5-2016

Enabling Runtime Profiling to Hide and Exploit Heterogeneity within Chip Heterogeneous Multiprocessor Systems (CHMPS)

Eugene Cartwright
University of Arkansas, Fayetteville

Enabling Runtime Profiling to Hide and Exploit Heterogeneity within Chip Heterogeneous Multiprocessor Systems (CHMPS)

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

by

Eugene Cartwright
University of Arkansas
Bachelor of Science in Computer Engineering, 2009
University of Arkansas
Master of Science in Computer Engineering, 2012

May 2016
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council

Abstract

The heterogeneity of multiprocessor systems on chip (MPSoC) has presented unique opportunities for furthering today's diverse application needs. FPGA-based MPSoCs have the potential of bridging the gap between generality and specialization but has traditionally been limited to device experts. The flexibility of these systems can enable computation without compromise but can only be realized if this flexibility extends throughout the software stack. At the top of this stack, there has been significant effort for leveraging the heterogeneity of the architecture. However, the betterment of these abstractions are limited to what the bottom of the stack exposes: the runtime system.

The runtime system is conveniently positioned between the heterogeneity of the hardware, and the diverse mix of both programming languages and applications. As a result, it is an important enabler of realizing the flexibility of an FPGA-base MPSoC. The runtime system can provide the abstractions of how to make use of the hardware. However, it is also important to know *when* and *which* hardware to use. This is a non-issue for a homogeneous system, but is an important challenge to overcome for heterogeneous systems.

This thesis presents a self-aware runtime system that is able to adapt to the application's hardware needs with a runtime overhead that is comparable to a naive approach. It achieves this through a combination of pre-generated offline data, and the utilization of runtime data. For systems with diminishing hardware, the results confirmed that the runtime system provided high resource efficiency. This thesis also explored different runtime metrics that can affect the application on a heterogeneous system and offers concluding remarks on future work.

Acknowledgements

I would like to extend my thanks to my thesis committee, especially my advisor Dr. David Andrews for providing the tools necessary for my research.

Contents

List of Figures

List of Tables

Terms and Definitions

ABI Application Binary Interface. Refers to the calling convention and data layout of a particular processor-compiler pair.

API Application Programmer Interface. The defined interface of a piece of software, often times a library or operating system.

CISC Complex Instruction Set Computer.

CPU Central Processing Unit. A programmable hardware component, often referred to as processor or core.

DMA Direct Memory Access. Often referring to hardware devices that can perform memory-to-memory operations without processor assistance.

DPR Dynamic Partial Reconfiguration. The process of which a portion of the FPGA is reconfigured without disrupting the rest of the chip while active.

DSP Digital Signal Processor. A processor specialized for signal processing, often featuring vector and multiply-accumulate operations.

FIFO First-In, First-Out. A hardware component or data structure that exhibits First-In, First-Out behavior (e.g. a queue).

FPGA Field Programmable Gate Array. A hardware chip whose functionality can be changed post-fabrication.

GPU Graphics Processing Unit. A hardware component specialized for graphics processing.

HAL Hardware Abstraction Layer. A layer of software used to hide hardware-specific implementation details.

HDL Hardware Description Language (e.g. VHDL or Verilog).

HLL High-Level Language.

HW Abbreviation for Hardware.

ISA Instruction Set Architecture. Also known as the instruction set of a particular processor.

IPC Inter-Process Communication, when referring to software; or Inter-Processor Communication, when referring to hardware systems. Also used to abbreviate the average number of instructions that can be processed in a clock cycle.

I/O Input/Output.

Kernel The core component(s) of an operating system.

LIFO Last-In, First-Out. A hardware component or data structure that exhibits Last-In, First-Out Behavior (e.g. a stack).

MIMD Multiple-Instruction, Multiple-Data. A category of parallel programming/computational model in Flynn's Taxonomy that is represented by systems capable of executing different instruction streams that are able to operate on different streams of data simultaneously. Multiprocessor (multi-core) systems fit into the MIMD category.

MISD Multiple-Instruction, Single-Data. A category of parallel programming/computational model in Flynn's Taxonomy that is represented by a machine that executes multiple instructions on a single piece of data. It can be argued that pipelines and systolic arrays fit into this category.

MMIO Memory-Mapped I/O. The process of performing I/O through memory-mapped interactions such as reads and writes (loads and stores).

MPSoC Multiprocessor System-on-Chip.

OS Operating System.

RPC Remote-Procedure Call.

SIMD Single-Instruction, Multiple-Data. A category of a parallel programming/computational model in Flynn's Taxonomy that is represented by a vector processor that executes a single instruction on multiple data at once.

SISD Single-Instruction, Single-Data. A category of a parallel programming/computational model in Flynn's Taxonomy that is represented by a typical scalar processor.

SW Abbreviation for Software.

RISC Reduced Instruction Set Computer.

Chapter 1

Introduction

General computing systems have evolved over the past decade to support the increasing diversity of concurrent and parallel tasks. System designers continually face the challenge of increasing performance while decreasing the energy footprint of these systems. One component that maintains its presence in many of these systems is the general purpose processor (GPP), otherwise known as the central processing unit (CPU). CPUs have long provided a flexible architecture for targeting applications that span across multiple domains. With the support of mature compilers and increasing levels of abstraction provided by high-level languages [42, 3, 35, 45, 36, 43], CPUs can provide high user productivity and application portability. Due to the generality of the architecture, however, performance (and energy consumption) cannot best an application-specific integrated circuit (ASIC). The use of ASICs has generally led to loss of portability and a significant decrease in productivity. One of the design choices that CPU architects have made to increase performance is outfitting parallel functional units throughout a CPU's pipeline, coining the term *superscalar* processors. These additional resources, coupled with advanced branch prediction techniques enables CPUs to exploit instruction-level parallelism (ILP).

Many of today's applications such as those in the linear algebra and image processing domains, benefit from an architecture that supports the single-instruction, multiple-data (SIMD) model. CPU support for data-level parallelism (DLP) has been achieved through the inclusion of co-processors or (application-specific) accelerators within the processor [63]. The advantage of this *tight-coupling* implementation is that the use of this specialized hardware registers as another instruction that is part of the CPU's instruction set architecture (ISA). Considering compiler support for such instructions, this level of transparency upholds application portability and user

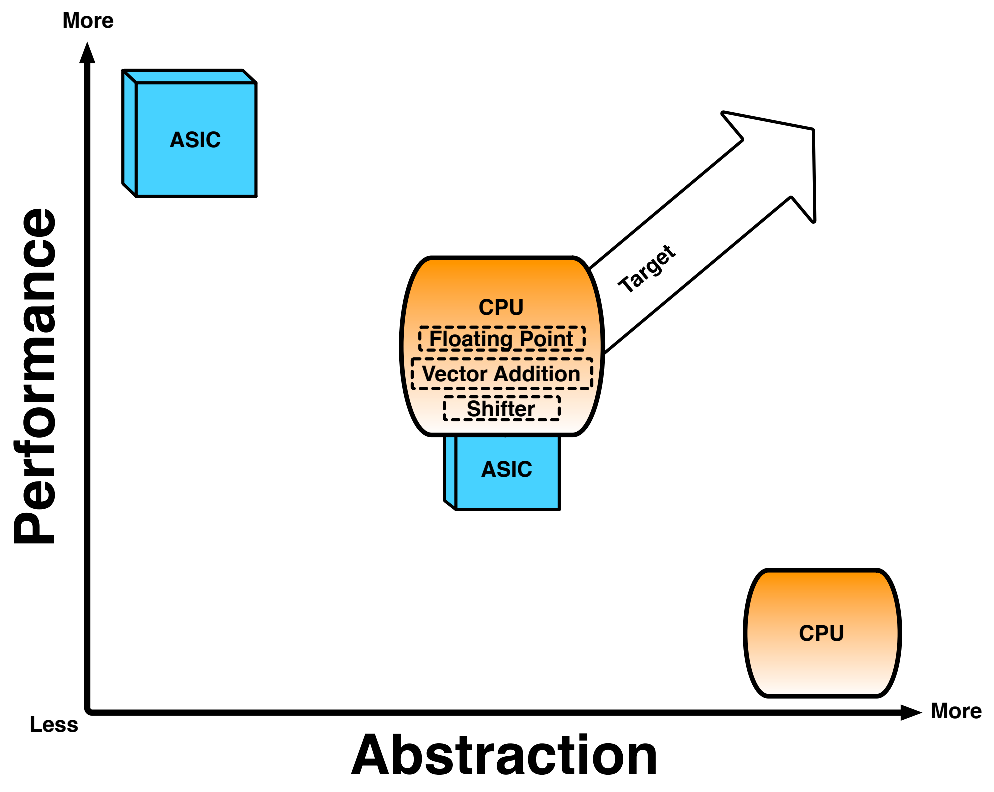

Figure 1.1: CPU vs ASIC design considerations

productivity, while providing a new avenue for program acceleration traditionally achieved through ASICs alone.

The reality is, many of these optimizations may or may not make it into a CPU depending on the constraints given to the CPU architect or the system integrator. Designers are always forced to strike a balance between generalized and application-specific, area and performance, and time-to-market and item costs. While the CPU does provide support for targeting most applications, one must consider which particular features a CPU may provide over others that would best benefit the targeted applications. Alternatively, with the advent of the many-core era, the opportunity arises for integrating multiple heterogeneous processors in the system. Therefore, the sacrifice of choosing a single configuration of a CPU for a multiprocessor system becomes moot as one can simply incorporate multiple configurations, capable of being optimized for different sets of tasks. The question poised here is does heterogeneity bring performance advantages of ASICS all the while maintaining CPU generality, and why or why not?

1.1 Why Heterogeneity?

More than a decade ago, a shift towards multiprocessor systems occurred in the general purpose computing community to combat the *power wall*: with increased processor clock speeds for faster performance came increased power (and heat) output [13, 50, 11]. These symmetric multiprocessor (SMP) systems were composed of identical processors (with identical bus and memory hierarchies) executing identical instruction streams independently, resulting in higher computational throughput over uniprocessor systems. New programming models for SMP systems emerged, enabling software developers to leverage transparent instruction-level parallelism (ILP) found in processors with *explicit* thread-/task-level parallelism (TLP) [35, 14]. Although technology scaling allowed vendors to increase the numbers of processors, application performance did not necessarily scale at the same rate. Amdahl's law [6] provides a means to calculate how much performance can scale for an application. Small fractions of the application that can be parallelized may benefit or not due to other overhead costs (communication and memory I/O). Introducing multiple, heterogeneous processing elements optimized for different sets of tasks within an application would enable performance to scale beyond SMP systems [32].

Heterogeneous multiprocessor systems, herein referred to as heterogeneous systems, have demonstrated their performance and energy benefits over their homogeneous counterparts [59, 30]. By including multiple heterogeneous processing elements, one can begin to place tasks onto specific components that provide greater performance and/or energy benefits. For example, the system may choose to conserve energy by mapping a longer running, memory-bounded task onto a low-performance and low-powered processor over a higher performance, high powered core. Examples of commercially available heterogeneous architectures today include ARM's big.LITTLE^{TM} multi-core architecture, Intel's QuickIA platform, and IBM's Cell platform [1, 15, 33]. However, due to the heterogeneity of these example platforms, a high-priority, CPU-bound tasks have the opportunity to be scheduled onto a processor with a high affinity. Contrasting this with a homogeneous platform, the design may consume more power, degrade in performance, or both due to minimal flexibility of where best to schedule tasks.

Micro-Benchmark Task Descriptions	
Name	**Description**
mul32	32-bit integer multiplication
mul64	64-bit integer multiplication
idivu	32-bit Unsigned integer division
idiv	32-bit Signed integer division
fpu	32-bit Floating point instructions. Includes square root function.

Table 1.1: Descriptions of the tasks used in Table 1.2.

System heterogeneity, otherwise referred to as system asymmetry, can arise from multiple sources and does not necessarily translate into a system composed of multiple processing elements, each provided by a different vendor. Processors can introduce asymmetry in simple features such as clock frequency, or can encompass more micro-architectural differences such as the number of general purpose registers (GPR), inclusion of different co-processors, and in/out-of-order instruction execution capabilities [37]. Table 1.2 provides a simple example of the performance effects of processor heterogeneity. The description of each task is outlined in Table 1.1.This example provides the execution time of several micro-benchmarks running on a 6 processor system: 1 host processor that schedules work onto 5 slave processors. Each micro-benchmark focuses on one type of task. The idea here is to isolate one task at a time to provide better insight into the decisions a runtime system or operating system (OS) may need to make when presented with a diverse set of tasks in its ready-to-run-queue (R2RQ). While all of the slave processors in this system are derived from the same base design, each one is configured to include a different co-processor optimized for one of these tasks. As a result, the instruction set is slightly modified amongst them. As the table highlights, this can translate into a significant performance delta amongst these processors for a particular benchmark. This table also suggests the energy benefits that can be gained by scheduling with this constraint. If all processors expel the same amount of power, scheduling a task that minimizes rather than prolongs the execution of

Micro-Benchmark Performance on a Heterogeneous system					
	Proc. 1	**Proc. 2**	**Proc. 3**	**Proc. 4**	**Proc. 5**
mul64	303.14 (1x)	130.07 (2.3x)	**23.11 (13.1x)**	213.35 (1.4x)	303.02 (1x)
mul32	138.84 (1x)	**15.24 (9.1x)**	**15.17 (9.2x)**	138.61 (1x)	139.01 (1x)
idivu	45.97 (1x)	46.10 (1x)	45.88 (1x)	45.94 (1x)	**7.43 (6.2x)**
idiv	44.69 (1x)	44.34 (1x)	45.39 (1x)	44.68 (1x)	**6.89 (6.6x)**
fpu	**104.56 (18.3x)**	115.56 (16.7x)	1058.22 (1.8x)	718.44 (2.7x)	1931.18 (1x)
shift	7099.59 (1x)	7098.70 (1x)	7098.85 (1x)	**554.41 (12.8x)**	7099.37 (1x)
	Execution times (μs) and speedup relative to slowest time.				

Table 1.2: Micro-benchmarks executing on heterogeneous processors.

this task would net less energy consumed by the system.

The previous example demonstrated the potential performance gains from a heterogeneous multiprocessor system but is often not representative of a system today. Often, this multiprocessor system is also augmented with hardware accelerators external to the multiple processors. Hardware accelerators excel at DLP, offering a higher performance-to-power metric over general purpose processors [31, 23]. This leaves more *branchy* or speculative code and coarse-grained tasks (i.e. TLP) to processors. However, designers may work with a limited amount of area and accounting for any and all types of ASIC components within their design is not realistic. Hence, the appeal of CPUs. Alternatively, an enticing heterogeneous platform for providing exploration of both general purpose and application specific components is a Field Programmable Gate Array (FPGA).

FPGAs have the potential to bridge the gap between the benefits of general purpose processors and ASIC-like components. The increasing densities of the reconfigurable fabric allow users to implement complete chip heterogeneous multiprocessor (CHMP) systems [16]. The malleability of the FPGA allows CHMP systems to be built that are tailored to the exact types of parallelism that may be available in each application. Execution of unmodified software applications can benefit from familiar architectures equipped with general purpose processors integrated as either reconfigurable *soft*, or *hard* intellectual property (IP) cores. However, FPGA users wishing to extract additional performance from the application can do so by including application-specific components onto the FPGA. Without having to commit to silicon as is done with an ASIC design cycle, this allows a more cost-effective solution. Leveraging this flexibility beyond processor heterogeneity, however, has been limited to FPGA device and tool experts as it imposes a high learning curve for the general software community. This requirement is continually being relaxed by new developments in high-level synthesis, re-useable IP, and programming models and abstractions [17, 41, 40]. Nevertheless, there still remains several challenges pertaining to FPGA-based CHMP systems and the heterogeneity it exposes.

1.2 Issues of Heterogeneity

Exploiting the use of heterogeneous components in a system faces several challenges. Today's commercial operating systems (OS) generally offer little to no support for system heterogeneity, and it is usually up to the software programmer to exploit this flexibility. As heterogeneous systems become more of a commodity item, it is important to address these challenges if one is to see a continued benefit in their use [4]. The following sections provides several challenges, specifically pertaining to CHMP systems on FPGAs.

1.2.1 Application portability and complexity

Programming for heterogeneous systems has traditionally come at the cost of both application complexity and portability. To fully exploit the performance a heterogeneous system can provide without support of the runtime and/or the OS, programmers are required to know and leverage detailed knowledge of the underlying hardware. For example, where and when to schedule a task based on multiple variables such as performance characteristics that can vary from one FPGA-based CHMP configuration to another. Without any heterogeneous hardware abstractions, accounting for such intricacies increases application complexity. Furthermore, including platform-specific optimizations within the source also affects application portability and requires it to be modified when moving between one system configuration to another. It is infeasible to account for all possible system configurations and doing so increases application complexity.

As noted previously, users of FPGA-based CHMP systems are required to know the hardware in order to design and use application-specific components within their systems. Consequently, the adoption rate of this heterogeneous platform by the general software community has been slow as a result of the imposed high learning curve. As will be discussed in Section 2.1, there is an abundance of research addressing the need for easier programmability. However, the use of user-provided hardware should coincide with application portability and complexity. Additionally, runtime decisions such as dynamically mapping tasks to the *appropriate* resource (i.e. accelerator or CPU) presents another challenge.

1.2.2 Dynamic Mapping

Determining how the application should be mapped onto heterogeneous systems before runtime is referred to as static mapping. Statically mapping the application is not sufficient for both homogeneous and heterogeneous systems alike [47]. Luk et. al note that not knowing the workload size renders performance estimation offline inaccurate [39]. Generating and optimizing a mapping scheme offline without knowing the data size can result in poor mapping. For example, allocating a small data-set onto hardware optimized for larger data sets can degrade performance (and increase power). Additionally, the heterogeneity of the system naturally exhibits asynchronous and indeterministic behavior. This can prove difficult to model the diverse components and their interactions in the system without making several (static) assumptions. This challenge is compounded further when considering FPGA CHMP systems. In addition to enabling FPGA users to realize their system designs on the chip itself through hardware synthesis, FPGAs also supports systems to change their hardware during runtime through a process known as dynamic partial reconfiguration (DPR) [64].

DPR offers an opportunity to shape the platform to the application's needs as it executes, gaining additional performance and/or energy savings wherever applicable. The process of dynamically using additional hardware can unlock further speedup in comparison to static heterogeneous designs [32]. Furthermore, this can be advantageous on more resource-constrained CHMP systems by swapping in-and-out hardware. This offers greater flexibility for designers when trading hardware resources/area for performance.

To accommodate FPGA CHMP systems utilizing DPR, dynamic mapping is needed to reflect scheduling decisions based on runtime parameters such as workload sizes and current system configuration. It is also needed for the different phases of an application. Not doing do so can lead to non-optimal performance, and even worse, system failure as components assumed to be present may no longer exist. Consequently, this makes scheduling (or in general, programming) for such systems much more difficult as there are more scheduling decisions to explore. Navigating through this multidimensional search space can lead to high runtime overheads, thus increasing the application execution times further [51].

While knowing the intricacies of a relatively small heterogeneous system may suffice for statically mapping an application, targeting larger heterogeneous systems becomes less attainable to do so by hand. What is needed is a dynamic solution where, ultimately, the mapping decision is done (or further aided) by the runtime system. This enables better mapping solutions that appropriately reflect the state of the system and pending tasks. Additionally, offering a dynamic solution can scale the performance of an application further over time. In the case of FPGA CHMP systems, as the necessary hardware diminishes or increases online through the use of DPR, the runtime can accommodate the decrease/increase in computational power and map the application appropriately. Similarly, when better performing processing elements are introduced in the system, dynamically mapping the application against the current system configuration can immediately leverage this resource.

1.2.3 Interoperability

Interoperability challenges within heterogeneous systems arises from the diversity of the components. For example, processors that differ in ISAs implement equivalent operations differently in hardware. As a result, this varies execution and memory access times amongst them. Another example is the application binary interface (ABI), which dictates the intricacies of a processor at the machine code level such as data storage and retrieval. The difference of ABI here can result in data misinterpretation. The introduction of custom hardware accelerators within FPGA CHMP system exacerbates this issue further, and gives a stronger need for some degree of standardization within a heterogeneous system.

Standardization in the form of hardware/software abstractions allows different components to interact within the system. Without such abstractions, this can lead to code complexity and non-portability. Standardization of OS services such as task management and synchronization should be provided for a machine that focusses on parallel execution. In addition to this, system configuration changes as a result of exploiting DPR should not introduce any interoperability issues and disrupt the flexibility of FPGA CHMP systems. Therefore, a need for an abstraction layer that seamlessly connect heterogeneous components at the HW/SW boundary is necessary and facilitates the flexibility offered by heterogeneous components.

1.3 Thesis Statement

Programming support for heterogeneous multiprocessor systems remains a challenge today despite the increasing ubiquity of these systems over the years. One of the attractions towards heterogeneous systems lies within their diverse hardware, gaining opportunities for hardware task specialization to further maximize performance/energy savings [19, 29]. Consequently, this has encouraged programmers to include more platform specific information within the source code to take advantage of these performance/energy savings. Without higher levels of abstraction and/or middleware support, this eliminates the idea of application portability, increases application complexity, and thus decreases programmer productivity. This thesis proposes to investigate if these trends can be reversed for FPGA-based heterogeneous multiprocessor systems by considering the following questions:

- Can platform specific details be moved from the source code and be encapsulated under platform neutral policies of a portable programming model to reinstate portability?
- Can the complexity of the software design process be reduced by enabling automatic profiling during runtime?
- Can dynamic mapping occur at reasonable overhead costs with high accuracy of fitting task placement? How best to determine task specialization for processors? Can some of this information be determined offline?
- Can resource efficiency be increased despite increasing levels of system complexity, by introducing additional self-awareness within the operating system?

Thesis: For FPGA-based heterogeneous multiprocessor systems, a self-aware runtime system can replace the need for a user to engage in hand profiling and explicit thread scheduling of an application on different configurations of heterogeneous resources. Through the integration of a self-aware runtime system in one's design, a heterogeneous multiprocessor system can better utilize its resources and match the performance of a homogeneous system where resource constraints are non-existent.

1.3.1 Thesis Contributions and Organization

To support the thesis statement, this work provides the following set of contributions:

- Integrated identification of processor (and hardware accelerator) specialization within the scheduler that results in little additional runtime overhead in comparison to a naive approach.

- Modified the hthreads compilation flow to parse platform configuration, including but not limited to MicroBlaze processor configuration and attached hardware accelerators.
- Integration of an accelerator library that enables users to utilize attached accelerators through software function invocations.

The remainder is organized as follows. Chapter 2 surveys the current state of the art approaches on multiple issues of heterogeneous computing, and concludes with FPGA related work. Next, Chapter 3 provides a detailed discussion of this thesis approach to addressing some of the issues raised in Chapter 1. A discussion of this thesis' evaluation against the proposed approach is presented in Chapter 4. Finally, the thesis concludes by reflecting back on the questions proposed and offers some potential future work that can follow this thesis.

Chapter 2

Background

Heterogeneous MPSoC systems are a commodity in today's society, and can be found in high performance computing data-centers to embedded devices within automobiles. Their heterogeneity, offering both flexibility and specialization, provides opportunities for additional levels of performance (and/or energy savings). However, achieving this continues to be met with the difficulty in programming such systems. Many have approached this problem with various solutions such as unified programming languages (and tools) and self-aware runtime systems. Others have incorporated both of these examples into programming frameworks. Still, very few have approached the reconfigurable domain. The following sections offer details on state of the art approaches in each of these categories.

2.1 Programming Languages and tools

To begin, many of the programming languages and tools (libraries and compilers) presented here attempt to provide one or more of the following:

- Device/platform abstractions for easier programming, and subsequently increasing user productivity.
- Platform portability between systems with both the presence or absence of *preferred* hardware.
- Easier accessibility of potential performance gains of the heterogeneous *component* while balancing programming effort.

Amongst these characteristics, the important goal many of the following languages and tools tend to focus on is enabling easier programmability. Therefore, many forgo addressing efficient utilization of resources. The absence of runtime information make programming languages and their use offline, not a complete solution for the programmability of heterogeneous systems. Many of these languages still require the programmer to explicitly identify where computation should execute and/or greedily schedule *all* work on a particular device (e.g. GPU vs. CPU). Nevertheless, the research presented here should be regarded as supplemental work as it provides a transparent front-end interface to a heterogeneous system.

2.1.1 C++ AMP:

With the increasing use of general purpose graphics processing units (GPGPUs) on desktops, it has been made possible to accelerate non-graphics related applications on a cost-effective hardware solution. Microsoft proposed the C++ Accelerated Massive Parallelism (C++ AMP) project, enabling users to exploit DLP on these types of data-parallel hardware. The targeted hardware has generally included GPUs. However, modern CPUs containing support for SIMD (vector) operations allows the C++ AMP runtime to fall back to such hardware whenever GPUs fail to provide the computation [42].

