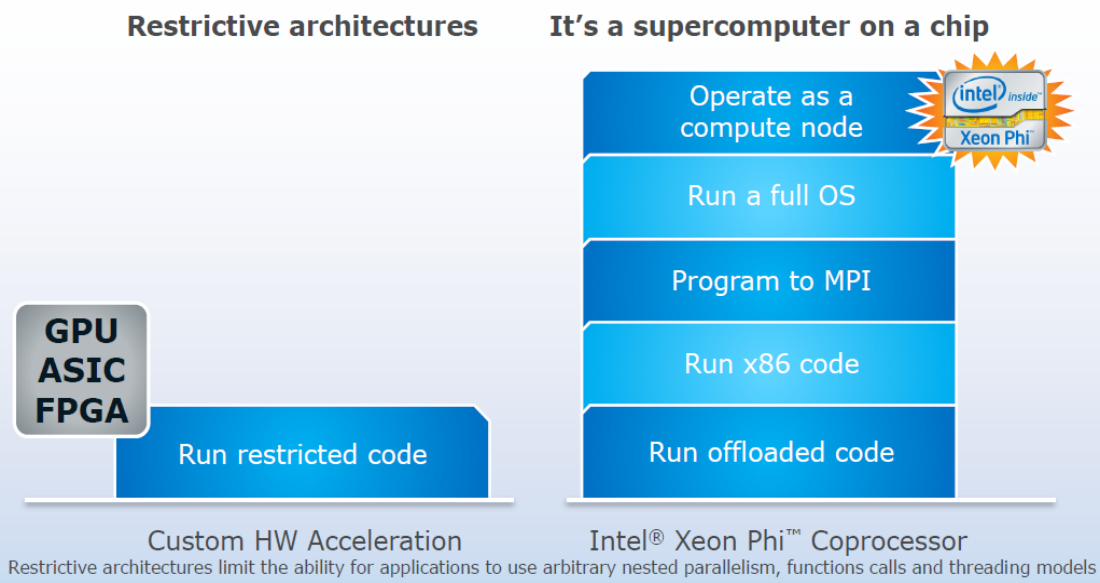

Figure 1.1: Intel Xeon Phi Coprocessor compared to traditional methods. https://www.lrz.de/services/compute/courses/x_lecturenotes/MIC_GPU_Workshop/Intel-01-The-Intel-Many-Core-Architecture.pdf, Used under fair use, 2015.

One example of such a recently released "OS-capable" device is Intel's® Xeon Phi Coprocessor [16]. Figure 1.1 showcases the competitive edge Xeon Phi has over current accelerator hardware. An honourable mention should also be given to AMD's Project Skybridge [1]; though this CPU has been cancelled for different reasons [38], it is a prime example of where heterogeneous computing could be tomorrow. Just as cellphones have consolidated many everyday-devices (e.g., alarm clock, calculator, watch, etc.) all into one platform, we envision heterogeneous platforms to evolve similarly albeit in the form of consolidating various platform configurations into one super versatile machine.

Taking advantage of parallelism and heterogeneity can result in significant performance gains but require substantial developer effort and deep knowledge of the underlying framework being utilized to achieve maximum performance. It also doesn't help that each heterogeneous-ISA hardware usually has it's own preferred or even required programming paradigm when developing code for these various hardware, implying that developers must spend additional time if they want to port between the various options available. A few examples of these semantically diverse programming paradigms include CUDA for Nvidia GPUs, OpenCL, OpenACC, and OpenMP 4.0. Any experience with these is fairly tedious if not unpleasant from the developer point of view; memory consistency between host and devices must be managed manually, taking away developers valuable time and attention away from focus on desired logical flow of an application and additionally spawning a new place for software bugs to manifest and hide, a potential nightmare for any unsuspecting developer.

Though the number of vendors starting to pursue general purpose OS-capable devices is increasing,

no one has stepped up to the plate to propose a standard for how applications should be structured, compiled, and executed on a given heterogeneous platform despite the respective vendors themselves. OpenCL is an attempt to solve this gap, but even this paradigm contains handicaps inherent with a large amount of boilerplate code that the developer is forced to use to effectively exploit the architecture. Note that OpenCL is only compatible with CPU + GPU and Xeon + Xeon Phi platforms, any other configuration for moving an application around is not allowed (e.g., rescheduling an application directly from one GPU to another GPU) since OpenCL is based on the "offloading" programming model. Ideally, the developer should only have to focus on creating the desired logic of his application, assuming a symmetric multiprocessing (SMP) environment while a compiler tool-chain and runtime support/library would manage code transformation to conform with the target heterogeneous platform as well as extracting maximum performance.

We thus seek to address the missing link developers have been waiting for to make heterogeneous platforms viable and easy to work with. At this point, we do not seek to have support for the "upgrading-of" legacy binaries (a model assuming no source is available either because the application is old or is restricted containing proprietary elements). Instead, we assume that the developer is always starting from source and interested in working with a heterogeneous platform that could potentially have up to *N* different devices and/or architectures. Therefore we seek to make the work contained herein be as generic as possible to support any combination of heterogeneous devices in the future. It is structured such that porting to a new device or architecture is only needed once per device as opposed to once per application. In addition, to gain more traction within the compiler community, we propose to develop the compiler framework by extending common software tools included in Linux, which is synergistic with the proposed Popcorn Linux OS.

1.3 Research Contributions

Though the possible directions to pursue with heterogeneous systems are limitless (e.g., performance, power, energy-savings, etc.) this work focuses on the performance and throughput advantage gained by leveraging a heterogeneous-ISA platform compared to traditional approaches. In this thesis the following contributions are presented for the Popcorn project:

1. A heterogeneous-ISA application Profiler is designed and implemented for Popcorn Linux's Xeon - Xeon Phi platform. Written using LLVM [21], it automatically analyzes memory access patterns and searches for highly parallel regions in an input application to provide the best partitioning in regards to performance.

2. A heterogeneous-ISA application partitioner is also designed and implemented to compliment the application Profiler and perform the needed code transformations to make an application have the necessary components for heterogeneous-ISA migration within Popcorn Linux. The partitioner is implemented in C using the ROSE Compiler Framework [27].

3. A compiler framework specific to the Xeon - Xeon Phi Popcorn Linux platform is designed and implemented to create compatible FAT heterogeneous-ISA binaries. Underlying heterogeneous-ISA support for commonly needed application librarys (such as the Math Library, *libm* and the C Library, *libc*) are given clever modifications to resolve conflicts inherent during link-time for a heterogeneous-ISA binary. In addition, a linker tool is implemented as part of the compiler framework to realize the final binary.

4. A comprehensive evaluation of the performance gained on the Xeon - Xeon Phi prototype platform compared to current methodologies and mature models aimed at high performance computing (HPC) is provided. This evaluation showcases what improvements are acheived using the proposed compiler framework.

1.4 Thesis Organization

This thesis is organized as follows:

- Chapter 2 presents background on the Popcorn Linux Operating System for the support of heterogeneous platforms. Note that this chapter is an excerpt taken from of a previous work I am a co-author of. It expands in detail for parts relevant to the compiler framework not mentioned before because of space constraints. In addition, it discusses the heterogeneous-ISA platforms chosen for this work.

- Chapter 3 presents related work in the area of heterogeneous compilers and runtimes.

- Chapter 4 presents the first segment of the Xeon - Xeon Phi compiler framework, specifically, the application Profiler. Using the LLVM framework, we are able to profile applications for advantageous partitionings on the Xeon - Xeon Phi system.

- Chapter 5 presents the second half of the Xeon - Xeon Phi compiler framework. This chapter describes the ROSE source-to-source transformation partitioner tool. It dicusses the needed changes to make an application *migration-ready*. It also presents the heterogeneous compiler/linker needed to create heterogeneous FAT binaries for the Xeon - Xeon Phi platform.

- Chapter 6 presents compiler runtime adaptations needed for heterogeneous execution through the manipulation of various libraries. This chapter describes the modifications in both the libraries and the compiler to solve compatibility issues for our two experimental platforms.

- Chapter 7 presents a detailed evaluation of the work discribed in this work. We compare this work to several other standard approaches currently used in heterogeneous platforms to validate our methodology. We also give an analysis of the overheads associated with the proposed methodology. It also presents the specifications of the platform used for those wishing to reproduce our results.

- Chapter 8 presents conclusions and wraps up the work. Future work is discussed here.

Chapter 2

Background

2.1 Popcorn Linux: A beast of the Heterogeneous

This Chapter contains select material from a paper in which I am a co-author [4] in order to inform the reader of underlying assumptions for this thesis.

To properly set the stage for the work in this thesis, it is necessary to describe what exactly the work described hereafter is actually supporting. For this work to have a meaning it is needed to acknowledge that the landscape of heterogeneous computing is evolving as depicted in Figure 2.1.

Proliferation of new computing hardware platforms that support increasing number of cores, as well as increasing instruction set architecture (ISA) heterogeneity, creates the opportunity for system software developers to question existing software architecture. The trend of specialization in new devices is becoming progressively *less* specialized and *more* general-purpose!

Figure 2.1: Evolution of device specialization over time.

Popcorn Linux is a possible answer to the future of heterogeneous computing. Popcorn Linux is a replicated-kernel Linux based operating system that bridges the gap in heterogeneous-ISA platforms for the application developer. Popcorn provides inter-architecture thread migration and Distributed Virtual Shared Memory (DVSM) across the entire heterogeneous-ISA platform. Popcorn aims to make applications believe, similar to a distributed OS, that multiple OS instances are infact a single computational entity, forming a "Single-System-Image". This illusion of a single-system-image and thus a single execution environment is a very beneficial simplification for both the developer and application. The Popcorn architecture aims to specifically provide the following three principles:

1. Transparency: The user should not see any kernel/processor boundaries, but a single system on which applications can run everywhere and use all the available resources on the platform. Developers life is simplified and can focus on an application's logic while assuming an SMP system; the compiler framework (the focal point of this thesis) takes care of application partitioning.

2. Load Sharing: Produced application executable should be able to run on any architecture by containing code for each architecture present in the heterogeneous-ISA platform. The operating system would handle migration, while the compiler framework would provide a compatible binary. Threads for an application should be able to migrate intra- and inter-architecture partition without being limited to a specific architecture partition.

3. Exploiting Asymmetries: Although sometimes it may be desirable to mask the asymmetries, the OS should still provide and expose architectural distinctions such that an application is able to exploit the architectural asymmetries present on the heterogeneous-ISA platform for maximum performance.

These principles lead to the Popcorn architecture in Figure 2.2 and Figure 2.3.

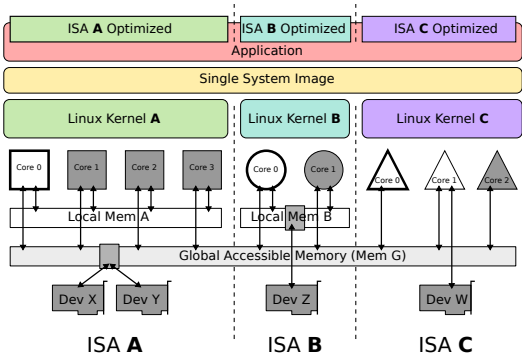

Figure 2.2: Heterogeneous-ISA generic hardware model, and Popcorn software layout.

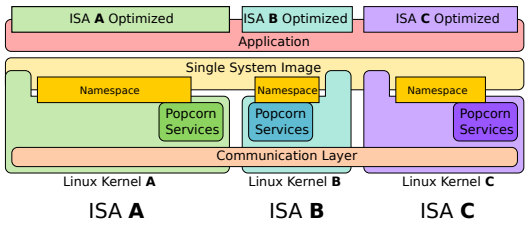

Figure 2.3: Popcorn kernel- and user-level software layout.

Hardware Model

The Popcorn Linux software architecture (along with the compiler framework described herein) is designed to work with a generic hardware platform, depicted in Figure 2.2, which attempts to abstract current and emerging hardware. Popcorn assumes a hardware model where processors of the same ISA are grouped together, and different ISA processors share access to a global, eventually consistent, memory (*Mem G* in Figure 2.2). Computational units of the ISA group may have exclusive access to a memory area (*Mem A* and *Mem B*) as well as across ISA groups, the same memory 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.

Software Layout

Figure 2.2 shows Popcorn's software architecture and how it is layered on top of a generic hardware model and the single-system-image. It illustrates a single application compiled with the Popcorn compiler framework (described herein) that is running on the Popcorn Linux operating system. The application is multi-threaded, and different threads potentially run on different kernels (and therefore on different ISA processors).

Operating System Architecture

The operating system consists of multiple kernel instances, one is compiled for each different ISA present in the heterogeneous-ISA platform. Therefore the kernel code must be portable to any ISA that could be included in the platform. Kernels interact to provide applications with the illusion of a single operating system, a "single-system-image" amongst the different kernels in the platform as if it were an ordinary SMP machine, as shown in Figure 2.2. 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 DVSM, so that multi-threaded applications, written for shared memory architectures can continue to function.

A communication layer glues kernels together and provides basic data conversion between ISAs, as in Figure 2.3. 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 2.3, 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.

2.2 Architectures & Connectivity

In theory, a heterogeneous platform can exist in a wide variety of different configurations ranging from same-ISA heterogeneous cores to a completely disjoint-ISA platform. Each configuration with it's own set of benefits and drawbacks; even if some configurations are theoretically possible, they physically are not simply because the hardware to provide such a configuration and the connectivity neeeded between devices does not exist. Since this work seeks to demonstrate the pervasiveness of heterogeneous platforms, we implement our solution on commodity hardware in order to show both the possibilities with heterogeneous hardware and the viability the presented work. Figure 2.4 summarizes possible ISA platform setups.

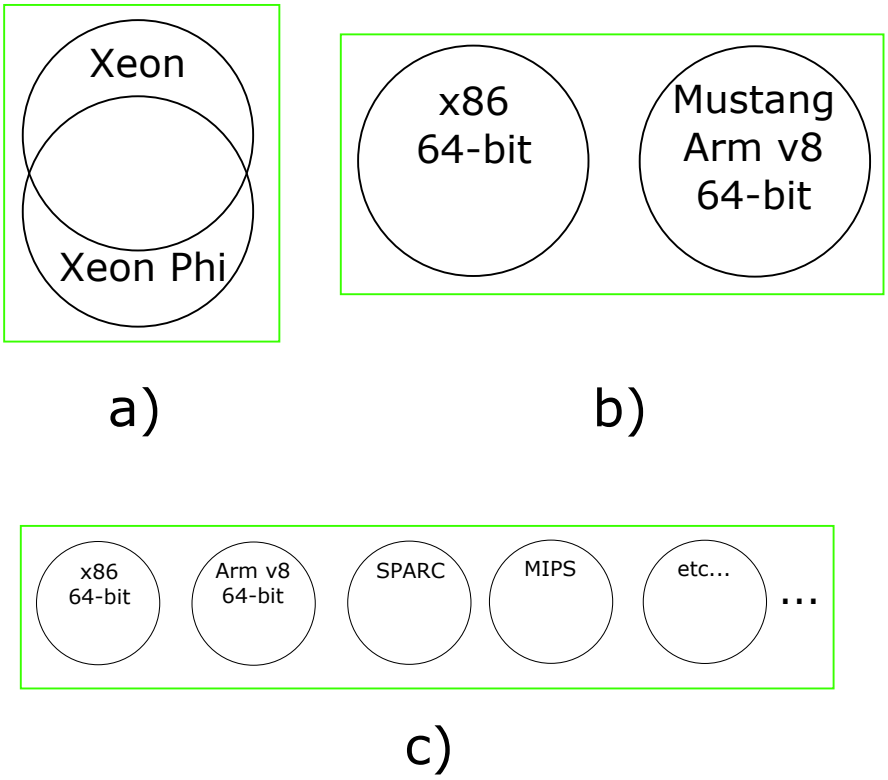

Figure 2.4: Overlap of instruction set architectures (ISA) for heterogeneous platforms. In (a) the two processors share a common ISA, but each has special purpose features (such as vector masks in the case of Xeon Phi coprocessor). In (b) the two processors have their own ISA and are completely disjoint; no features are shared. (c) depicts the end goal of what kind of heterogeneous platforms future compiler tool-chains will need to provide for.

2.2.1 Intel's Xeon + Xeon Phi

Our first experimental platform consists of two Intel processors, the Xeon E5, an 8-core 64-bit processor, as the host platform CPU connected with a Xeon Phi 3120A, a 57-core 4-way hyper threaded coprocessor, over PCIe. Throughout this work the term host platform, host architecture, and host kernel are used interchangeably; these terms all refer to the Xeon x86 architecture for the test platform used in this work. At the same time this work could be extended, in which case, using hardware available today, the host architecture would refer to the main anchoring architecture which provisions the rest of the devices/accelerators that have different architectures. Note that this does not imply a master worker relationship between devices using the Popcorn Linux OS. In the future, we predict that the terms host/non-host will fade as systems become increasingly heterogeneous.

This platform, when setup with Popcorn Linux, is one which exhibits an overlapping-ISA configuration (see Figure 2.4 (a)). To go into greater detail, both the Intel Xeon and the Intel Xeon Phi

coprocessor make use of the base x86 64-bit ISA, however, both processors have their own implementation for performing floating point arithmetic. The Xeon Phi coprocessor features vector masks along with vector instructions which use a special extra source, known as the write-mask, sourced from a set of 8 registers called vector mask registers in order to do highly parallel computation [15]. The Intel Xeon does not have these registers or instructions and instead uses owns means for Floating Point Unit (FPU) computation. Thus even the case of a simpler heterogeneous-ISA model such as an overlapping-ISA platform has a number of implications. Things such as computational implementations and other general functionality usually supported by libraries must be taken into consideration. These implications will be discussed in later sections as well as how the compiler tool-chain accounts for them.

Since the two processors are connected over PCIe, this greatly simplifies one of the biggest concerns in heterogeneous computing, namely, moving data. At the same time the PCIe bus can be identified as the largest bottleneck of the heterogeneous system during the process of migrating data to where it is needed. However, messaging and synchronization of data across the Xeon and Xeon Phi are beyond what the compiler must provide to enable migration therefore it is beyond the scope of this work.

Chapter 3

Related Work

In conjunction with the goal of making Popcorn Linux a versatile operating system for future heterogeneous systems, able to support any combination of *N*-architectures within the same platform, this work is somewhat aimed to be broad on purpose for that reason. At the same time this work retains focus on key components that are indifferent to the number of distinct architectures/devices present on the target heterogeneous platform. Technically, the age of heterogeneous computing is already here. Single-ISA heterogeneous CMPs have been introduced into the market such as ARM's big.LITTLE processor [2] and Nvidia's Kal-El processor [25]. An even more widely known heterogeneous CMP is Intel's integrated graphics CPU architecture, Sandy Bridge[41]. However, none of these architectures allow migration between core types to occur at arbitrary places in the code.

Since the work in this thesis interacts with many different aspects of system software, research related to this work falls into several categories. The first section discusses profiling and quantifying application performance characteristics (i.e., file I/O, loops, memory access patterns, and compute kernels) with respect to hardware characteristics (i.e., number of cores, memory subsystem, device locality, functional units in processors, etc.) available for which an application will be run on.

The next section pertains to compiler toolchains/frameworks developed for similar systems. Since heterogeneous-ISA platform technology (not to be confused with "heterogeneous systems" commonly consisting of only CPUs and GPUs) is for the most part still in its infancy, there do not exist any direct competitors for comparison of compiler toolchains/frameworks in this respect – Popcorn Linux is the first multiple kernel OS for heterogeneous platforms to be successfully implemented in hardware while most other works rely strictly on emulation.

Following this, the next section explores what has been done in terms of providing underlying support for heterogeneous systems as a whole. This includes various approaches to solve functional heterogeneity in multi-core architectures.

Finally, the last section discusses approaches that tackle the challenge of heterogeneity on the system scale; several works resolve this challenge by building operating systems from the ground

up to introduce methodologies which allow the system to cooperate and benefit as a whole!

3.1 Application Characterization/Profiling

Several previous works give motivating evidence and hints that heterogeneous-ISA platforms could out-perform traditional homogeneous SMP platforms by evaluating their methodologies on systems that are heterogeneous of varying degree.

Even on a homogeneous-ISA, heterogeneous-core platform, it is not sufficient to arbitrarily migrate tasks between processors in an attempt to make up for performance bottlenecks one processor exhibits but the other processor does not. Inefficient partitioning/mapping can easily result in application slowdown and ruin performance.

