5-2016

Achieving a better balance between productivity and performance on FPGAs through Heterogeneous Extensible Multiprocessor Systems

Abazar Sadeghian
University of Arkansas, Fayetteville

Achieving a better balance between productivity and performance on FPGAs through Heterogeneous Extensible Multiprocessor Systems

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

By

Abazar Sadeghian
Iran University of Science and Technology
Master of Science in Electrical and Electronics Engineering, 2007

May 2016
University of Arkansas

This thesis is approved for recommendation to the Graduate Council

______________________________-

Dr. David Andrews, Ph.D.
Thesis Director

______________________________-

Dr. Miaoqing Huang, Ph.D.
Committee Member

______________________________-

Dr. Gordon Beavers, Ph.D.
Committee Member

Abstract

Field Programmable Gate Arrays (FPGAs) were first introduced circa 1980, and they held the promise of delivering performance levels associated with customized circuits, but with productivity levels more closely associated with software development. Achieving both performance and productivity objectives has been a long standing challenge problem for the reconfigurable computing community and remains unsolved today. On one hand, Vendor supplied design flows have tended towards achieving the high levels of performance through gate level customization, but at the cost of very low productivity. On the other hand, FPGA densities are following Moore's law and and can now support complete multiprocessor system architectures. Thus FPGAs can be turned into an architecture with programmable processors which brings productivity but sacrifices the peak performance advantages of custom circuits. In this thesis we explore how the two use cases can be combined to achieve the best from both.

The flexibility of the FPGAs to host a heterogeneous multiprocessor system with different types of programmable processors and custom accelerators allows the software developers to design a platform that matches the unique performance needs of their application. However, currently no automated approaches are publicly available to create such heterogeneous architectures as well as the software support for these platforms. Creating base architectures, configuring multiple tool chains, and repetitive engineering design efforts can and should be automated. This thesis introduces Heterogeneous Extensible Multiprocessor System (HEMPS) template approach which allows an FPGA to be programmed with productivity levels close to those associated with parallel processing, and with performance levels close to those associated with customized circuits. The work in this thesis introduces an ArchGen script to automate the generation of HEMPS systems as well as a library of portable and self tuning polymorphic functions. These tools will abstract away the HW/SW co-design details and provide a transparent programming language to capture different levels of parallelisms, without sacrificing productivity or portability.

Acknowledgment

I would like to thank Dr. David Andrews for giving me the opportunity to work in his research laboratory. Additionally, I would like to thank my thesis committee, Dr. Huang and Dr. Beavers. Finally, I would like to thank my labmates in the CSDL lab for all the team work we had during the last 5 years.

Contents

List of Figures

List of Tables

Terms and Definitions

HEMPS Heterogeneous Extensible Multiprocessor Systems

ESL Electronic system Level design, tools and methods to abstract the HW details away.

SoC System-on-Chip. A system in which all components are co-located on a single chip.

MPSoC Multi-processor System-on-Chip. A system-on-chip having multiple processors.

AMBA Advanced Microcontroller Bus Architecture

AXI Advanced eXtensible Interface

IP Intellectual Property

VLSI Very-large-Scale Integration

VHDL VHSIC Hardware Description Language

Tcl Tool Command Language

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

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.

RISC Reduced Instruction Set Computer.

CLB Configurable Logic Block. A programmable block within the architecture of Field Programmable Gate Arrays (FPGAs).

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.

HW Abbreviation for Hardware.

SW Abbreviation for Software.

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.

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

OS Operating System.

RPC Remote-Procedure Call.

RTL Register Transfer Level. An abstraction level of the hardware design process.

LUT Lookup Table. A digital building block used to implement N-bit binary functions via lookup operations.

ASIC Application Specific Integrated Circuit.

CPU Central Processing Unit. A hardware component, often referred to as processor or core that processes instructions of a program.

DLP Data Level Parallelism. A form of parallelization that emphasizes distribution of data on parallel processingelements.

TLP Task/Thread -Level Parallelism. A form of parallelization that emphasizes on the distribution of threads/tasks across parallel processing elements.

FSM Finite State Machine. Model of behavior of a finite number of states that includes actions and transitions between states.