2.1.2 Bolt:

Bolt is a C++ Standard Template Library (STL) that aims for accelerating user applications on accelerated hardware [3]. Bolt includes optimized implementations of common algorithmic patterns such as reduce and transform, and currently has a few back-ends for generating code to languages such as C++ AMP, OpenCL and Intel Thread Building Blocks. Having been brought to market by a GPU supplier, Advanced Micro Devices (AMD), Bolt is optimized specifically for AMD GPUs. However, like C++ AMP, the library includes fall-back mechanisms to CPUs. Bolt can dynamically query the platform, favoring the GPU device due to the assumption that it will provide a higher level of data-parallel performance. Additionally, the memory hierarchy is not imposed on the user unlike OpenCL. As a result, programmers are presented with a linear address space found on familiar homogeneous architectures.

2.1.3 Intel Thread Building Blocks:

The Intel Thread Building Blocks (Intel TBB) is a C++ STL designed for supporting parallelism across multiprocessor systems [35]. Unlike the previous discussed thus far, Intel TBB does not target other types of data-parallel hardware such as GPUs. Users write their application in terms of tasks and the library maps these tasks onto threads (CPUs assumed to be homogeneous) for more finer-grained parallelism. The Intel TBB runtime also provides load balancing of tasks and encourages users to reduce the granularity of tasks in order to provide the task-based scheduler greater opportunity for load-balancing. Currently, Intel TBB integrates load-balancing through task-queuing and work-stealing. Work stealing allows one to mask out any potential latencies attributed from a long running task, offering better load-balancing [12].

2.1.4 OpenMP:

OpenMP is an API for C/C++ and Fortran programmers enabling them to write parallel, shared-memory code for multiprocessor systems [18, 44]. Unlike task-based models such as Intel's TBB, OpenMP is a directive based parallel programming model: programmer's annotate parallel sections of the source code and leave implementation specific details up to the compiler (i.e. how to execute the annotated code in parallel). OpenMP is being adapted to extend support to heterogeneous multiprocessor systems such as the IBM Cell [45], and general purpose graphics processing units (GPGPUs) [65, 49]. However, it was becoming important to provide a standard set of directives for these accelerators. As a result, new emerging projects designed to naturally accommodate accelerators within the system such as GPUs include OpenACC [2] and OpenHMPP [22].

One of the dynamic properties of OpenMP (and its derivatives) can be found in its support for dynamic threading. When dynamic threading is disabled, the OpenMP greedily creates the maximum amount of threads when parallel code regions are encountered. When dynamic threading is enabled however, the runtime can create a variable number of threads. Currently, the OpenMP standard does not automatically adjust this variable number based on the runtime environment and is up to the programmer to define this number. Therefore, dynamic as defined here simply means user-configurable as opposed to self-adaptable.

2.1.5 Cilk/Cilk Plus:

Cilk is another directive-based programming model aimed at providing a reasonable ratio between user productivity and performance. Cilk (C) or Cilk Plus (C++) was designed to quickly and easily parallelize code for both general multi-core processors and processors with SIMD extensions/vector support. It achieves this through its simple code annotations all the while maintaining the serial nature of existing code bases programmers are familiar with. As with directive-based models, parallel regions must be explicitly annotated for use during compilation (and runtime). However, the user is relieved from deciding where such tasks will run in parallel. Intel's Cilk Plus shares similar scheduling policies with Intel's TBB: both take a greedy approach by assigning as many *worker threads* as possible throughout the system, and both support randomized work-stealing across all thread's double-ended queues (deques) to provide reasonable system load-balancing. Like Intel's TBB, Cilk Plus also does not support processor affinity and processors are assumed to be homogeneous.

2.2 Programming Frameworks

Programming frameworks can be defined as a collection of software (libraries) that provides an environment in which one can program a targeted platform. Programming frameworks differ from programming languages in that they guide the user in using its software to craft the application;

programming languages can too, use existing software but the decision is at the programmer's discretion.

Since programming in these frameworks entails making heavy use of their Application Programming Interfaces (APIs), much of the implementation-specific details are abstracted. These types of frameworks can provide their own virtual memory hierarchies and communication protocols atop the existing hardware platform. Furthermore, programming frameworks enables compiler and programming language developers to target such frameworks. This provides an opportunity for programmers to utilize legacy languages to target platforms that wouldn't otherwise be supported on its own.

2.2.1 OpenCL:

OpenCL is a portable/cross-platform parallel programming framework designed to aid programmers in directing computation across a range of heterogeneous computational devices [36, 53]. These devices are but not limited to CPUs, GPUs, digital signal processors (DSPs) and FPGAs. OpenCL can be used to exploit both TLP as well as DLP.

The execution model of OpenCL is similar to a task-based model however, OpenCL defines tasks as kernels. Instances of kernels assigned to devices are referred to as *work-items*, which are ultimately grouped into a *work-group*. Since OpenCL devices can contain many processing elements (PEs) such as a GPU or a multi-core CPU, workgroups are assigned to these devices where work-items are then computed individually on the processing elements (PEs) of that device.

Alongside its execution model, OpenCL adopts a memory model that consists of four types of memories: private, local, constant, and global. Private memory is reserved for each individual PE on the OpenCL device. For example, the private cache for each core in a multi-core CPU can be considered as private memory. Local memory is temporal memory that is shared amongst PEs within the device, but not globally accessible by other OpenCL devices. Constant and global memory are similar such that they are accessible by all OpenCL devices but differ in that constant memory is a read-only memory.

OpenCL exposes both the execution model and memory model to the programmer. Through the use of OpenCL's application programming interfaces (APIs), the memory hierarchy in addition to the heterogeneity of the device is abstracted. This allows portable code to be generated, however it is still possible for OpenCL programs to be *tuned* to specific platforms. This is realized through explicit management of where computation should occur by the programmer. Although one can query each device to obtain optimal working data sizes, OpenCL does not support automatic inference of where best to assign such tasks and the associated data. Additionally, one must explicitly perform memory management. When queuing workloads across different OpenCL devices, one must take care to transfer data between the system's global memory and the device's local memories.

2.2.2 Heterogeneous System Architecture:

The Heterogeneous System Architecture (HSA) [4] is a specification created to unify heterogeneous computing components and provide support for familiar programming models such as C++, C++ AMP, OpenCL, and OpenMP. HSA was created to address many of the existing challenges within heterogeneous systems, namely:

- Power consumption - the opportunity of leveraging the heterogeneity to reduce power is becoming more important as devices become smaller.
- Scalable performance - the accessibility of devices offering higher levels of computational throughput necessitates the performance to scale in order to warrant such device integrations.
- Programmer productivity - avoid soliciting a high learning curve by providing a familiar programming model to maintain and/or increase programmer productivity (and program portability).

- Program portability - the need to maintain program portability across heterogeneous systems to avoid rewriting code is important. Not doing so can negatively affect the programmer's productivity.

HSA was founded by the HSA Foundation, comprising of companies such as ARM, Qualcomm, AMD, and many others. Among them, AMD has demonstrated HSAs on their Accelerated processing units (APUs) consisting of both CPU and GPU, alternatively labelled as *latency compute unit* (LCU) and *throughput compute unit* (TCU) respectively. LCUs are a generalization of a CPU whereas TCUs can refer to GPGPUs and FPGAs. Traditionally, programming such devices required two disparate programming models. To remedy this, HSA foundation proposes a common intermediate representation referred to as the HSAIL, enabling high-level programming languages and compilers to target. The HSA also abstracts the distributed memory hierarchy that can arise from multiple heterogeneous devices. By adhering to the HSA specification, companies can enable programmers to leverage their devices in a very transparent and portable fashion. This is in contrast to more popular programming frameworks such as OpenCL, or CUDA by Nvidia [43], where the memory hierarchy is exposed and explicit memory management is required.

2.2.3 PLASMA:

The authors in [46] recognize the myriad of specialized accelerators and the differing programming models each brings to high-performance computing systems. As mentioned previously, this introduces distributive memory and heterogeneity, and requires the programmer to identify scenarios that warrants their use. Load-balancing and computation scheduling also presents a challenge in these systems where tasks can optionally execute on more general-purpose hardware (e.g. CPU). To address such problems, the authors propose a programming framework, referred to as PLASMA. The framework consists of an Intermediate Representation (IR) allowing existing legacy programming languages and tools to target the PLASMA platform, a compiler supporting multiple device specific back-ends, and a runtime to manage computation throughout the system. The authors integrate static partitioning within their work; hand-partitioning computation across both a (multi-core) CPU and GPU. This partitioning does not take into account data-sizes and other runtime data that may affect scheduling. The authors mention that the runtime system can be replaced to handle said dynamic partitioning and scheduling.

2.2.4 Dandelion:

Dandelion is a programming framework for distributed heterogeneous systems [48]. Users targeting the Dandelion system write code in the .NET LINQ framework. Dandelion then maps tasks from the user's source onto cluster machines, onto which those machines map work onto their heterogeneous resources: multi-core CPUs and GPUs. The inclusion of heterogeneous hardware, namely GPUs, in addition to distributed computing introduces varying programming models. To insulate the programmer from varying implementation-specific details, Dandelion includes three execution engines: a distributed cluster engine, a multi-core CPU engine, and a GPU engine. Each of these engines receives a data-flow graph dictating the order of computation and synchronization. Computation mapping begins at the cluster level to an available machine; followed by the machine level where computation is mapped to either the multi-core CPU or the GPU. Currently, the mapping to a CPU or GPU is determined offline. Furthermore, the Dandelion does not address resource availability and assumes each machine in the cluster is homogeneous with respect to one another.

2.2.5 OpenCPI:

OpenCPI is a software framework whose programming model is component-based [34]. This programming model abstracts a computing resource or a collection of computing resources (e.g. millions of gates on an FPGA) as a component. The programming model, coupled with a well-defined interface between components enables application portability and component interoperability. As a result, OpenCPI focuses on bringing component-based programming onto real-time systems where it is natural for components to be swapped in and out (e.g. aerospace and military grade applications). OpenCPI provides a middleware solution for targeting such heterogeneous systems that are composed of CPUs, FPGAs, DSPs, and other emerging technologies.

2.3 Runtime Systems

According to [10], runtime systems do what programming languages cannot. That is to say, runtime systems have the advantage of utilizing the available runtime variables and resolved data sizes during runtime. Programming languages cannot extract information and optimize the scheduling of work on information that is not yet available.

Runtime systems provides several core services for the program and can be viewed as an extension of the programming language itself. Core services can include but are not limited to operating system calls, exception handling, stack management, scheduling, and profiling. Due to this, the position of runtime systems provides an appropriate opportunity to insert additional self-aware services such as processor affinity discovery, and resource-aware scheduling. The related works that follow, exhibit this autonomous behavior but are able to provide a level of transparency a programming language alone cannot do. Note, runtime system and scheduler are used interchangeably throughout the remaining of this thesis. The scheduler exists alongside many other core services that the runtime can provide.

2.3.1 Elastic Computing:

The elastic computing system (ECS) is a combination of a programming framework and profiling tools that currently targets desktop systems composed of CPUs, GPUs, and FPGAs. ECS aims at providing transparent, portable and adaptable computing [57, 58]. To achieve this, ECS utilizes a function library that stores a collection of commonly used functions referred to as elastic functions. Elastic functions differ from traditional functions in that they include implementation decisions within themselves based on runtime data such as input data and (available) computing resources. This enables them to provide a level of transparency to the user without requiring them to specify implementation details within their code, hence rendering their application non-portable.

Unlike other works that specify a single implementation for a given function [39, 10], ECS designed the elastic function library to include multiple implementations for each function. This presents an opportunity for ECS to explore alternative implementations of a given function and provide the fastest implementation based on runtime parameters. For example, choosing between *insertion sort* and *quick sort* when the size of the data is small, or if an FPGA with bitonic sort is provided. Furthermore, ECS enables the user to provide suggestions when using elastic functions that could benefit the runtime on choosing the most appropriate implementation. For example, if the data is mostly sorted, it may be best to choose *insertion sort* as compared to randomly sorted data would profit the most using *quick sort*.

ECS identifies the appropriate implementation (and parallelization strategies) offline through a processor know as *implementation assessment*. Implementation assessment is performed at installation time of ECS, and stores empirical results that take any architecture effects such as memory bottlenecks, cache configurations, and interconnect architectures. To begin, ECS exhaustively samples for all combinations of computing resources, every implementation for every elastic function to build an implementation performance graph (IPG). IPGs are two-dimensional piecewise linear functions that given a work metric, an estimate of execution time is provided. Work metrics are an abstract representation of the amount of work/computation that will be incurred for the input parameter(s). For elastic functions that have a single input parameter, a mapping is made on assigning this parameter as the the work metric. For elastic functions composed of multiple parameters, the work metric should identify the subset of parameters that affect the function's execution time. In both cases, the work metric is assumed to be statically defined by the one designing the specific *implementation* of an elastic function.

To avoid expensive comparisons during runtime when deciding which implementation to choose when the application invokes an elastic function, ECS additionally employs *optimization planning*. Optimization planning is performed at ECS installation time, after implementation assessment. It involves combining all IPGs for a particular elastic function, and creating a new graph that reflects the fastest implementation for varying regions of the work metric. This function performance graph (FPG) enables the runtime to lookup a single source of performance information, circumventing a possibly large overhead for navigating elastic functions that include numerous implementations. Additionally, ECS also employs *work parallelization planning* (WPP) that can use FPGs to determine how to effectively partition the elastic function across multiple computing resources. This enables ECS to intelligently distribute work amongst computing resources rather than greedily assigning work to all, possibly negatively affecting performance. This information is also collapsed into a single graph, referred to as the work parallelization graph (WPG) for fast lookups during runtime. However, WPP does not take into account resource contention and other affects that arises when multiple instances are executing, and it is assumed the performance remains the same as it would when one implementation is executing on a single resource.

While ECS presents many unique solutions on adaptable computing, ECS targeted platform does not revolve around reconfigurable computing. ECS does demonstrate the use of FPGAs, however, there is no documented support for partial reconfiguration. Additionally, ECS does not queue work to devices as ECS does not attempt to minimize the overall/absolute execution time of the application. Rather, ECS aims at providing the fastest execution time for a given task, relative to the current state of the system.

2.3.2 Harmony:

Harmony is a collection of runtime services that provide dynamic inference of concurrency, application portability, application scalability, and efficient utilization of heterogeneous resources through dynamic partitioning [20]. Harmony provides a task-based programming model that directs users in creating *sequentially specified imperative programs*. That is, applications are specified as a sequence of control structures and compute kernels. This follows very similarly to StarPU discussed in Section 2.4.3, whose applications are also described as a task control-flow data graph. Meta-data is also supplied from the user (and can be supplied through compilers), which provides runtime data to infer concurrency from a sequentially written program without resorting to pragmas such as those found in OpenMP and OpenACC. Some of this meta-data includes read and write memory locations used to determine concurrency or anticipate data dependencies between compute kernels, a pointer to the next kernel which provides a more explicitly controlled dependency, a user provided function that enables the runtime to determine kernel complexity/estimate performance based on input-data, binary implementations for compute kernel for at least one architecture (similar to StarPU's codelet, or SPREAD's switchable threads), and more.

These dependency graphs are generated during runtime and in order to balance performance with the overhead of scheduling and the generation of these graphs, Harmony limits the size of the scheduling window or how many kernels it manages at any given time before adding more to the graph. Harmony also collapses multiple kernels into one on these dependency graphs to reduce the overhead of managing multiple kernels on the graph. This is done by using a kernel's estimated execution time. If it is found that the execution time is less than some threshold, the compute kernels are either combined with a previous kernel on the dependency graph or the next one dictated by program order. As a result of combining multiple compute kernels, it enables the system to schedule more compute kernels with less overhead.

To estimate performance for a given compute kernel, Harmony uses a combination of logging historical execution latencies for that kernel on various hardware, and computing offline regressional models over such logged data and meta-data. Users can indicate through meta-data, which inputs do affect the complexity of the compute kernels. Harmony can then aggregate this data with latencies already recorded (historical execution times on different hardware) to form a multivariate regressional model. They also indicate that they can improve results further if the user additionally adds ”user-supplied estimates” of kernel complexity. However, most of this discussion on performance monitoring and optimization in [21] does not provide sufficient detail as to how the authors used these regressional models and how much overhead it generates. Although they mention such techniques are possible for their runtime system, they ultimately do not demonstrate the use of input data and latencies to shape a performance curve with a multivariate regressional model.

2.3.3 Qilin:

The authors in [39] recognize the importance of heterogeneity and its advantages on today’s multiprocessor systems. They also mention that in order to take advantage of this heterogeneity, efficient utilization is necessary. Their solution includes automatically mapping computation to a CPU and GPU configuration where most appropriately. This is done transparently to the user, giving way to provide portable and scalable code across different system configurations.

To begin, Qilin leverages two threading-based APIs: Intel TBB for scheduling parallel workloads onto CPUs and NVIDIA CUDA for (NVIDIA) GPUs. As a result, Qilin does not support heterogeneous processors. Qilin’s programming model requires the programmer to explicitly write parallelizable code through Qilin’s APIs. The rationale behind this is it alleviates the complexity *of extracting parallelism from serial code, and instead can focus on performance tuning* [39]. Qilin programs compute on two types of data: QArray and QArrayLists. QArrays are multidimensional arrays and QArrayLists are list of QArray objects. Qilin programs can be written with two sets of APIs: Stream-API and Threading-API. The Stream-API approach includes utilizing common data-parallel functions (and overloaded operators) that exist within the Qilin library. A few examples include *sqrt()*, *BLAS()*, *shift()*, and *sum()*. Threading-API enables the user to provide their own function to execute on QArrays and QArrayLists. This is done by writing both Intel TBB code as well as NVIDIA CUDA code, providing a wrapper function that includes both source code, and then calling the wrapper function at runtime to bootstrap into either or both of them. Additionally, the user can annotate data as partitionable. This enables the runtime to automatically partition and merge the data across both the CPU and GPU.

Qilin contains an online compilation flow when targeting CPUs and GPUs. Qilin API calls are translated into TBB or CUDA source code, from which are then passed to the appropriate compiler during runtime. The motivation for online compilation is that it allows the final binary to reflect the current runtime environment and/or data sizes that are unknown offline. This also leverages the optimizations of the respective compilers when the input data is known. However, the authors do not provide further insight on runtime overhead for online compilation and how it affects performance.

Performance modelling on Qilin is similar to that of StarPU 2.4.3, a linear performance model is built from empirical runtime data collected overtime. For the current system configuration, Qilin additionally stores such data offline for later use. If there exists tasks within a program of which no stored data exist, Qilin first trains on the data. This is performed by first dividing the data in half, where half of the data is assigned to the CPU(s) and the other half for the GPU. Qilin further divides the data into *m* subparts on each of the two hardware, whose execution on those subparts are recorded and used to approximate performance for it. Additionally, the fraction of the amount of work assigned to the CPU, β, is also found by computing the intersection of both the CPU's and GPU's performance equation line. This is utilized in partitioning a single task across both types of hardware for better load balancing. If $\beta \leq 0$, the entire argument size is offloaded to the GPU. If $\beta \geq 0$, the CPU is given the entire argument size. The authors recognize that the computational throughput of the CPU and GPU working cooperatively is assumed to be the same in cases where one works alone in the system (i.e. one is idle and the other is busy). The authors attempt to account for this but provide insufficient details on the chosen method.

2.4 Reconfigurable Systems

The purpose of this section is to provide the state of the art that addresses heterogeneity on reconfigurable systems. Much of what has been discussed earlier will apply here as well. Besides targeting a new platform, capabilities such as DPR enables these systems to dynamically change its hardware to that of which an application requires. This brings a challenge previous works mentioning support for FPGAs have not explored. While this flexibility can provide the hardware on-demand, having the capability does not dictate that one should always exploit it. The following works provides exploration on this front and highlights more of the ingrained challenges as a result of a reconfigurable platform.

2.4.1 AMAP:

With the increased densities of FPGAs, application designers have much more flexibility on where portions of their application can execute: either in hardware or software or both. The authors in [52] agree that deciding how to partition the application at compile time misses opportunities for runtime optimizations. This static partitioning can not maximize optimization based on unknown function parameter values. Furthermore, changes to the system's environment (e.g. power or resource availability) can further affect partitioning decisions. To address this, [52] introduces an adaptive mapping algorithm (AMAP) that dynamically partitions the application during runtime based on previous execution times. The authors argue that by deferring this decision as late as possible (ALAP), better partitioning results can be achieved over compile time techniques.

2.4.2 SPREAD:

SPREAD is a partially reconfigurable and adaptable system, and programming model that targets streaming applications such as multimedia and cryptographic applications [56]. SPREAD's programming model is thread-based and the authors include support for hardware threads through a hardware thread interface (HTI). The HTI enables both hardware and software threads to coexist during runtime. Additionally, SPREAD supports reconfigurable point-to-point communication channels to facilitate the pipeline parallelism needed for streaming-based applications. Another optimization that is included is caching runtime partial reconfigurations in order to amortize runtime overhead and increase the reuse of dynamically allocated resources.

In addition to dynamic resource allocation for hardware threads and reconfigurable point-to-point streaming channels, SPREAD integrates runtime adaptability through the seamless switching between hardware and software threads. Like ReconOS [38], every hardware thread that is created has a software delegate thread referred to as a stub thread that can be used for hardware monitoring and data stream redirection. SPREAD additionally defines a switchable thread, which contains both hardware and software implementations for a given thread function. This enables hardware/software threads to fall-back to software/hardware based on the application's needs during runtime such as higher priority resource allocation. However, these software implementation threads are limited only to a single processor, of which the OS resides on.

2.4.3 StarPU:

When hardware accelerators are introduced into the system, a common approach towards utilizing it is to offload *all* work tuned for that device. Many programming models support this approach as discussed in Section 2.1. The authors in [10], believe that scheduling tasks across the entire system rather than simply offloading (and queuing) them all to a predetermined accelerator provides greater benefit in terms of performance as well as mitigating *dark silicon* [24]: a term used to describe the increasing inefficiency of a system due to underutilization of components. Their solution is a runtime system called StarPU, which targets both numerical kernel libraries and parallel programming languages. Additionally, StarPU presents a task-level programming model, where the user writes the implementation of tasks and describe any dependencies between them. To enable execution of these tasks on any device, StarPU uses *codelets* which are containers for including multiple device-specific binary implementations for the given task. StarPU also does not require explicit marshalling of data similar to OpenCL. StarPU relieves this explicit memory management as the runtime ultimately has control of determining where best to schedule the task. Other transparent optimizations include data prefetching, and employing a write-back policy (delaying writes back to memory, thus localizing the data).

The authors of StarPU make further argument that a single scheduling policy does not suffice for all applications and believe that the runtime should be flexible to allow multiple scheduling policies to be utilized. To accommodate multiple scheduling strategies between the runtime system and the heterogeneous resources, the authors make use of queues. When the application submits tasks to be scheduled, tasks may be enqueued/dequeued from a single FIFO (e.g. greedy policy) or multiple dedicated FIFOs (e.g. work-stealing policy). Much of the analysis uses the heterogeneous earliest finishing time (HEFT) policy. The HEFT scheduling policy tries to minimize the completion time for the system by summing both pending tasks and the new task's estimated execution times for all hardware resources. This can naturally lead to the scheduling of certain tasks onto heterogeneous resources that demonstrate an affinity for such tasks (provided the data granularity is ideal for that resource). In order to demonstrate this, the authors estimate task completion time dynamically online through performance (and data) modelling [9].

Modelling the performance of tasks by hand can be tedious as it requires detailed knowledge of both architecture and algorithm. This is further exacerbated for a system with much heterogeneity. Pre-calibrating performance models through benchmarking suites also does not present a viable solution as it does not represent a realistic workload, does not take into account the dynamic behavior of the system, or both. The authors decided on a history-based approach, where actual performance data is captured and referenced during runtime for the benefit of scheduling future scheduled tasks. The authors do assume some regularity hypothesis in many algorithms utilized *fixed sizes of data, or limited sets of data*. Execution information for a given kernel and hardware are stored and referenced in their respective online tables. The parameters to a given kernel pass through a hash function that ultimately generates an index into the tables. The authors claim this history-based approach is a simpler than multivariate performance models that distinguish table entries for data sizes such as a 1024×50 matrix vs. a 50×1024 matrix.

2.4.4 CaaS:

The Core as a Service (CaaS) project was designed as a service-oriented architecture (SOA), similar to what is used in the networking domain, and targets an MPSoC [55]. CaaS defines two types of tasks: *scheduling servants* or scheduling agents which disperse tasks to *computing servants*. Computing servants receive and service the tasks. Computing servants are categorized into software servants such as general purpose processors (MicroBlaze), and hardware servants such as DSPs or custom hardware accelerators. Similar to Harmony, CaaS utilizes a directive-based programming model referred to as the flexible programming model (FPM) [54]. CaaS can automatically infer inter-task dependencies through high-level programming annotations captured during compilation and utilized during runtime. Such annotations include the specification of directionality (inputs/outputs) on a function's parameters.