Saez et al. [31] develop two versions of an Heterogeneity-Aware Signature-Supported (HASS) scheduling algorithm. They are HASS-S and HASS-D, which handle analysis and scheduling statically (HASS-S), compared to handling analysis and scheduling dynamically (HASS-D), respectively, during runtime for an asymmetric multi-core platform. Specifically their algorithm is based upon the idea of "architectural signatures" of an application. They explain that an architectural signature is a compact summary of architectural properties of an application and may contain information such as memory access patterns and instruction-level parallelism (ILP) along with characteristics of hardware present within the platform such as variation in clock speed. With these two parameters the scheduler schedules threads onto a given core based upon its estimated performance. They also introduce a novel concept (denoted as "optimistic rebinding") where thread assignment is further optimized by "finding" a good partner thread to swap cores to execute on instead of traditional thread assignment. Evaluating both HASS algorithms showed that they outperformed an IPC-Driven algorithm as well as out-performing a heterogeneity-agnostic scheduler by as much as 12.5%.

Our work follows a similar path but goes to the next level of comparing compute kernels. Moving across cores versus moving across heterogeneous architectures involve very different factors that impact total cost. Even though our algorithm only performs static analysis for now, it is critical to have an accurate guage of the migration cost associated with moving across architectures and whether migrating can offset this upfront cost. Our work puts extra effort to accurately derive this cost while Saez et al. don't even have to consider this cost.

Going a step closer to real heterogeneous systems, Kofler et al. [20] work with a heterogeneous CPU and GPU platform for which the authors propose and implement an automatic, problem-size sensitive compiler-runtime framework as part of Insieme, which can auto-generate multi-device OpenCL code and optimized task partitioning for a given application. They argue that the performance capability of individual devices can vary significantly across different applications and problem sizes on a heterogeneous platform. The framework consists of two phases declared as training and deployment phases. During the first phase, the training phase, it utilizes machine

learning via Artificial Neural Networks (ANN) to build a task partitioning prediction model based on static program features extracted from the intermediate-representation (IR) of the application. The framework has the benefit of being universal in the sense that any previously unseen target architecture can be supported by generating a new model for it in the training phase. The source-to-source transformation is performed offline via the Insieme Compiler [10] using a generated problem size sensitive model with the help of machine learning to determine the best task partitioning for the given application. To test the viability of their approach, the authors used a selection of 23 programs from various sources and varied input problem sizes in order to create their training patterns. Once the training phase completed, running two simple benchmarks resulted in 22% and 25% performance improvement compared to an execution of the benchmarks on a single CPU or a single GPU respectively using the Insieme Runtime System responsible for the execution and scheduling of the generated programs. Interestingly enough, over 25% of their 355 training patterns deliver best performance when using a hybrid task partitioning.

Compared to Popcorn Linux, Kofler et al.'s framework does not allow for the gathering and merging of writes from different devices; Popcorn solves this using a replicated kernel with Distributed Virtual Shared Memory (DVSM).

On the other hand, the Merge framework proposed by Linderman et al. takes a different approach and proposes a general purpose programming model for heterogeneous multi-core systems [23]. Their framework replaces current ad-hoc approaches to parallel computing on heterogeneous platforms with a rigorous, library based methodology that can automatically distribute computation across heterogeneous cores to achieve increased energy efficiency and performance. Rather than writing and/or compiling an application for the heterogeneous target at hand, the programmer only needs to express computations using architecture-independent, high-level language extensions based on the map-reduce pattern. The Merge framework consists of three components, namely, a high-level parallel programming language based on map-reduce patterns, a predicate-based library system for managing and invoking function variants for different architectures, and a compiler and runtime which implement the map-reduce paradigm by dynamically selecting the best available function variant. As can deduced, this framework's backbone is reliant on map-reduce that then produce different variants of compute kernel's in a given application. Though porting an application such as Blackscholes can take up to four days to be compliant with their framework, the authors were able to achieve 3.7x-8.5x speed ups relative to the reference implementation on a heterogeneous CPU-GPU platform.

While the Merge framework is a library-based approach, our Profiler/Partitioner compiler framework uses the direct approach mapping the application directly onto the predetermined best fit architecture. The tradeoff is that the library-based compiler will have only minimal information about any given function-intrinsic, and thus will miss some of the optimizations exploited our direct approach takes.

3.2 Heterogeneous Runtime Support/Migration Techniques

A significant amount of engineering goes into designing adequate and efficient migration techniques for heterogeneous systems – moving applications across cores and ISA boundaries is where most runtime overhead of an application originates for heterogeneous systems!

Reddy et al. [28] presented a comprehensive study of heterogeneous architectures and the challenges inherent with supporting and running applications on such platforms. The authors discuss a variety of issues including process migration, feature enumeration, and virtual memory & paging that would have to be considered for a true heterogeneous-aware OS. They then analyze several methods that can be used to bridge the functional issues that arise in heterogeneous systems. The authors propose three design models (Restricted Model, Hybrid Model, Unified Model) and discuss the benefits and drawbacks of each proposed model. The findings in this work both reveal the numerous hurdles that Popcorn Linux needed to overcome to be what it is today and emphasize the need of a symbiotic evolution of hardware and software for heterogeneous systems to fulfill their potential as the next generation in high performance computing.

The two experimental platforms used in Reddy et al's work are minimally heterogeneous, the authors citing that their first experimental platform consists of processors from different microarchitecture families (but still using the same underlying ISA) while the second platform uses identical processors but with certain features disabled to "emulate" heterogeneity. Thus this authors work does not address heterogeneous-ISA platforms. On the other hand, our work is tested and verified on a platform with processors that have different ISAs.

Dynamic Binary Rewriting for Migration (DBRM), presented by Georgakoudis et al. [13], is a low-level thread migration methodology for the case of a shared-ISA heterogeneous platform where some cores are performance enhanced (PE) (i.e., those cores extend the baseline ISA with instructions that accelerate performance-critical operations). The application is compiled for the underlying ISA of the platform and DBRM occurs during runtime, filling in the gaps during execution if a thread is rescheduled onto a PE core by rewriting assembly to reflect compatible instructions for the core the thread has been assigned to. Note that the authors limit the scope of their work to statically compiled applications only and are not able to handle dynamically loaded libraries. The authors evaluate their migration methodology using the SPEC CPU2006 and Rodina benchmark suites, showing that a DBRM-enabled scheduler can improve performance by 2%-30% compared to an oracle scheduler that statically matches threads to cores using a-priori knowledge. They further investigate their methodology to reveal a breakdown of its overheads. It is no surprise from reading similar works that binary analysis and assembly translations accounts for the majority of the overhead.

Our approach forgoes the expensive overhead associated with performing DBRM during the runtime of an application and instead utilize a FAT binary to contain all needed implementations of a given candidate compute kernel. We argue that storage is cheap and outweighs the overhead of performing DBRM. In addition, our approach does not have to worry about a core missing a PE instruction implementation as our tool compiles the compute kernel natively for each architecture

present on the target heterogeneous-ISA platform.

DeVuyst et al. [8] present their approach for execution migration on heterogeneous-ISA CMPs utilizing Dynamic Binary Translation (DBT) and stack transformation to bridge the gap between architectures (in their use case they use a small, low-power ARM core and a large, high-performance MIPS core). Their approach is achieved in four steps: rescheduling the process on another core, changing page table mappings to facilitate access to the code for the migrated-to core, performing binary translation until a program transformation point (in their work this is at any function call site), and transforming the program stack for execution on the new architecture. The first two steps are common operating system responsibilities; the last two are unique to heterogeneous-ISA migration and DeVuyst's attempt to bridge the gap between architectures in the most memory-coherent manner possible. Memory consistency between architectures was a major topic in their work, and most of their modifications to their compiler tool reflect this as they present compiler techniques to minimize the amount of program state kept in ISA-specific form to enable fast migration. Even so these modifications came at a price of having to disable some optimizations usually desired when compiling (in their case GCC's RTL optimizations had to be disabled). In addition, the authors have a runtime component (called the Stack Transformer) which facilitates translating the current stack for native execution once a transformation point is reached after performing DBT. Improving their approach by inserting more transformation points to conclude DBT as soon as possible (the most expensive component of their migration approach) the authors were able to achieve a total loss of performance of under 5% compared to native execution, regardless of migration frequency between architectures.

It should be noted that their evaluation was completely carried out in simulation using the M5 Processor Simulator whereas our work has been implemented in real hardware better showing the viability of heterogeneous-ISA platforms even if a bit more engineering work is required to bring concepts to a reality. The difference between the authors ARM - MIPS platform and our Xeon - Xeon Phi platform is that DeVuyst's platform requires state transformation and DBT in order to successfully migrate a task between architectures, our approach does not need state transformation as we take advantage of equivalence points to minimize the amount of effort required by the OS to migrate a task thus lowering the overheads associated with migration.

Another work actually builds upon DeVuyst et al.'s work, but instead of focusing on the migration mechanism, it seeks to identify the best heterogeneous designs for a given workload for varying power budgets and performance. Venkat et al. [35] considers several axes of ISA diversity including: code density, dynamic instruction count, register pressure, and floating-point/SIMD support and explains the benefits and drawbacks of the three target ISA's chosen a priori – Thumb, Alpha, and x86-64. Similar to Popcorn, the authors opt to use a unified address space and utilize FAT binaries with multiple target-specific code sections and common target-independent data sections. Their compilation strategy leverage's LLVM common intermediate representation (LLVM bytecode) to enforce target-independent type legalization which ensures a consistent view of global data when compiling the multi-ISA FAT binary. Interestingly, the authors also generate a set of transforms that can be applied by the runtime environment at the time of migration during compile-time. These transforms are routines that reconstruct the target-specific program state of a basic

block (live registers and stack objects). The authors create a modified version of DeVuyst et al.'s migration mechanism for the purpose of utilizing DBT and stack transformation with the help of the generated transforms mentioned earlier. Using the SPEC CPU2006 benchmarks, the authors were able to achieve an average energy savings of 21.5% and an average reduction of 27.8% in the Energy Delay Product (EDP) using Heterogeneous-ISA CMPs compared to single-ISA heterogeneous CMPs. In addition, they observe an additional speedup of 11.2% due to migration on a heterogeneous-ISA CMP at phase boundaries in contrast to a 4.6% speedup due to migration on a single-ISA heterogeneous CMP.

Note that similar to DeVuyst, this work is based upon simulation of a heterogeneous platform – it is not implemented in hardware as Popcorn Linux.

Kessler et al. [19] elaborates on three approaches to obtaining portability and increased levels of abstraction for the programming of heterogeneous multicore systems. The authors first describe a library approach combining the SkePU skeleton programming library and leverage the StarPU heterogeneous runtime system. Skeletons are generic components derived from higher-order functions that describe common computational structures and are mapped to one or more StarPU tasks, generating task-parallelism for the runtime system. Their second approach, explores the use of Codeplay's Offload C++ language and compiler traditionally used by game developers. By hiding implementation details in generic C++ classes, it enables complex C++ accelerator code to be embedded inside target applications without major code changes. The third approach builds on the PEPPHER component model and transformation system to encapsulate user-provided code into different implementation variants along with XML meta-data for different architectural components of a heterogeneous manycore system. During runtime the PEPPHER framework can generate a performance model using historical performance data to choose and schedule the best variant possible during runtime. Using OpenCV image processing code as their benchmark, the authors were able to reduce execution time by a factor of up to 3.14 using the third approach with PEPPHER compared to an implementation using Intel Threading Building Blocks (TBB). They also used the ROSE compiler framework to implement their source-to-source compiler.

While PEPPHER is similar in functionality to our compiler framework performing source-to-source transformation to prepare code for migration, their approach as a whole does not treat devices within the heterogeneous platform as general purpose but only as accelerators to perform offloading. Assuming the offloading model in heterogeneous platforms severly limits what can be acheived with them. By Popcorn Linux being a replicated-kernel OS, opportunities such as load balancing and energy savings open up to be explored and exploited.

3.3 Other Heterogeneous Compilers and Runtimes

Dandelion [30], presented by Rossbach et al., is a recent approach trying to unify the wide variety of programming techniques usually needed for different devices (such as CPUs, GPUs, FPGAs, and even the cloud) typically part of a heterogeneous system. The authors utilize the .NET Lan-

guage INtegrated Query (LINQ), a general language integration framework, as the backbone of their compiler and runtime system. This allows them to support a wide variety of data-parallel operators which could be run on a multitude of devices for the developer. Dandelion's runtime uses statically generated "execution plans", a data flow graph representation of the application, from its compiler to help automatically and efficiently map application computational blocks onto their heterogeneous system for the best performance. Their mapping scheme consists of 3 layers: the cluster execution engine, the machine execution engine, and the GPU execution engine; each layer is of finer granularity orchestrating the assignment of vertices from the execution plan onto components making up that layer (e.g. machines within the cluster layer, processors within the machine layer, cores of a given CPU or GPU within the GPU layer). In their evaluation, the authors were able to achieve up 6.4x speed up using the Dandelion system compared to CPU parallelism only on a single machine. They also empirically affirm that memory management and transfer overheads are one of the most impacting costs in heterogeneous systems.

Though Dandelion is able to cross-compile for a variety of devices like GPUs and FPGAs to take advantage of a heterogeneous platform, it contains an offload model mentality. It depends on low-level GPU runtimes such as CUDA and therefore has limited support for dynamic memory allocation. In addition, Dandelion assumes that user-defined functions must be side-effect free. Popcorn Linux along with the compiler framework presented in this work provides it's own runtime mechanisms that collectively side-step the limitations that Dandelion has.

A project dubbed Liquid Metal [3] gives its take on how the headaches linked with heterogeneous systems could be solved via the use of a Java Virtual Machine (JVM) and a new device agnostic programming language developed by Auerbach et al. called Lime. Lime is a Java-compatible object-oriented language with new features such as: ability to define a unit of migration between CPUs and accelerators, statically compile efficient code for accelerators in a given heterogeneous system, and orchestrate data flow across the system. Limes strong points in design consist of strong isolation and abstract parallelism, therefore decreasing the learning curve needed to be successful with Lime. Using Lime with the Liquid Metal compiler and runtime system enables seamless co-execution of the resultant programs on CPUs and accelerators that include GPUs and FPGAs. Liquid Metal features dynamic runtime-partitioning, since the result of the Liquid Metal compiler is a collection of artifacts (created from the applications computational kernels) for the various architectures on the heterogeneous platform. As a result, the runtime can choose from a large number of functionally-equivalent configurations for co-execution for the compute kernels. This allows Liquid Metal to advantageously adapt to changes in the platform or program workloads, availability of resources, and other fine-grained dynamic features. Execution is handled via any modern JVM and runtime-partitioning occurs using a task graph where elements are the compiled artifacts for each compute node in the program. The authors even put in additional effort to design and implement an interactive development environment (IDE) to aid with using Lime.

Though Liquid Metal and Lime provide a new way of programming heterogeneous platforms, it still leaves the question of very large/complex and/or legacy applications should be handled. It is a given fact that industry very rarely spends effort in porting applications unless a new proven standard has been ushered in [14]. Popcorn Linux along with its compiler framework takes care

of these common concerns, providing all the necessary tools to prepare any C application to be compatible with Popcorn Linux and the given heterogeneous-ISA platform. All that's needed is the original source code.

3.4 Heterogeneous Operating Systems

Several other works propose operating systems and compiler techniques to realize viable migration within heterogeneous platforms to achieve high performance as well as flexibility in load-balancing.

Helios [24], an operating system proposed by Nightingale et al., is composed of multiple "satellite kernels", where a microkernel is instantiated for each CPU present on the heterogeneous platform that has a different ISA or performance characteristics. Together with each microkernel, Helios forms the notion of a "single-system-image" by exporting a single, unified namespace. Helios implements both local message passing (LMP) and remote message passing (RMP) to provide transparent, unified interprocess communication, independent of where a process or service executes. The authors highlight a common theme within heterogeneous computing – the placement of applications can have a drastic impact on performance. Helios attempts to solve migration costs by exporting an affinity metric that is expressed over message-passing channels: a positive affinity indicates to the OS that two components (e.g. process and a service or process and another currently executing process) will benefit from fast message passing and therefore should execute on the same satellite kernel and the opposite for a negative affinity. The authors explicitly state that Helios does not strive to optimally map processes to a graph of active processes on a system, but instead base scheduling on their affinity metric with respect to the location of other processes with which it wishes to communicate. Helios simplifies application deployment on a heterogeneous platform by implementing its own two-phase compilation strategy for applications to be run on its OS. The first step transforms source into a common intermediate language (CIL), followed by compiling for each ISA present using a derivative of the Marmot [12] compiler called Bartok. The authors argue that packaging applications using CIL has two advantages over FAT binaries. Developers using FAT binaries must choose ahead of time which architectures to support as a result as more instruction sets are supported the size of the binary will grow in size. In addition, CIL already contains infrastructure for efficiently supporting multiple versions of a method. This in turn allows an application to be portable, taking advantage of device-specific features if they are present, and still functioning if this device is missing by falling back to a another implementation. Running a SAT solver benchmark using their affinity metric and being automatically offloaded resulted in the application running 28% faster than when sharing the CPU with a disk indexer benchmark.

Popcorn Linux and Helios are similar in several regards. Both Popcorn Linux and Helios use replicated kernels to form their respective OSs. However, kernels in Popcorn are peers, while in Helios they are structured in a master/worker relationship. Moreover, Helios utilizes explicit message passing to enable inter-processor-communication (IPC) whereas Popcorn Linux is shared memory

POSIX compliant. All Helios applications must be written in Sing# in order to be compiled to CIL; our compiler framework can prepare any C application into a compatible binary for a target heterogeneous-ISA platform.

Barrelfish [5], as proposed by Baumann et al., is a new future thinking operating system based upon the authors OS model coined as "the multikernel." Recognizing the trend in rising core counts and increasing hardware diversity leads the authors to identify three design principles for their multikernel model. These principles are: making all inter-core communication explicit, make OS structure hardware neutral, and view state as replicated instead of shared. The authors give compelling reasoning that in order to fully exploit the heterogeneity present in tomorrow's platforms, the traditional OS model needs to be rewritten. They observe that platforms are increasingly resembling networked systems, and should be treated as such. They argue that explicit message passing can outperform shared memory for updating shared state. By making the OS structure hardware neutral, it leaves minimal hardware specific aspects needed to be created or ported when a developer desires to add in a new device to their heterogeneous platform. Specifically, the only components that would need to be implemented for a newly added device would be the messaging transport mechanisms and the interface to hardware. In addition, the Barrelfish team developed their own compiler framework named Hake [29], essentially a Haskell embedded domain-specific language, to build both Barrelfish itself and applications for Barrelfish. By additionally writing a Hakefile for a target application, the Hake compiler will parse this file and generate a Makefile that will create a Barrelfish compatible binary. Finally, by making the OS state viewed as replicated instead of shared across the multiple kernels, it allows for improved system scalability by reducing load on the system interconnect, contention for memory, and overhead for synchronization as the system resembling more an event-driven system.

Even though Barrelfish claims to be portable [32] on heterogeneous platforms, no heterogeneous-ISA platform deployments are mentioned in their work. Popcorn is currently portable on the Xeon-Xeon Phi heterogeneous platform as demonstrated in this thesis.

Chapter 4

Heterogeneous-ISA Application Profiling

The first part of this work addresses the profiling of applications across the available processing resources in a heterogeneous platform. This thesis targets a platform with an Intel Xeon processor and an Intel Xeon Phi coprocessor. Moreover, the applications we focus on are written using OpenMP. OpenMP [7] is a traditional multicore/shared-memory parallelism model that targets thread-based parallelism in CPUs. To reiterate from the introduction, one of the driving forces of using heterogeneous hardware and systems is to leverage individual advantages of a given architecture.

Even though Popcorn Linux provides the illusion of a single operating system environment among different architectures through thread and process migration, it delegates to the user the decision of migrating an application to a different architecture. The profiling tool introduced in this chapter fills that gap; it finds an ideal partitioning that obtains the most performance with reference to the strength of memory coupling between functions within the given application. The definition of memory coupling in regards to this work is:

Definition 4.1. The amount of data (i.e., variables, arrays, etc.) used by a given function A, that is also used by a successive second function B within the same application. A high or large memory coupling value denotes that given functions A and B are highly coupled with each other as a result of sharing and/or passing each other a significant amount of data. A low or small memory coupling value denotes the opposite and therefore functions A and B do not share and/or pass a significant amount of data to each other.

Note that the proposed profiling tool is static and dependent on the input data. Although a single input data is usually representative of the call graph for certain applications, the input data can alter the runtime behaviour and therefore the tool should be re-run for each different input.

Memory coupling between functions of a target application is highly relevant to heterogeneous

platforms, especially when their current bottleneck hinges on the migration cost moving threads and data across architectures. This specifically pertains to heterogeneous platforms constructed of processors that do not share memory, since if processors do indeed share memory, there is no need to move data back and forth as it is already accessible to the other cores. No heterogeneous architecture today is cache-coherent, however this may change in the future. Nonetheless today's heterogeneous-ISA processors themselves can be cache-coherent within a heterogeneous platform, or even use shared memory (such as AMD APUs, and the Xeon - Xeon Phi) to get closer to closing the gap of memory coupling.