Like Harmony and StarPU, the user writes a sequentially written program, which executes on a master (main) processor. Users can also utilize FPM-library calls which are functions designated to execute on computing servants (either within software or hardware). As these non-blocking calls are encountered, the runtime scheduler determines task dependencies by utilizing the function's annotations in order to automatically execute a subset of them concurrently. Additionally, it is mentioned in [54] that one of the functions for the scheduler is to estimate the total execution time for each computing servant. This can be used to decide which target computing servant should execute the given task based on earliest finishing time. However, no details are given on this automatic adaptive mapping scheme, and how execution time is estimated and its cost for doing so.

In addition to automatic detection of task dependencies, Caas also provides profiling mechanisms and runtime partial reconfiguration. Profiling mechanisms are used to provide feedback to the programmer on what sections of their code is executed frequently. It was suggested that this feedback can be provided to the runtime's scheduler for more efficient task mapping. Partial reconfiguration is also provided as a runtime service and is invoked on an FPM library call-by-call basis when the target resource is currently not present. Furthermore, CaaS does not provide any details on mitigating the overhead of partial reconfiguration with computation. Therefore, whether invoking the partial reconfiguration service is warranted does not weigh in on the scheduling decisions of the runtime. This is up to the user who is granted explicit control as to where computation should occur though the FPM-library calls.

2.4.5 CAP-OS:

Hübner et. al. in [28, 27] reports on an adaptive OS referred to as the Configuration Access Port OS (CAP-OS). The authors argue that traditional task scheduling made on resource (processor) availability and priority does not suffice for systems with runtime reconfigurable hardware. Their work addresses task scheduling onto reconfigurable hardware consisting of processors, and directly/indirectly attached accelerators for real-time systems.

CAP-OS' programming model is task-based and transparently resolves any task inter-dependencies through the generation of task graphs. For every task graph, a global deadline is provided. Additionally, each task is augmented with meta-data. This meta-data includes the task ID, predecessor/successor tasks, algorithm type or hardware constraints, partial bitstream file or software binary, and execution and reconfiguration times to name a few. The specification of an algorithm type aids in maximizing the reuse of previously allocated resources during runtime. It does so by assigning tasks with similar hardware requirements and/or specific hardware configurations. Unfortunately, there are no details provided as to how the hardware resources are identified to the appropriate algorithm type.

CAP-OS classifies tasks into three categories: software tasks (processors), codesign tasks (processors augmented with hardware accelerator(s)), and hardware tasks (hardware accelerators directly attached to the system's interconnect). For both codesign and hardware tasks, CAP-OS' toolchain [26] guides the user in profiling their code for estimating execution time and for including hardware task implementations. While software and hardware tasks execute on both processor and hardware accelerators respectively, codesign tasks provides an opportunity for increased system adaptability and further scheduling flexibility. CAP-OS leverages this flexibility by taking into account appropriate targets to schedule tasks in order to satisfy timing constraints. This same technique of an earliest deadline first (EDF) scheduling policy mirrors StarPU's HEFT scheduler.

Chapter 3

Implementation

This chapter describes the implementation details in this thesis. The chapter begins by elaborating HybridThreads (hthreads), a micro-kernel of which this entire thesis is built. Following this are specific details on the compilation framework and also limitations/assumptions about hthreads. Next, information is given on the polymorphic function library, its use in enabling users for quick and portable use of hardware accelerators, and the runtime decisions to provide a level of transparency to the user. After this follows a section describing how this thesis goes about a minimal overhead of determining processor specialization for a thread. Finally, this chapter concludes by describing the runtime system and how it uses the previously aforementioned details to map threads to processors.

3.1 HybridThreads (hthreads)

HybridThreads (hthreads) is a hardware/software co-designed micro-kernel operating system that provides a POSIX threads (Pthreads) compliant programming environment, and abstracts both general purpose processors and custom hardware accelerators as threads under a unifying, heterogeneous platform [8]. An example of an hthreads system is shown in Figure 3.1. At a minimum, hthread systems contain one processor upon which a portion of the OS resides. Here, *software threads* can be created and executed on this main processor, referred to as the host processor. Hthreads also supports *hardware threads*, which are defined as custom hardware accessible through a Hardware Abstraction Layer (HAL). The HAL is a finite-state machine (FSM) referred to as the Hardware Thread Interface (HWTI), and serves as the communication layer between the OS and the custom hardware. In addition to abstracting the execution of threads on custom hardware, the HWTI supports synchronization primitives such as mutex locks and conditional signals, extending the Pthreads compliance from software to hardware threads.

The HAL extended the threading model onto custom hardware but the FSM within the HAL posed a problem for scalability. Continuing to expand OS service support through additional hardware application portable interfaces (APIs) was increasingly expensive in area and latency. To address this, hthreads moved the functionality of the HWTI to software using general purpose processors slaved to the host processor. As opposed to directly attaching to the interconnect, custom hardware attached directly to slave processors. The slave processors coupled with memory-mapped Block RAM (BRAM), is collectively referred to as the Virtual-HWTI (V-HWTI) [5, 7]. Through the inherent abstractions of software, this provided a more scalable solution for future extensions of OS services: adding a new OS system call did not require any hardware design and verification within hthreads. The use of general purpose processors also increased the flexibility of system. Users have the option to write threads in ways such that the thread runs entirely on the processor, the custom hardware or distributively on both. Provided the necessary software abstractions for utilizing custom hardware attached to slave processor allows programmers to write portable code, agnostic to the type of hardware it executes on. This presented new avenues of research; one in particular included integrating heterogeneous slave processors. However, the question that was raised is how can the host seamlessly target threads onto multiple heterogeneous processors during runtime?

3.1.1 Heterogeneous Compilation Flow

Thread creation on an hthreads' heterogeneous MPSoC platform is achieved through an automated compilation flow. Multiple ISA-specific binaries are produced by compiling the original source file for said targeted architecture(s), *flattening* or decompressing the binary to ensure a contiguous memory layout within its binary output, embedding this modified binary

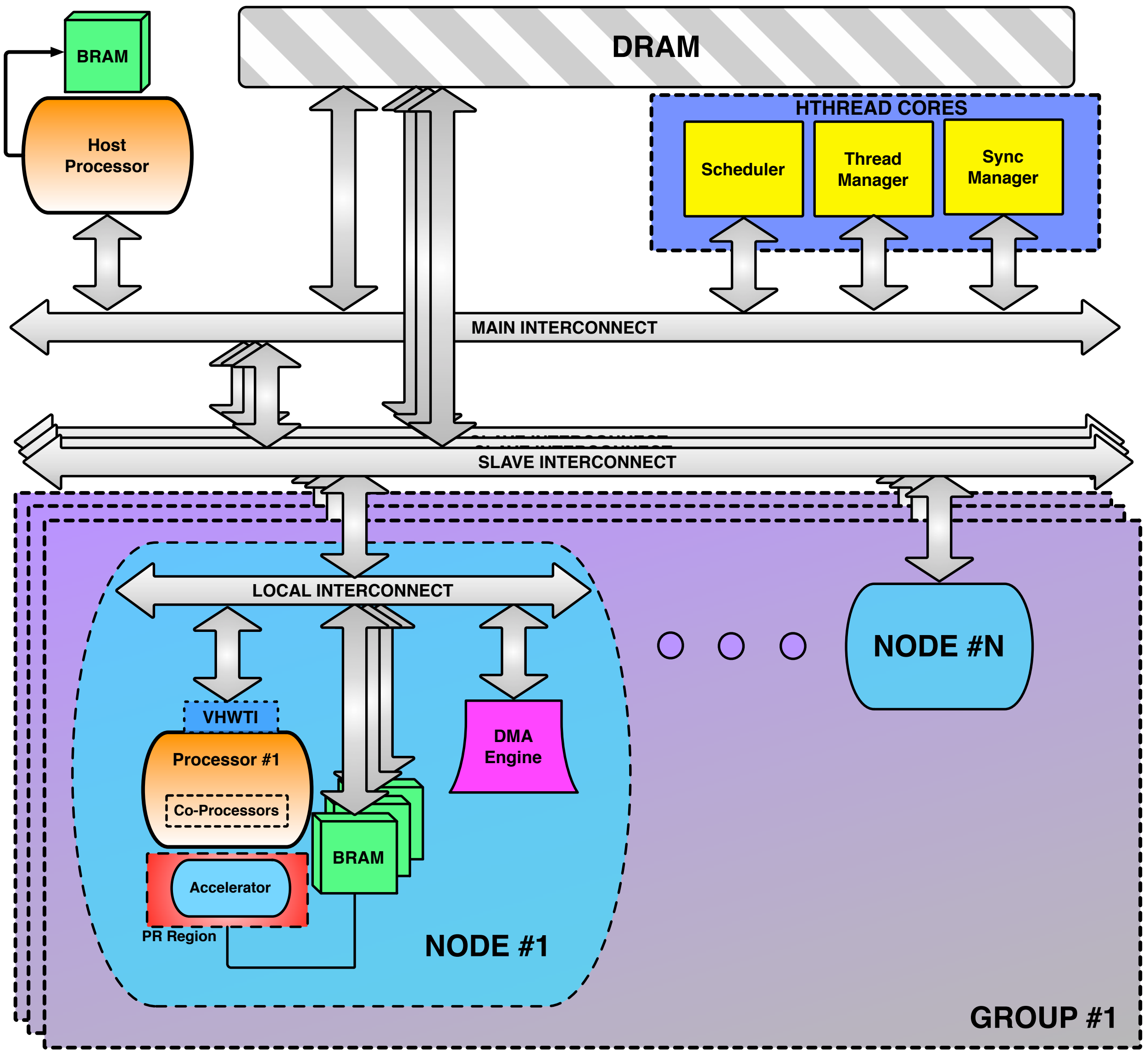

Figure 3.1: An example of an hthreads system

within the host processor's binary as a data structure, and recording the offsets at which point within the embedded binaries do *thread functions* reside for each heterogeneous processor. With this information, the runtime can appropriately supply the correct address of thread functions to the corresponding processors. As the name implies, thread functions are functions that threads can execute. Within the compilation flow, they differentiate themselves from other functions only in their name declaration: the name of thread functions are suffixed with *_thread*. This allows for easy symbol identification when compiling the user code and recording thread offsets in the embedded binaries. Figure 3.2 [5] from Agron [5] gives a high level overview of this compilation flow and shows how a programmer can write a single source file without processor specific optimizations, and target many different architectures during runtime.

The early version of the heterogeneous compilation flow was created and automated by Agron [5]. At the time, it only supported similarly configured MicroBlaze slave processors but provided a proof of concept that supporting heterogeneous processors between host and slave processors was possible. As part of this work, the compilation flow was modified to extend heterogeneity amongst slave processors. The details of this expansion is provided in Section 3.3.1. However, there still remains a few support limitations to this heterogeneity at the slave processor level.

3.1.2 Limitations of hthreads

While the hthreads platform serves as a vehicle for heterogeneous MPSoC exploration, there exists a number of limitations that should be mentioned. The following are a few that are relevant to this thesis. First, hthreads has predominately targeted Xilinx FPGAs. As a result, the hthreads microkernel that exists as IP cores have evolved to utilized the interconnect with each new generation. Currently, the hthreads platform supports the new AMBA busses (AXI4-Lite, AXI4-Stream) [60]. Additionally, hthreads supports the first-party MicroBlaze [63] processor that is designed to utilize the AMBA bus (and previous Xilinx interconnect offerings). No support has

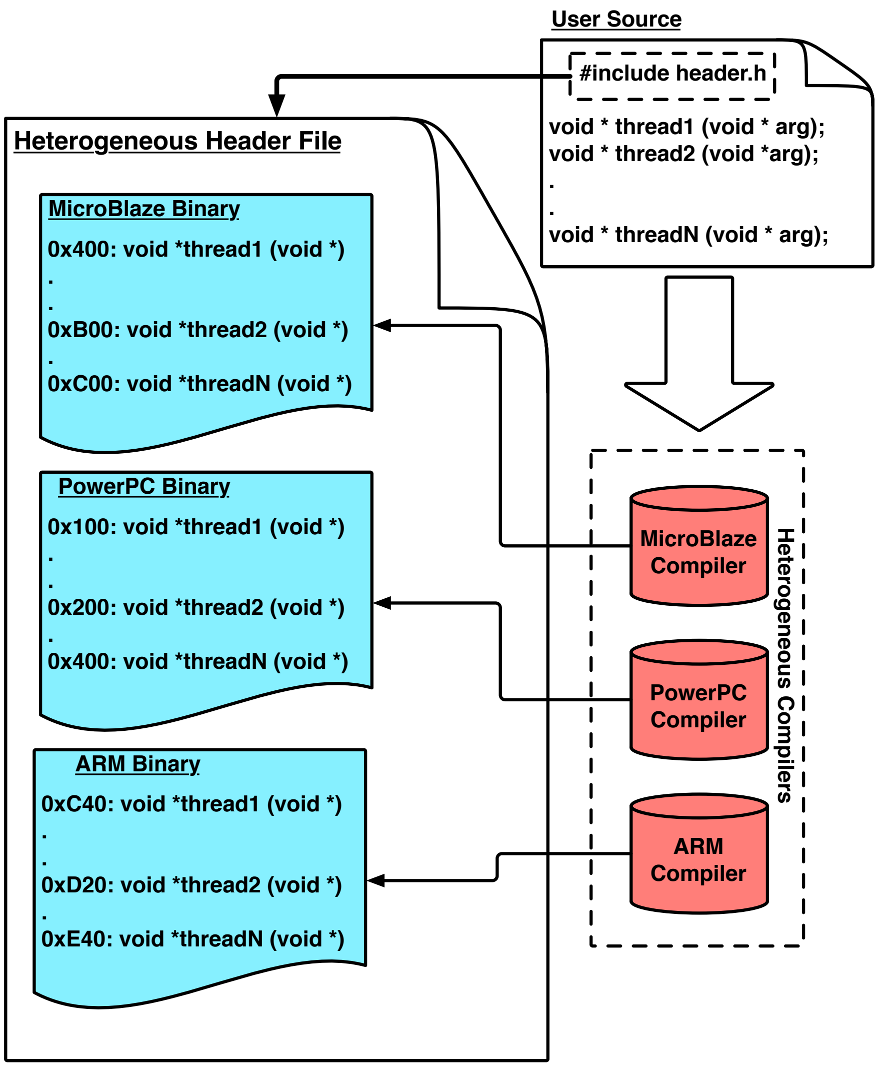
User Source
#include header.h
void * thread1 (void * arg);
void * thread2 (void *arg);
.
.
void * threadN (void * arg);
Heterogeneous Header File
MicroBlaze Binary
0x400: void *thread1 (void *)
.
.
0xB00: void *thread2 (void *)
.
0xC00: void *threadN (void *)
PowerPC Binary
0x100: void *thread1 (void *)
.
.
0x200: void *thread2 (void *)
.
0x400: void *threadN (void *)
ARM Binary
0xC40: void *thread1 (void *)
.
.
0xD20: void *thread2 (void *)
.
0xE40: void *threadN (void *)
MicroBlaze Compiler
PowerPC Compiler
ARM Compiler
Heterogeneous Compilers

Figure 3.2: A flow chart describing the embedding process for heterogeneous binaries.

been provided for other third-party *soft* processors such as the Leon3 or ZPU [25, 66] as either they have not included support for interconnects such as the AMBA bus, no supported and complete build tool-chain, or were generally too large for the hthreads platform.

Another missing feature is thread preemption on the slave processors. This is currently supported on the host processor as software threads can be created with a variable priority. In contrast, hardware threads carry a fixed priority and are immediately dispatched to the slave processor (or the custom hardware) once received by the thread scheduler. Implementing this feature can provide opportunities of thread/computation migration across slave processors in scenarios where more computationally-capable processors become available for use.

Finally, thread creation onto slave processors is only supported by a single host processor. Although an SMP variant of hthreads exists that allows two host processors to create and schedule threads, these threads are software threads limited to executing onto the host processors. As hthreads moves towards supporting more than one application to execute at a time, the sharing of slave processors may be an important step in continuing the scalability of the platform.

3.2 Polymorphism - Dynamic and adaptable behavior

It was explained in Section 3.1, that hthreads unifies both software and hardware threads through the use of the HAL. It was also mentioned that slave processors can serve as the HAL for custom hardware. Using a processor (or software) as the front end to the rest of the system for custom hardware provides great flexibility. An example is allowing the programmer to construct a thread such that a portion executes on the slave processor and a portion executes on the attached custom hardware. However, leveraging this flexibility within a thread can introduce platform details as assumptions of the hardware must be made at time of writing. Another issue is knowing *when* to make use of custom hardware. Section 1.2.2 highlighted that depending on the hardware resource a thread requires, there may be an intersection point at which executing the function in software compared to shifting the work to a hardware accelerator exists. Furthermore, this point is not fixed

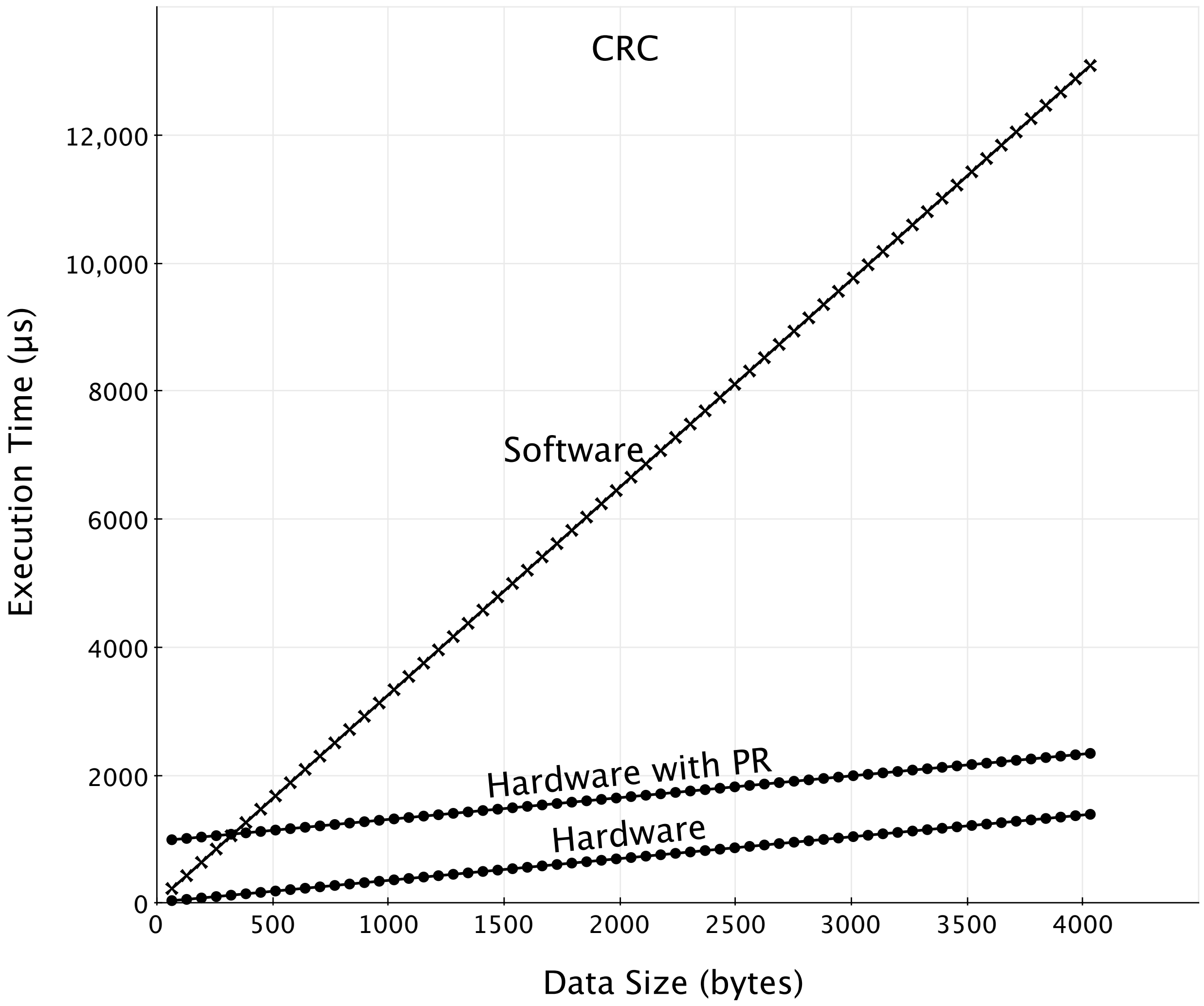

Figure 3.3: CRC Polymorphic Function

and varies between hardware accelerators (see Figures 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9). This is further exacerbated for a heterogeneous system where the performance curves amongst processors themselves can differ from one another, thereby introducing greater variance to performance.

This work introduced polymorphic functions, which were created to offload these decisions from the programmer to the runtime system. Polymorphic functions have been investigated in other research [20, 56, 10]. They are wrapper functions that coordinate the handling of data and perform partial reconfiguration (PR), utilize pointer information for the execution of the function in software, and provides the control flow for deciding how and when to execute the function: hardware or proceed in software.

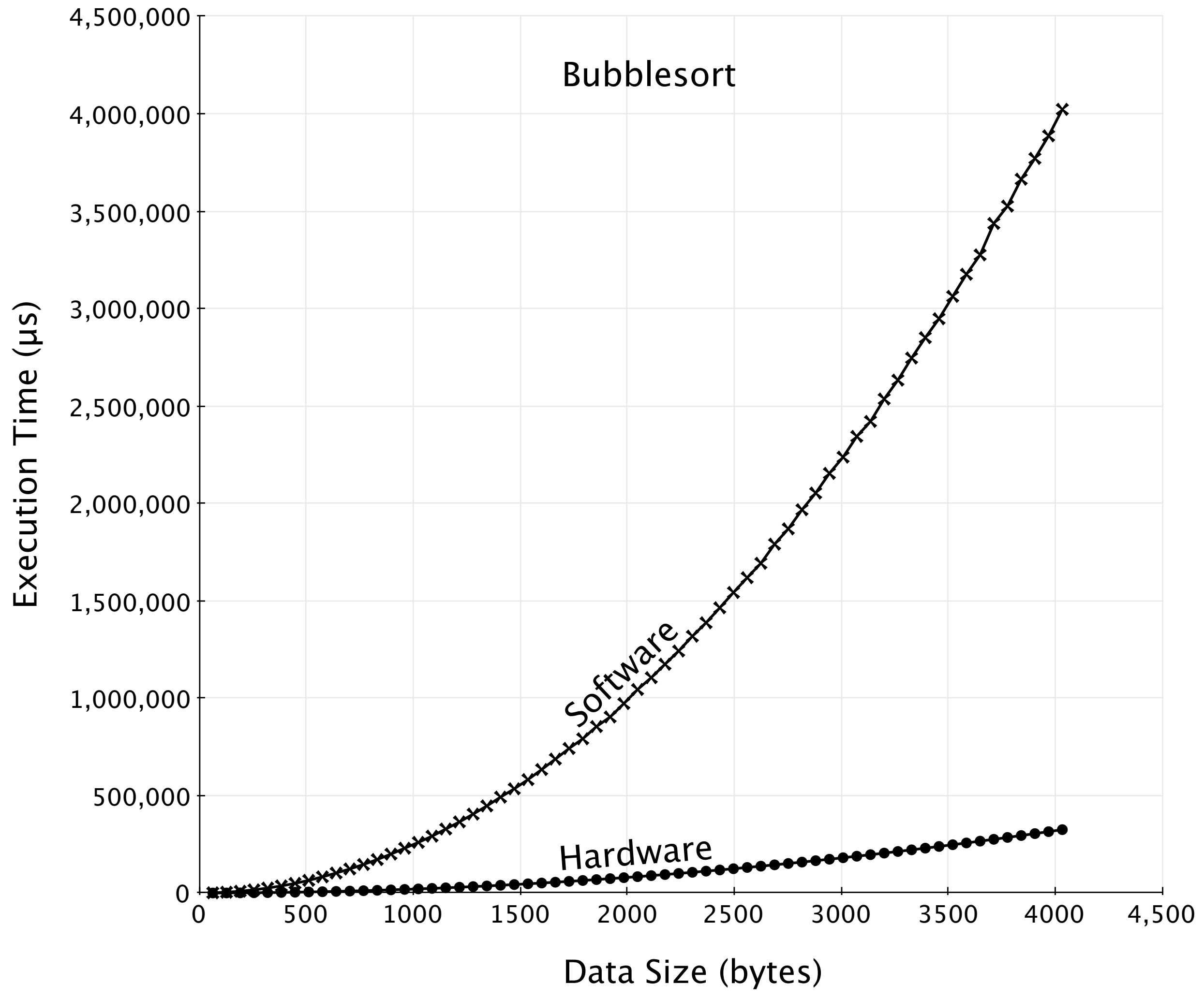

Figure 3.4: Bubblesort Polymorphic Function. Hardware with PR overhead matches Hardware plot.

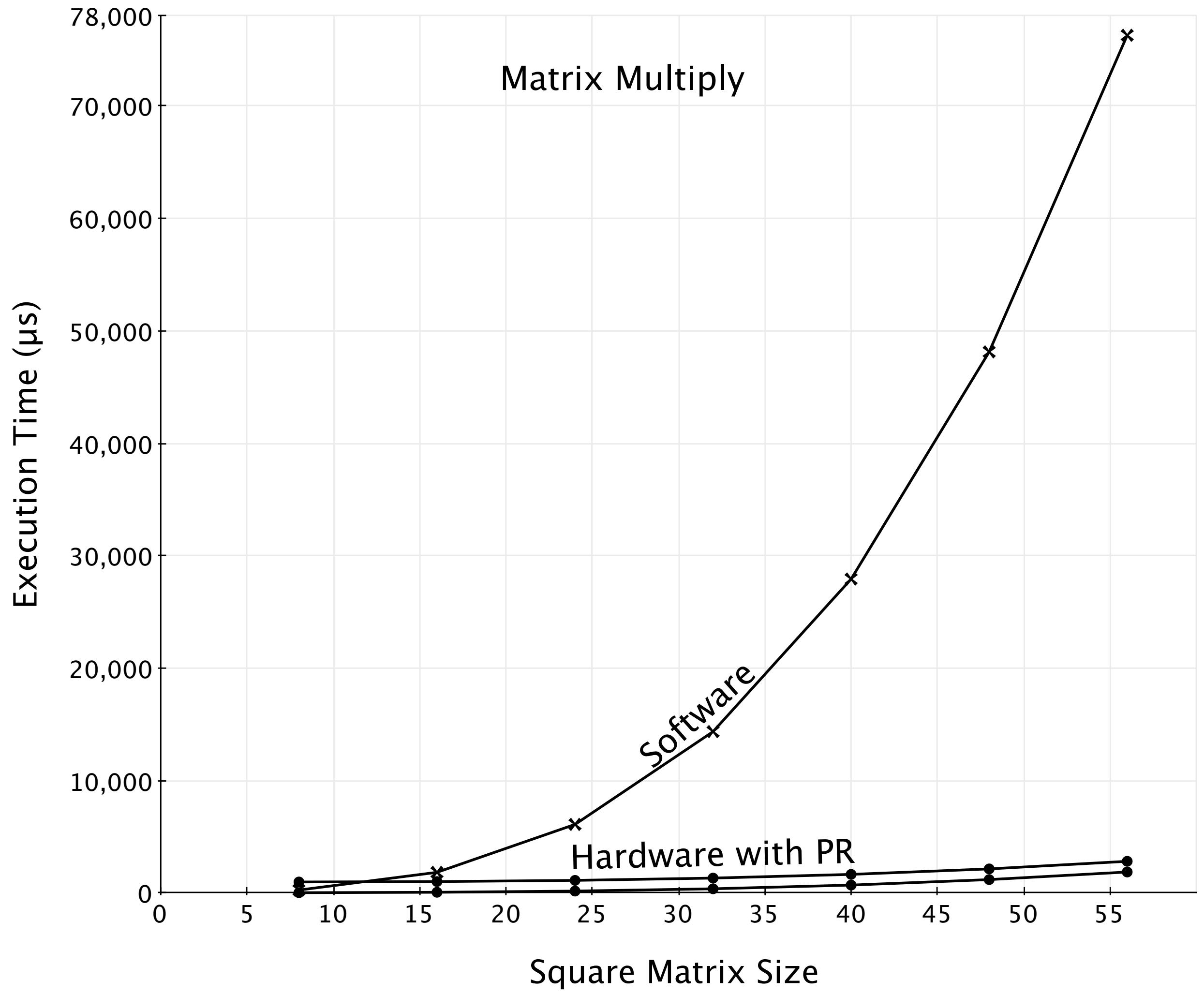

Figure 3.5: Matrix Multiply Polymorphic Function

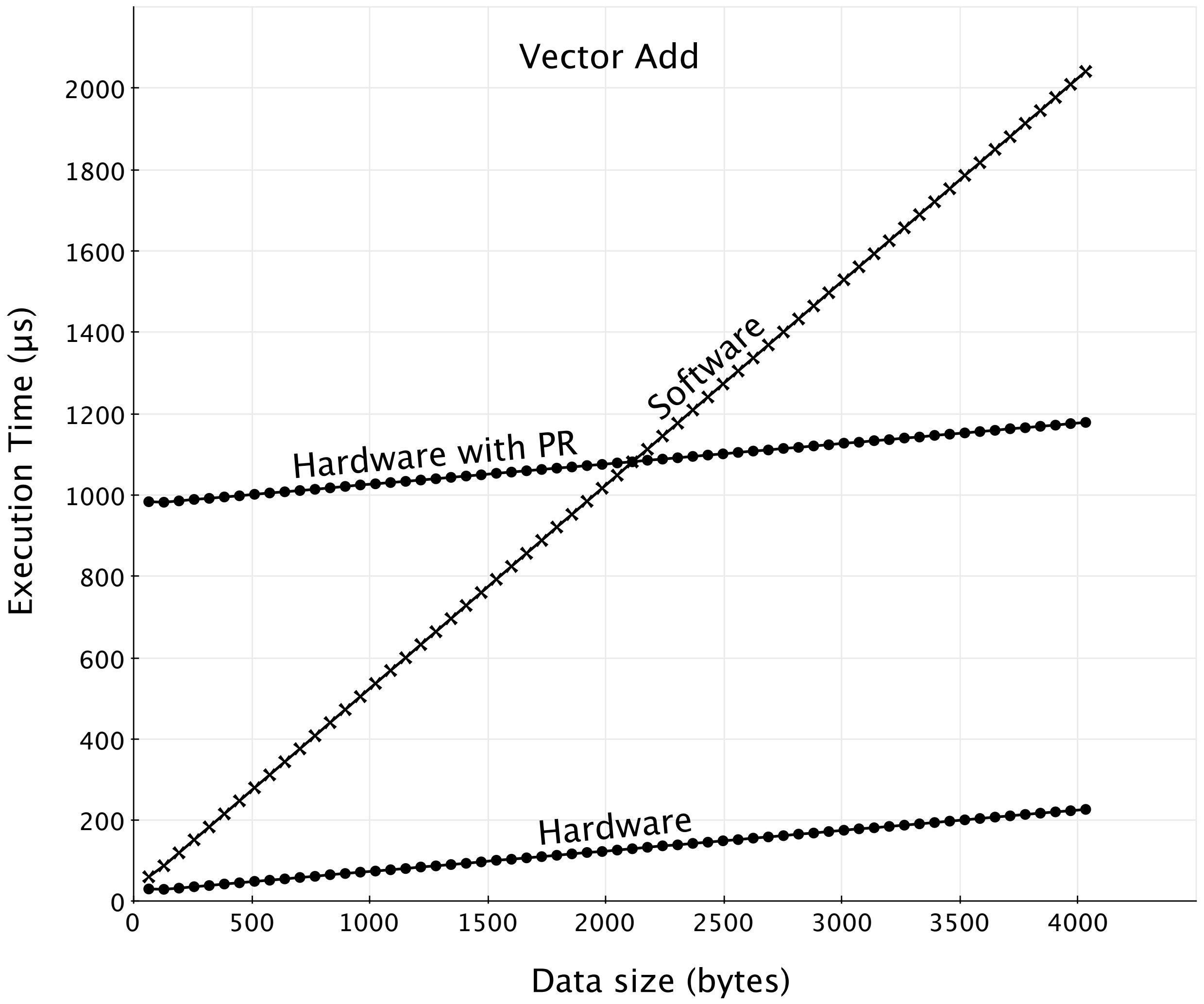

Figure 3.6: VectorAdd Polymorphic Function

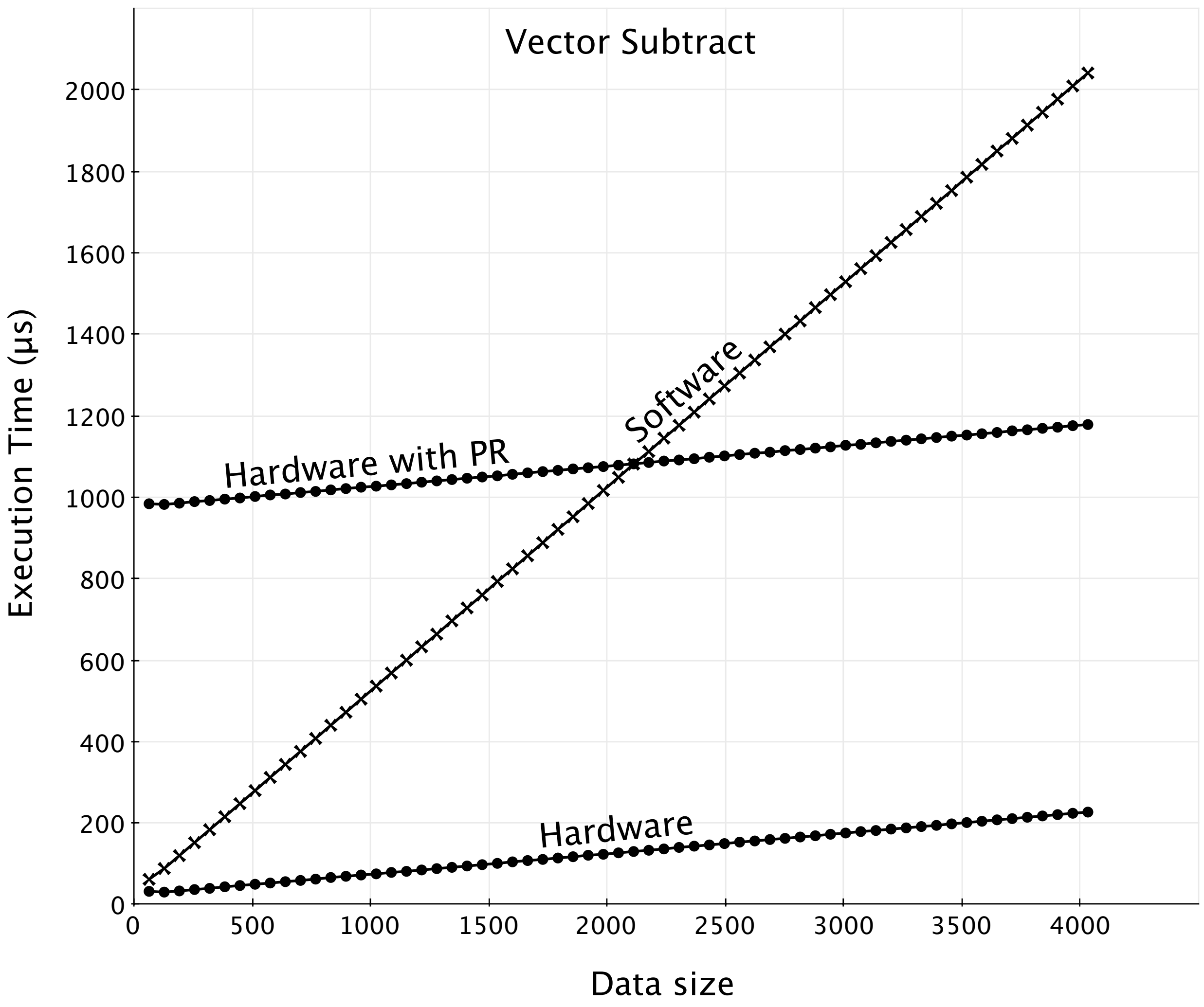

Figure 3.7: VectorSubtract Polymorphic Function

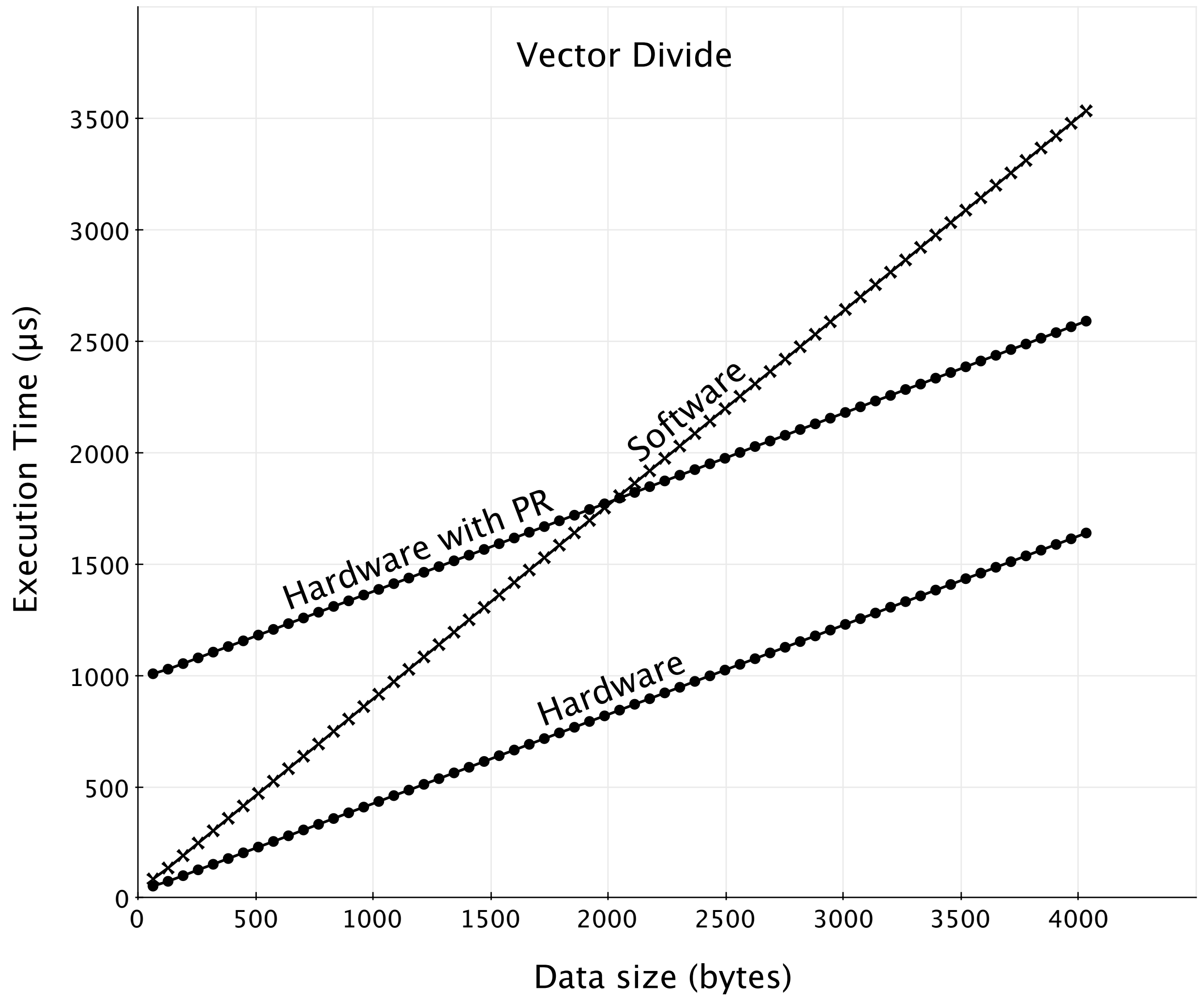

Figure 3.8: VectorDivide Polymorphic Function

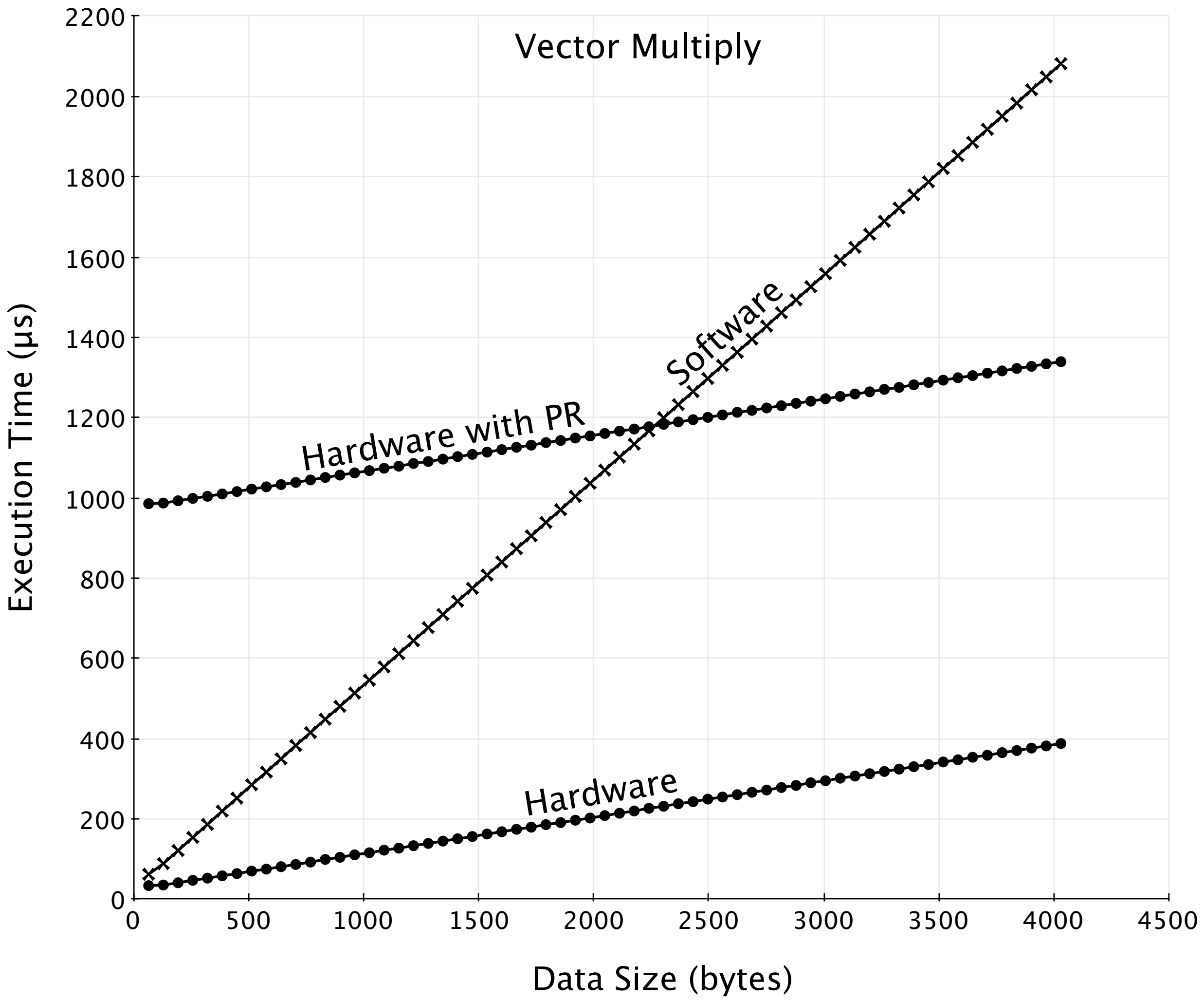

Figure 3.9: VectorMultiply Polymorphic Function

The control logic within a polymorphic function decides when to execute in hardware or software based on the runtime information. The runtime information used within a polymorphic function call are as follows:

- Software and hardware execution times
- Partial reconfiguration overhead
- Current hardware accelerator attached to the processor (if any)
- Determine whether accelerator is static or dynamic

This list does not represent a complete list but only those that are currently utilized when invoking a polymorphic function. For the current functions in the polymorphic function library as seen in Figures 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9, it suggests that it is always best to execute in hardware if the necessary hardware exists. However, when the hardware does not exist, each slave processor checks whether it can swap in the needed accelerator in the case it has a dynamic accelerator attached. If so, the slave processor then compares the hardware and software execution times of a polymorphic function along with the PR overhead, to decide whether or not to execute the function in software or perform PR of the necessary accelerator and transfer computation directly onto it. The entire process for invoking a polymorphic function is provided in Figure 3.10.

The runtime system keeps information about the performance of polymorphic functions for each heterogeneous processor within the tuning table shown in Figure 3.11. The tuning table is populated once at boot (similar to what is done in [58]) and is used when comparing the software and hardware executions of a polymorphic function. The tuning table is stored offline for each system configuration to avoid repeatedly generating it at every bootup. Currently, there exists a single PR controller, responsible for receiving commands by the processors in the system and performing PR on the dynamic accelerators. The performance data that is generated assumes little to no bus, memory, and/or PR controller contention. This is a non-issue however, as the systems integrating PR are currently limited to a maximum of 16 slave processors.

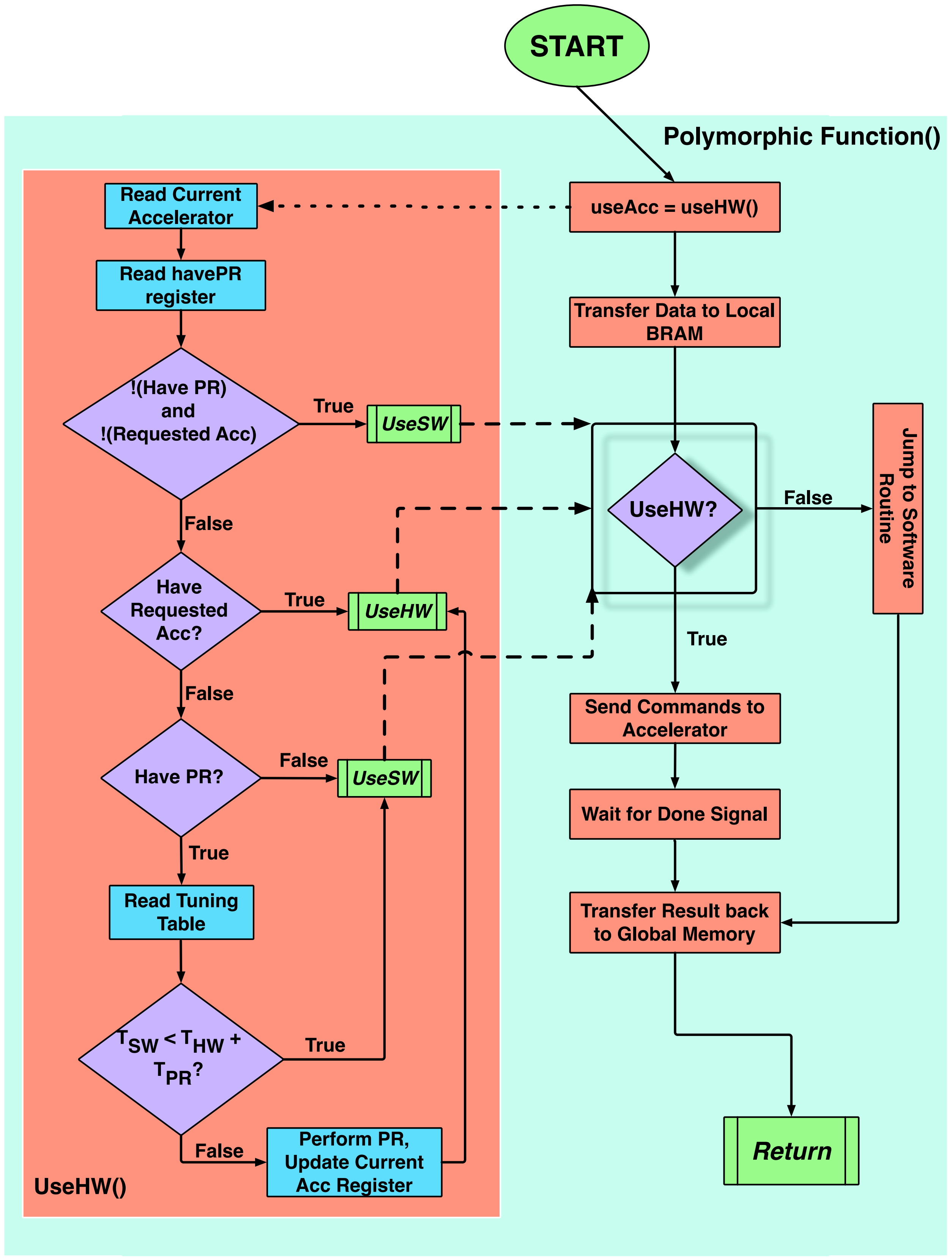

Figure 3.10: Flow chart describing polymorphic function invocation.