Migrating across different architectures provides the advantage that particular functions could potentially see a speed up by being mapped to one architecture versus another such that the migration cost is amortized throughout the program runtime. On the other hand, a function within the same program could also have terrible performance by being mapped to a different architecture and experience slowdown. By analyzing the input program by the Profiler tool presented in this work, this fatal scenario is avoided. As of this writing, characterizing and associating given architectures to having affinity for certain computation types has not been studied but could be a further work. Running the profiling tool is a one-time cost per application so this cost is constant compared to the number of architectures present on the heterogeneous platform.

The rest of this chapter discusses the Page Tracking Library and the application Profiler and is structured as follows:

- Section 4.1 presents the design of a generic heterogeneous application Profiler.
- Section 4.2 goes into detail of how the Profiler was implemented, how a page tracking library was implemented to accurately predict the costs of migration, and how graph theory is used to produce an accurate model that would give favourable partition solutions for the Xeon - Xeon Phi heterogeneous platform for a given target application.
- Section 4.3 gives small examples illustrating what the Profiler produces.

4.1 Profiler Design

Several preliminary experiments had been performed to better gauge the cost of the Popcorn Linux Operating System on Xeon - Xeon Phi platform. This tool is completely concerned with task migration as it is the most influential part of the formula to decide which compute kernels are worth migrating in the first place. Table 4.1 summarizes the averages of the timing information gathered.

Cost Name	Cost
Migration Cost	900 μs
Page Fault Cost	50 μs

Table 4.1: Costs associated with Popcorn Linux (from `cost.h` of the Page Tracking Library explained later)

The *migration cost* is the cost in micro-seconds associated with moving one thread from kernel 0 to kernel 1 (or vice-versa). The *page fault cost* is the cost in micro-seconds associated with bringing in a missing page to kernel 1, from kernel 0's memory. To obtain the *migration cost*, we average the cost of 20 migrations (operating system overhead only), invoked by a simple micro-benchmark which ping-pongs a simple function across architectures keeping track of the elapsed time needed to complete the migration, and then divide this time by the number of threads for each thread count of 1, 4, 8, 57, 114, 228. Because the Xeon - Xeon Phi configuration utilizes 8 cores for the Xeon, we therefore additionally weigh this average by the thread count divided by 8 (where any weight less than one is rounded up to 1). Finally, to get an all-around migration cost for any potential scenario, we take the mean of these values.

Number of Threads	Avg. Migration Cost
1	0.899 μs
4	1.103 μs
8	1.896 μs
57	0.369 μs
114	0.355 μs
228	0.437 μs

Table 4.2: Costs associated with Popcorn Linux (from `cost.h` of the Page Tracking Library explained later)

When calculating the best partitioning for the given application and in order to run the Profiler once, we should not only consider the system software overheads but also the relative computational capacity of one processor island in comparison to the other. Therefore we added the following parameters for each function when calculating a possible partitiong. The *xeon compute cost* is the cost in nano-seconds of performing on average one compute operation per memory access. The *xeon parallelism* is the number of cores available on the Xeon. The *phi compute cost* is the same value except for the Xeon Phi coprocessor. We measure it as eleven times (11x) slower. The *phi parallelism* is the number of cores available on the Xeon Phi.

We now discuss how Popcorn Linux achieves task migration.

Popcorn Linux introduces inter-kernel user-space task (thread and process) migration in Linux. A task migration consists of copying the task state from one kernel to another kernel in the

heterogeneous-ISA platform. A kernel-level migration service runs on each kernel and 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 called upon by an incoming inter-kernel task migration request. Using this technique, the pool adds minimal resource overhead to the system. Figure 4.1 shows the initial migration cost varying the number of concurrent migrating threads. The blue line shows that successive migrations are up to 35 times faster, but performance number dwindles as more concurrent threads are added.

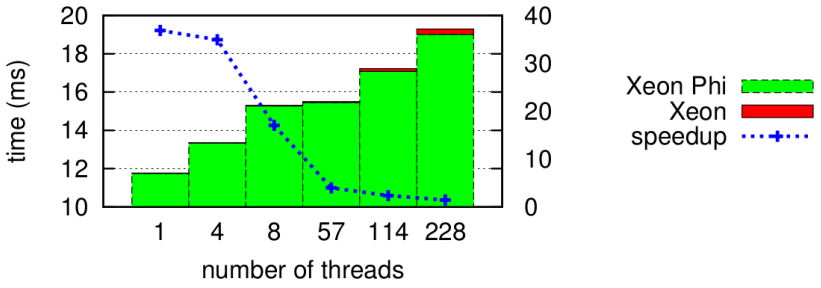

Figure 4.1: The bar graph shows average per-thread first migration OS cost in milli-seconds, for Xeon and Xeon Phi (refer to left side units). The blue line shows the speed up for subsequent migrations (refer to right side units).

4.1.1 Obtaining Processor Relations for Profiling & Partitioning

To obtain the performance relation between the Xeon and Xeon Phi processors needed for profiling, we performed the following experiment. Since the target set of applications that would be interesting to evaluate on this heterogeneous platform were already known, specifically we were interested in the NASA NPB benchmark suite [9], so we used that to our advantage and used that as the basis for the experiment. The procedure involved running a wide spectrum of benchmarks of varying nature and complexity on both the Xeon and Xeon Phi; this was so that the resulting equation for determining optimal partitionings (described in the following sections) would be decent "in-general" and not be specifically tuned for a particular use-case or type of application, and therefore potentially avoid profiling and partitioning to be skewed in certain manner. The applications were compiled natively with maximum optimizations enabled and configured to utilize the optimal number of threads per processor design (8 for Xeon and 228 for Xeon Phi). With these runtimes documented, we then divided the runtime by the number of threads utilized to see contribution per-thread. Finally, we took an average for both the Xeon and Xeon Phi runtimes and divided Xeon Phi numbers by Xeon numbers to see the speed-up or slowdown ratio between the two processors. Figure 4.2 shows the average performance ratio of Xeon Phi compared to Xeon. As can be observed the BT and SP benchmark of the NPB suite performance ratio both follow a similar linear trend as the input problem size is increased. It is expected that as the input problem size is doubled, so will the amount of time needed to perform the benchmark; as Class is

increased, the Xeon Phi takes approximately 11 times, 22 times, and 33 times longer to complete the benchmark respectively for classes A, B and C. The rest of the NPB suite follows a similiar trend.

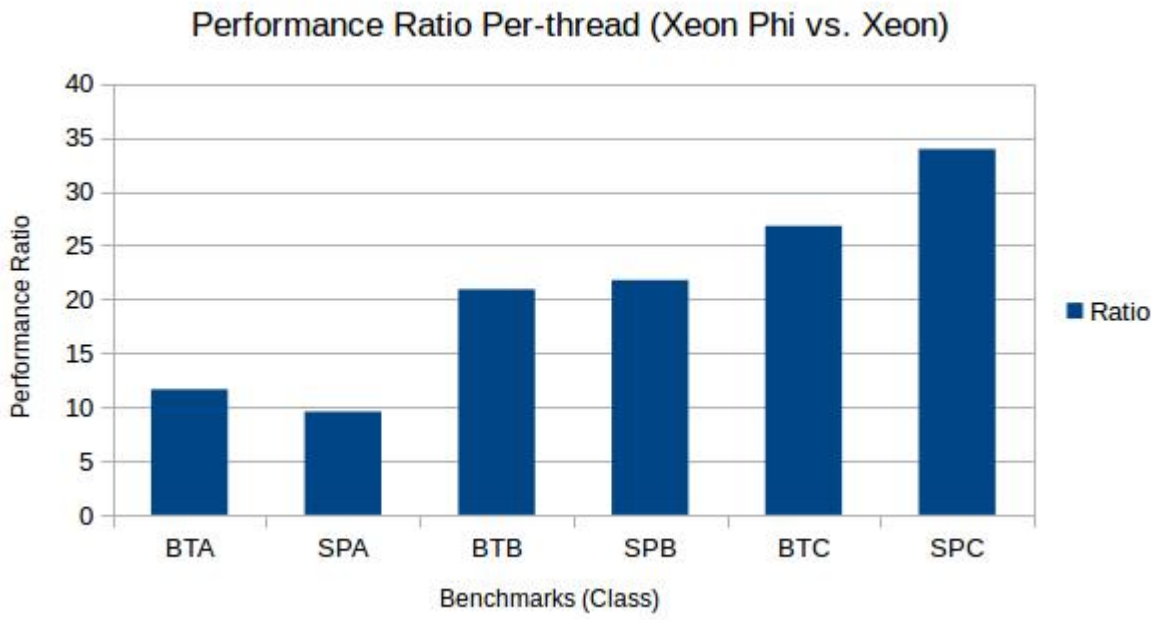

Figure 4.2: This figure shows the ratio of the additional time needed for Xeon Phi to complete the BT and SP benchmarks for each input problem size (Classes A, B, and C).

4.1.2 Design Principles

Once we explored the capabilities of the Xeon - Xeon Phi platform, we then decided upon some desired qualities our program Profiler should resonate and exhibit once completed. We came up with the following design principles.

- **Be generic.** Machines are being packed with more and more processing elements everyday, thus becoming more heterogeneous as a result. The Profiler tool should have a versatile interface to which a developer can easily add/modify/delete architectures and/or devices to consider when performing profiling on a target application.
- **React to interference.** How multiple applications are scheduled and mapped on a many-core system, such as a heterogeneous platform running Popcorn Linux, can influence inter-application interference for critical shared resources. Undesired interference can be introduced if poor/bad application-to-core mapping policies are in place, thus potentially hampering performance and efficiency. The Profiler should be at least naïvely aware of such inter-

ferences and predict which application phases could be most vulnerable to this phenomenon in the presence of other applications. A possible solution is to provide hints highlighting these vulnerable code regions and suggest to schedule them onto another core/architecture.

- **Accurately predict if migration will amortize migration costs.** It is known that migration of a target application to different architectures is the most expensive operation on a heterogeneous-ISA platform. The Profiler should be able to predict whether executing a given compute kernel on another architecture/device will amortize the migration cost and give better performance than by staying on the host kernel.

- **Be aware of the hardware topology.** Different heterogeneous-ISA platforms could have various configurations regarding shared memory and cache-coherence. In the current implementation of Popcorn Linux if cache-coherent shared memory is present, it is utilized by default. If shared memory is not available, Popcorn's DVSM is used in order to maintain a replicated state among kernels. Depending on the cache configuration, the Profiler should automatically select the appropriate cost model to use when profiling applications to correctly reflect the costs incurred when migrating across architectures connected by different means (e.g., PCIe, multi-socket, network, etc.). In addition, the result of the Profiler can vary according to the number of cores available on each architecture.

4.2 Implementation

To begin, we decided to implement the Profiling tool to initially use a static approach. We realize that using a static approach comes with limitations a dynamic approach would not encounter: static analysis is very rigid, is *not* problem size invariant, is *not* able to interact with features such a real-time scheduling, is *not* able to accurately account compute costs associated with external library functions, etc. These additional features are left as a future work and are beyond the scope of this thesis. The profiling tool is written using LLVM [21] (last tested against LLVM/Clang trunk revision 212187) and has been implemented as a LLVM pass. LLVM passes perform transformations and optimizations that make up the compiler as well as build analysis results from the transformations.

The profiling tool walks over every function in the target application and annotates every function call and every variable access with a call to a created page tracking library. We do not necessarily care about specifics, like which exact address is being accessed by which function, but rather care about the amount of data and memory pages that need to be fetched by a function in order for it to execute. The obvious reason being that the amount of data transferred during heterogeneous-ISA execution migration directly correlates to the incurred overhead of migration time. More specifically, we care about the number of pages needed to transferred – specifically per migration. This is because one page is the unit of allocation that the DVSM protocol handles when migrating data between architectures. This is why the Page Tracking Library was created. In Linux, system memory is divided into units called pages which consist of $4KB$. In addition, a memory address is

"page-aligned" if it's the beginning of a new page. An optimization discussed in Chapter 6 goes into more detail of why pages are important.

4.2.1 The Page Tracking Library

In order to encapsulate the core mechanisms of the Profiler and keep it simple for the LLVM pass, it was ideal to create functions as part of a small library. These functions would be called for each and any call of the target application's LLVM bytecode that manipulates memory in some shape or form. It is implemented in C/C++. To give a short summary, the library was organized as follows:

- `cost.h` – A header that provides the costs of performing actions using Popcorn Linux. These costs include migration cost, page fault cost, Xeon compute cost and the Xeon Phi compute cost all used the in Profiling algorithm when deciding whether or not it is beneficial to partition a given function to the Xeon Phi.
- `interface.h` – Implements the core page tracking functions described below.
- `graph.h` – Provides access to requires Boost Graph Library [22] Functionality. To avoid the need to use the Boost Graph Library directly in the main code we have this abstraction layer. It provides graphs where each vertex can have a name and a partition assignment, edge properties are templated. The main purpose of Graph<> is to provide a way to record edge properties. Not all functions make sense for all template types, this should only be a "problem" for FlowNetworks, but it's not actually a problem because those are only used in one place in a very specific way. Partitioning (global min-cut or s-t min-cut) is only implemented for UndirectedGraph. Most functions are implemented in `function_graph.h`, but a few functions which have per-template specializations are implemented in `graph.cpp`. As Boost is massively templated this class is lightly templated, however at the very bottom of this file three different template combinations typedef'd, these represent the only current use cases of the Graph<> class.
- `page_tracking.h` – Implements functions that record which function accesses which pages during runtime. The main purpose of this class is to provide information about the function that previously accessed a page, i.e. the function that currently *owns* a page. So when a function accesses a page, it is recorded as the new owner, and the previous owner is returned. Of course, the current function may already be the owner of a page. The point of this is that if two functions end up on separate kernels we need to know how many page faults this will cause so that we can choose a partitioning that minimizes the number of faults (balanced with other costs).
- `function_graph.h` – Implements functions which record properties of the call graph. More specifically this class records which functions call which and how often, and which functions access pages *owned* by other functions. Most of this work is actually done by

Graph (`graph.h`) and PageTracker (`page_tracking.h`), this class wraps these together and builds a cost model on top of them.

The following are the core tracking functions implemented in `interface.cpp` that are used to track which functions call which, and which memory addresses each function accesses:

- `ptrack_init(void)` – This function is called once before any other function in the target application. It initializes data structures used to store the memory access patterns during execution and registers a callback to the destruction function that is called upon completion of the target application.
- `ptrack_enter_func(const char *fname)` – This function is called upon entering a function. It can be used to keep track of which function the annotation's regarding memory access are occurring.
- `ptrack_call_func(const char *caller, const char *callee)` – This function is called just before a `caller` calls a `callee`. It is used for keeping track of the call chain and determining the strength of coupling between functions.
- `ptrack_memory_read(const char *fname, const void *addr)` – This function is called just before the current function, `fname`, reads from address, `addr`. It denotes that when the current function is being run, a page fault will occur. This will give an estimate of how much data is needed by this function.
- `ptrack_memory_write(const char *fname, const void *addr)` – This function is called just before the current function, `fname`, writes an address, `addr`. It denotes that when the current function is being run, the followed data will be invalidated. AKA This will give an estimate of how much data is affected by this function.

It should be noted that LLVM inserts functions for certain tasks (e.g., memset for initializing memory). These functions have nothing to do with partitioning, as they can happen locally for any architecture, and we do not see the implementation of them, so the Profiler has been modified to quietly ignore them. Early along the initial implementation, it was observed that failing to ignore them slightly biases against partitioning to put a given function on a different architecture, as these instructions tend to be widely used, and thus add a false cost of potentially putting a function on a different architecture.

Since applications tend to use libraries and functions associated with those libraries, those functions can potentially also affect how target applications should be partitioned between architectures. However, our page tracking library cannot track the memory or computational cost of library functions for obvious reasons. It is possible that the source of external libraries is not accessible, therefore the tool is not able to modify it appropriately and recompile. Mostly, this doesn't matter as functions like `printf()` are not interesting or relevant for our profiling. The GNU math

library, *libm*, however, is! Computational kernels frequently use math library functions, and assuming that they have a cost of zero is clearly wrong. We must realize that since we are considering scientific applications, their contibution in the application is substantial as they are a key player, but even so math library functions are quick to execute. We therefore assign a constant value for all math related functions, but realize that future work could possibly give the profiling tool a better way of accounting for the costs of library functions and improve the Profiler's introspection. In addition, to get desired behaviour of having I/O on the host architecture (e.g., it doesn't make much sense to have these operations occurring on an accelerator) we decided to pin all functions related to I/O (i.e., `fopen`, `fclose`, `fwrite`, `fread`, etc.) to the Xeon processor.

The Page Tracking Library utilizes the Boost Graph C++ Library [22], in order to represent the target application memory access pattern once all data has been recorded as a call-graph. Using the produced call-graph, *s-t* minimum-cut [11] (*s-t* Min-Cut), a common graph operation, can be performed and yield the most advantageous application partitioning from the memory access profile. In graph theory, a traditional minimum-cut of a graph is an operation that results in a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge that is minimal in some sense such as minimum edge weight. A conceptual illustration of a minimum-cut is below in Figure 4.3. Using information from the Page Tracking Library an *s-t* Min-Cut is performed to determine on which architecture each function in the target application should be run for best performance.

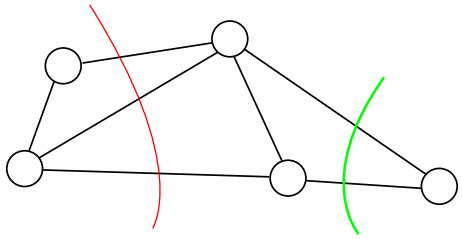

Figure 4.3: A graph and two of its cuts. The dotted line in red represents a cut with three crossing edges. The dashed line in green represents one of the minimum cuts of this graph, crossing only two edges.

4.2.2 Profiler Schema

The overall schema of the Profiler can be illustrated as below in Figure 4.4:

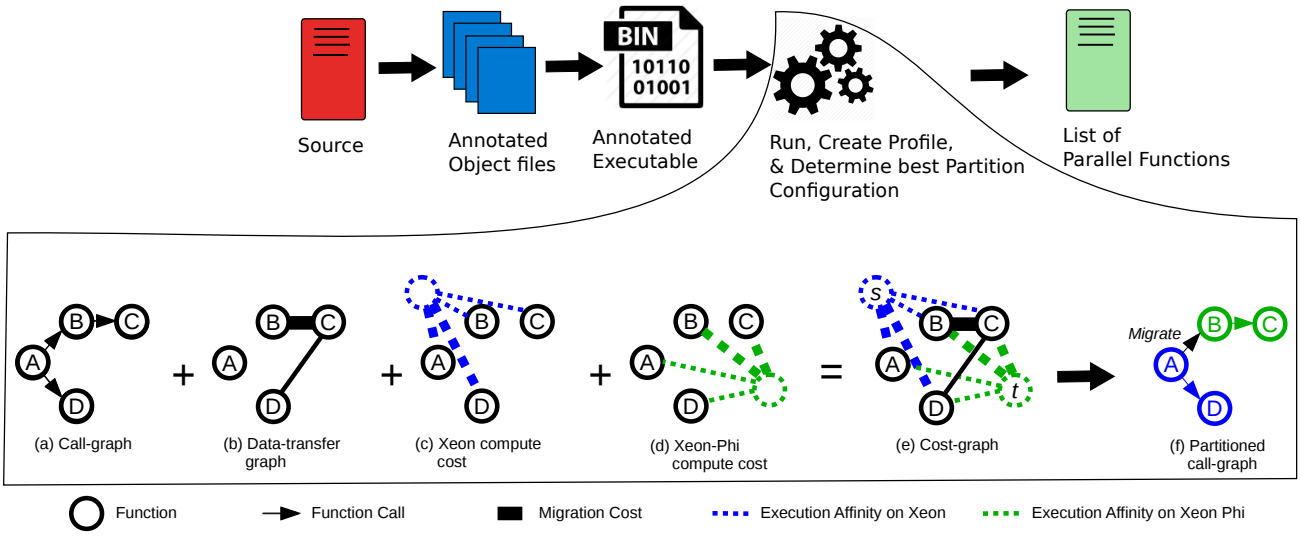

Figure 4.4: A complete schema of the heterogeneous-ISA Application Profiler

To better illustrate how the Profiler internals work, we describe a target application's profile graph as follows:

- The information used to create these cost graphs for each benchmark is obtained by using a LLVM pass to annotate every function call and memory access with a call to the Page Tracking Library mentioned earlier. In turn, this library then generates the necessary information 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 4.4 (a). If a function is not executed during the profiling run it will not be considered for migration. Each vertex in the graphs represents a function (*A*, *B*, *C*, *D* in Figure 4.4 $(a) - (e)$) within the target application. In (a) a call-graph is formulated to discern which functions call upon which to realize relationships and dependencies between functions in the target application.