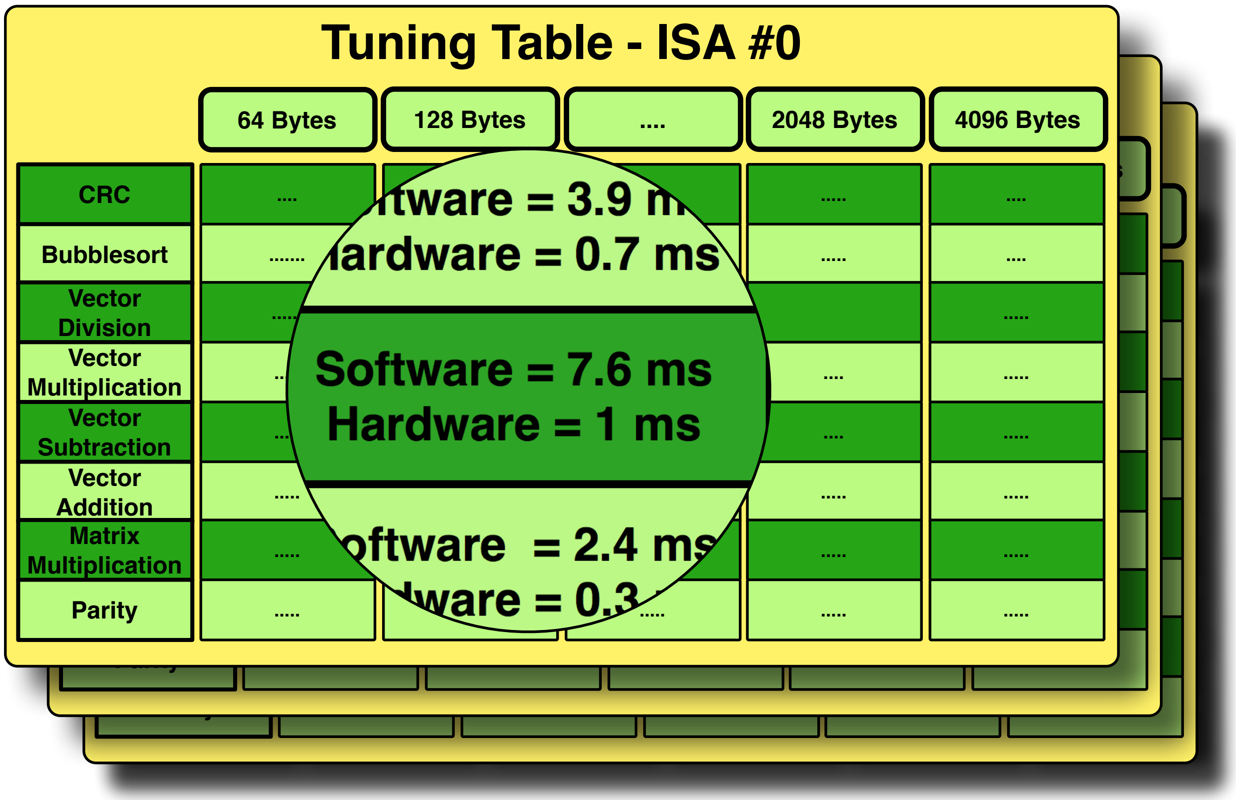

Figure 3.11: Three-dimensional tuning table storing polymorphic function performance data.

3.3 Determining processor specialization - Thread affinity

While a polymorphic function's performance data can be automatically generated or loaded at boot, performance data for the entire thread body remains unknown until runtime. One of the goals set forth by this thesis is to not require the end user to manually profile their code in an effort to determine where best to map a thread. Additionally, while profiling threads can be automated, it is assumed that the polymorphic function library and the code that it includes remains static and is not under rapid development as a user's source code. The polymorphic function library only needs to be generated once for a platform and can be loaded into memory for said platform for *any* user program wanting to take advantage of it. Furthermore, one of the use-case scenarios this work targets is operating systems, or time-sharing software that executes for a long period of time. Whenever a new program/task is introduced into the system, it is infeasible to require the entire system to halt in order to benchmark this new task.

To support this level of performance-aware transparency, this thesis combines information generated offline by the compiler with runtime data used when scheduling a thread. The items that are generated offline include information about the platform, a processor-priority list for each thread, and a polymorphic function call-graph. All three of these topics are discussed in the following sections.

3.3.1 Platform Analysis

A process that was added within the heterogeneous compiler was the platform analysis stage. A similar operation to what the commercially available Xilinx XSDK tool provides, platform analysis allows the compiler to grab information about how an FPGA MPSoC platform was configured in order to determine how to compile one's program against it. This information includes how many (slave) processors are in the system, what accelerators are attached to the processors (if any), is PR supported, and how each slave processor is configured (specifically,

```
<PROCESSORS>
    <PROCESSOR Hthreads_ISA="mblaze" HWVERSION="9.5">
        <PARAMETER NAME="HWVERSION">
            <OPTION VALUE="9.5" FLAG="-mcpu=v9.5"/>
        </PARAMETER>
        <PARAMETER NAME="C_USE_FPU">
            <OPTION VALUE="0" FLAG="-msoft-float"/>
            <OPTION VALUE="1" FLAG="-mhard-float"/>
            <OPTION VALUE="2" FLAG="-mhard-float -mxl-float-convert -mxl-
                float-sqrt"/>
        </PARAMETER>
        <PARAMETER NAME="C_USE_BARREL">
            <OPTION VALUE="0" FLAG="-mno-xl-barrel-shift"/>
            <OPTION VALUE="1" FLAG="-mxl-barrel-shift"/>
        </PARAMETER>

        ...... File truncated for brevity ....
```

Figure 3.12: Compiler flags for the MicroBlaze corresponding to configured parameters.

what co-processors are enabled). Figure 3.12 represents the file used by the compiler to map certain values of a MicroBlaze (version 9.5) parameter to compiler flags that should be passed during compilation. Additionally, having knowledge of whether the system supports PR or not dictates whether the appropriate PR support libraries are provided within the compilation build. The result of platform analysis has enabled the hthreads compiler to support differently configured slave processors.

3.3.2 Instruction Flagging

The platform analysis stage also enabled the capability to flag certain instructions, known as the instruction flagging stage. The instruction flagging stage during compilation reads the information passed on from the platform analysis, specifically enabled co-processors on each slave processor, and flags the co-processor instructions for each thread function. The assumptions made here is that hardware is generally faster than software and all processors share a common base architecture. If instructions that make use of co-processors are found during compilation, then this processor should be given more weight, hence the flagging of instructions. Similar to Figure 3.12, Figure 3.14 previews the file used in this stage to support this flagging process.

```
typedef struct {
    Hbool idiv;
    Hbool fpu;
    Hbool barrel;
    Hbool mul;

    // Reserved for future use
    Hbool idiv_ratio;
    Hbool fpu_ratio;
    Hbool barrel_ratio;
    Hbool mul_ratio;

    // First possible accelerator, used
    Hint first_accelerator;

    // PR preference: valid if multiple
    // polymorphic functions called.
    Hbool prefer_PR;

} thread_profile_t;
```

Figure 3.13: Thread profile structure that describes what co-processors a thread can use for a given platform or a set of slave processors.

Based on the targeted platform, this allows the compiler to generate a *thread profile* (Figure 3.13). A thread profile provides information about what co-processors *can* a thread function utilize considering the given set of slave processors in the targeted platform and how they are configured. This information is stored within the first 4 parameters of this data structure. As a result of this information, finding a suitable processor can be generalized to one whose enabled co-processors intersects with these values in a thread profile. This generalization provides a simple estimation of finding a suitable thread-to-processor mapping. It also does not add any overhead cost to the runtime as the compiler generates a priority array list for each thread function, sorted by most preferable to least preferable processor with respect to co-processor utilization. During runtime, the scheduler can index into this ordered array when scheduling threads.

3.3.3 Polymorphic Function Call Graph

A final addition that was added to the heterogeneous compiler was the generation of a function call graph. This stage has been written to apply to any function symbol, but is used in this thesis to

```
<PROCESSORS>
    <PROCESSOR Hthreads_ISA="mblaze" HWVERSION="9.5">
        <PARAMETER NAME="C_USE_DIV" VALUE="1" OPCODE="010010" STRUCT_ENTRY="
            idiv">
            <OPTION VALUE="idiv"/>
            <OPTION VALUE="idivu"/>
        </PARAMETER>
        <PARAMETER NAME="C_USE_HW_MUL" VALUE="1" OPCODE="010000"
            STRUCT_ENTRY="mul">
            <OPTION VALUE="mul"/>
            <!-- C_USE_HW_MUL = 2 -->
            <OPTION VALUE="mulh"/>
            <OPTION VALUE="mulhu"/>
            <OPTION VALUE="mulhsu"/>
            <OPTION VALUE="muli"/>
        <PARAMETER NAME="C_USE_FPU" VALUE="1" OPCODE="010110" STRUCT_ENTRY="
            fpu">
            <OPTION VALUE="fadd"/>
            <OPTION VALUE="frsub"/>
            <OPTION VALUE="fsub"/>
        </PARAMETER>

        ...... File truncated for brevity ....
```

Figure 3.14: Instruction flagging information for the MicroBlaze's co-processors.

select polymorphic functions only. With this information, it is possible to know whether a thread requires partial reconfiguration. If it is discovered by this stage that a thread makes multiple, *different* polymorphic calls, then it is assumed PR is preferred due to slave processors containing at most one dynamic/static accelerator. If it is found that only a single polymorphic function call is invoked, partial reconfiguration is optional. However, if there are no polymorphic functions called within a thread, then this thread should not consume a processor which has a static/dynamic accelerator (unless it was found that this processor has a high affinity for the thread).

In all of these cases, this information should be conveyed to the runtime system. This is done using the thread profile data structure previously shown in Figure 3.13. The field, *first_accelerator* describes whether or not a processor calls a polymorphic function. It also provides a way for the scheduler to reduce the amount of PR needed by matching a processor with the initial accelerator it may need. The other field, *prefer_PR* is used by the runtime to indicate this thread may call multiple polymorphic functions. It is not guaranteed that the thread will as the polymorphic functions may all be conditionally invoked. If PR is preferred however, threads should be mapped to processors that have dynamic accelerators. Whether the polymorphic calls execute in hardware or software will depend on the runtime performance data of that call. However, having the option for reconfiguration may prove beneficial for times *when* it is needed.

3.4 Runtime System

As outlined in Section 3.1.1, the compiler supplies the runtime system the correct address of thread functions for each heterogeneous processor. This information alone is not sufficient for self-awareness and for dynamically mapping threads to more capable performing processors than others. As was discussed in Section 1.2.2, generating a static mapping for threads to processors can also be a dangerous solution, particularly for a system whose hardware varies over its uptime. Therefore, information about the current state of the system must also be taken under consideration when placing a thread onto a processor. The following sections provide details on thread scheduling against a heterogeneous system with combinations of static, dynamic, or no accelerators attached to each processor.

3.4.1 Two-tiered Scheduler

Previously, the scheduling policy within the hthreads runtime system began with a processor availability lookup, immediately scheduling a thread once an available processor was found. Since this policy pre-existed the polymorphic function library and slave processors were considered to be identical at the time, there was not a need to analyze which processor would be best to execute a thread: all processors would produce similar results. Now, with the introduction of heterogeneity amongst slave processors also brings asymmetrical performance. This simple scheduling policy used previously would not suffice. The scheduler must now take into account several new parameters: how the processor is configured, if there are any external hardware accelerators attached (as part of the polymorphic function library), and what resources does a thread require. Additionally, the system state, namely the attached accelerators can change through DPR and the runtime must be aware of such changes when considering new threads to schedule on slave processors.

To address this, the scheduler within this thesis is a two tiered scheduler. The first tier of the scheduler that creates and schedules threads onto slave processors, executes partially in software on the host processor and partially in hardware. This scheduler attempts to map threads onto processors where the accelerated resources that it uses such as hardware accelerators or co-processors are maximized. Once a thread begins execution on a slave processor, further scheduling is required when invoking a polymorphic function call. When a polymorphic function executes, the second tier of the scheduler directs computation onto the attached hardware accelerator or proceeds in software. If the first tier is successful in mapping a thread to a processor with the appropriate resources, this potentially provides a high resource *hit-rate* as the hardware it needs currently exists on the slave processor. Therefore, the second tier supports the first in case the hardware does not exist (see the previous section, 3.2, for more details). Therefore, it is important that the first tier, from here on referenced as the scheduler, maps threads to processors with the needed resources. These processors are referred to as the *ideal* processors.

3.4.2 Mapping threads to processors

The strategy towards mapping threads onto ideal processors is to utilize information on processor specialization for each thread. As previously mentioned in Section 3.3.2, the compiler generates an ordered list from most preferred to least preferred processor for every thread function. This enables the runtime to consider processor availability, with implicit priorities assigned to processors. If the application and slave processors are sufficiently diverse, this ordered list can map threads to their processors at a higher rate than choosing the first available processor. Other bodies of research have chosen instead to track historical data [10] or benchmark [39] tasks when input sizes can vary as is the case with this thesis. With the approach taken in this thesis, the scheduler does not introduce additional overhead as it operates on pre-generated information. However, this data generated by flagged instructions alone is not sufficient for mapping to ideal processors. The distribution of those flagged instructions is another metric that should considered. The impact of using this metric is expounded in Section 4.3.1.

Threads invoking polymorphic functions also provide another strategy for mapping threads to processors. Polymorphic functions implicitly provide hints as to what external hardware accelerators it will use and whether or not PR will be needed (Section 3.3.3). The flow chart in Figure 3.15 summarizes the final scheduling policy the runtime uses when mapping threads to processors. The scheduler begins by first referencing an ordered list of processors, represented as the set, $P = \{p_0, p_1, \ldots p_{N-1}\}$. Processor p_0 represents any processor in the system and does not necessarily equate to processor id 0. From this list, the runtime can index from most preferable to least preferable processor, checking the availability and assigning a *rating* to each. There are three rating values: Good, Better, and Best. There is a fourth rating value, perfect, that is implied when the system finds an ideal match for the thread. Processors that fall short of an ideal match belong to one of the three values mentioned.

If an ideal match is not found, this rating mechanism allows the runtime to compare a processor's external hardware resources to the thread's hardware requirements. Internal hardware resources such as the MicroBlaze's enabled co-processors have already been taken into account through the ordering of the processor set, P. Once the runtime has traversed the entire list, it can find the next appropriate processor to map a thread. For example, if a thread prefers PR, finding a processor that has a dynamic accelerator is preferable. On top of this, knowing what polymorphic function will be invoked first also leads to matching a processor that has that initial hardware accelerator. This eliminates the need for PR once a thread begins execution. If such a processor exists, according to the flow chart, this represents the ideal case. If it is found that the processor's current hardware accelerator does not match the first invoked polymorphic function, but has a dynamic accelerator attached, it passes as the next appropriate option. Therefore, its number

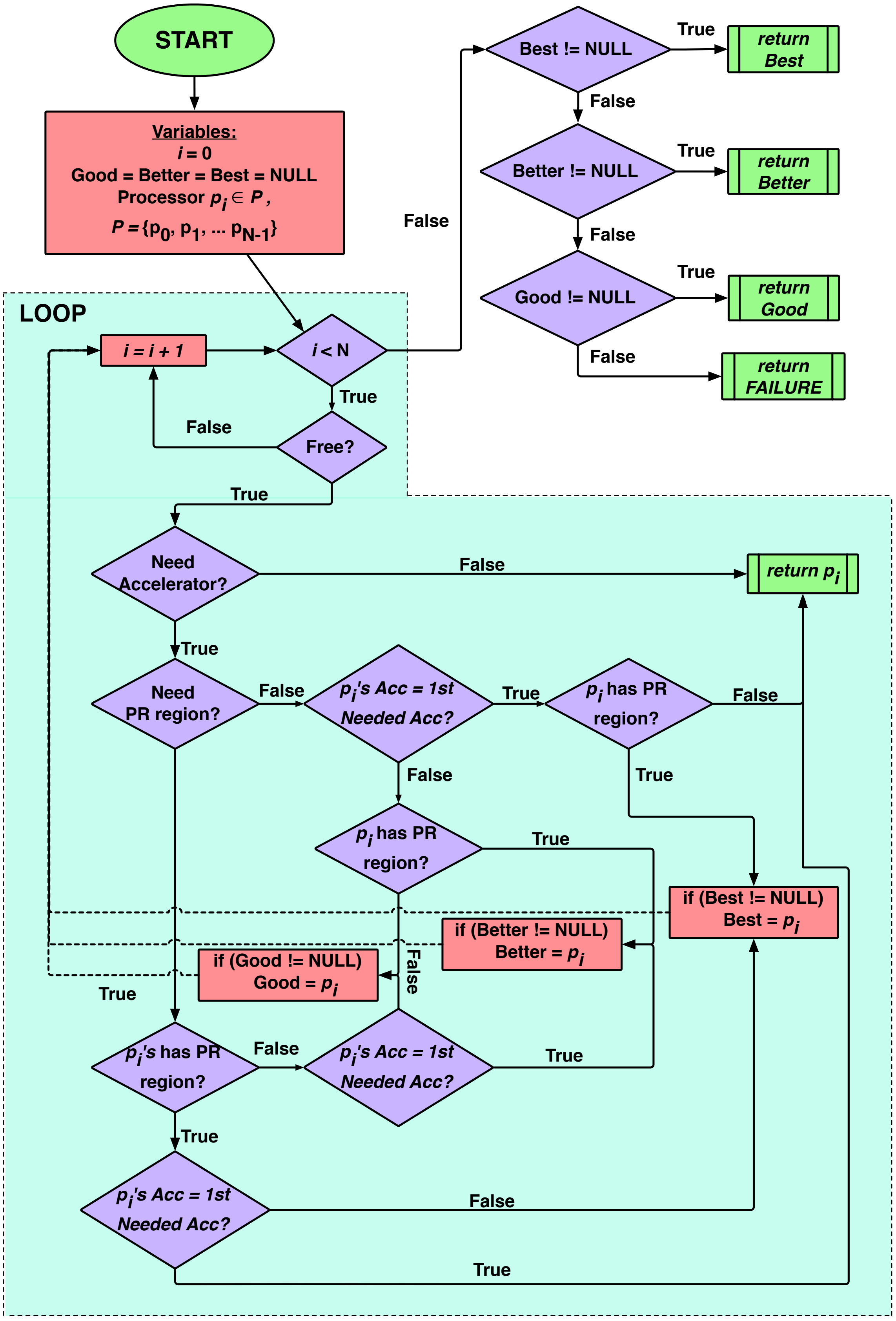

Figure 3.15: Scheduling flowchart for threads with Polymorphic functions

would be assigned to the variable *best* if it has not already been assigned. The reason why the runtime does not overwrite a previously assigned processor number is due to the initial ordered list of processors. More appropriate processors in terms of their internal hardware support for a thread, are assigned to these values first. Overwriting these values as the runtime traverses more into this list only leads to the processor reversing the intended preferences.

Chapter 4

Evaluation

This thesis presents a series of tests to evaluate the thesis' claims and questions put forth in Section 1.3. This chapter begins by first explaining how an FPGA-based MPSoC system was configured to be representative of a heterogeneous system. Although some of the testing utilized different heterogeneous systems (and in some cases, a homogeneous system) for simplifying the test, the underlying heterogeneous architecture remained the same. Next, a discussion is given on the execution of a homogeneous task load by a heterogeneous system and provides a comparison to the equivalent amount of work given in a heterogeneous task workload. Following this are brief discussions of runtime metrics that influence the performance of the self-aware runtime. Finally, the chapter concludes with tests drawn from real-world applications on a larger heterogeneous system, and provides a brief look at the resource efficiency achieved by the proposed self-aware runtime system.

4.1 Creating a heterogeneous system - Evaluation Setup

Earlier on in this body of research, the boards that were used coupled an IBM PowerPC hard processor and the FPGA fabric co-joined onto a single chip. A heterogeneous multiprocessor system could be realized by instantiating on the FPGA, soft processors that worked in tandem with the external, hard processor. However, these specific boards were retired from the vendor's toolsuite. It should be noted that Xilinx does support recent boards with similar hard and soft processor configurations in their toolsuite today [62]. However, the FPGA fabric included onto these boards are not as dense and therefore, would not be able to support large (10+)

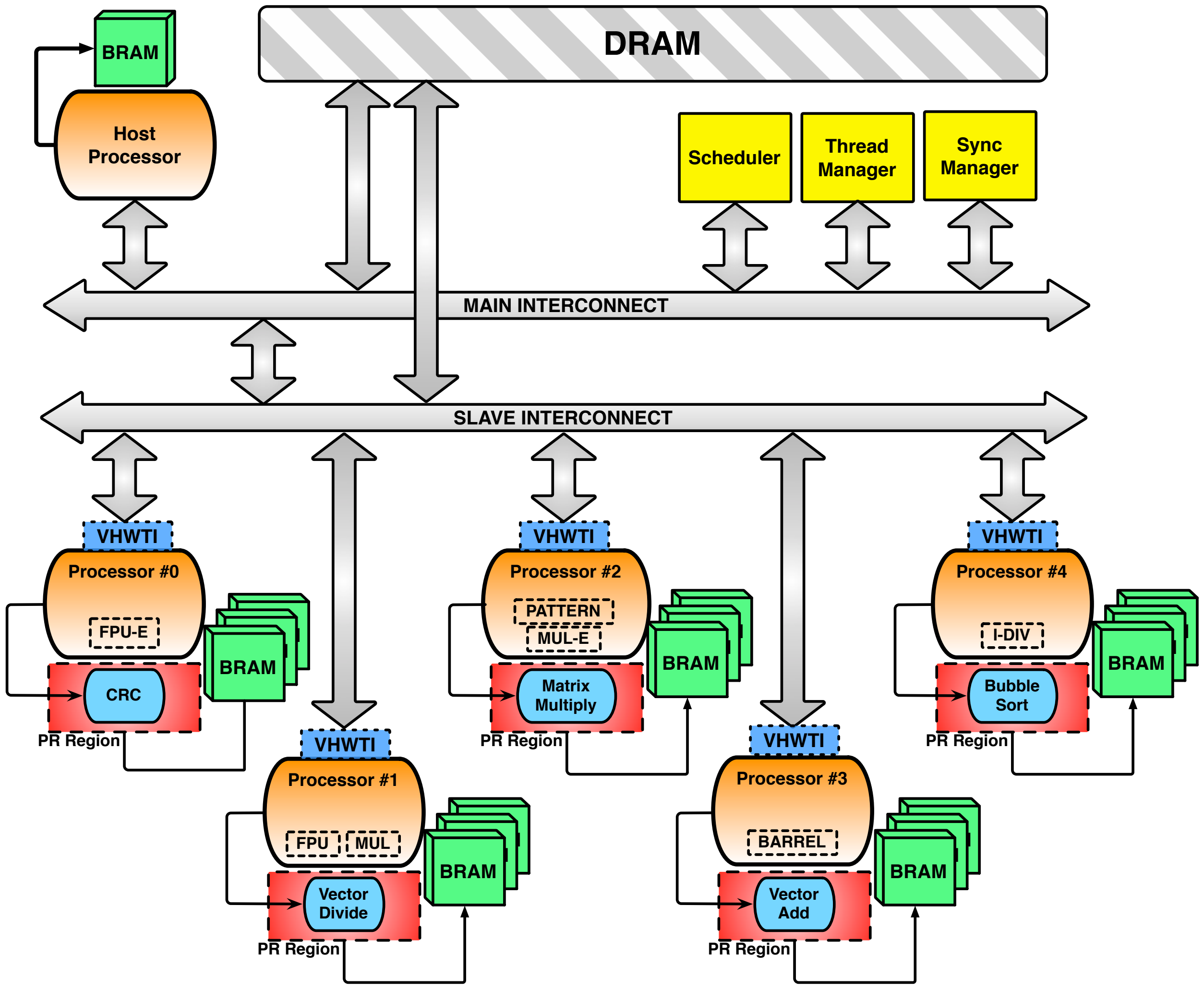

Figure 4.1: Example system containing 5 heterogeneous slave processors.

multiprocessor systems integrated with several PR regions. To accommodate larger designs, this work has used a combination of both the Xilinx Virtex 7 and Kintex 7 [61] FPGA evaluation boards. These boards do not contain any hard processors, but heterogeneous processors can be instantiated as soft processors.

To create processor heterogeneity from a soft IP cores, this thesis leverages the reconfigurable MicroBlaze [63] processor. The MicroBlaze allows the system designer the opportunity to configure it in ways that can affect performance (and area). For example, a MicroBlaze configuration can cherry-pick from several co-processors. Available co-processors include a 32-bit Floating Point Unit (FPU), 32-bit FPU-Extended (FPU-E) enabling additional

FPU instructions such as square root and floating point comparisons, 32-bit and 64-bit integer Multiplier (MUL and MUL-E), integer pattern comparator (PATTERN), Barrel shifter (BARREL), and a 32-bit integer divider (I-DIV). Additionally, tightly coupling reconfigurable regions to the MicroBlazes further created a greater degree of heterogeneity as new, but diverse support for external hardware acceleration can exist statically or change during runtime. Access to these PR regions can be modified during runtime to emulate static, dynamic, or no accelerators attached. Figure 4.1 shows an example system that containing 5 slave MicroBlazes configured with different co-processors to yield a heterogeneous MPSoC. The co-processor selections in all 5 slave processors were generated to cover all of the available co-processors and to ensure different binary outputs for each processor. There was also a need to provide an ample mix of heterogeneity, and less so on overlap. Contrary to approaches where systems are generated specifically for a single application, these systems were generated to execute multiple applications, and it is the goal of the self-aware runtime system to maximize performance with the available resources.