- Each edge's weight between two vertices in Figure 4.4 (b) represents how tightly coupled two given functions are in terms of data transfers and page-faults incurred since data transfer is the most impacting cost when migrating between architectures. 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 4.4 (b) between two functions A and 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. In addition there is one configurable variable used in calculating the edge weight defined as bias. In this equation, bias controls how aggressively the tool attempts to partition a program.

It is implemented by increasing the compute cost of a vertex (node); this in turn reduces the relative cost of a data transfer, thus making partitioning more attractive. Trial runs of the Profiling tool performed on the NASA NPB suite show a good bias value is somewhere between one and five inclusive.

- To calculate the weight associated with a functions affinity with respect to the available architectures, as in Figure 4.4 (c) and (d), it is based on known constant (static) costs/specifications known about the heterogeneous system, in our case the Xeon - Xeon Phi platform (Refer to Table 4.1). Again, a higher (bigger) weight represents higher affinity between a function and a given architecture indicating that the function should be mapped to that particular architecture for most performance benefit. A low affinity between a function and a given architecture suggests that the function may have more benefit being mapped to some other architecture in the heterogeneous platform. The specifications used in the calculations include migration cost (the cost to migrate a thread from kernel 0 to kernel 1 or vice-versa), page fault cost (the cost to bring in a missing page to kernel 1, from kernel 0's memory), Xeon compute cost (the average compute per memory access), Xeon parallelism (the number of cores Xeon has), Xeon Phi slowdown (the slowdown of Xeon Phi compared to Xeon running a single-threaded program), Xeon Phi parallelism (the number of cores Xeon Phi has).

- Source and Sink nodes represent the various possible processors or instruction set architectures present within the heterogeneous-ISA platform. In our Profiler schema Figure 4.4 (c), (d), and (e) nodes s and t map Source and Sink nodes to the Xeon and Xeon Phi processors respectively. It is simple to extend our experimental platform to the source and sink model of a flow network as there are only two different architectures present and each can arbitrarily represent either the source or the sink while the second represents the node chosen (e.g., Xeon can be the source and Xeon Phi the sink or vice-versa).

The four graphs are combined into a single cost graph, shown in Figure 4.4 (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 runtime 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 runtime 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 diverse 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 4.4 (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 [11]. We map Figure 4.4 (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.

Before we dive into how we perform an *s-t* Min-Cut operation, it is necessary to be aware of a few terms that are used. The core algorithm implemented in the Page Tracking Library which decides the most advantageous partitioning for the target application based on the memory access profile and is based upon the Max-Flow Min-Cut Theorem [11] described below.

Definition 4.2. A **partition** of a set *X* is a set of non-empty subsets of *X* such that every element *x* in *X* is in exactly one of these subsets.

Definition 4.3. A **network** is a directed graph $G = (V, E)$ with a source vertex $s \in V$ and a sink vertex $t \in V$. Each edge $e = (v, w)$ from v to w has a defined capacity, denoted by $u(e)$ or $u(v, w)$. It is also useful to also define capacity for any pair of vertices $(v, w) \notin E$ with $u(v, w) = 0$.

Theorem 4.4. *The maximum possible flow from left to right through a network is equal to the minimum value among all simple cut sets.*

Theorem 4.5. *Max-Flow Min-Cut: In any network, the value of max flow equals capacity of Min-Cut. (Ford-Fulkerson, 1956)*

An *s-t* Min-Cut operation is the same as a Min-Cut operation, except that the operation additionally satisfies that nodes *s* and *t* reside on opposite sides of the cut and thus in different partitions. This *s-t* Min-cut is determined using max-flow/min-cut duality. What this means is that we map the resulting profile graph to a flow network, solve for max-flow using the Max-Flow Min-Cut Theorem, and then map that flow back to a min-cut. To go into more detail, once the flow-network's max-flow is obtained, we take the residual network (i.e., every edge which not at maximum capacity in the max-flow) and perform a reachability search from the original source node, to map the graph back to a min-cut of the original graph. Every node that we can reach goes in one partition, every node that we can't goes in another. This gives us the optimal partition's for Xeon and Xeon Phi. For those curious, the runtime of the Ford-Fulkerson algorithm is $O(m|f|)$ if all capacities

are integers, where m is the number of paths that exist in the graph and $|f|$ is the max-flow of the graph.

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.

4.3 Results

The Profiler optimizes the target application with the maximum settings using the `-O3` parameter. We also use the `-fno-inline` CFLAGS parameter so that every source function exists and is tracked. Disabling inlining of functions will have a conflicting affect on other optimizations, but leaving function inlining enabled will result in inaccurate partitioning decisions. After compiling to LLVM bytecode with maximum optimizations, the tool then applies the previously described annotation pass functions to each source file where needed. Then linking occurs and the target application can be run. With the annotations now inserted, running the application will record each function and it's memory accesses therein. Adding annotations enormously slows down the program, but luckily this is a one time cost per application for the user. The initial implementation could be sped up for loops that access memory serially by adding a new hook function to the Page Tracking Library and exploiting LLVM bytecode loop information. Generally however, the need to know the exact memory accessed at every single point makes optimizations hard.

Once the target annotated application completes, the second phase of the Profiler begins as it is safe to begin processing all the data that was recorded and produce an analysis that give the best partitioning of the target application between the Xeon and Xeon Phi. The analysis produces a few results: a text file with a list of functions that are deemed advantageous to put on the Xeon Phi coprocessor, a .DOT (graph description language) file which graphically represents the information collected, as well as a PDF file to visually show the application call-graph and along with their calculated costs. The best determined partitioning for the target application is also displayed to the user.

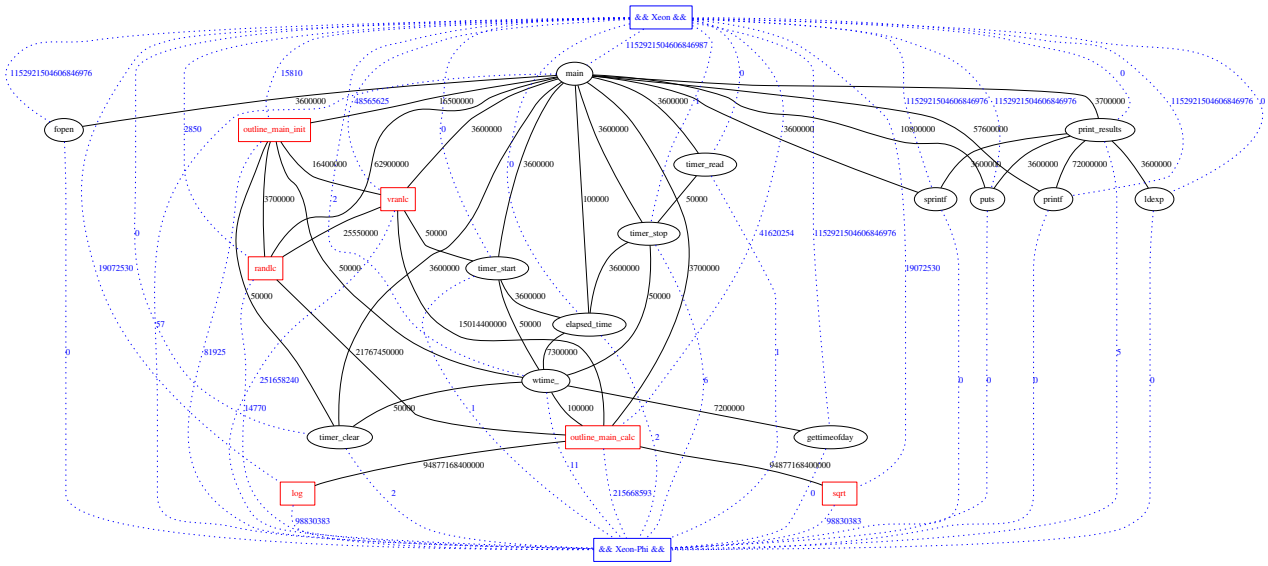

Figure 4.5: A sample call graph for the EP NPB benchmark that is generated by the Application Profiler.

Figure 4.5 shows a sample DOT graph from a generated PDF file representing all profiling data collected for the NPB EP benchmark. The two unique *s* and *t* nodes are denoted in blue. As mentioned earlier all the nodes in this graph (besides the two in blue boxes) are various functions that are used in the target application. The resulting best partitioning is denoted by the function names contained in red boxes. These are the functions and subfunctions that have been deemed advantageous to be run heterogeneously should be compiled for the Xeon Phi coprocessor to be migrated to that architecture during runtime. The remaining functions contained in black circular nodes are functions that are not worth migrating onto the Xeon Phi and therefore do not need to be transformed. If inspected carefully, it can be observed that certain edges have massively large weights compared to other edges. It was mentioned earlier that all I/O functionality would be pinned to the host (Xeon) architecture when performing the *s-t* Min-Cut computation to determine optimal partitions. We acheived pinning the desired I/O functions by manually assigning them the maximum edge weight between the host architecture when creating the cost graph.

In addition, a production version of the Application Profiler would be combined with the Heterogeneous-ISA Application Partitioner (to be presented in the next chapter) to form a 1-step process for the developer. However, for the prototype it was earlier to debug the compiler framework as separate entities, isolating problems to resolve issues faster. Figure 4.6 illustrates this connection.

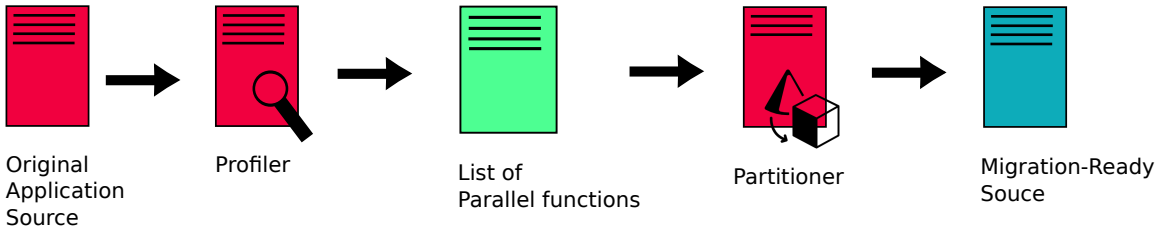

Figure 4.6: This figure illustrates how the Profiler and Partitioner are connected.

It has been left for future work to implement detection of multi-application intereference and automatic cache configuration detection. With this tool, instead of the developer needing to guess and check various possible partition configurations for the target application, they only need to run the Profiler once to obtain the optimal configuration, with no user intervention required besides a few initial settings.

Chapter 5

Heterogeneous-ISA Partitioner

While the Xeon - Xeon Phi Profiler enables the developer to see what functions have advantages being run on different processors available within a given heterogeneous-ISA platform, it doesn't perform the code transformations needed for an application to be *migration-ready* during runtime to take advantage of the platforms heterogeneity. When we say that an application is *migration-ready*, we mean that the application's source has been successfully transformed and additional files have been included during recompilation to allow migration to occur at function boundaries (the purpose of this tool). Even with this newly collected information, it still leaves the developer with the tedious and error-prone task of performing porting for their target applicationsto to a given parallelism programming paradigm. In addition, several Popcorn Linux specific code blocks and files need to be added to any target application to be compatible and *migration-ready* on Popcorn Linux. We envision Popcorn Linux to be a self contained and easily deployed solution for heterogeneous systems. By having an automatic Partitioner, we provide a component that performs the tedious tasks a user would otherwise need to worry about when attempting to use Popcorn Linux. Therefore, this tool fosters the programmability of such emerging platforms. Note that the component we are referring to can be integrated into the C library when the entire system software will support heterogeneous-ISA platforms, thereby removing the burden from the developer.

This chapter presents the second half of the Xeon - Xeon Phi compiler framework, namely the Application Partitioner. The Application Partitioner's duty is to take a developer's original source code, and, using the information produced by the Application Profiler, properly transform the designated functions to become able to migrate between Xeon and Xeon Phi during runtime. The Application Partitioner will also need to satisify dependencies created by partitioning as well as add in Popcorn Linux specific modifications to enable seamless migration. While performing all these tasks, the Partitioner must be careful to maintain original application semantics in order to not introduce new unintended bugs or unsupported corner cases.

This chapter is structured as follows:

- Section 5.1 discusses the design behind re-factoring developer code (by performing source-

to-source code transformation) in order to a) be compatible with the underlying OS, Popcorn Linux, and b) support process migration during runtime.

- Section 5.2 gives the implementation details for performing the source-to-source code transformation step-by-step, what additional code is needed for process migration to be possible and why, as well as the complete compilation strategy for producing a heterogeneous-ISA binary ready to be run on the Xeon - Xeon Phi platform featuring the Popcorn Linux OS, including modified runtime support libraries that will be further described later in Chapter 6.
- Section 5.3 presents sample source-to-source code transformations on a smaller scale in order to demonstrate what this process looked like for the benchmarks that are included in this work; we also present the evaluation of our profiling, partitioning, and code transformation techniques in this section.

5.1 Design

With the need to add-in additional Popcorn Linux specific code blocks to a target applications source and compiling additional files when creating a heterogeneous-ISA binary, it drives the motivation for creating a tool that would automatically take care of this and require minimal user intervention during the transformation process. Performing porting for numerous benchmarks would otherwise be time-consuming and user-effort could better spent elsewhere. With the Application Partitioner the following goals should be realized:

- **Be Compartmentalized.** In the larger scheme of things, the scale of complexity of the Xeon - Xeon Phi compiler framework steadily grew throughout the engineering phase of developing this framework to be production ready for the desired set of benchmarks to bring to light how good Popcorn Linux and the compiler framework were with respect to given competitors (e.g., Barrelfish, Intel LEO, and OpenCL implementations). The further the framework progressed, the more complex and minute nuances plagued our implementation that had to be figured out and resolved in order to progress. These nuances occurred in the entire spectrum of the compiler framework and early on it was determined to be of best interest to adopt a modular design of components. This would allow for straightforward access to apply remedies for discovered bugs without the risk of encountering a domino effect where one fix breaks something else. As a result, the application Profiler and application Partitioner are completely disjoint, but the Profiler feeds information directly to the partitioner as an input parameter.
- **Be Automatic.** A few Popcorn specific code blocks need to be inserted in order to enable process migration between architectures for the given target application. This consists of wrapping function calls and redirecting variable assignment. Depending on the complexity of the target application, this could easily become very troublesome for a developer to deal

with manually by hand. The application partitioner described below steps in and provides the porting of a target application without user intervention.

- **Be Modular.** Thinking ahead, it is important to consider possible configurations of heterogeneous-ISA platforms down the line. When implementing this part of the compiler framework, it was paramount to make it modular and generic, able to be adapted to add, remove, and modify the interface that directs how the target application would be compiled. This is so that creating binaries for any given heterogeneous-ISA platform would be easy to update and as easy as physically swapping in and out various heterogeneous hardware. As a result, both the partitioner and compiler have interfaces and options that provide a means for the user to update which architectures and/or devices that they wish for the binary to support. This feature proved to be very useful in the development stage, as it provided a means to debug the target application's different ISA compiled versions by being isolated from each other and allowing them to tested independently.

For the given configurations presented in this work (Xeon - Xeon Phi) only one partition file (explained in the implementation section) is needed to be generated as there exists only one "other" architecture present within the heterogeneous-ISA platform and to compile separately for (besides the host architecture). In a production version of this application partitioner tool, the tool would be capable of performing all the source-to-source code transformations needed to support *N* target architectures and/or devices. In this production scenario there would exist as many partition files as "other" non-host architectures present on the heterogeneous-ISA platform.

5.2 Implementation

The Xeon - Xeon Phi partitioner presented in this work is implemented using the ROSE Compiler Infrasture [27]. ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large scale C(C89 and C98), C++(C++98 and C++11), UPC, Fortran (77/95/2003), OpenMP, Java, Python and PHP applications and was developed at the Lawrence Livermore National Laboratory (LLNL). Using its API, we were able to construct a source-to-source compiler that performs two passes of transformations on the target application before being ready for the final heterogeneous-ISA compilation for Xeon - Xeon Phi.

ROSE works by first reading the target application's source and generating an Abstract Syntax Tree (AST). The AST generates a graph representing the structure of the target application composed of nodes that contain the intermediate representation (IR) of the source. The API then provides and interface to manipulate and change the tree as desired; this is how source-to-source transformation occurs.

5.2.1 Adding Hooks

The first pass connects the Application Profiler to the Application Partitioner (refer to Figure 4.6) by taking in the results of the Application Profiler (a text file with a list of functions that should be transformed to be *migration-ready*) and going through the application source appending `#pragma popcorn` above the listed function's definitions. This is acheived by creating a new node into the AST with this content and traversing the AST until the designated function definition is found. This process is repeated until all functions listed in the Profiler output file have been found. This creates a hook for the second pass to easily recognize which functions should be transformed. Figure 5.1 depicts this transformation.

```
...

int compute(int a, int b)
{
  int c;
  c = a + b;
  ...
  return c;
}

void print_result(char *n){
...
```

```
...

#pragma popcorn
int compute(int a, int b)
{
  int c;
  c = a + b;
  ...
  return c;
}

void print_result(char *n){
...
```

Figure 5.1: $Before$ (left) and$After$ (right) adding code transformation hooks to a sample program

5.2.2 The Main Pass

The second pass performs the majority of the heavy lifting for the Application Partitioner. In order to stay aligned with the third goal of the partitioner to be modular, we create a new file (hereafter called partition file) for each different ISA we will be compiling for except for the designated "host" architecture. In this work only one partition file is created as the Xeon Phi ISA is the only additional "non-host" architecture present in our target heterogeneous-ISA plaftform. The partition file will be populated during the second pass of Application Partitioner and will contain functions and those function's respective dependencies of the given target application that have been deemed beneficial in utilizing migration to a given target ISA present on the given heterogeneous-ISA platform. This population process is repeated for each ISAs designated partition file, based on the results from the Application Profiler and is described in more detail below. Dependencies that need to be copied over to a partition file from the original target application's source include global objects used within the target function such as arrays, pointers including their parallel programming meta-data (in our test case OMP pragmas such as `#pragma omp threadprivate`), as well as functions called within target functions. It should be mentioned that the partitioner can handle

any amount of recursive function calls and will copy all involved children functions. In addition, the partitioner has been implemented to recognize and correctly handle traditional recursion (e.g., if a function is recursive, the partitioner will not copy the function more than once, thus avoiding going into an infinite loop).

Now we will go into more detail of the source-to-source transformation that occurs in the second pass once the creation of partition files for each of the present ISAs has completed. The process described below is iterative, an iteration is executed for each `#pragma popcorn` and this search-and-transform process is repeated until no more `#pragma popcorn` hooks are found. Once a `#pragma popcorn` is found partitioner begins the source transformation process begins.

The source-to-source transformation is composed of several steps. We will discuss each of them separately and after each step, a figure is presented to depict what new changes have been made to the sample application to give a better understanding of the code transformation process. For easy readibility as well as debugging each of the steps are implemented as a specific function in our ROSE Partitioner tool. Table 5.1 lists each of these steps, notes which function is associated with this step, and gives a short description.

Tranformation Step	**Function Associated with Step**	**Description**
Input and Output Argument Recognition	`makeStruct4Func(funcDecl);`	Creates the structures needed to encapsulate function parameters
Function Rewriting to Handle Arguments on the Heap	`makePopcornFunc(funcDecl, st);`	Transforms target functions to be *migration-ready*
General Cleanup	`removeOrigFunc(funcDecl);`	Removes old function definitions
Adding Stub Code to Enable Migration	`mainMigrateTransform(project);`	Replaces all instances of target functions to enter a migration wrapper

Table 5.1: Main Code Transformation Steps

For the rest of the implementation please refer to the following code in Figure 5.2 as a reference to what the original application source looked liked before it underwent transformation.

```
/* Original Source Code */
#include <string.h>
...

double compute( int a, int b)
{
  double c;
  c = a + b + 3.14;
  ...

  return c;
}
...

int main(int argc, char *argv){
  int a, b;
  double s;
  ...

  s = compute( a, b);
  printf("output after compute: %f\n",s);
  ...

  return 0;
}
```

Figure 5.2: Original source code of sample application

5.2.3 Input and Output Argument Recognition

This is the first step that occurs. This function creates a structure (`struct`) that will hold the `funcDecl`'s function parameters (as well as the return value if the function is not of type `void`) and is the unit which gets fetched when execution migration occurs. The parameters of `funcDecl` are obtained by parsing the function's function declaration using the AST. This structure definition is added to both the partition file and in the source file where the original function is located such that all files which will reference this structure will have its definition. Note that a structure is created individually for each `#pragma popcorn` annotated function with the following naming convention: `functionName_migrate`, where `functionName` is replaced with the actual name of the function currently being transformed. This approach is scalable as more architectures are included within the heterogeneous-ISA platform, only one structure per function is needed.

```
...

extern struct compute_migrate;

double compute(int a, int b)
{
  double c;
  c = a + b + 3.14;
  ...
  return c;
}

void print_result(char *n){
...
```

```

struct compute_migrate
{
  int a;
  int b;
  double return_val;
}
```

Figure 5.3: Source (left) and Partition File (right) after `makeStruct4Func`

5.2.4 Function Rewriting to Handle Arguments on the Heap

This is the next step that occurs in the loop and is responsible for creating the *migration-ready* version of the function that has been marked with the Popcorn pragma. This includes changing the function's name to include a suffix (i.e. `_x86`, `_phi`, etc.) to prevent compiler conflicts later in the compilation process, modifying its input parameters to only be the structure (created in `makeStruct4Func(funcDecl);` associated with that function, and changing it's return type to be void now that the return value will be stored in the migration structure instead. This step also modifies the body of the given function to reflect the changes in the function prototype. This means changing all references of input parameters in the function to refer to its migration structure counterpart (e.g., `input1` becomes `ctx->input1`, etc.) for all original input parameters. Refer to Figure 5.4. In addition, this step recursively searches for child function calls within the Popcorn pragma'ed function, records them, and adds static copies of function definitions to the partition file to resolve dependencies. The last part of this step checks and modifies the return segment of the function. Since the return value, if any, will now be stored in the migration structure, the code needs to reflect this so the return statement is modified to be stored in the structure, therefore the return structure is eliminated and return type is changed to void.

```
void compute_x86(struct compute_migrate *ctx)
{
  int a = ctx->a;
  int b = ctx->b;
  double c;
  c = a + b + 3.14;
  ...
  ctx->return_val = c;
}
```

Figure 5.4: The *migration-ready* version of `compute_kernel` after `makePopcornFunc`

5.2.5 General Cleanup

Once the modified *migration-ready* function has been created, the next step appends it to the original functions source location as well as partition file and removes the old function as that one is now obsolete. The orginal function can be optionally left in during transformation but adds no value to the target application at this point.

```
...

struct compute_migrate
{
  int a;
  int b;
  double return_val;
}

void compute_x86(struct compute_migrate
    *ctx)
{
  int a = ctx->a;
  int b = ctx->b;
  double c;
  c = a + b + 3.14;
  ...
  ctx->return_val = c;
}

void print_result(char *n){
...
```

```
struct compute_migrate
{
  int a;
  int b;
  double return_val;
}

void compute_phi(struct compute_migrate
    *ctx)
{
  int a = ctx->a;
  int b = ctx->b;
  double c;
  c = a + b + 3.14;
  ...
  ctx->return_val = c;
}
```

Figure 5.5: Source (left) and Partition File (right) after `removeOrigFunc`

5.2.6 Adding Stub Code to Enable Migration

This step does the heavy lifting of searching throughout the entire application source and replacing all calls to Popcorn pragma'ed functions with the migration hint function, the function through which migration to another architecture occurs. This step also takes care of adding neccessary calls to populate the migration structure that gets passed into the migration hint function to use migrated and used by the Popcorn pragma'ed function. In addition it takes care of adding calls for populating the variable which originally received the return value redirecting it from the migration structure; note that this additional process is not invoked if the Popcorn pragma'ed function has return type void.

```
#define _GNU_SOURCE
#include <sched.h>
#include "migration_addins.h"

#include <string.h>

...

struct compute_migrate
{
  int a;
  int b;
  double return_val;
}

void compute_x86(struct compute_migrate *ctx)
{
  int a = ctx->a;
  int b = ctx->b;
  double c;
  c = a + b + 3.14;
  ...
  ctx->return_val = c;
}

...

int main(int argc, char *argv){

  struct compute_migrate *compute_ctx;
  compute_ctx = ((struct compute_migrate *)(malloc(sizeof(struct
    compute_migrate ))));

  ...

  compute_ctx -> a = 1;
  compute_ctx -> b = 2;
  migration_hint(compute_ctx,compute_x86,compute_phi);
  s = compute_ctx -> return_val;

  printf("output after compute: %d\n",s);

  ...

  free(compute_ctx);
  return 0;
}
```

Figure 5.6: The resulting source code after undergoing all stages of the transformation

To highlight the code transformation, all function calls of functions that have been deemed worthy to migrate and now *migration-ready* versions, are replaced with a function call the function, `migration_hint` which invokes process migration between kernels. A code block is shown below.

```
migration_hint(funcName_ctx, funcName_x86, funcName_phi);
```

This is an example of the Xeon - Xeon Phi implementation of the `migration_hint`. The first parameter is the function's structure, created in the first step and contains `funcName`'s original function parameters. The second parameter is always designated to be a pointer to the host-architecture version of `funcName_x86` (In our implementation the Xeon x86 version). In our implementation, the third parameter is a pointer to the Xeon Phi architecture verision of `funcName_phi`.

```
migration_hint(funcName_ctx, funcName_arch0, funcName_arch1,
                    funcName_arch2, ...  );
```

Figure 5.7: Generic Implementation of `migration_hint`

In a production version of the Application Partitioner, the `migration_hint` function would be configurable to handle any number of N different architectures for a developer that is targeting a given heterogeneous-ISA platform. The current implementation is able to be reconfigured to handle additional architectures with trivial developer effort.

Once the partitioner has come out of this loop, minor touch-ups on the source are performed. This includes declaring header files and a few macros needed for migration functionality. An important distinction is needed to treat/compile the correct versions of library functions (specifically from the math library as these functions can potentially be called from any partition). As a result supplimental code transformation needed to be performed, but this process will be described in detail in the Xeon - Xeon Phi runtime support section of the Heterogeneous-ISA Runtime Support chapter. This concludes the source-to-source transformation process; modified source is now ready to be compiled and linked to form the heterogeneous-ISA FAT binary [39].

5.2.7 Xeon - Xeon Phi Compiler/Linker

Once source-to-source code transformation has been performed on the target application, it is ready to be compiled and linked for the Xeon - Xeon Phi platform.

We automated the process to complete both the compile and link steps needed to create a heterogenousa-ISA FAT binary of the target application. The automation process has 3 modes of operation and 2

variable inputs for creating binaries. Mode 0 denotes the creation of heterogeneous-ISA executables with the ability of migration between architectures. The other two modes, 1 and 2, denote the creation of homogeneous-ISA executables (no ability to migrate but still able to run on the Popcorn Linux platform. The `migration_hint` function forces the application to run the native version of the target function instead of performing `sched_setaffinity()` and migrating the threads before executing the non-native version of the target function) that are compatible with only one host architecture, Xeon or Xeon Phi respectively. The two variable inputs were for fine tuning of experimental results and denote to the compiler how many cores to enable for use on each architecture in order to test a wide spectrum of configurations on the Xeon - Xeon Phi platform with respect to the number of CPU cores to be utilized. Depending on the mode selected, the binaries are assigned discernable names to help the developer understand the configuration of the executable produced without much effort.

This notation is as follows: `bName.bClass.version.xCores.xpCores.kernel`, where `bName` is the benchmark name, `bClass` is the benchmark input problem size (in the case of NPB), `version` to descipher which mode was used to compile the benchmark and can have values `Heterogeneous` or `Homogeneous`, `xCores` denotes how many Xeon cores have been enabled, `xpCores` denotes how many Xeon Phi cores have been enabled, and `kernel` is for the heterogeneous-ISA case where the kernel states for which architecture the given benchmark file is for (in our case it can take values `Xeon` or `Phi`, denoting that the file has been compiled using the Xeon or Xeon Phi ISA, respectively).

After the output names are set, traditional compilation flags are set such as optimization level, enabling libraries, including a few customly created macros for manipulating the OMP library at run time. Note that each architecture's partition uses the most aggressive optimization option (`-O3`) when being compiled to gain the most performance benefit. The previously mentioned custom macros include:

`-DNUM_HOST_CORES`: the number of cores to be enabled for use on the host architecture (in this case Xeon)

`-DNUM_REMOTE_CORES`: the number of cores to be enabled for use on the target architecture (in this case Xeon Phi)

`-DFIRST_REMOTE_CORE`: this macro is tied in with Popcorn Linux's OS single system image and the way it sees the heterogeneous platforms available resources. It denotes the assigned number of the first CPU of the target architecture to be migrated to. This is utilized when migrating calling the function `sched_setaffinity()` which takes in the process's PID and CPU affinity mask and uses that to 'reschedule' the process onto the target architecture.

Once all flags are set for the mode selected, it is time to actually compile the source with the appropriate flags into object files. This is accomplished in one (optionally two) step(s). The first step is that all C source files are compiled for the host (Xeon) architecture. During this step, all host architecture object files have the libomp symbol `__kmpc_atomic_float8_add` defined to point to the Xeon ISA implementation. This is a patch for a nuance created by the OMP Library

for proper multithreaded execution and will be described in further detail in the chapter regarding heterogeneous-ISA runtime support, Chapter 6.

Since the first action performed by the automation process compiles all C source files for the host architecture, this inadvertently also includes all non-host partition files (in our case this includes the parttion file created for Xeon Phi) which is undesired if compiling for a heterogeneous-ISA FAT binary (mode 0). If mode 0 was selected, we delete the object's created for the non-host partitions and recompile for their respective ISAs (in our case we recompiled for Xeon Phi using Intel's ICC compiler with *k1om* as as the target architecture.

During this step we also redefine the libomp symbol `__kmpc_atomic_float8_add` to point to the Xeon Phi ISA implementation. This is make sure the correct implementation is used for proper multithreaded execution.

Assuming that we are compiling for a heterogeneous-ISA FAT binary, we run into the issue that compilers such as `GCC` were not designed to link object files of varying ISAs. We therefore need to manipulate the ELF's Magic Number [40] value for the each partitions generated object file to match the host architectures Magic Number configuration. Essentially a files Magic Number, not usually relevant in high performance computing yet is an obscure obstacle we discovered working with heterogeneous platforms, is meta data located in the first few byte of a file that specifies the architecture type of a given file. In our case, it prevents mixing of object files of different architectures during compilation which we actually do want to do to form a heterogeneous-ISA binary. With the value changed the compiler will successfully create the host architecture version of the target application, no longer preventing compilation. But this is only half of a complete heterogeneous-ISA FAT binary. In addition, a binary for each target architecture (in our case just Xeon Phi) must also be compiled, but a short-cut can be taken here. Since we have already compiled object files for each partition using it's respective ISA and we will *not* be executing any other functions on a target architecture besides the ones specified by the Profiler into the host-architecture binary, we need make a copy of the host-architecture binary and modify it's Magic Number to satisfy the architecture requirement for execution.

Note that if mode 0 was not selected this re-compilation of non-host partition files can be side-stepped and linking can occur immediately after object file creation. Also if the mode was selected to compile for a non-host ISA, the binary would be compiled for that architecture.

There is one more catch when dealing with heterogeneous-ISA FAT binaries in Popcorn: each executable must be in the same system directory location on each architecture. Another caveat, though not really a limitation, is that execution of the heterogeneous binary must begin on the host (Xeon) architecture of the platform. This design decision was motivated by providing simplicity to the user and also to provide potential future work for the Heterogeneous compiler, specifically the potential of a ubiquitous (all-encompassing) executable for the heterogeneous platform for maximum flexibility.

5.2.8 Add-in Files & Migration Mechanism

In addition to the needed source-to-source code transformation provided by the Application Partitioner, an additional module is necessary to be compiled together with the target application to correctly produce a *migration-ready* FAT binary for Popcorn Linux. This source file provides the implementation of functions that take care of process migration as well as a "primer" migration function. Taking advantage of Popcorn Linux's global namespace, we are able to "see" the majority of all hardware and software resources (i.e., processor cores, PID's, etc.) on the heterogeneous-ISA platform. We mention this because the mechanism to migrate a process onto a different architecture is through the use of `sched.h` macros and functions. The implementation of `migration_hint` is depicted below.

```
Inputs:
  *ctx: Functions associated structure holding original functions parameters
  *f:   Pointer to Xeon compiled version of function to be migrated
  *g:   Pointer to Xeon-Phi compiled version of function to be migrated
void migration_hint(void *ctx, void (*f)(), void (*g)() ) {
  int kernel,i;
  kernel = pick_kernel();
  cpu_set_t host, target;

  CPU_ZERO(&host); CPU_ZERO(&target);
  const int host_cpus = NUM_HOST_CORES;
  const int target_cpus = NUM_REMOTE_CORES; //xeon phi is 228
  const int offset = FIRST_REMOTE_CORE; //8

  for (i = 0; i < host_cpus; i++) {
    CPU_SET(i, &host);
    //0,1,2,3
  }
  for (i = offset; i < (offset+target_cpus); i++) {
    CPU_SET(i, &target);
    //4,5,6,7,...
  }
  if (kernel == 0){
    // 0 is host (Xeon)
    f(ctx);
  }else{
    //1 is target arch (Xeon Phi)
    //go to target
    #ifdef _OPENMP
      omp_set_num_threads(target_cpus);
    #endif
    #pragma omp parallel
    {
      sched_setaffinity(0,sizeof(cpu_set_t),&target);
    }
    //run that versions of function
    g(ctx);
```

```
38      //come back
39      #pragma omp parallel
40      {
41        sched_setaffinity(0,sizeof(cpu_set_t),&host);
42      }
43    }
44  }
```

Listing 5.1: `migration_hint` Implementation

The macros `NUM_HOST_CORES, NUM_REMOTE_CORES, FIRST_REMOTE_CORE` are defined during compilation by the compilation script described earlier. The function works as follows. First, executing `pick_kernel();` gives which kernel/architecture the process should be migrated to for the given function. In our case, there is only one other architecture possible to migrate to, the Xeon Phi and thus returns the value 1. With the help of `CPU_ZERO` and `CPU_SET`, we are able to create the desired CPU mask for the given platform, and thus dictate onto which CPUs the process and/or its respective threads should be scheduled. Once the CPU mask is set, it is passed to the function `sched_setaffinity()` which takes care of performing the needed syscalls to perform the process/thread migration.

Below is an excerpt of the description for `sched_setaffinity()` from Linux Programmer's Manual [18]:

> A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits. For example, by dedicating one CPU to a particular thread (i.e., setting the affinity mask of that thread to specify a single CPU, and setting the affinity mask of all other threads to exclude that CPU), it is possible to ensure maximum execution speed for that thread. Restricting a thread to run on a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to execute on one CPU and then recommences execution on a different CPU.
>
> A CPU affinity mask is represented by the cpu_set_t structure, a "CPU set", pointed to by mask. A set of macros for manipulating CPU sets is described in CPU_SET(3).

`sched_setaffinity()` is wrapped within a `#pragma omp parallel` block in order to support multi-threaded migration. The desired function to be run on the different architecture is then initiated by calling `g(ctx);`. The target function will now be executed on the Xeon Phi architecture. Once it completes, `sched_setaffinity()` is called once again, this time with the host CPU mask configuration to bring the threads back to the host architecture (Xeon). With additional architectures present on the target heterogeneous-ISA platform, the function `pick_kernel();` is a perfect candidate for modifying to create a user-defined migration model with the various options available to the developer. In addition, enabling for the use of additional different architectures or devices would require minimal developer effort to add in additional hooks following the current model for our Xeon - Xeon Phi platform.

Chapter 6

Heterogeneous-ISA Runtime Libraries

Having an automatic compiler framework that properly partition target applications to be *migration-ready* for a heterogeneous-ISA platform is just one piece of the puzzle to possessing the capability of fully exploiting a heterogeneous-ISA platform.

Making the argument that today's applications are intricate and complex, they usually rely on support from various libraries to fill in functionality that would otherwise be too tedious to also implement alongside the application logic. In other words, why re-invent the wheel when someone else has already created a library to provide the functionality you desire in your application. It is general knowledge that the kernel (in our case Popcorn Linux) governs access to memory, the filesystem, hardware, and the privileges for using these resources. The standard C library provides the actual C function interfaces userspace applications see. It is responsible for thread creation, managing memory allocation and synchronization operations, and other commonly needed functionality using the lower-level interfaces the kernel provides. Finally, the standard C library also gives access to pure library routines of the C language like `strstr`, `sqrt`, `exp`, etc. Therefore the standard C library is a necessary core component for any non-trivial application. Normally, this type of discussion about libraries is not a major concern or real interest to most developers and computer scientists in this day and age. The reason being that most current heterogeneous devices such as GPUs, FPGAs, and assorted accelerators simply provide the developer "their" device's library API when they purchase the device. Sure the developer has to now port his/her applications to conform with the new devices API, but manufacturers assume this is acceptable as the speedup that the hardware provides is a satisfactory compromise for this time investment. We claim that this is not sufficient.

Providing support for heterogeneous-ISA OS-capable platforms is a whole different story. Our initial survey of core libraries showed that finding ones which have multiple implementations across various architectures proved difficult to come by, and in our case, forcing us to even perform porting ourselves in some instances. Because of the predicted significant effort required to port multiple libraries for the heterogeneous-ISA platforms, we decided to narrow our scope and only target core and absolutely necessary libraries for this work. This included libraries such as the C

standard library, the Math library, and libraries to enable multi-threading. For most benchmarks presented in this work, we were fortunate enough to not be required to port any additional obscure libraries. All things aside, this caveat is still a harsh reality that developers wishing to use our work should realize. Though on the bright side, libraries only need to be ported once per architecture.

In addition, at this point it should be noted that all applications as well as libraries (i.e., *glibc*, *libm*, *pthreads*, etc.) used and presented in this work and all others to be run on Popcorn Linux are compiled ***statically*** as dynamic compilation is not yet supported for Popcorn Linux. The reasoning behind this design decision is for a multitude of logical reasons. Since we are supporting a heterogeneous-ISA platform, the complexity inherent with dynamically loaded libraries is much greater than static libraries and the amount of engineering needed is beyond the scope of this thesis. Even though we are well aware it makes the size of a heterogeneous-ISA FAT binary explode, static libraries give much more direct access for performing modifications and resolving issues during compiling and linking of binaries. Static linking also comes with it's other already known benefits of portability and slight performance advantage over dynamically linked programs. As Popcorn Linux and its compiler framework continues to mature, it is a working goal to support dynamically linked libraries.

To obtain the best performance gains possible, it should also be noted that all applications and libraries used and presented in this were compiled with maximum and most reasonably aggressive optimizations for each respective architecture *enabled*. In order to make sure that our compiler framework had maximum optimizations enabled, we verified that all loops in our various test cases had been vectorized. We checked this using the `-vec-report1` option available in Intel's ICC compiler.

One interesting caveat that came up during benchmarking was a specified option for the compiler. The GCC option `-mcmodel=` can have one of several values including `small`, `medium`, and `large`. This option specifies and tells the compiler to generate code for the small, medium, or large code model respectively, where the `medium` value specifies that the program is to be linked in the lower 2*GB* of the address space and to place small symbols there as well, where symbols with large size are put into large data or bss sections and can be located above 2*GB*. The `large` code model allows the compiler to make no assumptions about addresses and sizes of sections. By default, the Nasa Parallel Benchmark Suite has this option flags set to `-mcmodel=medium` and so we tuned our compiler framework to enable the same flag.

During compilation of the larger class input problem sizes of NPB, an error occured consisting of the message: `relocation truncated to fit`. It was discovered that the combination of compiling NPB statically and compiling the large input problem size data sets (larger than 2*GB*) into the binary, triggered a relocation overflow error at link time. We discovered that applications built with `-mcmode=medium` should be linked against shared libraries. Since Popcorn Linux does not support shared libraries, we had to rely with the second solution. We fixed this by rebuilding all the libraries we used specifying the same memory model option as the application. Therefore both the libraries and application were compiled with the same memory model. This was an easy problem to solve, but interesting nonetheless. This issue is referenced and resolved in

an article at Intel's Developer Zone website [17].

This chapter presents the runtime support implemented for the Xeon - Xeon Phi platform. We describe the challenges associated with each heterogeneous-ISA platform. We then go into implementation details for various nuances discovered along the way while debugging to currently have flawless execution for the benchmarks utilized for this work. We also present compiler modifications related to the runtime support here instead of the compiler partitioner chapter for easier understanding.

6.1 Design

Even in the case of an overlapping-ISA platform a variety of changes were needed to enable seamless migration for any target application. The following sections will discuss each change that was necessary for Xeon - Xeon Phi in detail and why it is needed for each library. These changes include modifying the following libraries:

1. *libc* (including pthreads to enable multi-threading)
2. *libm*
3. *libomp*

Because of initial personal design decisions along with taking the path of least resistence to implement a working prototype, a few design decisions were made for us. For example, because the Intel Manycore Platform Software Stack (MPSS) version that was shipped with our Xeon Phi utilized GNU libc 2.13, we had to use it. We adapted all of our code based on this and also performed modifications to run the system on this version. Antonio Barbalace also ported our Popcorn Linux kernel from Linux kernel 2.6.38.8 to 3.2.14 which is the Intel baseline Linux kernel.