4.2 Executing homogeneous task loads

It is common for a single application during its runtime to spawn homogeneous tasks to *divide and conquer* a workload. This is particularly true for single instruction, multiple data (SIMD) code such as loops that act on independent data. This thesis also examines not only how well a heterogeneous task load may execute on a heterogeneous FPGA-based MPSoC, but must evaluate the performance in comparison to a homogeneous task load.

To create this scenario, the heterogeneous multiprocessor system shown in Figure 4.1 was used. While the system contains PR regions, the following example leverages only processor heterogeneity defined by the presence/absence of different co-processors in each processor. Additionally, the tasks that were chosen targets one of the MicroBlaze's co-processors if available. This set of tasks were used in an earlier motivating example found in Section 1.1).

Homogeneous vs. Heterogeneous Task Load		
	Task (# of operations)	**Execution Time (μs)**
Homogeneous Load	shift (960)	1,588
	fpu (144)	1,895
	idivu (30)	252
	mul64 (1441)	4,529
	mul32 (1201)	1,155
Heterogeneous Load	shift, fpu, idivu, mul64, mul32	828

Table 4.1: Performance of homogeneous tasks scheduled on all processors vs heterogeneous load of these tasks scheduled onto a heterogeneous system.

Each task (except *idivu*) was given a large enough data size to ensure the execution time would amortize the initial task scheduling overhead. This data size is measured in Table 4.1 as the number of operations as opposed to memory footprint. For each of these tasks stated in the homogeneous load, the number of operations for the corresponding task in the heterogeneous load is multiplied by N processors, where N in this example is 5. That is, if the shift task involved 960 shifting operations on each of the 5 processors in the homogeneous task load, the corresponding task within the heterogeneous load executing on a single processor performed $N * 960 = 4800$ shifting operations.

In both heterogeneous and homogeneous task loads, dynamic scheduling of tasks with consideration of processor specialization was used. As a result, it is possible for the scheduler to reuse a previously scheduled but currently available slave processor in the homogeneous test. This was done to investigate equal scheduling policies on both task sets. Unlike scheduling tasks on the first available processor, scheduling with processor specialization allows these tasks to execute quickly on processors that are more *fit* or hardware-equipped for the task: processors that have accelerated hardware for specific operations. This combined with the diversity of the processors in this system, allows each task to map to one or more processors capable of accelerated execution. Introducing a more complex system by integrating additional processors will only increase system flexibility due to increasing opportunities to map tasks to ideal processors. Thus, the heterogeneity of the system can continue to perform favorably for heterogeneous task loads.

Shift Task (Homogeneous Load) - Detailed view		
Thread	**Processor**	**Execution Time (μs)**
1	3	121
2	0	1,454
3	1	1,455
4	2	1,455
5	4	1,454
Total Thread Create Overhead:	111 μs	
Total Time:	1,588 μs	

Table 4.2: Details on the shift benchmark executed on a heterogeneous system.

On the other hand, scheduling several of the same tasks (especially in succession of one another) as is done in the homogeneous task set does not perform favorably on a heterogeneous system. This is due to the higher contention of similar fit processors these threads may need to be mapped to. In terms of scheduling overhead, both of the task sets involved scheduling 5 threads into the system. For a specific task, such as shift, both executed the same amount of shifting operations. While one can infer that some of the homogeneous tasks are being mapped to less ideal processors resulting in poor performance, there is a need to quantify how less ideal are these processors.

Table 4.2 highlights why scheduling consecutively the same task across multiple threads (albeit, with a $\frac{1}{5}$ workload size each) performs poorly in contrast to scheduling the entire task's workload on a single processor as is the case in the heterogeneous task load. The table details the 5 shift thread's execution time, which processor the thread mapped to, total OS scheduling overhead, and total running time. As shown, the scheduler immediately schedules the first thread to the processor(s) for which it is most fit. In this case, processor #3 has a hardware barrel shifter. However, it is the only processor with such accelerated shifting capacities. Although it finishes in 121 μs, the overhead of scheduling a single thread requires 20-25μs. As the table shows, the combined thread creation overhead was 111 μs which is less than Processor #3's execution time. Therefore, the OS was able to schedule 4 additional threads within that 121 μs execution window. If there were a 6^{th} instance of the shift task, the OS would be able to reuse Processor #3 by the

Unsigned Integer Division/idivu Task (Homogeneous Load) - Detailed view		
Thread	**Processor**	**Execution Time (μs)**
1	4	17
2	0	94
3	4	15
4	1	95
5	4	16
Total Thread Create Overhead:	110 μs	
Total Time:	252 μs	

Table 4.3: Details on the idivu benchmark executed on a heterogeneous system.

time it schedules it provided that the workload size remained the same.

The moment the data size becomes smaller, and the execution of a task on a preferred processor becomes less than the task scheduling overhead, the system will reuse that processor. This is the case for the *idivu* task which has a 15-17 μs execution time a processor that has an integer divider unit: processor #4. As shown in Table 4.3, processor #4 is reused several times as soon as it becomes available. While the the execution time reported for processor #4 is less than scheduling a thread in this system, this execution time does not take into account the slave processor performing additional thread exiting procedures as well as the hthreads scheduler IP core issuing a ”GO” command to the awaiting slave processor. While a small dataset may seem unlikely as programmers typically divide a large dataset in order to warrant the overhead of scheduling on multiple processors, it is a possible scenario nonetheless.

4.3 Runtime metrics for effective heterogeneous scheduling

While knowing what particular instructions make use of certain hardware accelerators (internal or external) can help in determining where best to map a thread, there is additional information the runtime system can utilize. The following sections explores several additional sources of information that can be supplied to the runtime system. Whether it can provide additional benefit to a heterogeneous system is also investigated.

4.3.1 Instruction Distribution

Instruction distribution describes which instructions impact the thread's execution time and thus, impact the original processor mapping discussed in Section 3.3.2. For example, at compile time it may be determined through the instruction flagging stage that a thread performs both floating point and integer division, but the execution ratio between the two operations may be 10:1. Therefore, when deciding between a MicroBlaze with an integer division co-processor and one with a floating point unit only, knowing this information can help determine where best to schedule this thread. In this case, suffering the loss of integer division on the latter processor to address the floating point instructions that dominates the thread's execution is the correct decision.

To test this, instruction distribution was manually added to a slave's meta-data profile managed on the host at runtime. This distribution was found by running the thread shown in Figure 4.2 on a MicroBlaze that contains any of the necessary co-processors the thread may require. While this is a manual procedure, it should be noted that it can be incorporated in runtime through a combination of thread execution history and performance profiling features offered within the MicroBlaze. At the time of this writing however, such features in the MicroBlaze exist but do not work reliably to provide confidence in the profiling data it captures.

The thread in Figure 4.2 contains a finite amount of shifting operations, coupled with floating point operations encapsulated within the body of a loop. If it is assumed that both types of operations that make use of their respective hardware execute in a similar amount of time, it can be concluded that the floating point operations have the potential to dominate this thread's execution time. To demonstrate this, this thread was first mapped to a processor based on only the number of instructions generated by the compiler through the instruction flagging stage. In this case, 18 32-bit shifting operations as the values being shifted are 64-bits each and 2 floating point operations. As a result, the runtime system automatically gives more weight to the processor that contains a barrel shifter. For the system shown in Figure 4.1, processor #3 contained a barrel shifter unit, and processor #0 and #1 contained a floating point units. Alternatively, if the

```c
typedef struct {
    Huint max;
    Huint shift_num;
} volatile fpu_shift_t;

void * fpu_shift_thread (void * arg) {
    fpu_shift_t * data = (fpu_shift_t *) arg;
    Huint max = data->max;
    Huint shift_num = data->shift_num;
    Huint i = 0;
    volatile float sum = 0.0F, temp = 2.0F;

    volatile Hlong m = 0x1, j = 0x2, k = 0x3,l = 0x7;

    // Finite number of shifting operations
   m = m >> shift_num;
    j = j >> shift_num;
    k = k << shift_num;
    l = l << shift_num;

    // Variable number of floating point operations
    for (i = 0; i < max; i++) {
        // Floating point multiplication
        temp *= temp;
        // Floating point addition
        sum += temp;
    }

    return (void *) SUCCESS;
}
```

Figure 4.2: Instruction example: the floating point instructions may be dominant depending the number of times the for-loop executes.

Instruction Distribution Example		
Scheduling optimization	**Processor**	**Execution Time (μs)**
Scheduling based on static number of compiled instructions	3	663
Scheduling knowing the distribution of compiled instructions	0	85

Table 4.4: Results of executing a thread that includes an uneven distribution of shifting operations (18) and floating point operations (2400).

scheduler maps the thread to processor #0, a faster execution is observed for the thread as given in Table 4.4. This is true as long as the floating point operations dominate the thread's execution time.

4.3.2 Resource-Awareness

Another runtime optimization that can be integrated in order to reduce PR overhead, and thus, reduce the contention in accessing the PR controller is knowing ahead of time, what polymorphic functions are called within a thread and whether it prefers a processor with a PR region (discussed previously in Chapter 3. By design, slave processors report to the hthreads runtime system what accelerators (if any) are attached to them and whether it is partially reconfigurable (i.e. if it has a dynamic accelerator). If it is partially reconfigurable, slaves report the currently attached accelerator once a thread finishes execution. Scheduling with this information compliments known information on which co-processors a thread utilizes. This is because the ultimate goal is to treat both internal and external hardware accelerators similarly. The scheduler addresses this by considering a thread's use of co-processors first as that remains static, and then determines whether the processor is connected to the appropriate external hardware (i.e. static or dynamic accelerator).

The first test critiques on whether or not knowing ahead of time 1) which external hardware accelerator may be used as a result of a polymorphic function call and, 2) knowing if partial reconfiguration is possibly needed in the case of multiple polymorphic function calls. To isolate this test from other factors, the slave processors always attempt to execute in hardware. Hence,

```
int main() {
  init_host_tables();

  // Start timer
  start = hthread_time_get();

  // Create detached threads
  thread_create(&threads[0], worker_sort_vector_thread);
  thread_create(&threads[1], worker_crc_thread);
  ....
  ....
  thread_create(&threads[127], worker_sort_thread);

  // Wait until slave processors have executed all queued threads
  while (get_num_free_slaves() != NUM_OF_AVAILABLE_HETERO_CPUS);

  // Stop timer
  stop = hthread_time_get();

  return 0;
}

// Example of thread function
void * worker_sort_thread( void * arg) {
  Data * myarg = (Data *) arg;

  // SORT
  if (poly_bubblesort(myarg->sort_data, myarg->sort_size))
    return (void *) FAILURE;

  return (void *) SUCCESS;
}
```

Figure 4.3: Pseudocode for the synthetic benchmark.

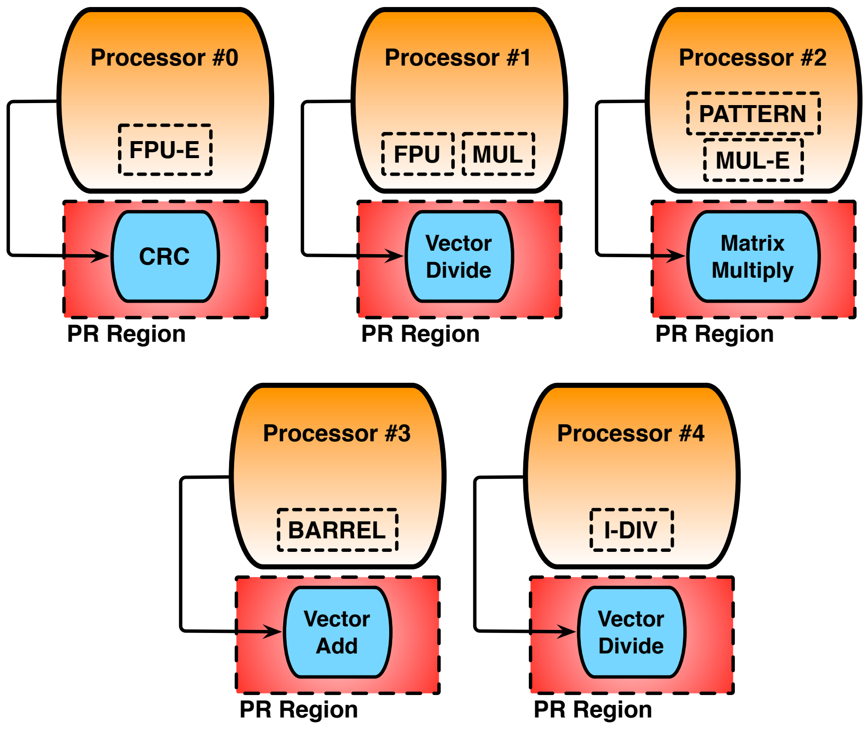

Figure 4.4: Platform A

they do not retrieve profiling data for the polymorphic function. If partial reconfiguration functionality exists at that slave processor, it will perform partial reconfiguration in order to execute the function in hardware. If the slave does not have the external resources to execute a polymorphic function in hardware, it will default to software execution. This test, shown in Figure 4.3, creates 128 detached threads in the system. These threads perform various polymorphic functions. In the case of worker_crc_thread, it calls the CRC polymorphic function only but for worker_sort_vector_thread, it calls both the bubblesort and vector addition polymorphic functions. Once this arbitrary amount of 128 threads finishes, the host processor records and displays the execution time, and optionally checks the results and displays the thread to processor mapping.

Furthermore, there was a need to vary the system configuration during this test to observe any changes that arise with less hardware resources/diminishing levels of system complexity. Therefore, this test was executed on three platforms derived from Figure 4.1. The first system, contains all 5 slave processors with dynamic accelerators attached. The second system contained

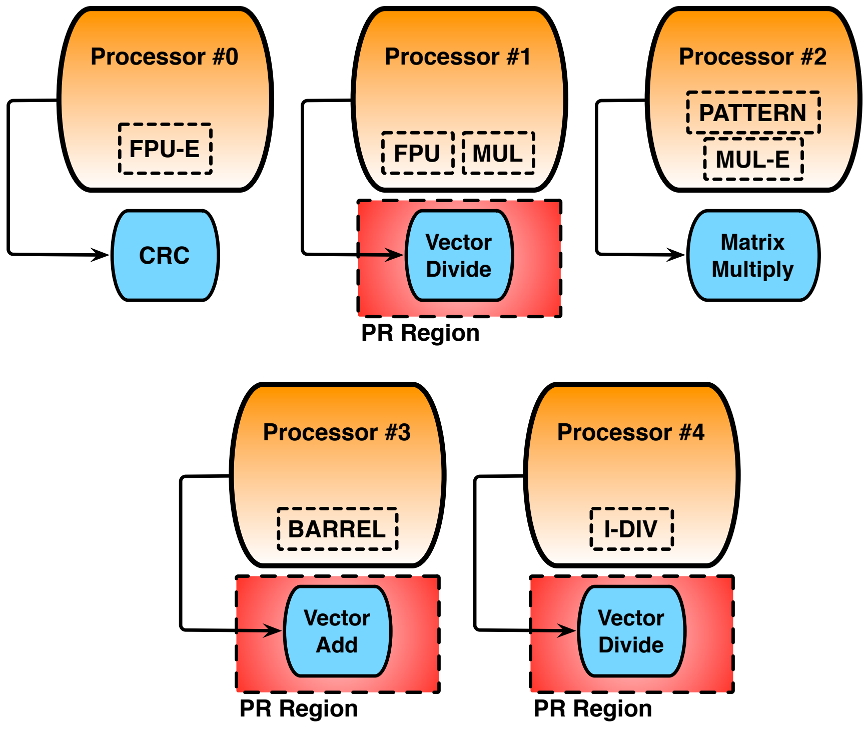

Figure 4.5: Platform B

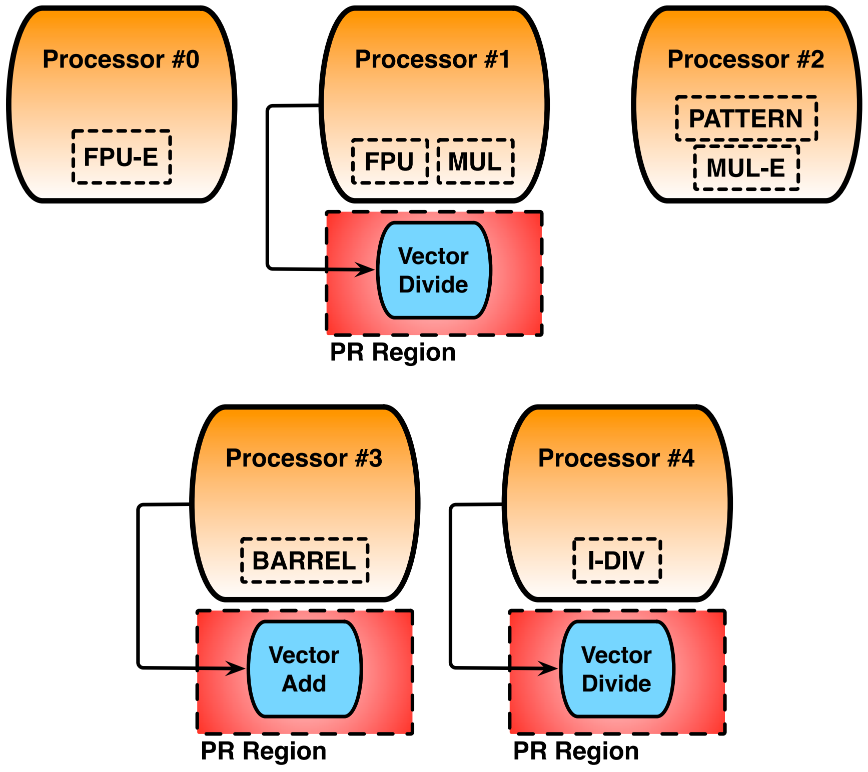

Figure 4.6: Platform C

Resource-Aware Scheduling			
	Platform A	**Platform B**	**Platform C**
Base scheduling	2,626.31	6,758.16	6,767.92
Base scheduling with random selection	2,861.24	6,687.99	6,688.45
Resource Aware-Scheduling	2,611.59	2,606.92	2,629.43
	Execution Time (ms)		

Table 4.5: Results of executing 128 detached threads that call various polymorphic functions, randomly. Data sizes for each call was also randomly chosen.

three processors with dynamic accelerators, and two with static accelerators. Finally, the third system contained three processors with dynamic accelerators, and two with neither static or dynamic accelerators. A simplified figure of these three systems are provided in Figures 4.4, 4.5, and 4.6.

The results in Table 4.5 confirm that knowing ahead of time what resources a thread *may* use, will deliver better overall system performance. This is seen when one considers moving the application from Platform A to Platform B and Platform C. During this shift, some of the slaves lose the ability to perform PR (Platform B), and some no external hardware accelerators (Platform C). This can force slaves to execute a polymorphic function in software or undergo the cost of PR even if there is an available processor with the appropriate resource. The arrangement of these disadvantaged processors were purposely configured between the three test systems to show this worst case. In the base scheduling scheme, the runtime examines each available processor beginning with Processor #0 to Processor #5, and schedules a thread once an available processor is found. Processor #0 in both Platform B and C do not have PR functionality or an accelerator at all. Processor #0 is the first processor under consideration for the base scheduler and if available, a thread will be mapped. As a result, the performance worsens on the base scheduler when moving from Platform A to Platform B or Platform C. To provide a more fair assessment to the resource-aware scheduling, results are included for the base scheduling policy with random selection. That is, instead of the base scheduler identifying which processor is available beginning with #0, it chooses a processor number at random to check its availability. While this scheduling policy does marginally improve over the base scheduling policy in both platforms B and C,

Resource-Aware Scheduling - PR and Software Ratio Details			
	Platform A	**Platform B**	**Platform C**
Base scheduling	75% / 0%	73% / 23%	78% / 28%
Base scheduling with random selection	82% / 0%	83% / 11%	84% / 12%
Resource Aware-Scheduling	58% / 0%	49% / 0%	77% / 0%

Table 4.6: PR percentage when executing in Hardware/Percentage of software executions out of the 154 polymorphic functions called across the 128 threads.

performance degrades in platform A. The base scheduling policy, is likely to reuse a processor with the appropriate hardware. This can be seen when one considers that the base scheduler performed PR 75% of the time during its 154 calls to polymorphic functions compared to the base scheduler with random selection invoking partial reconfiguration 82% of the time.

The alternative resource-aware scheduler on the other hand, provides very consistent results when moving between different configurations. To have a better understanding of why this scheduler avoids any performance loss through the loss of resources, a comparison can be made with the number of partial reconfigurations needed when deciding to execute in hardware and the percentage of software executions over the total amount of polymorphic functions invoked (154). As shown in Table 4.6, both of the base schedulers in Platforms B and C begin to map threads onto processors that are forced to execute in software due to a lack of resources whereas the resource-aware scheduler consistently maps threads to slaves with appropriate resources. Randomly selecting a processor candidate when scheduling a thread reduces the amount of software execution needed over the base scheduler. However, as explained previously, it became more likely to need partial reconfiguration. As a result, the high frequency of partial reconfiguration and resulting overhead can be observed across all three platforms for the base scheduler with random selection.

The PR percentage provides this thesis a way to verify whether the resource-aware scheduling minimizes the amount of PR within the system. Platform A presents an opportunity for this as all slave processors are equipped with a PR region, and under this testing procedure, there is a bias of performing PR if the hardware is not currently loaded. Additionally, all scheduling policies resulted in 0% software execution. Therefore, it can be concluded from Table 4.6 that the resource-aware scheduler successfully decreased partial reconfiguration required by 23% over the base scheduler. Platform B also shows a reduction in partial reconfiguration and is greater than Platform A. The reason for this is based on the priority of certain hardware features to what a thread needs. As pointed out earlier in Section 3, for threads that invoke a single polymorphic function, the resource-aware scheduler gives a greater priority to processors that have a static accelerator, reserving processors with a dynamic accelerator/PR region to threads that invoke multiple polymorphic functions. Since Platform B contains a mix of these two processor configurations, the resource-aware scheduler has better opportunity to map a thread to these *ideal* processors. As a result, the resource-aware scheduler was able to map threads to their ideal processors 32% of the time in Platform B whereas in Platform A that is filled with only PR regions on slaves, this decreased to only 4% of the time.

4.3.3 Forward Progress Threshold

A runtime metric that was added out of necessity was a threshold that indicates the number of free slaves needed before making any forward progress with scheduling. The synthetic benchmark in Figure 4.3 creates more threads than there are processors in the 3 platforms, A, B and C. In this scenario, scheduling a thread as soon as a processor becomes available would make deciding where best to schedule the thread moot (unless such a thread mapping is queued). Therefore, the Forward Progress Threshold (FPT) presents a way for the scheduler to have alternate options when scheduling a thread.

To provide a fair comparison, the FPT was applied to both the base scheduler(s) and the resource-aware scheduler. Before scheduling a thread, the scheduler checks if the available amount of processors is less than some percentage. For the previous tests, this was set to be 50% of the slaves, or 3 out of 5 processors. Unlike the base scheduler with random selection and the resource-aware scheduler, this limits scheduling threads only to processors #0 through #2 for the

Scheduler Policy: Forward Progress Threshold = 20%			
	Platform A	**Platform B**	**Platform C**
Base scheduling	1,675.44	5,467.24	5,467.24
Base scheduling with random selection	1,875.89	5,525.24	5,525.26
Resource Aware-Scheduling	1,675.46	6,618.14	6,653.15
	Execution Time (ms)		

Table 4.7: Results of scheduling 128 detached threads, whereby threads are released only when 20% or more of the processors are available.

base scheduler as it always checks for the first available slave beginning with processor #0. As Table 4.7 highlights, when decreasing this constraint by reducing the FPT to 20% or 1 or more available slave processors, performance improves for all scheduling policies in Platform A. This is to be expected as all processors in Platform A contain a PR region, and all have a bias according to the test setup to execute polymorphic functions in hardware. Since these threads are only composed of polymorphic function invocations, the performance gained on Platform A simplifies to which thread has the appropriate external resource to minimize partial reconfigurations. In the case of Platform B and C however, performance is much worse for resource-aware scheduling over the previous 50% FPT.

To understand why this behavior is observed, metrics such as PR ratios, software execution ratios, and the number of times a thread was mapped to an ideal location according to the flow chart discussed in Chapter 3 are again analyzed. For Platform A, all threads executed in hardware as all processors contained a dynamic accelerator. All three scheduling policies performed similarly with respect to their PR ratios as Table 4.8 reveals. The slight variation in performance between the base schedulers can be attributed to the additional overhead of randomly selecting a processor. This became evident once the percentages of mapping a thread to an ideal processor, to a processor that is the next best candidate, and so on were compared.