Significant work went to create a compliant methodology for discerning appropriate implementations of functionality provided by the Math library for a given architecture. In a heterogeneous-ISA binary is essential that a function compiled for a particular architecture is *only* linked to functions compiled for the same architecture. As noted earlier, the most divergent characteristics between the Xeon and Xeon Phi occur in the way ISA extensions are handled. Both the Xeon and Xeon Phi represent vector processors within their own respective instruction set, capable of operating on instructions that consist of one-dimensional arrays of data (i.e., vector instructions). However, the Xeon and Xeon Phi vector sizes are different, with the Xeon possessing smaller vector size compared to the Xeon Phi. This in turn has a significant impact on the performance of mathematical computations; therefore it should be no suprise that the Xeon Phi excels at SIMD operations compared to the Xeon even if it has a lower operating frequency.

Finally, not much design per se was needed for the OMP library. Instead, mostly pure engineering effort was needed to make it compatible on a heterogeneous-ISA system.

6.2 The C Library, *libc*

In this work *libc* is GNU *libc* (*glibc*). Since the C Library consists of a large spectrum of core functionalities, it was imperative to have a working implementation for Popcorn Linux applications. Thoughout the development process few but necessary changes were made to the *glibc* library. It should be noted that though the Math Library, *libm*, is traditionally a part of *glibc*, the changes performed were quite unique and therefore called for their own section which follows.

One notable change performed in libc, in order to improve the performance of the page fault coherence algorithm, was to pad the `pthread_mutex` data structure to $4kB$ to ensure that only a single instantiation can exist on a page. As supporting concurrent writes to a single page across multiple kernels is expensive, false sharing should be avoided. Often multiple mutexes are created together and this modification prevents multiple mutexes from existing on the same page.

In addition, several changes were made to adapt *glibc* for Popcorn Linux in terms of pthreads. These changes were performed in the native POSIX thread library (also a part of *glibc*).

Other changes included stripping the library down to the bare minimum in terms of functionality. These modifications were done to simplify the prototype evaluation and debugging process.

The fact that the needed changes performed were minimal demonstrates the "transparency" and wide applicability of our approach.

6.3 The Math Library, *libm*

The design constraint of using a single FAT binary presented a unique challenge to execution of external library functions. Typically it is known that an executable can only have one instance of a given symbol when being compiled and executed in order to avoid ambiguity, otherwise an exception is raised and linking is aborted. As mentioned in Section 5.2.5 describing Compiling and Linking for Xeon - Xeon Phi, external libraries such as the Math library, are compiled for each architecture on the system and then linked in with the appropriate object file (Xeon implementation to Xeon object file, and similarly Xeon Phi to Xeon Phi), each of which forms an executable that is part of the single FAT heterogeneous-ISA binary. The question that immediately comes to mind when known that several implementations of the same function are present is how are symbol name clashs avoided?

In order to account for incompatibilities between different architectural implementations for the various math library routines, it was needed that each compiled executable (one for each architecture as part of the FAT heterogeneous-ISA binary) had only its architectures respective implementation linked in, for Xeon and Xeon Phi respectively. This isolation was acheived as a two-part solution:

1. The library source would contain both implementations in the same archive file.

2. The compiler would perform additional code transformation to:

    - Provide prototypes of all implementations of the various Math library routines with unqiue names to provide distinguishability.
    - Include additional macro logic in the source of the target application in order to associate to correct implementation of each of the math library routine during compile time.

Creating a heterogeneous Math library was a daunting task, combining C implementations, assembly, utilizing macro, and performing Magic Number conversions in file meta data to combine it all together into one functional entity. Take a step back to the external library source code itself. Libraries are filled with optimized routines; most are written in assembly and therefore are architecture dependent. For the experimental evaluation for Popcorn, *libm* was needed for most benchmarks and therefore needed to be revamped to work with heterogeneity, in our case Xeon (x86) - Xeon Phi (x86 extended). The way in which a heterogeneous library was created for Popcorn is as follows.

For each respective implementation of Math Library functions, we changed the function names to include a prefix (`__xeon_popcorn_` for the Xeon implementations and `__phi_popcorn_` for Xeon Phi implementations) to be easily distinguishable and prevent symbol name clashing. Similarly to how a heterogeneous-ISA binary is created, the Math Library implementations were compiled using their respective native compiler (gcc for Xeon and icc *k1om* for Xeon Phi). Then the same scripts that modified the Xeon Phi partition file Magic Number meta data is also applied to the Xeon Phi compiled Math function's object (*.o) files meta data. Once the Xeon Phi implementation is compiled to object files, each file's Magic Number bytes were overwritten. Magic Numbers [40] are a block of byte values used to designate a file type in order for applications (including gcc) to be able to detect whether or not the file the application plans to parse and consume is of the proper format. Initial compilation debugging resulted showed that a binary with different Magic Number than what is assigned for the target architecture results in both linking (for when trying to link in a library with an object file compiled for a foreign architecture) and runtime errors (the binary itself has foreign architecture Magic Number values) presumably an expected error to protect the user. It was resolved soon after that the "Magic Numbers" values are 0x3E and 0xB5 for Xeon and Xeon Phi, respectively. The Xeon Phi compiled objects Magic Number byte's are overwritten to be 0x3E so that the linking stage of the compilation process of the heterogeneous binary succeeds. Finally, after the magic byte swap has occurred, all object files from both Xeon and Xeon Phi are archived into a single library. Then all object files are compiled into a heterogeneous-ISA Math Library archive, now containing both implementations. At the same Homogeneous-ISA of libraries are also created for extended accessibility of running homogeneous-ISA binarys as well. An illustration of the math library compilation process is depicted in Figure 6.1.

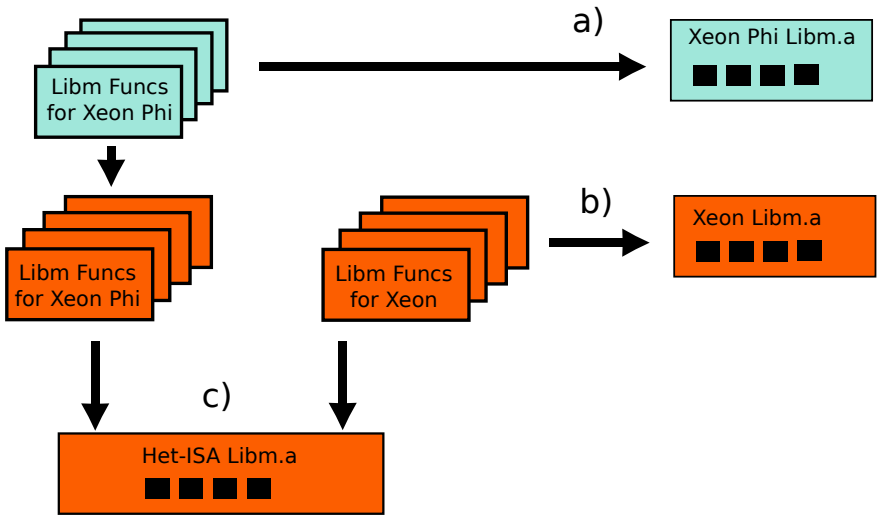

Figure 6.1: This depicts how various versions of the Math library are formed. Boxes in cyan depict object files compiled using *k1om* gcc for Xeon Phi, while boxes in orange depict object files compiled using regular gcc for Xeon thus having a different magic number than the object files compiled for Xeon Phi. First in a) Xeon Phi implementations are collected into a Homogeneous-ISA Xeon Phi Math Library, then Xeon implementations are collected into a Xeon Math Library in b). Finally before combining to create a Het-ISA Math Library c), the Xeon Phi object files meta data are processed to match the Xeon's object file Magic Number meta data.

The second part of our solution lets us effectively connect the correct implementation of a math function to the correct compilation pass, when compiling for each architecture. Our solution consists of including addition macro logic in the source code of the target application in order to achieve correct association of implementations of each of the Math library routines. The following code snippet shows how this requirement was implemented:

```
#ifdef __xXEON
extern double __xeon_popcorn_pow(double x,double y);
#define pow __xeon_popcorn_pow
#elif def __xPHI
extern double __phi_popcorn_pow(double x,double y);
#define pow __phi_popcorn_pow
#endif

#ifdef __xXEON
extern double __xeon_popcorn_sqrt(double x);
#define sqrt __xeon_popcorn_sqrt
#elif def __xPHI
extern double __phi_popcorn_sqrt(double x);
#define sqrt __phi_popcorn_sqrt
#endif
```

Figure 6.2: Distingushing Math Library Implementations (`pow` and `sqrt`)

The code in Figure 6.2 shows how we provide per-cpu optimized versions of `pow` and `sqrt`. **NOTE:** The rest of the math library routines (i.e., exp, fabs, etc.) were wrapped in a similar manner.

As can be observed there exist two disquishing macros `__xXEON` and `__xPHI` which can are selected during compile time when creating the object files of the target application for each architecture. During compile time, the script that creates the executables for each architecture passes in a defining macro option `-D` along with which architecture it is currently compiling for (e.g., the script would pass `-Dx__XEON` when compiling the Xeon executable and `-D__xPHI` when compiling the Xeon Phi executable) which collectively form the heterogeneous-ISA binary. Finally, this structure can be easily be extended to cater for additional architectures.

6.4 The OpenMP Library, *libiomp*

OpenMP, the main parallelism method supported by and benchmarked for the Xeon - Xeon Phi heterogeneous platform, is provided by libiomp, the Intel OpenMP runtime library. However, it is not provided to be compatible with heterogeneous platforms right out of the box, so this is the second library that had to be modified and rebuilt to become compatible with the Xeon - Xeon Phi platform.

Many implementations of OpenMP exist (e.g. GOMP, Intel's libiomp, LLVM's, etc.), however in our test case we are limited to using Intel's version of OpenMP (libiomp) as it is the the only version compatible with Xeon Phi's tools. We automated the process to compile an unstripped version of this library for Xeon Phi using GCC that targeted *k1om*. This library is open-source. More information can be found in [6].

One issue discovered only after initial benchmarking had begun was a problem in the accuracy of the benchmarks. Even though all migration issues had been fixed, when the NPB suite was run, most of the benchmarks "failed" verification. Isolating the problem, it was found that multi-threading was causing a race condition, giving incorrect results due to a lock, because each architecture had its own respective implementation. To remedy this, a simple macro (`__MIC` was used to toggle between implementations when compiling the OMP library. If __MIC was defined, it meant that the Xeon Phi implementation was being compiled, otherwise the Xeon implementation was used. Figure 6.3 shows this code modification. Two assembly files were created with each architectures respective implementation of `__kmp_test_then_add_real64`. For better readibility to see changes made in the code base, the suffixes `popcorn_phi` and `popcorn_xeon` were added.

```
#ifdef __MIC
#define LOCK_FUNC2 __kmp_test_then_add_real64_popcorn_phi

extern kmp_real64 LOCK_FUNC2 ( volatile kmp_real64 *p, kmp_real64 v );

void __kmpc_atomic_float8_add_phi( ident_t *id_ref, int gtid, kmp_real64 * lhs
    , kmp_real64 rhs ) { 0; ; if ( (0) && (__kmp_atomic_mode == 2) ) { if (
    gtid == (-5) ) { gtid = __kmp_get_global_thread_id_reg(); };
    __kmp_acquire_atomic_lock( & __kmp_atomic_lock, gtid ); (*lhs) += (rhs);
    __kmp_release_atomic_lock( & __kmp_atomic_lock, gtid );; return; }
LOCK_FUNC2( (lhs), (+ rhs) );
}

#else
#define LOCK_FUNC1 __kmp_test_then_add_real64_popcorn_xeon

extern kmp_real64 LOCK_FUNC1 ( volatile kmp_real64 *p, kmp_real64 v );

void __kmpc_atomic_float8_add_xeon( ident_t *id_ref, int gtid, kmp_real64 *
    lhs, kmp_real64 rhs ) { 0; ; if ( (0) && (__kmp_atomic_mode == 2) ) { if (
     gtid == (-5) ) { gtid = __kmp_get_global_thread_id_reg(); };
    __kmp_acquire_atomic_lock( & __kmp_atomic_lock, gtid ); (*lhs) += (rhs);
    __kmp_release_atomic_lock( & __kmp_atomic_lock, gtid );; return; }
LOCK_FUNC1( (lhs), (+ rhs) );
}

#endif
```

Figure 6.3: Distingushing OMP Library Implementations for `__kmp_test_then_add_real64`

6.5 Putting It All Together

Finally, at this stage we have all the profiling data for optimal execution, *migration-ready* partitioned source code, and finally the core dependencies provided by libraries. The compiler frame-

work described in this work provides all the necessary pieces to create a heterogeneous-ISA binary for the Popcorn Linux Xeon - Xeon Phi platform.

Chapter 7

Experimental Evaluation

For our evaluation of the compiler framework for Popcorn Linux on the Xeon - Xeon Phi platform, we measured the speedup achieved compared to running homogeneously using only one of the architectures (i.e., Xeon or Xeon Phi), an OpenCL version, or utilizing Intel's Language Extension for Offload (LEO) [37]. The Xeon - Xeon Phi platform results have been published at the ACM 2015 EuroSys conference [4].

7.1 Hardware

All experiments were run 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. 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. Refer to Saif Ansary's MS thesis for more information on this configuration decision. This configuration was used for all experiments presented herein.

7.2 Software

The Popcorn Linux prototype platform 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. We backported part of the namespace code from Linux 3.8 since the namespace code on Linux 3.2.14 was not complete. 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 for statically-compiled NPB applications. The Linux distribution used in the experimental system was CentOS 6.5.

7.3 Results

The goal of this evaluation is to show that our system software enables optimal exploitation of a heterogeneous-ISA platform, automatically. We seek to provide evidence that our system software consisting of the compiler framework, when paired with a heterogeneous-ISA OS, Popcorn Linux, is able to produce advantageous compute kernel code placements on available cores for the Xeon - Xeon Phi platform. This section compares our system software approach to the most predominantly used mature programming paradigms for offloading as well as homogeneous approaches.

Just to reiterate, our heterogeneous-ISA FAT binaries were compiled using a combination of GCC and ICC (*k1om*) for Xeon and Xeon Phi partitions respectively.

To compare to homogeneous-ISA implementations of the chosen benchmark applications (compiled to be run natively on either Xeon or Xeon Phi only), we did not modify any code but simply compiled with the appropriate compiler with maximum optimizations enabled.

To compare our work to Intel's LEO, we manually port the same set of benchmark applications and insert the additional offloading paradigms as needed. We insert the LEO calls at the same code locations as they are inserted in the OpenCL implementation.

Finally, we also included the OpenCL version of the SNU NPB benchmark suite as an additional competitor.

7.3.1 Running with Code Analysis

The following Figures 7.1 through 7.15, depict the execution times (in seconds) of various benchmarks from the NPB suite. Specifically we include the EP, IS, SP, and BT benchmarks in this work. For each benchmark, we report the execution time for different input data sizes (denoted as Class A, Class B, and Class C catagories) using various combinations of available cores on the Xeon Phi processor (4, 8, 57, 114, and 228). For more precise results, each data point was averaged over 10 separate runs; the maximum deviation per sample never exceeded 11ms. Note that in these graphs, lower is better.