For Platform B and C, the base schedulers perform marginally better than their previous results found at an FPT of 50%. The additional savings of time was recovered from a lower minimum slave processor count the scheduler needed to wait on. However, the resource-aware

Scheduler Policy - PR and Software Ratio Details: Forward Progress Threshold = 20%			
	Platform A	**Platform B**	**Platform C**
Base scheduling	79% / 0%	78% / 10%	78% / 10%
Base scheduling with random selection	79% / 0%	77% / 16%	77% / 16%
Resource Aware-Scheduling	78% / 0%	72% / 9%	72% / 10%

Table 4.8: PR percentage when executing in Hardware/Percentage of software executions out of the 154 polymorphic functions called across the 128 threads.

scheduler performs worse than the two base schedulers and much worse from its previous results in Table 4.5. Although, Table 4.8 shows that the PR ratios are lower than the two base scheduling policies indicating that there is more reuse of hardware accelerators and less partial reconfigurations, the difference is insignificant. Furthermore, the metric describing the amount of times a thread was mapped to its ideal processor, the next best candidate, etc. were very similar across the three schedulers for Platforms B and C. The reasoning for the performance degradation of the resource-aware scheduler is tied to the FPT percentage. When the resource-aware scheduler is only given a single choice in the case where the FPT = 20% compared to several (3) choices with an FPT = 50%, scheduling with resource consideration defaults to similar performance of that of the base schedulers. Flexibility is needed for exploiting the performance of heterogeneous architectures. Without this, threads are mapped to processors with no consideration of resources needed to accelerate the thread. As a result, the resource-aware scheduler performs similarly to that of the base schedulers in platforms B and C, with one exception. There is a 1,000 *ms* difference. This difference is from software executions between the resource-aware scheduler and the base schedulers. Threads were mapped onto processors that are less ideal to execute the polymorphic function(s) than others in the case of the resource-aware scheduler.

To elaborate, the fact that the system contains heterogeneous processors results in differing software execution times amongst them. On platform B and C, for example, the two possible locations where software execution can or will occur are processor's #0 and #2. This is due to the lack of a dynamic accelerator or the correct static accelerator. To prove that this is the source of the 1,000 *ms* difference between the resource-aware and base schedulers, each processor's software execution of a polymorphic function was timed and tallied up for all threads executing on them. Since thread execution across these processors overlap, viewing the processor whose software execution total is largest provides an adequate estimate as to the differences amongst the scheduling policies. For the base scheduler with random selection, the total time recorded for all software executions of polymorphic functions occurred on Processor #2 was 5,382.46 *ms* out of the 5,525.24 ms total execution time. In contrast, the resource-aware scheduler resulted in Processor #0's total software execution time of 6,609.72 *ms* out of the 6,618.14 *ms* total execution time. A difference close to 1,000 *ms* confirms why the resource-aware scheduler performance is an additional 1,000+ *ms* worse than the base schedulers.

4.4 Putting it all together

In this section, we look at combining the ideas of the previous sections and applying them to real-world tasks. From this, this thesis would like to draw several key points: how well does the system scale when introducing additional heterogeneity/complexity.

4.4.1 Executing real-world tasks

The previous tests included micro-benchmarks, written to target each of the co-processors in the MicroBlaze. As such, the majority of their runtime utilized the co-processor instructions and less on memory I/O or other unaccelerated operations. Similarly for examining resource-awareness, the tasks that were scheduled had execution times dominated by the use of hardware accelerators attached to slave processors. These previous tests were conducted to show the potential of a heterogeneous system and how important it is to have a self-aware runtime system to manage it. The purpose of this section is to evaluate the self-aware runtime system further using a sample of non-synthetic tasks such as those used across multiple domains in research and industry. The tasks that are used, the co-processors within the MicroBlaze they leverage, and their descriptions are all described in Table 4.9. Additionally, the fastest and slowest times are also shown for each task.

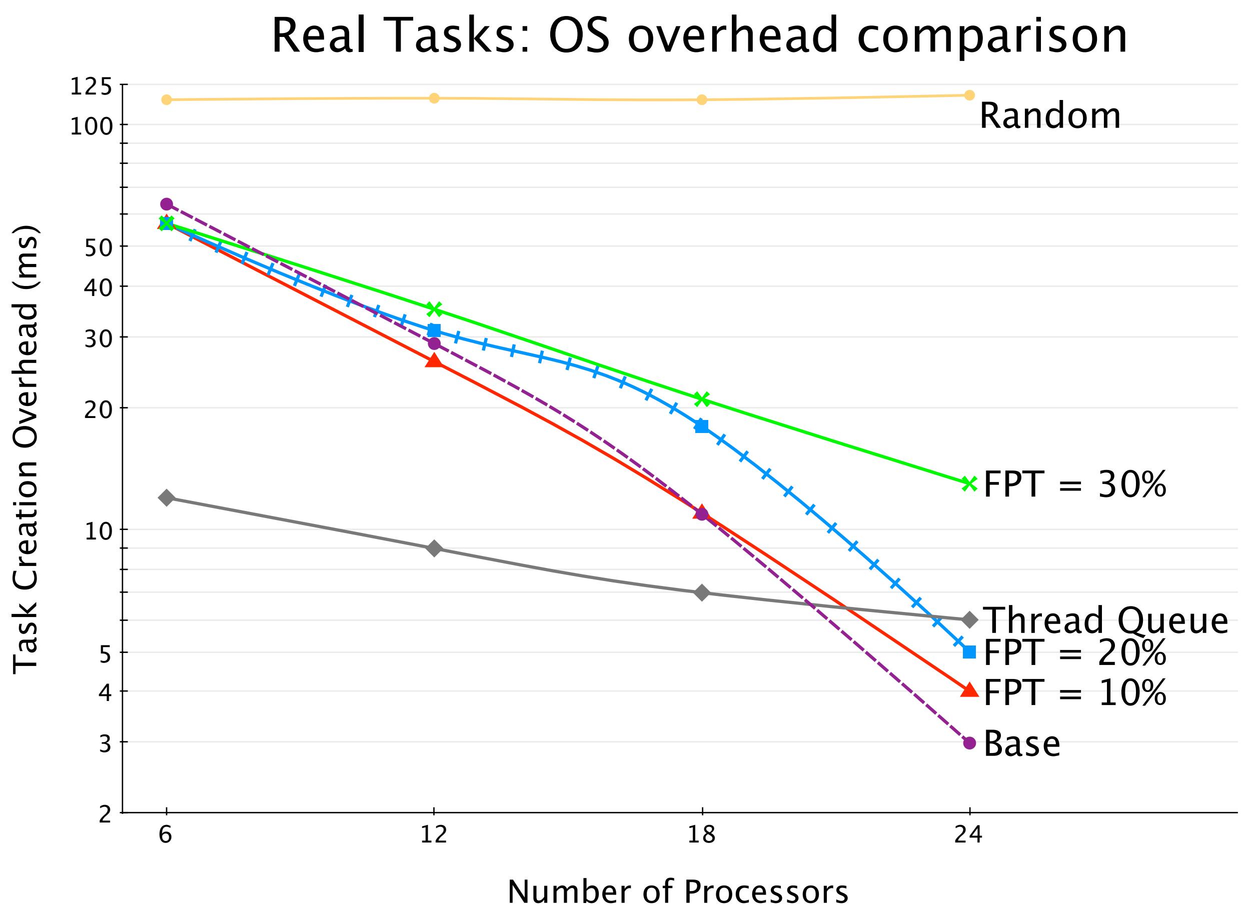

Figure 4.7: Task Creation overhead for all variants of schedulers.

Task Description Table				
Name	**Co-processors**	**Description**	**Fastest Time(ms)**	**Slowest Time(ms)**
Distance	Floating Point	Given two floating point arrays,it computes the distances between pairs of points in both arrays.	4.5	21.5
Find Max	Barrel Shifter	Given two integer arrays, it determines the max value for each pair of values in both arrays.	6	13
Histogram	Integer Division	Given an array of integers, it computes a frequency count for a configurable amount of ranges.	13	27
Matrix Multiply	32-bit Multiplier	Performs matrix multiply on two square matrices.	10	16
Mandel Brot	Floating Point	Given an array of floating point values, it determines whether the complex number for that value belongs to the Mandelbrot set.	4.6	5,541
Approx. Pi	Floating Point	Approximates Pi using the Nilakantha series.	0.4	18

Table 4.9: Table providing the name, MicroBlaze co-processor support, and descriptions of each real-world task used in this thesis. Fastest/Slowest Time is the time recorded for a MicroBlaze in the 24 processor system used in Section 4.4.1

The first test that was executed consisted of scheduling an arbitrary amount of threads onto a 6 to 24 heterogeneous processor system. The amount of threads scheduled onto the system totalled 25 threads (5 Approx. Pi, 3 MandelBrot, 5 Histogram, 3 Distance, 4 Matrix Multiply, 5 Find Max threads). Although there were multiple instances of the same tasks, all of the tasks were arranged in a random order when delivering them to the task. This was to avoid a sudden burst of homogeneous tasks all requesting similar resources. The FPT values for this tests ranged from 10% to 30% for comparison to the base scheduler with random selection.

Another variant of the self-aware runtime system was also added to the system that integrates thread queues within the scheduler. Instead of stalling until a certain percentage of slave processors became available, queuing threads would allow the runtime system to have a faster response time when scheduling threads and possibly lead to faster execution times over the runtime variants utilizing the FPT values. This can be seen in the total task creation overhead in

Figure 4.7. The figure shows, as the system becomes smaller, the overhead is larger when using an FPT value compared to the thread queuing. However, as the system becomes larger, the scheduler utilizing FPT has greater opportunity for mapping threads as the number of available slaves needed becomes less. Thread queuing would also provide similar overhead times, but the functionality of thread queuing currently executes in a software thread that involves CPU context switching of the main thread on the host processor. Hthreads does provide fast CPU context switching due to the hardware implementation of a thread manager and scheduler, but the CPU context resides in main memory across a shared bus. This overhead slightly decreases however, from 6 to 24 processors due to the higher availability of alternative preferred processors to map to, and thus, less context switching on the host processor. The base scheduler with random selection remains fairly constant through the processor range in this Figure. This is largely due to the high overhead of randomization currently used.

Figures 4.8, 4.9, 4.10, 4.11 and 4.12, all show the execution times for the self-aware runtime with thread queuing compared to the other scheduling policies used in this thesis. The first note of importance is the comparison to the base scheduler in Figure 4.8. A speedup is observed across all 4 systems with thread queuing enabled. For the system with 12 heterogeneous processors, the speedup is significantly larger than the other systems. This was also seen in Figure 4.9, but for a different system (24 processor system). The reason for these results is due to the thread mapping between the Base scheduler, and the base scheduler with randomization. As can be seen in Table 4.9, some of the tasks involved in this test had execution times ranging in the 1,000's of milliseconds. If the scheduler were to map a thread onto one of these processors, it would dominate the execution time. This is the case for both base schedulers in comparison to the self-aware scheduler with thread queuing.

The comparisons between thread queuing and utilizing the FPT values track very similarly between each other. As the FPT value decreases, the scheduler makes faster decisions at the expense of not having as many options when mapping a thread to a processor. As a result, thread

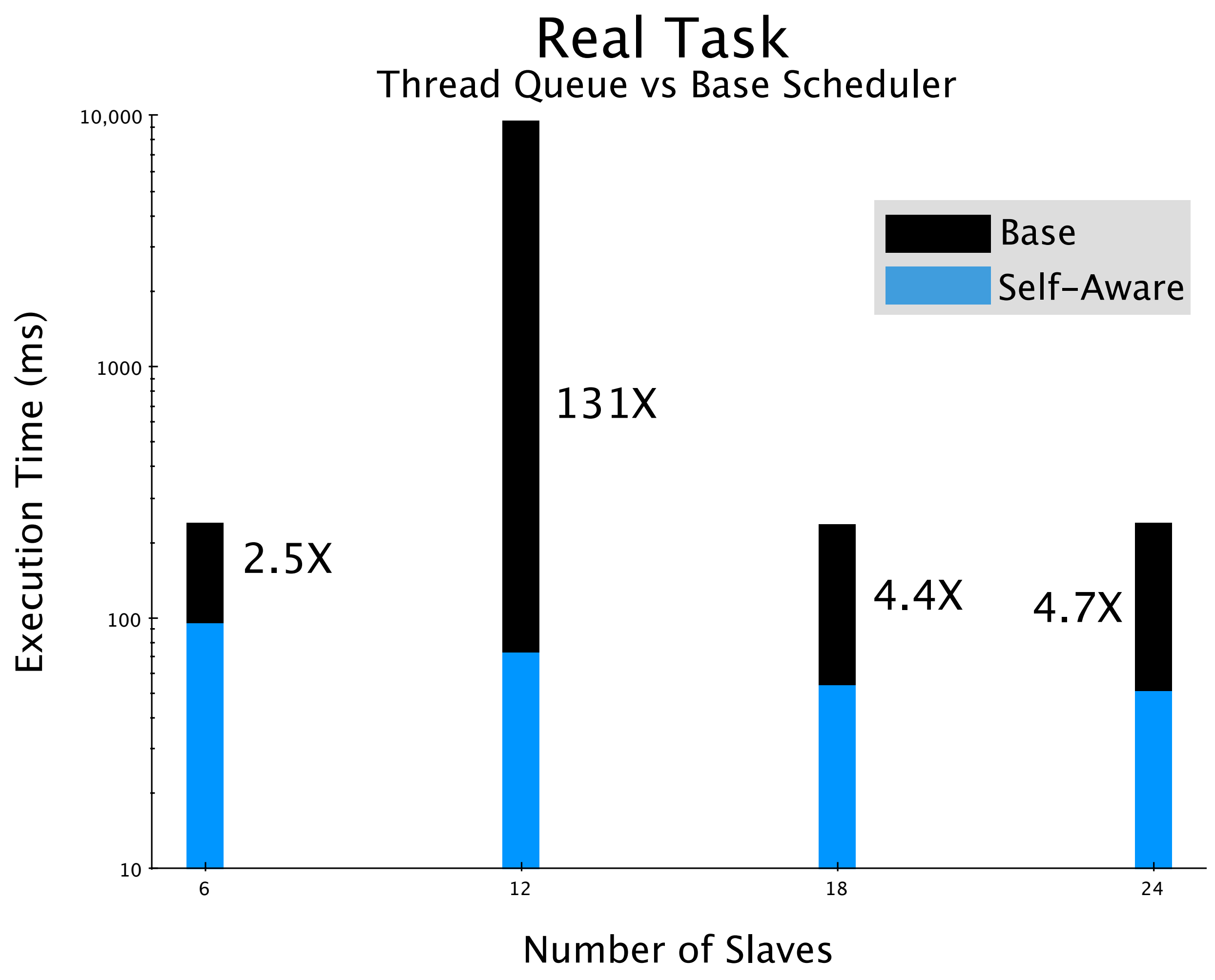

Figure 4.8: Results of the Self-aware Scheduler with Thread Queuing vs Base Scheduler. Speedup is also shown.

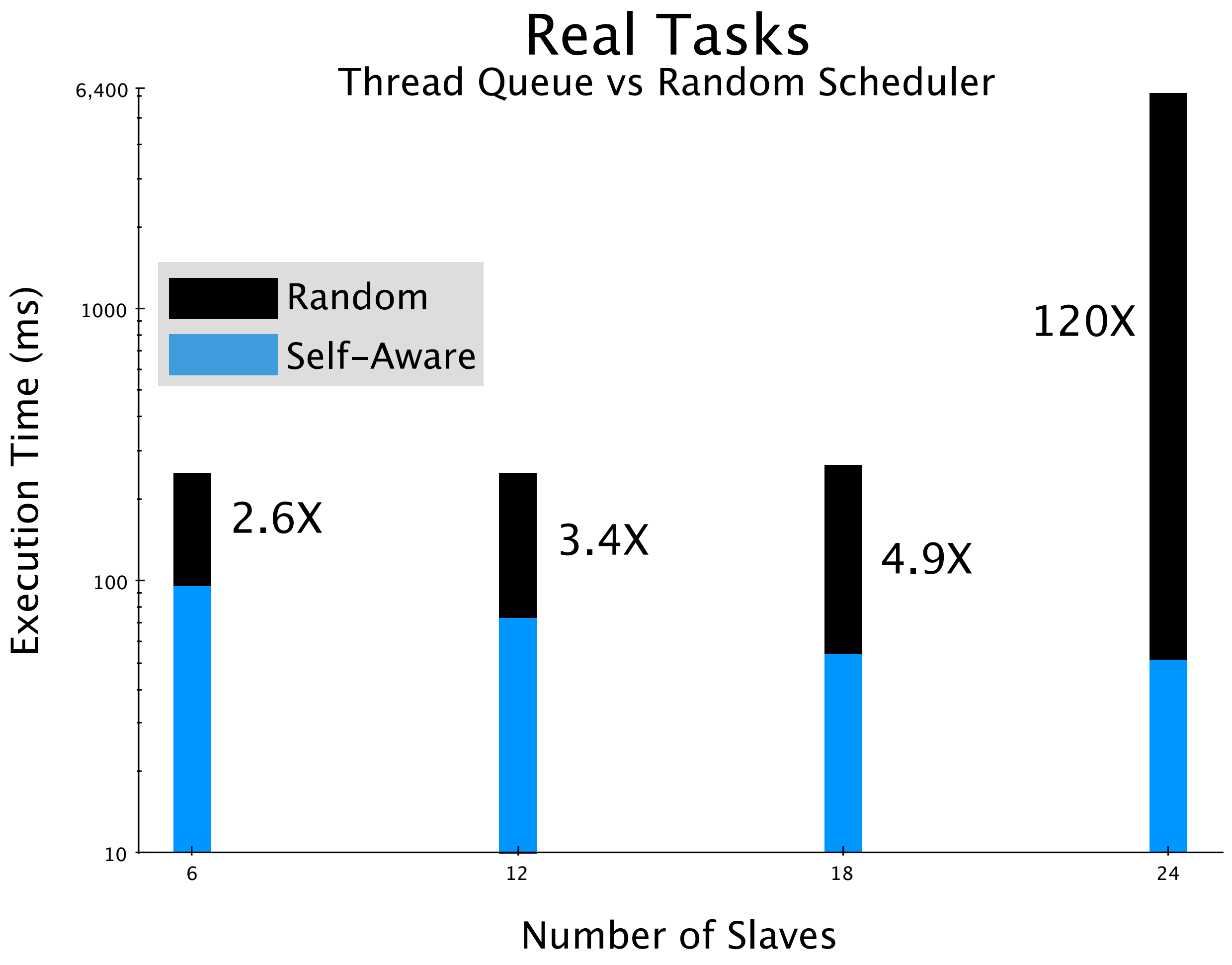

Figure 4.9: Results of the Self-aware Scheduler with Thread Queuing vs Base Scheduler with Random Selection. Speedup is also shown.

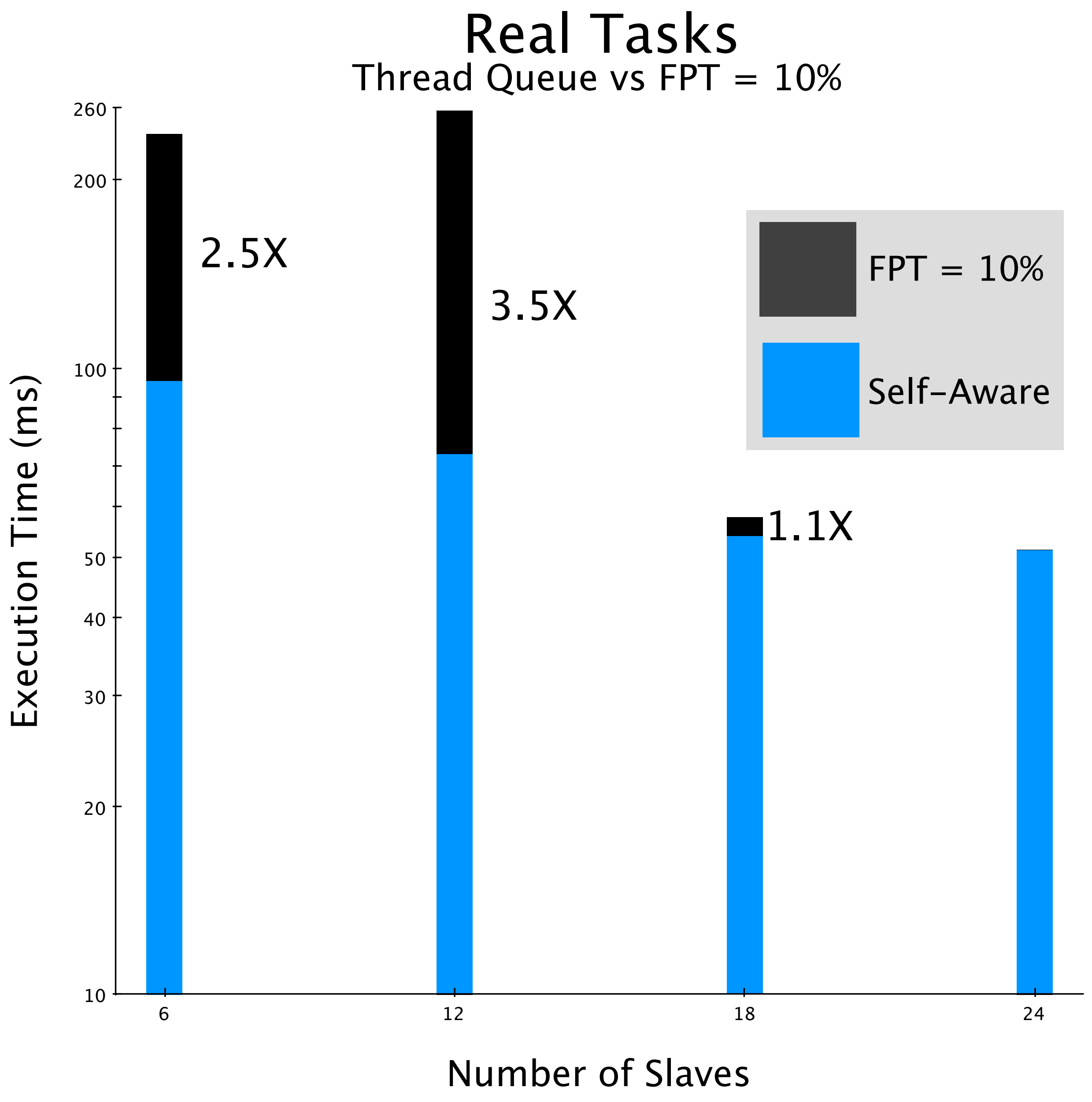

Figure 4.10: Results of the Self-aware Scheduler with Thread Queuing vs with FPT = 10%. Speedup is also shown.

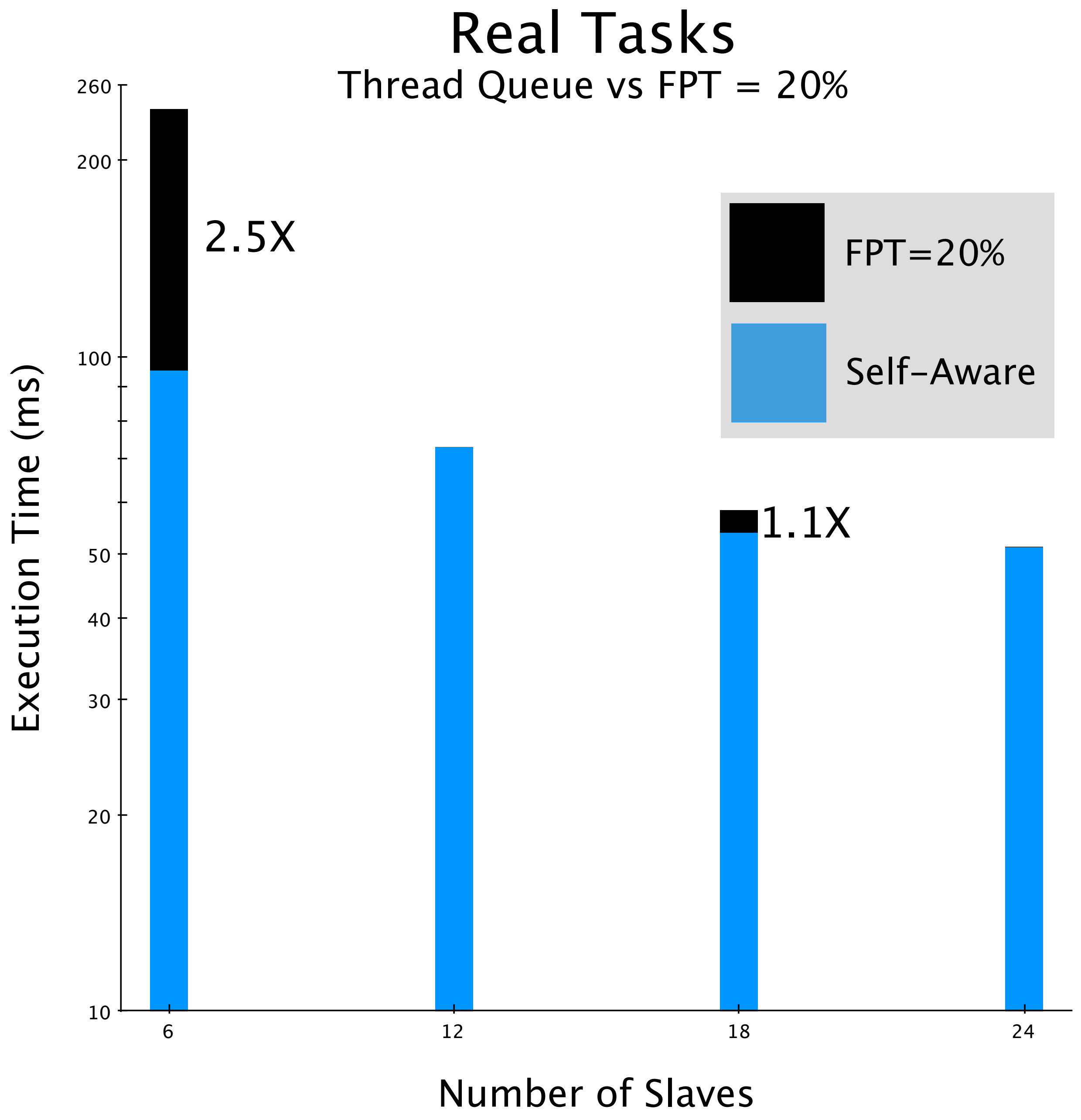

Figure 4.11: Results of the Self-aware Scheduler with Thread Queuing vs with FPT = 20%. Speedup is also shown.

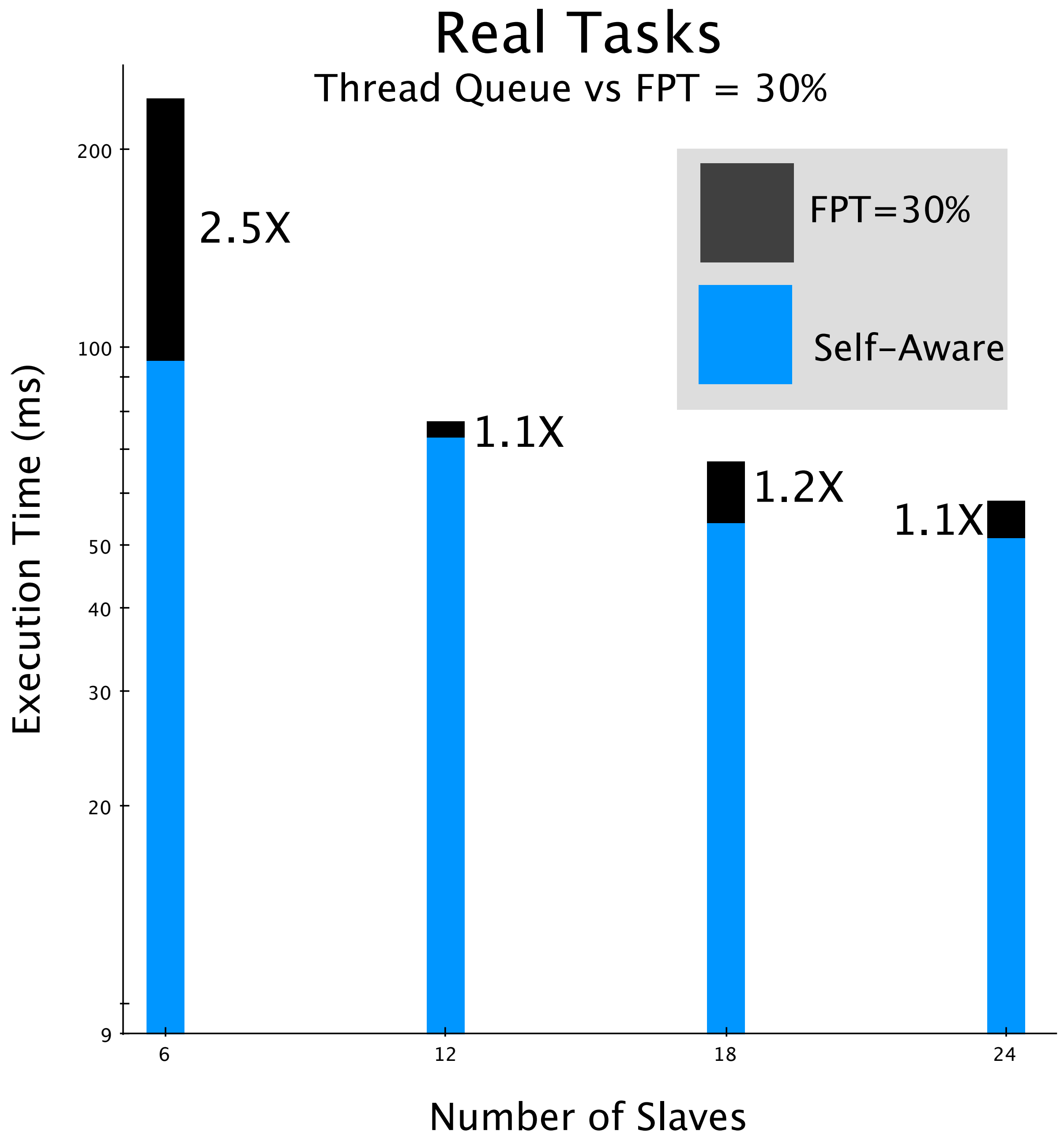

Figure 4.12: Results of the Self-aware Scheduler with Thread Queuing vs with FPT = 30%. Speedup is also shown.

queuing enjoys a healthy speedup on the smaller systems in Figure 4.10 as it can queue threads to preferred processors all the while making forward progress of scheduling other threads. As the system becomes larger, there is an increase in tasks executing in parallel. This leads to an increase in the amount of processors that are available, and thus faster scheduling decisions for the schedulers utilizing the FPT values. As a result, thread queuing has a marginal increase in speedup, even for larger systems.

What is not explicitly shown in these figures, but rather implied is the high unpredictability in the execution times for systems that have fast scheduling response times. It was mentioned that for the base schedulers, there was at least one instance where a mapped thread resulted in its execution dominating the entire application's execution time (120X and 131X speedup in Figures 4.9 and 4.8 respectively). The same is true for the self-aware system with a low FPT value (10%). Figure 4.10 shows for the 12 processor system, there was at least a non-optimal thread mapping that resulted in a 3.5X speedup for thread queuing. When comparing the Figures 4.11 and 4.12, this was not observed at the 12 processor system. This unpredictable behavior was also observed for the self-aware runtime employing thread queues. This variant represents both fast scheduling decisions as with the base schedulers, but also waits/queues threads to their appropriate processors. Since this is a heterogeneous system, threads that have more than one preferred processor do not necessarily execute similarly. For example, the approximating PI task can execute in 0.4 *ms* on the most preferred processor, 9.6 *ms* as the next best option, and finally, in 15 *ms* on the slowest option. For thread queuing, the scheduler will be presented with the first two options, but not the last as it represents an unaccelerated path of execution. However, the difference between the first two is significant to cause a slowdown if a decision is made on the latter processor. As a result, the execution results can be impacted by faster scheduler decisions.

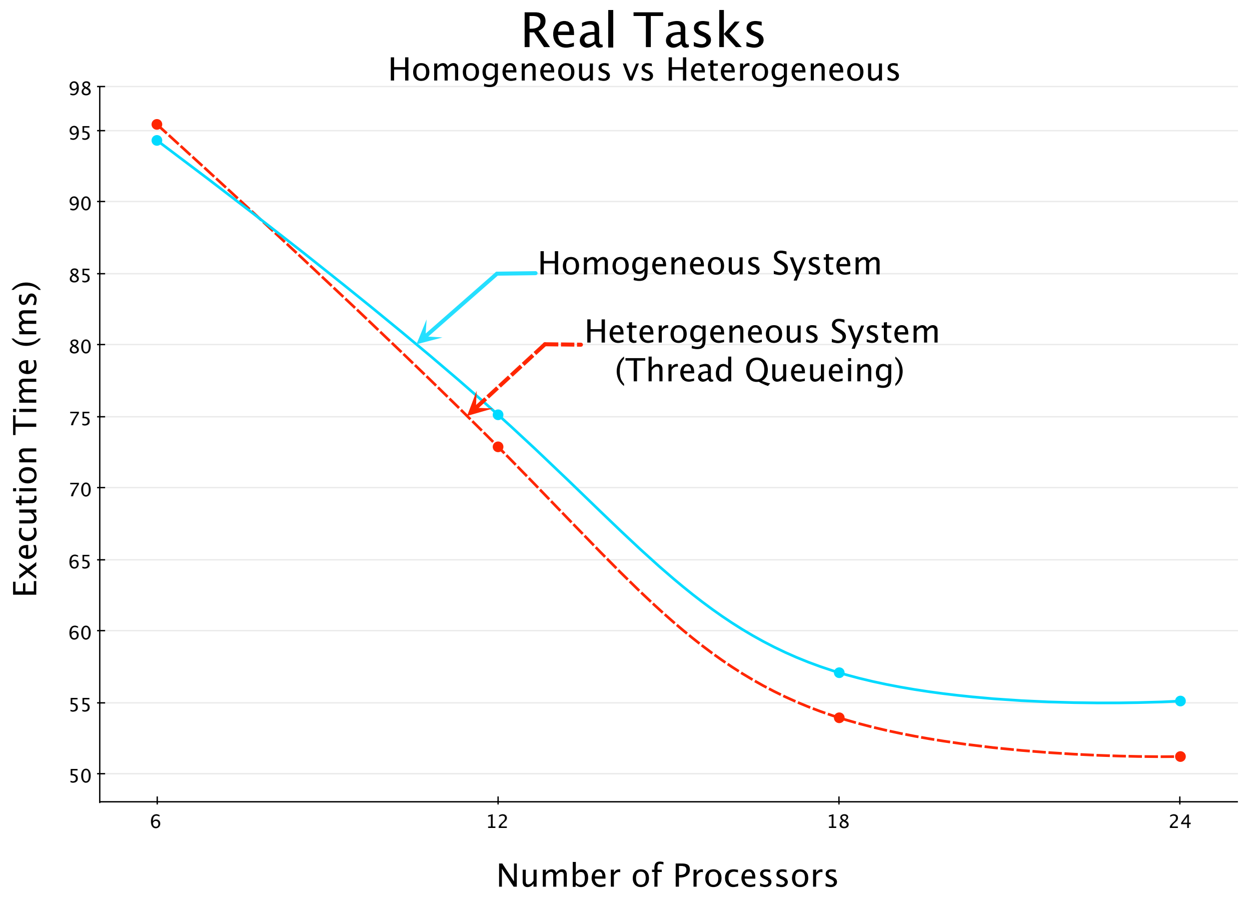

Figure 4.13: Execution results of the real-world test application on a homogeneous system vs a heterogeneous system with the self-aware runtime utilizing thread queuing.

4.4.2 Task mapping/resource efficiency performance

What has been an unanswered question up until now is how accurate are threads being mapped throughout the system relative to the best known solution. For a heterogeneous system, the solution space for mapping a thread is large and can vary between system configurations. To answer this, a comparison to a homogeneous system that includes all processors whereby each are configured with all optional co-processors. On the other hand, the heterogeneous system taken from Section 4.4.1, will have varying configurations for each processor. Both of these systems have a maximum number of 24 slave processors and the tests was executed by varying the enabled processors from 6-24. Additionally, the base scheduler for the homogeneous system was chosen as the identifying processor specialization is unnecessary: all processors perform similarly. The self-aware runtime with thread queuing was utilized on the heterogeneous system.

Figure 4.13, shows the final result of executing the same test in section 4.4.1: an application scheduling 25 real-world tasks across the system. While this test was meant to show how close in performance the heterogeneous system gets towards the homogeneous system, it surpasses the performance from 12-24 processors. This verifies that the runtime system maps tasks appropriately on the heterogeneous system despite the platform not having as many resources as the homogeneous system. A reason why it is able to perform marginally better is due to the forward progress the self-aware scheduler makes. While the base scheduler will *spin*, waiting for a processor to become available if all are busy, the self-aware scheduler with thread queuing will continue to cycle through the submitted threads, considering only preferred processors instead of the entire processor list for each. This is the reason why this difference becomes larger for larger systems as the self-aware scheduler may only consider a subset of processors when mapping a thread.

Chapter 5

Conclusion

This thesis has explored new challenges for emerging FPGA-based heterogeneous multiprocessor systems. FPGAs have the potential of striking a balance between generality and specialization for driving computation in the next generation of software (and those applicable today). However, management of such a platform has traditionally been hindered by design experts. The use of CPUs within FPGA designs has made system bring up possible for non-experts, and allows them to extend computation through special hardware, either through pre-compiled accelerator libraries, high-level synthesis, or other future enablers. However, the problem of knowing when and how to map to these heterogeneous resources is not always clear.

This thesis has shown that there is a need to have a runtime system that is cognizant of the available hardware, the varying affinities between threads and heterogeneous processing elements, and knows how and when to exploit this information to provide better scheduling decisions. This is particularly true if the adoption of much larger heterogeneous systems, especially those providing online reconfigurability, is to be realized as static scheduling decisions determined offline will not suffice. Furthermore, greedily scheduling tasks to as many processors available can also lead to poor thread mapping and thus longer execution times and higher energy consumption.

To summarize, this thesis has investigated the following questions:

Can platform specific details be moved from the source code and be encapsulated under platform neutral policies of a portable programming model to reinstate portability? Yes, these details that describe when and how to use resources and procedures of when such resources do not exist were moved to a combination of a runtime kernel and a hardware programming abstraction known as the polymorphic function library. Additionally, distinguishing processor heterogeneity during thread creation and thread mapping are transparently managed by the compilation flow and the self-aware runtime system.

Can the complexity of the software design process be reduced by enabling automatic profiling during runtime? Yes, the complexity of the software design process has been avoided through automatic identification of processor specialization of user created threads. As discussed in the future works (Section 5.1), this requires further investigation but it has been shown in this thesis that requiring the user to profile his/her source code can be avoided. The polymorphic function library, on the other hand, does require a manual process of profiling. However, it is our belief that through the distribution of a library, this effort can be reduced and further enhanced through the community over time.

Can dynamic mapping occur at reasonable overhead costs with high accuracy of fitting task placement? How best to determine task specialization for processors? Can some of this information be determined offline? Yes, the overhead costs can be kept at a minimum and achieve accurate task placement. Introducing thread queuing enabled the self-aware runtime system to consider preferable processors for threads but continue to make forward progress for other submitted threads. Additionally, some of the information needed to perform accurate task placement can be retrieved offline. However, as this thesis highlighted with instruction distribution, static information alone is insufficient for more complex tasks.

Can resource efficiency be increased despite increasing levels of system complexity, by introducing additional self-awareness within the operating system? Resource efficiency was increased through dynamically mapping threads to new hardware that offered an accelerated path of execution. These performance gains were seen when the processor count increased. However, such utilization of resources can further improve with a programming model that naturally supports heterogeneous systems. Introducing a task-based language would allow further abstraction of user-written tasks, but would also enable the runtime to infer how to further divide these tasks into subtasks for increase resource efficiency.

5.1 Future Work

This section discusses a few ideas of future work that have been mentioned throughout this thesis and provides more details on the impact it may have in furthering this body of research.

5.1.1 Opcode Flagging and Instruction Distribution

Flagging instructions as part of the compilation process provides a lightweight alternative for estimating performance on similar ISA processors, but falls short when including a mix of processors that have dissimilar ISAs. Operations that are equivalent amongst them may be implemented, executed, and completed differently. Furthermore, it was briefly shown in Section 4.3.1 that when considering a similar base ISA, deciding solely based on produced instructions by the compiler will still be insufficient at times when those instructions do not equally contribute to a task's execution time. Additionally, a task may exhibit different phases that corresponds to different behaviors during its runtime. For example, the beginning of its execution may involved heavy local computation followed by many memory I/O requests. Too many memory accesses could lead to dominating most of the task's execution time. In that instance, identifying which processor can provide a path of acceleration for the initial part of a task's execution becomes moot.

The MicroBlaze, and many modern day processors integrate hardware performance counters, and some even provide the capture of instruction traces for finer-grained profiling data. The MicroBlaze for instance, can be configured to count specific types of instructions, and can be stopped, resumed and reconfigured online. This can provide the runtime system presented in this thesis to *learn* a task's execution over time and modify the information that was generated offline for better task placement. A metric that can be used is the instructions per cycle (IPC). This metric may allow the comparison of dissimilar ISAs when deciding to map a thread onto a processor. Integrating support for these performance counters was initially started but later abandoned for unreliability and lacking vendor tool support.

5.1.2 Slave Processor Queues and Work Stealing

When presenting the results for this thesis, Chapter 4 introduced a variant of the runtime system that included thread queues. This was a software implementation that tried to eliminate the host processor from *spinning* until a certain amount of slave processors became available. As a result, execution times approached that of a system with all the necessary hardware at each processor. However, the software implementation brought high CPU context switching on the host which stalled it from making further forward progress on scheduling other tasks onto the system. Hardware queues would offer increased parallelism and scalability within the system as the host processor can push to the queue(s), and slaves can simultaneously pull, distributively. This can also enable future optimizations such as work/thread stealing at low runtime overhead costs. For example, the host processor currently manages two lists for each task: a preferred and non-preferred list of processors for this task. When scheduling a task, the runtime system checks availability of the preferred processors only. If none are available, the task is enqueued. Instead of this policy, non-preferred processors may be allowed to begin execution of said task. If the preferred processor were to become available soon after, the task could be stolen or migrated onto the preferred processor as long as the processor shares the same base ISA.

5.1.3 Adoption of another programming model

Programs written for hthreads uses the Pthreads library. As a result, writing the application for a multiprocessor system at a thread level still requires explicit memory management, and inter-thread communication, otherwise known as inter-process communication (IPC). As mentioned in Chapter 2, there is much research surrounding the area of easier methods of describing computation for (heterogeneous) multiprocessor systems. Some even focus on including optimization without the expense of losing portability. Adopting such languages and tools within this body of work represents a step in enabling future developments on this platform. Much of these languages and tools provide Pthreads backends, enabling the hthreads platform to support more than one. As a result, users can take advantage of the already existing research in these communities to quickly target existing or new applications on the hthread's platform and any other platforms the user's tools of choice targets.

References

[1] big.little technology: The future of mobile. Technical report, ARM.

[2] The OpenACC Application Programming Interface. http://www.openacc.org. Version 2.0, August 2013 Last accessed May 4, 2016.

[3] Advanced Micro Devices. *Bolt C++ Template Library*. Version 1.2 (July 2014) Last accessed May 4, 2016.

[4] Advanced Micro Devices. *Heterogeneous System Architecture: A Technical Review*. Rev. 1.0 (August 30th, 2012) Last accessed May 4, 2016.

[5] Jason Agron. *Hardware Microkernels - A Novel Method for Constructing Operating Systems for Heterogeneous Multi-Core Platforms*. Ph.D. Dissertation, University of Arkansas, August 2010.

[6] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[7] Erik Konrad Anderson. *Abstracting the Hardware / Software Boundary through a Standard System Support Layer and Architecture*. Ph.D. Dissertation, University of Kansas, May 2007.

[8] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, Ed Komp, and P. Ashenden. Programming models for hybrid fpga-cpu computational components: a missing link. *IEEE Micro*, 24(4):42–53, July 2004.

[9] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Pays-Bas, August 2009.

[10] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, INRIA, March 2010.

[11] David Bernstein. *The Future of Computing, essay in memory of Stamatis Vassiliadis*, chapter SOFTWARE ENABLEMENT FOR MULTICORE ARCHITECTURES, pages 12–18. Delft, The Netherlands, September 28, 2007.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[13] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, Jul 1999.

[14] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Last accessed May 4, 2016.

[15] N. Chitlur, G. Srinivasa, S. Hahn, P.K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, Li Zhao, N. Ijih, S. Subhaschandra, S. Grover, Xiaowei Jiang, and R. Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–8, Feb 2012.

[16] J. Cong, G. Reinman, A. Bui, and V. Sarkar. Customizable domain-specific computing. *Design Test of Computers, IEEE*, 28(2):6 –15, march-april 2011.

[17] J. Cong, V. Sarkar, G. Reinman, and A Bui. Customizable domain-specific computing. *Design Test of Computers, IEEE*, 28(2):6–15, March 2011.

[18] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[19] A. DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, Apr 2000.

[20] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[21] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: Runtime techniques for dynamic concurrency inference, resource constrained hierarchical scheduling, and online optimization in heterogeneous multiprocessor systems. Technical report, Georgia Institute of Technology, Computer Architecture and Systems Lab, 2008.

[22] S. B. R. Dolbeau and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. 2007.

[23] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. 41(2):78–85, February 2008.

[24] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[25] Aeroflex Gaisler. Leon3 Multiprocessing CPU Core. http://www.gaisler.com/index.php/products/processors/leon3. Last accessed May 4, 2016.

[26] D. Gohringer, M. Hubner, T. Perschke, and J. Becker. New dimensions for multiprocessor architectures: On demand heterogeneity, infrastructure and performance through reconfigurability - the rampsoc approach. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 495–498, Sept 2008.

[27] D. Gohringer, M. Hubner, E.N. Zeutebouo, and J. Becker. Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable

multiprocessor architectures. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[28] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, and Jürgen Becker. Operating system for runtime reconfigurable multiprocessor systems. *Int. J. Reconfig. Comput.*, 2011:3:1–3:16, January 2011.

[29] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 162–170, New York, NY, USA, 2004. ACM.

[30] V. Gupta and K. Schwan. Brawny vs. wimpy: Evaluation and analysis of modern workloads on heterogeneous processors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 74–83, May 2013.

[31] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[32] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[33] IBM. The Cell Broadband Engine. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/cellengine/. Last accessed May 4, 2016.

[34] Mercury Federal Systemsm Inc. OpenCPI Technical Summary. http://opencpi.org. Last accessed May 4, 2016.

[35] Intel Corporation. *Intel Thread Building Blocks Reference Manual*. Last accessed May 4, 2016.

[36] The Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*. Last accessed May 4, 2016.

[37] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Enno Luebbers and Marco Platzner. ReconOS: An RTOS Supporting Hard- and Software Threads. *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, August 2007.

[39] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[40] Sen Ma, Zeyad Aklah, and David Andrews. A run time interpretation approach for creating custom accelerators. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–4, Sept 2015.

[41] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *Design Test of Computers, IEEE*, 26(4):18–25, July 2009.

[42] Microsoft Corporation. *C++ AMP: Language and Programming Model*. Version 1.2 (December 2013) Last accessed May 4, 2016.

[43] Nvidia. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda. Last accessed May 4, 2016.

[44] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Version 4.0 (July 2013) Last accessed May 4, 2016.

[45] Kevin OBrien, Kathryn OBrien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. In Barbara Chapman, Weiming Zheng, GuangR. Gao, Mitsuhisa Sato, Eduard Ayguad, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 65–76. Springer Berlin Heidelberg, 2008.

[46] Sreepathi Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Plasma: Portable programming for simd heterogeneous accelerators. Workshop on Language, Compiler, and Architecture Support for GPGPU, held in conjunction with HPCA/PPoPP, 2010. Last accessed May 4, 2016.

[47] Swamy Dhoss Ponpandi. *Polymorphic computing abstraction for heterogeneous architectures*. PhD thesis, Iowa State University, 2015.

[48] 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*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[49] T.R.W. Scogland, B. Rountree, Wu chun Feng, and B.R. de Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144–155, May 2012.

[50] John Shalf. The New Landscape of Parallel Computer Architecture. Technical report, NERSC Division, Lawrence Berkeley National Laboratory.

[51] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, April 2009.

[52] V. Sima and K. Bertels. Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–6, May 2009.

[53] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[54] Chao Wang, Xi Li, Junneng Zhang, Peng Chen, Xiaojing Feng, and Xuehai Zhou. Fpm: A flexible programming model for mpsoc on fpga. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 477–484, May 2012.

[55] Chao Wang, Xi Li, Junneng Zhang, Peng Chen, and Xuehai Zhou. Caas: Core as a service realizing hardware sercices on reconfigurable mpsocs. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 495–498, Aug 2012.

[56] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–1, 2013.

[57] John R. Wernsing and Greg Stitt. Elastic computing: A portable optimization framework for hybrid computers. *Parallel Computing*, 38(8):438 – 464, 2012.

[58] John Robert Wernsing and Greg Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 115–124, New York, NY, USA, 2010. ACM.

[59] W. Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 681–685, July 2004.

[60] Xilinx. AXI Reference Guide. http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Last accessed May 4, 2016.

[61] Xilinx. DS180: 7 Series FPGAs Overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. Last accessed May 4, 2016.

[62] Xilinx. DS890 (v2.6): UltraScale Architecture and Product Overview. http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Last accessed May 4, 2016.

[63] Xilinx. MicroBlaze Processor Reference Guide. http://bit.ly/1xFtH8q. Last accessed May 4, 2016.

[64] Xilinx. Vivado Design Suite User Guide: Partial Reconfiguration. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf. Last accessed May 4, 2016.

[65] XueJun Yang, Tao Tang, GuiBin Wang, Jia Jia, and XinHai Xu. Mptostream: an openmp compiler for cpu-gpu heterogeneous parallel systems. *Science China Information Sciences*, 55(9):1961–1971, 2012.

[66] Zylin. ZPU. http://opensource.zylin.com/zpu.htm. Last accessed May 4, 2016.