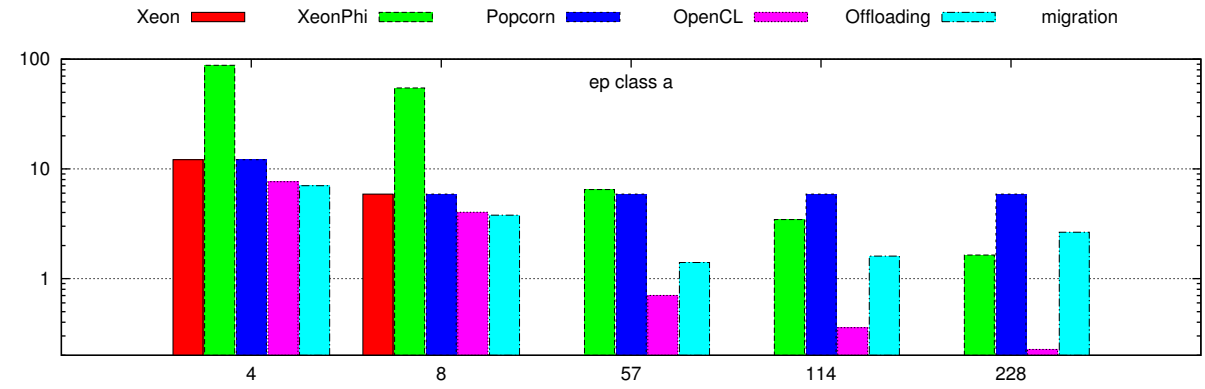

Figure 7.1: EP Class A

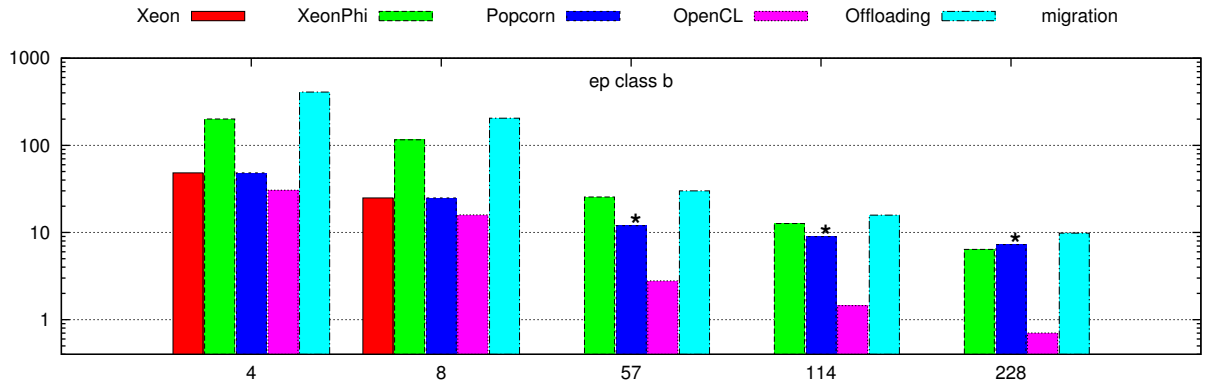

Figure 7.2: EP Class B

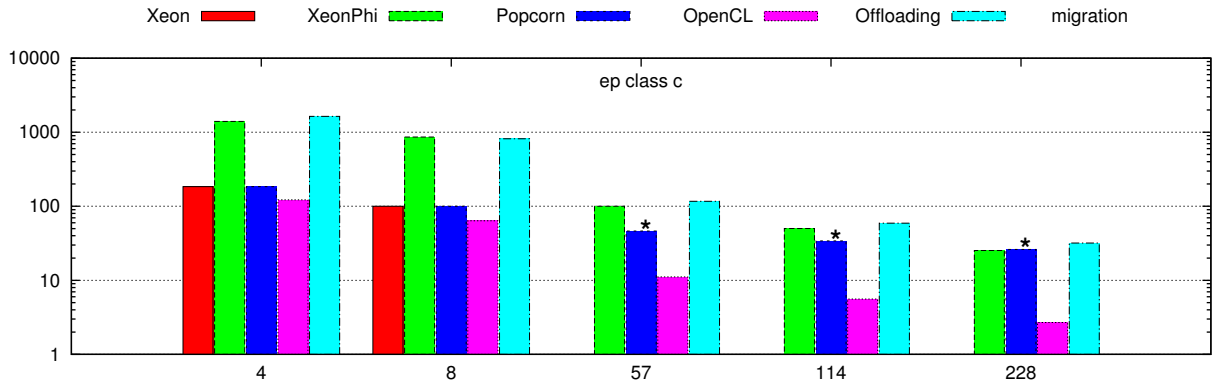

Figure 7.3: EP Class C

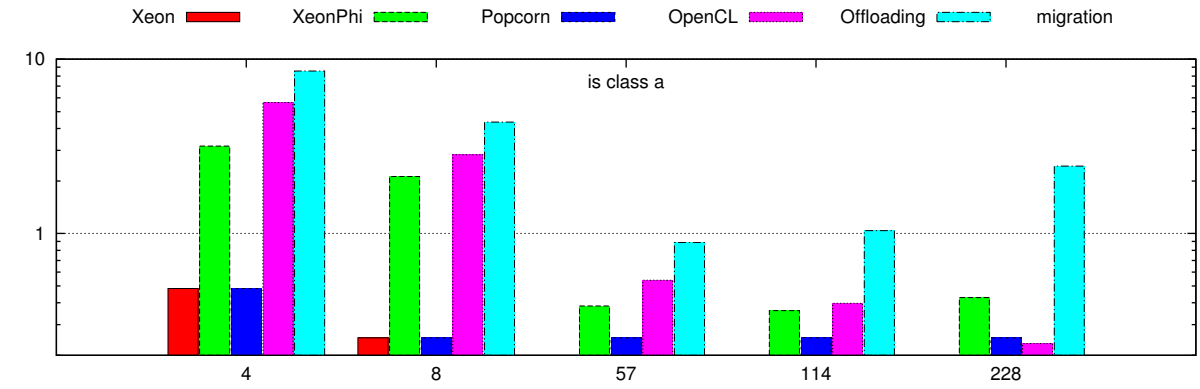

Figure 7.4: IS Class A

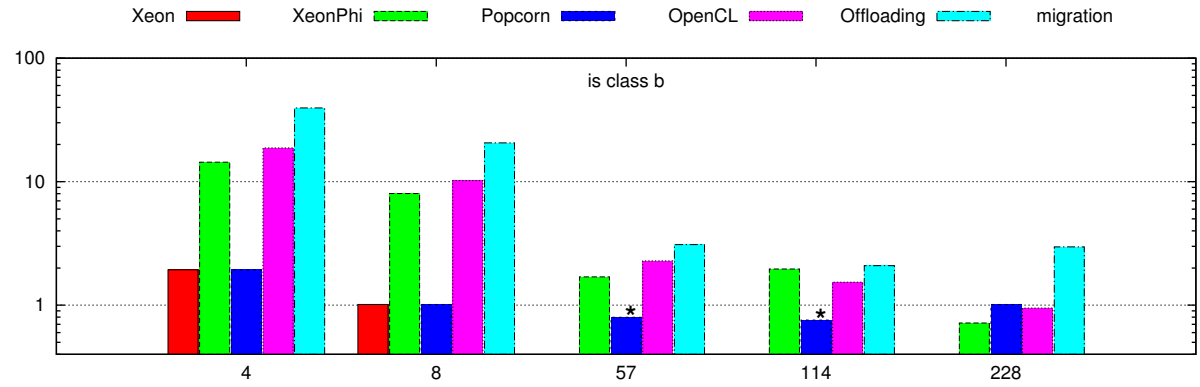

Figure 7.5: IS Class B

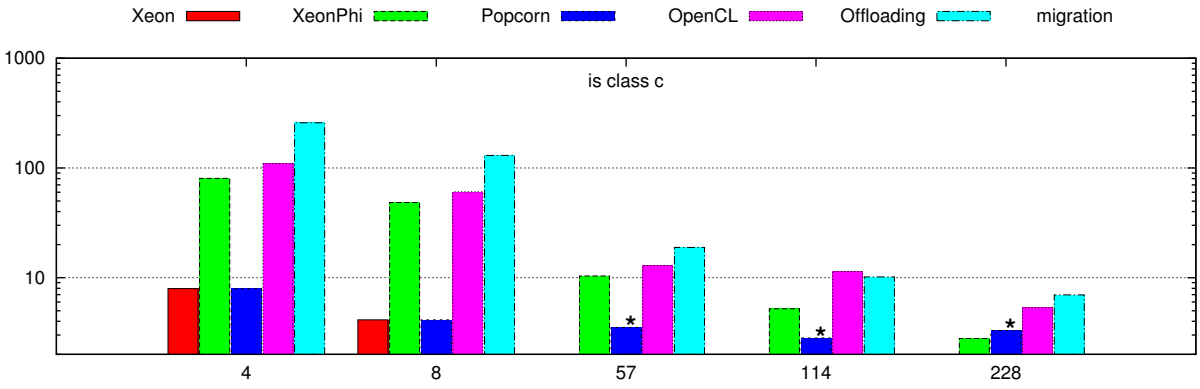

Figure 7.6: IS Class C

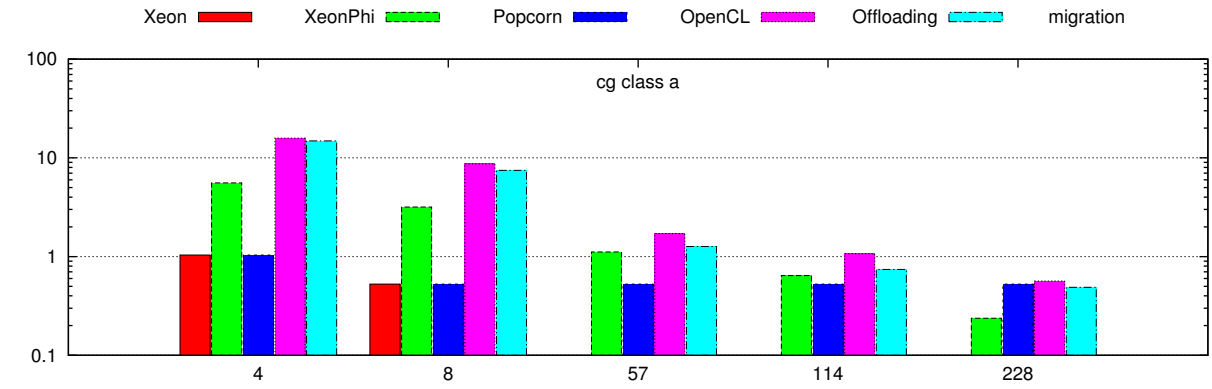

Figure 7.7: CG Class A

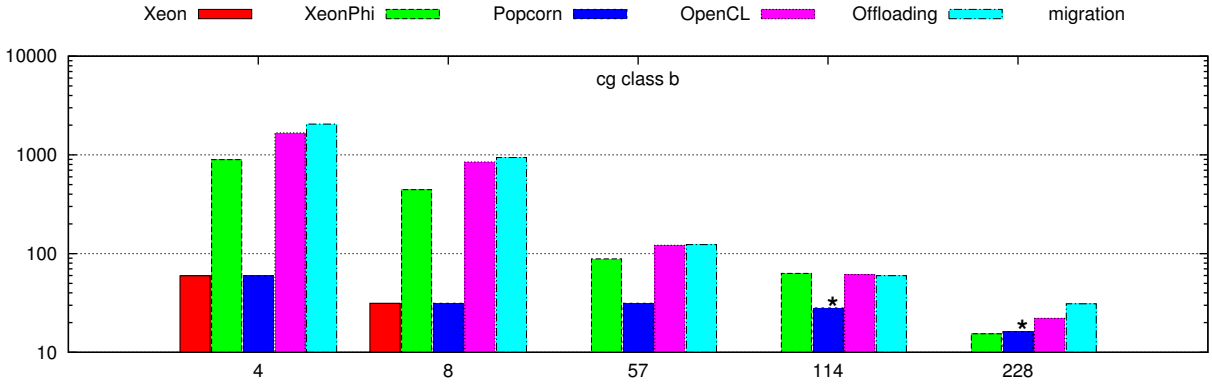

Figure 7.8: CG Class B

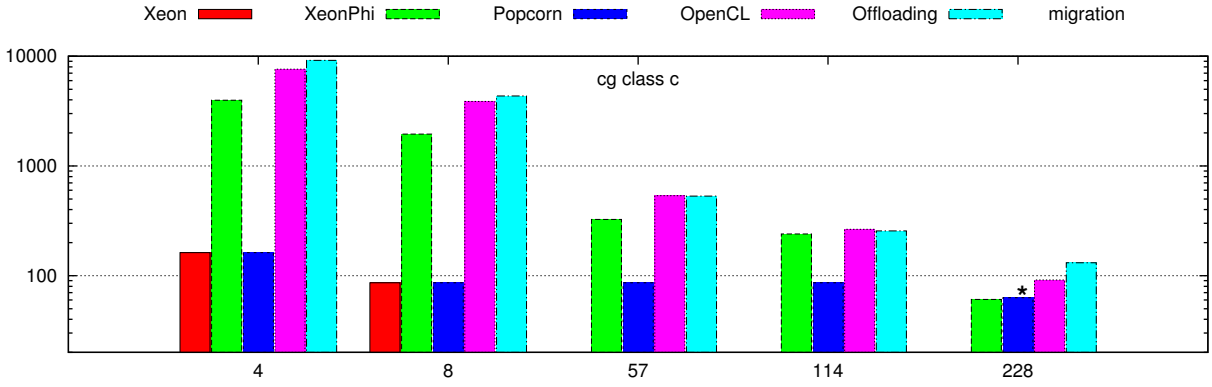

Figure 7.9: CG Class C

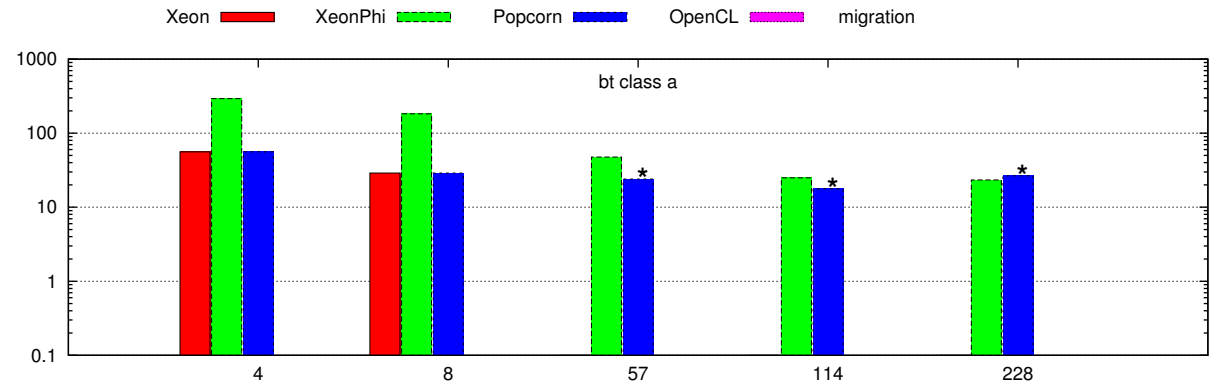

Figure 7.10: BT Class A

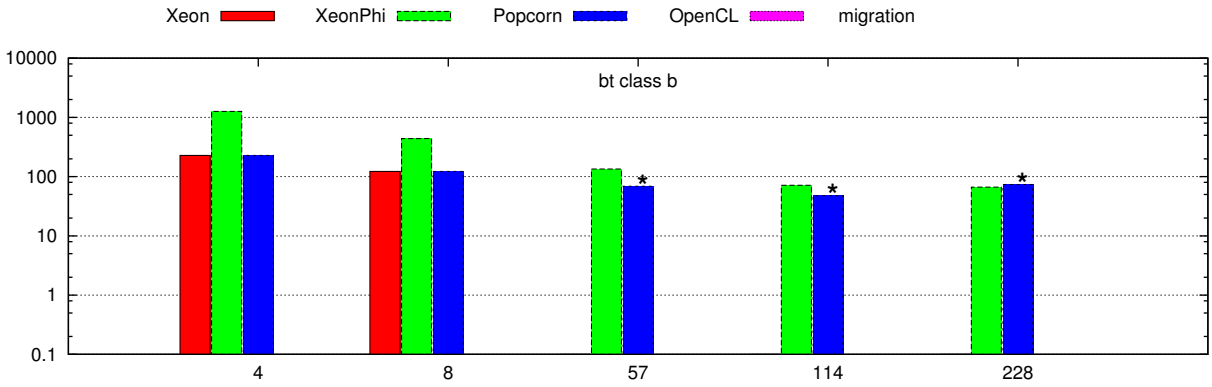

Figure 7.11: BT Class B

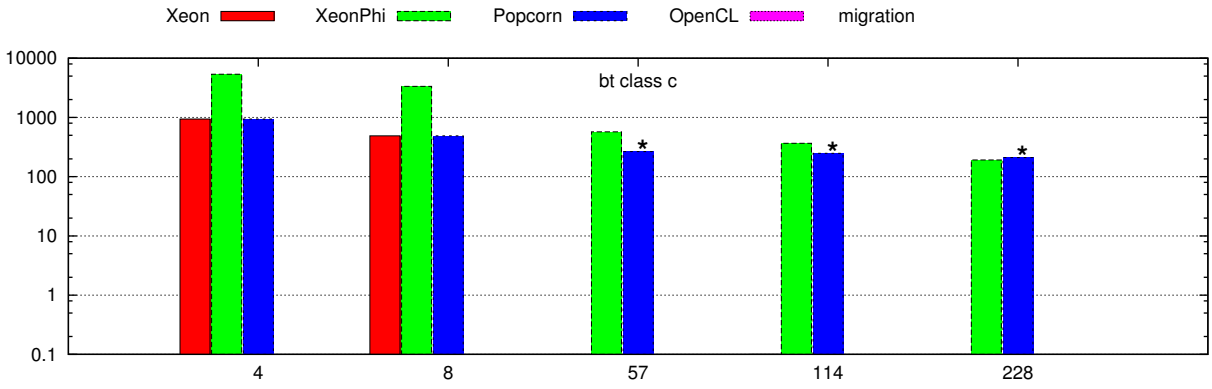

Figure 7.12: BT Class C

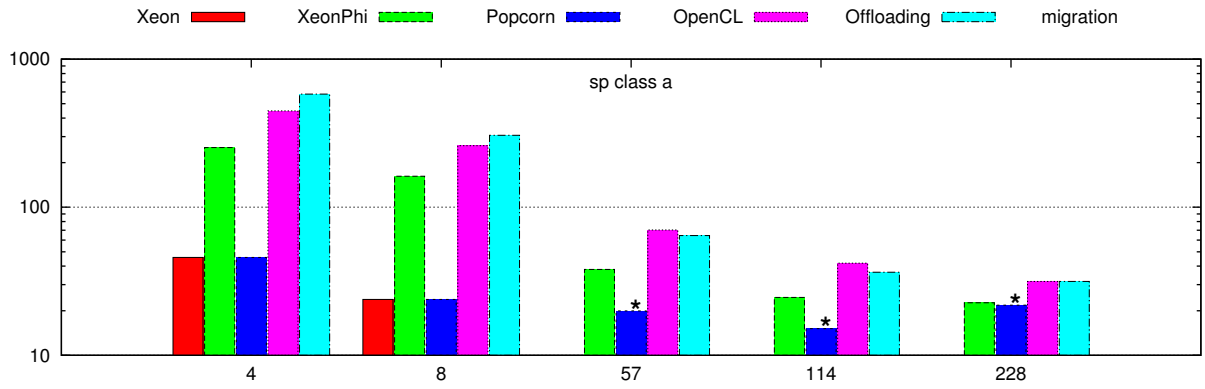

Figure 7.13: SP Class A

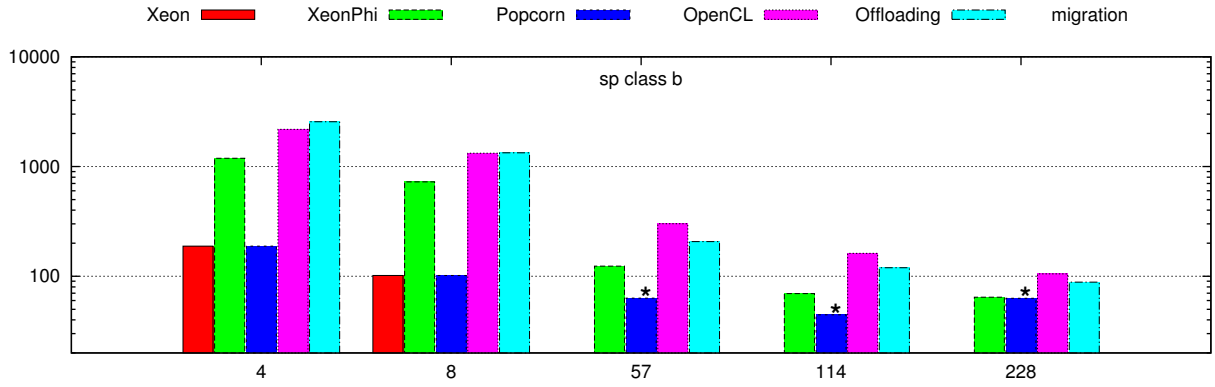

Figure 7.14: SP Class B

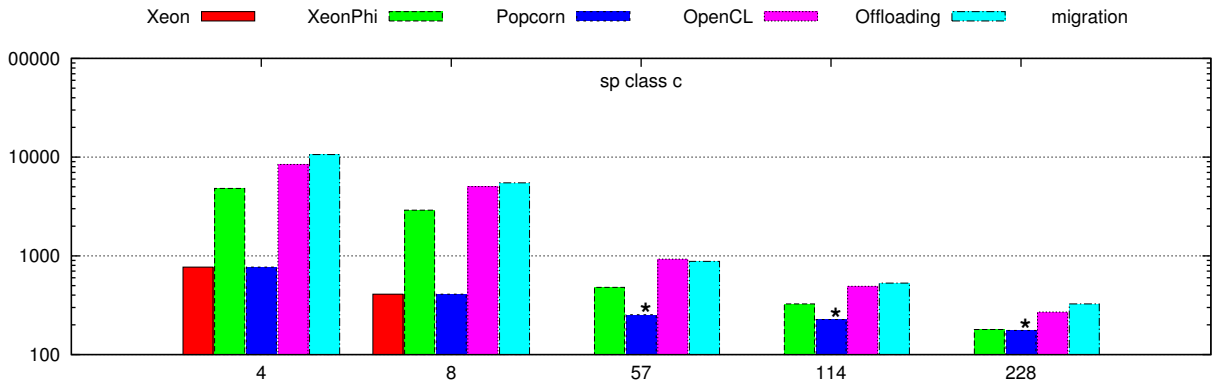

Figure 7.15: SP Class C

As denoted in the graph legend, an asterik above a given bar denotes that migration occured at least once during execution between Xeon and Xeon Phi for the given configuration/benchmark. Other

benchmarks from the NPB suite (i.e. UA) never migrate between Xeon and Xeon Phi for any input data size 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 (Figure 4.4), determines that the performance benefits outweigh the communication overheads due to migration. From the figures, it is clear that due to the large difference (almost 2x) in the clock frquency between the Xeon and Xeon Phi, the compiler frame never idicates to migrate 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 made available on the Xeon Phi, the behavior changes for each benchmark.

Figures 7.13-7.15 shows that for the SP benchmark, the partitioner decides to always migrate for 57, 114, and 228 threads, showing heterogeneous execution is more beneficial over both Xeon and Xeon Phi native execution for higher thread counts. With 57 threads, Popcorn is up to 53% and 46% faster than native execution on Xeon and Xeon Phi (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 and Xeon Phi (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 application Profiler/Partitioner part of the compiler framework, with its decision to migrate execution to the Xeon Phi for highly parallel, and therefore favorable, functions within the various NPB benchmarks.

Since the migration points are in the same locations in both the Popcorn and LEO implementations for all the benchmarks, this allows for a direct comparison of Popcorn and the LEO models and their respective performance. However, Popcorn is always faster than its competitors. In particular, Popcorn is up to 3.5 times faster on 57 cores for Class C benchmarks.

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 implementated in user-space between the Xeon and Xeon Phi. The same performance trends and conlusions 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 application Profiler of the compiler framework decides to never migrate 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 only on 8 Xeon cores.

On the other hand, OpenCL and LEO versions of the NPB benchmarks always migrate because these versions do not have any migration policy mechanism that can select which device or architecture to perform its' computations on. Therefore, the OpenCL and LEO versions are (in most cases) slower than Popcorn, with up to 12 times slowdown for IS Class A using 228 cores.

The EP Class B and Class C benchmarks (Figures 7.2-7.3 have the same trend as the SP Class B and Class C benchmarks (Figures 7.14-7.15). 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. Benchmark EP represents a special case for OpenCL in which Intel's analysis tool allowed us to discover its exploitation of extremely efficient mathematical fucntions that could not be used for the other versions of the benchmark, hence Popcorn's performance is slower.

An interesting case is observed in the CG benchmark, where 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 the application Profiler that is part of Popcorn's compiler framework. However, the OpenCL and LEO versions are not able to detect this and experience up to 6.2 times slowdown for 57 cores, Class C.

From the NPB suite, IS is its 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). The publication hypothesizes 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 (the IS benchmark contains only integer computations).

7.3.2 Lessons Learned

This evaluation illustrates that compute and memory intensive benchmarks written for SMP and running on a replicated-kernel OS with DVSM (Popcorn Linux) can take significant advantage of a heterogeneous platform when profiled and compiled with our Profiler and Partitioner. Additionally, Popcorn's system software infrastructure is more performant than a user-space offloading software infrastructure many developers are still in the midst of today. However, for short-running benchmarks there is no benefit in using due to the high communication overheads incurred when migrating between architectures. Moreover, Popcorn does not always have low migration overheads for high thread counts. This performance profile is application-dependent.

7.3.3 Framework Overheads

Once the resulting applications performance had been evaluated it was also of importance to see the impact of adopting this methodology. There are two interesting angles to investigate here.

7.3.4 Profiling Time

The first is to evaluate the time needed to perform profiling in order to have an optimal partitioning for a target application. Depending on the complexity of the application, the time will vary. Since the Profiler adds annotations to memory accesses, it can be naturally assumed that memory-intense applications will be affected significantly more than other applications such as ones that are compute-bound. A table, Table 7.1, shows the slowdown incurred to run an NPB benchmark for Class W which has been annotated for profiling by the profiling tool described in Chapter 4. The slowdown incurred is calculated by comparing the runtime of the annotated application to the runtime of a natively compiled vanilla version of the same application; it should also be noted that the same input problem size is used for both runs.

Benchmark	Avg. Runtime (s)	Avg. Profiling Time (s)
BT	2.07	2136.06
CG	0.23	208.43
EP	3.17	109.40
IS	0.05	44.79
MG	0.16	216.60
SP	4.48	2578.45

Table 7.1: Percentage Increase in Time to Perform Profiling Compared to Native Execution for Class W (The second smallest data size)

As can be observed from Table 7.1, the EP benchmark is affected the least from being annotated, compared to MG and BT which are up to 40x more affected due to being annotated. This bias is due to the number memory accesses performed during these benchmarks. Further inspection of the code source of MG and BT reveals that these benchmarks use fairly large data structure consisting of many elements whereas EP has barely any and is primarily compute-bound. The rest of the benchmarks, CG, IS, and SP fall somewhere in the middle of the road between these two extremes of data structure usage, and the resulting data reflects this trend.

Finally, an added benefit of the Profiler is that it only needs to be be run once to gather the information it needs to provide an optimal partitioning; this justifies the seemingly high overhead imposed by this solution as the information can be additionally reused to provide optimial partitionings on any heterogeneous architecture configuration. As a future work, it would be interesting to investigate ways of making the profiling phase faster while still providing enough information to make an informed and correct decision for an optimal partitioning.

7.3.5 Executable Size

The second interesting angle to explore and evaluate is to compare the difference in size of the binary footprint produced with compiling heterogeneous-ISA applications using the compiler framework described in this work. Ideally, the difference in footprint size should not differ significantly than the size that results by compiling the same application natively for a given platform. In reality though, space is cheap and abundant, therefore not a chief concern in most domains (It should be noted this fact is quite the contrary in the embedded systems domain, however the embedded domain is out of scope for this work).

In the following table, Table 7.2 gives a detailed report of the additional executable compilation configurations that are possible with our framework for a given application. Specifically, this table lists the percentage increase in binary footprint size with respect to a native, statically compiled vanilla version of NPB benchmark (hereafter refered as the native compilation). The compilations we compared are: the partitioned version of the NPB benchmark compiled statically to run homogeneously on Xeon only (denoted as *Xeon Static*), the partitioned version of the NPB benchmark compiled statically to run homogeneously on Xeon Phi only (denoted as *Xeon Phi Static*), and the partitioned version of the NPB benchmark compiled statically to run heterogeneously (it is the cumalative size of the executables that form the FAT heterogeneous-ISA binary, denoted as *Heterogeneous*). During this analysis, we tested various configurations for both homogeneous and heterogeneous-ISA scenarios. Out of these two scenarios we tested differences in multi-threading for 4, 8, 57, 114, and 228 concurrent threads. Only the 8, 57, and 228 thread configurations are listed because both the 4 and 8 thread configurations and the 114 and 228 thread configurations had the exact same binary footprint size. In addition, it should be noted that in the Heterogeneous-ISA configuration, multi-threading was never set to use 4 or 8 threads because this would be sub-optimal for exploiting the parallelism offered by the Xeon Phi coprocessor, thus these fields have been marks as *N/A*. A similar explanation is true for not testing 57, 114, or 228 threads on the Xeon only configuration as the Xeon is hyper-threaded to efficiently handle a maximum of 8 threads; configuring to use more than 8 threads would cause significant degradation of performance due to the huge increase in context switchs needed to be performed to maintain a larger amount of threads.

	Xeon Static	Xeon Phi Static	Heterogeneous
BT (A) 8	-1.64%	182.30%	N/A
BT (A) 57	N/A	182.32%	101.44%
BT (A) 228	N/A	182.31%	101.44%
BT (C) 8	-1.64%	183.85%	N/A
BT (C) 57	N/A	183.87%	103.15%
BT (C) 228	N/A	183.86%	103.15%
CG (A) 8	-7.46%	179.19%	N/A
CG (A) 57	N/A	179.21%	87.83%
CG (A) 228	N/A	179.20%	87.83%
CG (C) 8	-7.41%	179.36%	N/A
CG (C) 57	N/A	179.38%	87.94%
CG (C) 228	N/A	179.37%	87.94%
EP (A) 8	-14.28%	165.46%	N/A
EP (A) 57	N/A	165.48%	71.62%
EP (A) 228	N/A	165.47%	71.62%
EP (C) 8	-14.31%	165.39%	N/A
EP (C) 57	N/A	165.41%	71.56%
EP (C) 228	N/A	165.40%	71.56%
IS (A) 8	-0.35%	229.68%	N/A
IS (A) 57	N/A	229.73%	99.39%
IS (A) 228	N/A	229.71%	99.39%
IS (B) 8	-0.34%	229.72%	N/A
IS (B) 57	N/A	229.77%	99.42%
IS (B) 228	N/A	229.75%	99.42%
IS (C) 8	-0.35%	229.71%	N/A
IS (C) 57	N/A	229.75%	99.41%
IS (C) 228	N/A	229.73%	99.41%
MG (A) 8	-8.98%	175.20%	N/A
MG (A) 57	N/A	175.21%	98.13%
MG (A) 228	N/A	175.21%	98.13%
MG (B) 8	-8.98%	175.20%	N/A
MG (B) 57	N/A	175.22%	98.12%
MG (B) 228	N/A	175.21%	98.12%
SP (A) 8	0.52%	195.24%	N/A
SP (A) 57	N/A	195.26%	121.33%
SP (A) 228	N/A	195.25%	121.33%
SP (C) 8	-2.19%	193.48%	N/A
SP (C) 57	N/A	193.50%	119.58%
SP (C) 228	N/A	193.49%	119.58%

Table 7.2: Percentage Increase in Size of Binaries compared to x86 Vanilla Static Compilation

Just to shed light on the difference between dynamic and static compilations for the various configurations, this data point is also included. It should be no surprise that a dynamic compilation is much slimmer than a static one as not much information is stored in the actual binary since the library functions are fetched on demand during runtime. It was surprising to see that the Xeon Static compilation was actually smaller than the native compilation for not one but all the benchmarks to a varying degree. Before we collected this data, I predicted that out of all compilations, the Heterogeneous compilation would be the largest. This hypothesis was wrong; it was interesting to note that the Xeon Phi Static compilation had the largest footprint being larger than both the native compilation and the Heterogeneous compilation. In some cases the Xeon Phi compilation's footprint increase was up to 2 times larger than the increase for the Heterogeneous compilation. In conclusion, our FAT heterogeneous-ISA compilation adds only up to double the size of vanilla binary for the applications proposed – therefore the resulting space overhead is insignificant and in some cases it can even be smaller than a given compilation as is the case compared to a static Xeon Phi compilation.

Chapter 8

Conclusion

In this thesis, we have shown that emerging OS-capable heterogeneous platforms can run applications written for homogeneous-ISA multiprocessors; this is enabled by the presented hetereogeneous-ISA compiler framework, consisting of a Profiler, Partitioner, and runtime support. The framework successfully introduces heterogeneous-ISA migration during execution of compute and memory bound applications running on the Popcorn Linux operating system while requiring no rewriting from the developer. Using the Profiler leads to significant performance improvements on the Xeon - Xeon Phi platform and save time spent on application development by being an automatic process. Our evaluation showed that exploiting a heterogeneous-ISA platform such as the Xeon - Xeon Phi is more performant than running the same application homogeneously (exclusively) on one processor. Given the diversity present in heterogeneous-ISA platforms, we successfully provided runtime support for our experimental platform porting various core libraries, which enable heterogeneous execution for most applications. We also show that applications automatically processed and compiled with our toolchain benefit from the heterogeneous-ISA environment, performing better than manually partitioned programs. Moreover, the results demonstrate that the proposed programming model is more efficient and performant than other mature user-space "offloading" paradigms.

8.1 Contributions

This work consists of the following contributions:

1. **We implemented an application Profiler that analyzed memory access patterns of a given application and generates the most advantageous partitioning scheme for the Xeon - Xeon Phi platform.** The Profiler examined preliminary compiled object code and annotated all instances of memory access instructions. It then ran the application, keeping track of these annotations to create a graph representing the cost of migration across architectures between function calls. The logic for obtaining an optimal partitioning for a given

target application is implemented utilizing the ST-Min Cut algorithm. Once an optimal partitioning is found it is fed as input to the application partitioner.

2. **We developed an application partitioner that would prepare and compile heterogeneous-ISA binaries for the Xeon - Xeon Phi platform.** Using the information obtained by the Profiler, the partitioner is then able to perform source-to-source code transformation on a target application in order to enable heterogeneous-ISA migration on the Xeon - Xeon Phi platform. The second part of the partitioner then compiles and links in any additional runtime support needed (libraries) to create a heterogeneous-ISA FAT binary.

3. **Using the Xeon - Xeon Phi compiler framework, we were able to accurately predict the cost of migration to effectively schedule worth-while compute kernels onto the Xeon Phi.** Since the compiler framework supports multi-threaded applications, we were able to evaluate the effectiveness of exploiting a heterogeneous-ISA platform for OpenMP applications, such as the NASA Benchmark Suite [9]. We indeed found that running a given benchmark heterogeneously showed a substantial speedup compared to its homogeneous counterpart.

4. **We additionally implemented core runtime components to support our heterogeneous platform.** Namely, we implemented a heterogeneous version of commonly needed libraries for the Xeon - Xeon Phi platform. These libraries included the C Library, *libc*, and the Math Library, *libm*. Modifications included porting functionality that was implemented in assembly for either architecture, restructuring functionality to support migration while maintaining semantics, as well as enabling heterogeneous-ISA migration of multiple threads to support parallelism models such as OpenMP. In the cases where low-level assembly needed to be changed to resolve architecture-specific implementations, we follow the x86 architecture's organization.

5. **We evaluate the effectiveness of the implemented compiler framework.** The work described in this thesis is compared to currently used practices in high performance computing to guage how well it compares. We compare our methodology to OpenCL and Intel's LEO as well as homogeneous implementations. This comparison gives a twofold benefit of showing the performance that can be exploited by utilizing heterogeneous-ISA platforms as well as proving the viability of the compiler framework compared to traditional practices. We also evaluate the overheads associated with using our compiler framework to give a complete view of how this approach compares against competitors.

Our evaluations show that together with the compiler framework, Popcorn is up to 61% faster executing applications heterogeneously compared to native execution on the Xeon or Xeon Phi. For compute and memory-intensive benchmarks, running them on Popcorn can show up to a 52% performance improvement compared to the most performant natively executing benchmark. In addition, Popcorn applications outperform the Intel LEO model's implementation for all problem input sizes, partiticularly, being up to 3.5 times faster for the largest input problem size (Class C) in the SP benchmark.

General trends from the data indicate that the Popcorn platform continues to improve in performance as more resources are added to the system, indicating that Popcorn would scale very well in the long run. It also helps that the compiler framework's cost model can adapt the migration decisions to a variable number of processor cores active. This allows Popcorn and the compiler framework to be up to 6.2 times faster than the OpenCL and LEO models. This proves that Popcorn's programming model can be implemented in kernel space together with a compiler framework that results in faster mechanisms compared to a user-space "offloading" programming model that requires user-level daemons.

The Profiler and the effective time it takes to produce an optimal partitioning for a target application greatly depend on the nature of it. It is a given that the Profiler will never be faster than the original target application itself. For non-memory-intensive applications, such as the EP benchmark, the Profiler runtime was 3 times as long (a 3x slowdown) to produce an optimal partitioning compared to the original target applications runtime. This overhead skyrockets for memory-intensive applications, such as the MG benchmark, taking 135 times as long as the original to produce an optimal partitioning.

Given that storage is no longer a major concern in most domains, the difference in footprint size of a heterogeneous-ISA binary compared to native homogeneous compilation is insignificant. Going into more detail, using the compiler framework proposed in this work for any given application at worst case produces a heterogeneous-ISA binary that is 230% larger compared to its natively compiled vanilla counterpart. In the best case, the heterogeneous-ISA binary is only 71.5% larger than the original. As a side note, it is was interesting to observe that a homogeneous Xeon compilation (the application is partitioned but can only run on Xeon) using our compiler framework actually resulting in a decrease in size, up to 14% smaller! We believe that this occured from partitioning tool, in that the reorganization of the application code (while still maintaining that it is semantically correct) influenced the optimizations to better pack the binary. However, this is a hypothesis and would need to be investigated further to determine a discernible cause.

Coming full circle, we feel that our compiler framework exceeds most other approachs attempting to break into the heterogeneous domains. Furthermore, the benefits that this compiler framework offers to the developer far outweigh the minimal impact and overheads it imposes mentioned earlier.

8.2 Future Work

Although we demonstrated several strong research contributions built upon the Popcorn Linux Operating System, there are additional improvements and implementation details that can further solidify the research.

8.2.1 Application Profiler

As stated in section 4.1, the Profiler is a priliminarly implementation and therefore has a lot of room for improvement:

- The Profiler is currently implemented to analyze and give results from a static point of view, it has no regard or control over an application's performance when additional applications are running on the same machine. A dynamic approach to scheduling applications would yield even better performance in the case of multiple concurrent applications.
- The Profiler is input problem size dependent and therefore must be rerun if the application's input problem size is changed. The Profiler was implemented initially catering to legacy applications where not much information would change over the course of time; this makes the one time analysis cost no big deal and an acceptable set-back. Ideally, the Application Profiler would be input problem size *independent*. Only requiring the analysis phase to be run once regardless of how many different input sets an application has.
- The time taken to perform analysis is quite substantial. Even if the profiling phase of analyzing a target application is a one time cost to the user, that time could be decreased. Specifically, a low hanging fruit for this challenge would be to optimize the way data accesses are kept track of especially in the case of memory being accessed serially as in the case of loops.

8.2.2 Heterogeneous-ISA Compiler Framework

Although we were able to successfully provide a solution for the Xeon - Xeon Phi platform, this is still a very specific platform configuration and thus the compiler framework provided is not portable to other heterogeneous-ISA systems. It should be recognized that heterogeneous computing, especially for the case of heterogeneous-ISA platforms, is a fairly young concept. While a wide variety of solutions to this challenge such as OpenMP, CUDA, etc. have been proposed and embraced by the high performance computing community, there is still no clear winner. In order for Popcorn Linux and the compiler framework presented in this work to accel, the need to converge to a common interface is key. We believe that when heterogeneous computing matures, this compiler framework along with Popcorn Linux will be a strong competitor as well as helping mature the field of heterogeneous computing itself.

In the mean time though, there are a number improvements that could further the innovation the compiler framework provides:

- Note that this partitioner is not production ready, as it is only able to support heterogeneous-ISA platforms featuring 64-bit x86. Ideally, in the future it should be able to support any given combination of architectures and devices to provide a solid and genericsolution in

which to exploit any given heterogeneous platform's configuration. Given the amount of work that went into creating a sound migration mechanism for overlapping-ISAs, this will be no small task.

- In the case of a further improved heterogeneous-ISA compiler framework, each architecture possesses it's own strengths and weaknesses. Ideally, any given partitioning schema given either by static analysis or runtime should only exploit an architectures strengths. However, just as there are "good" combinations of architectures that become even more performant when joined together, there also exist "bad" combinations of architectures that should be avoided. These "bad" combinations would potentially arise from implementation details as to get the both of best worlds, it is often needed to compromise in some aspect to resolve ISA differences. This question therefore poses a new avenue of research to explore various heterogeneous-ISA configurations.

- A common challenge with software in industry is that source code for applications sometimes cannot be provided or cannot be modified for a variety of reasons. Some code is protected by Intellectual Protection (IP) or cannot be disclosed, other applications no longer have source or the source is fairly dated to the years that assembly and FORTRAN were the main programming models, leaving the binary file as the programs only tangible form. Finally, the application's source code could be so large that partitioning is not a viable option. For any of these ailments, its not to say that dealing with them is impossible but a significant upgrade would be required, with more intricate methods needed to perform the same changes.

- Applications that are too small to benefit from heterogeneity should automatically be recognized by analysis and reported to the user.

8.3 Further Evaluation

While we have evaulated the Xeon - Xeon Phi platform with the NAS Parallel Benchmark Suite [9], it would be beneficial to expand our evaluation to also include other arbitrary C language benchmarks (as our compiler framework currently only supports the C language) such as PARSEC [34].

Additionally, we would be very excited to evaluate our heterogeneous-ISA compiler framework on more than two different architectures. When hardware catches up to our expectations of pushing the envelope of heterogeneous processors and easy ways of connecting heterogeneous-ISA devices, it would be interesting to see the resulting performance together with Popcorn's heterogeneous runtime scheduler.

Bibliography

[1] AMD. Amd unveils ambidextrous computing roadmap, September 2015. http://www.amd.com/en-us/press-releases/Pages/ambidextrous-computing-2014may05.aspx.

[2] ARM. big.little technology, September 2015.

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

[4] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, page 29. ACM, 2015.

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

[6] OpenMP Architecture Review Board. Intel openmp* runtime library, September 2015. https://www.openmprtl.org/.

[7] The OpenMP Architecture Review Board. Openmp.org, September 2015.

[8] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 261–272. ACM, 2012.

[9] Nasa Advanced Supercomuting Division. Nasa parallel benchmarks, September 2015. https://www.nas.nasa.gov/publications/npb.html.

[10] University of Innsbruck DPS Research Group. Insieme compiler project, September 2015.

[11] Peter Elias, Amiel Feinstein, and Claude E Shannon. A note on the maximum flow through a network. *Information Theory, IRE Transactions on*, 2(4):117–119, 1956.

[12] Robert Fitzgerald, Todd B Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. *Software-Practice and Experience*, 30(3):199–232, 2000.

[13] Giorgis Georgakoudis, Dimitrios S Nikolopoulos, and Spyros Lalis. Fast dynamic binary rewriting to support thread migration in shared-isa asymmetric multicores. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and many cores*, page 4. ACM, 2013.

[14] Bronwyn H Hall and Beethika Khan. Adoption of new technology. Technical report, National Bureau of Economic Research, 2003.

[15] Intel. Intel xeon phi coprocessor instruction set architecture reference manual, September 2015. https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.

[16] Intel. Intel® xeon phi™ product family: Product brief, September 2015. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[17] Intel. Relocation errors, September 2015. https://software.intel.com/en-us/articles/avoiding-relocation-errors-when-building-applications-with-large-global-or-static-data-on-intel64.

[18] Michael Kerrisk. Linux programmer's manual, September 2015. http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

[19] Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1403–1408. IEEE, 2012.

[20] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 149–160. ACM, 2013.

[21] 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.

[22] Boost C++ Libraries. Boost.org, September 2015.

[23] Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. In *ACM SIGOPS operating systems review*, volume 42, pages 287–296. ACM, 2008.

[24] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[25] NVIDIA. Tegra processors, September 2015. http://www.nvidia.com/object/tegra-x1-processor.html.

[26] David Patterson. Future of computer architecture, September 2015. http://www.slidefinder.net/f/future_computer_architecture_david_patterson/patterson/6912680.

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

[28] Dheeraj Reddy, David Koufaty, Paul Brett, and Scott Hahn. Bridging functional heterogeneity in multicore architectures. *ACM SIGOPS Operating Systems Review*, 45(1):21–33, 2011.

[29] Timothy Roscoe. Hake, barrelfish technical note 003, November 2010.

[30] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[31] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71(1):114–131, 2011.

[32] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, page 27, 2008.

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

[34] Princeton University. Parsec, September 2015. http://parsec.cs.princeton.edu/.

[35] Ashwin Venkat and Dean M Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 121–132. IEEE, 2014.

[36] David G Von Bank, Charles M Shub, and Robert W Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1842–1874, 1994.

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

[38] WCCFTech. Amd cancelled project skybridge because global foundries terminated 20nm bulk – italian publication, September 2015. http://wccftech.com/amd-cancelled-skybridge-glofo-terminated-20nm/.

[39] Wikipedia. Fat binary, September 2015. https://en.wikipedia.org/wiki/Fat_binary.

[40] Wikipedia. Magic number, September 2015. https://en.wikipedia.org/wiki/Magic_number_

[41] Computer World. What intel's sandy bridge chips offer you, September 2015. http://www.computerworld.com/article/2512287/computer-processors/what-intel-s-sandy-bridge-chips-offer-you.html.