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

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

HWTI Abbreviation for Hardware Thread Interface.

V-HWTI Abbreviation for Virtual Hardware Thread Interface.

MPI Message Passing Interface.

OTS Off-the-Shelf.

RTOS Abbreviation for Real-time Operating System.

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).

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

Chapter 1

Motivation

ESL (Electronic system level) Design is an umbrella term for tools and methods that allow designers with software programming skills to easily implement their ideas in programmable hardware (like FPGAs) without having to learn traditional hardware design techniques. This enables programmers to access the performance potential of the FPGA through standard programming models on a heterogeneous chip multiprocessor system. This augments the FPGA traditional ability to provide performance increases through custom circuit implementations to bringing performance through scalable parallelism. The use of programmable components and runtime systems brings advancements in application portability and platform reuse. It also brings increased designer productivity; the historical Achilles heel of FPGA hardware design flows. Similarly, HLS tools are bringing similar performance and productivity advancements through their ability in generating custom circuit from sequential code to designers, even those with no hardware design expertise, to rapidly create accelerators. Recent work in middleware and operating systems support for dynamic reconfiguration is bringing new abilities to increase system performance and increase gate utilizations under a more unified virtual machine model. Although existing physical FPGA components can support these capabilities together, no unifying design flows, abstractions, programming models or runtime systems have yet evolved to enable their simultaneous use. As a result these capabilities are still treated largely as distinct and stand alone use cases for FPGA's. The ability to combine these capabilities can bring significant performance advantages to a widening group of designers, and that is what we are trying to address in this thesis work.

The big motivation here is basically how to make FPGAs as accessible as a typical CPU to SW developers. In other words, when a SW developer writes his application code on a typical CPU, they can simply compile the code, no matter which system they are using, and all the complexities of the CPU architecture is abstracted away from him. For example, the software developer does

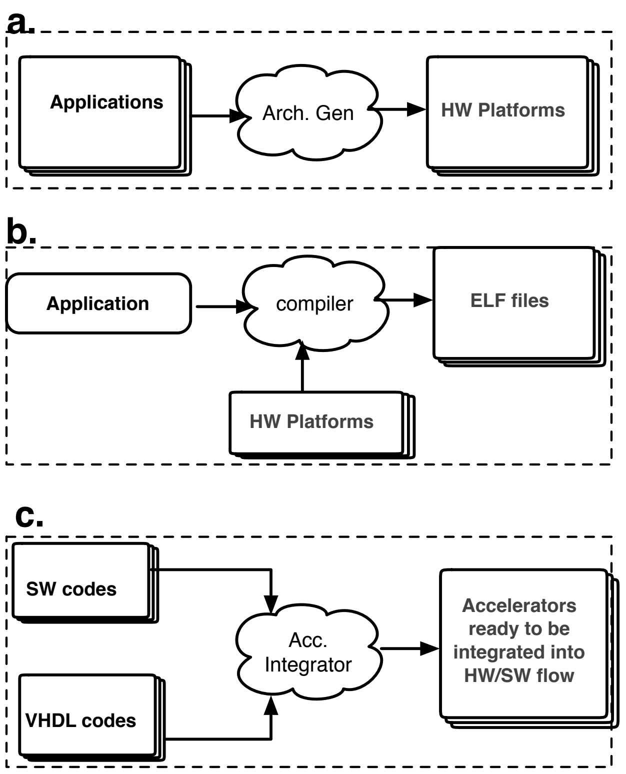


Figure 1.1: Three areas that should be addressed to achieve abstraction in FPGAs

not have to know if the system has a quad core intel CPU, or just a single core AMD CPU. Also he/she does not have to know about memory hierarchy or bus interconnect or DMA controller, etc. This all comes form abstraction. Basically, all the details of HW architecture is abstracted away from SW developer and the compiler tool chain takes care of enabling the notion of ”Writing code once and running it everywhere”.

However, despite this great productivity and portability brought to SW developers by a typical CPU, the performance still remains an issue. First off, the HW can not be customized to accelerate some portions of the code to take advantage of data level parallelism. Also, the number of general purpose processors is fixed and limited, hence limiting the maximum thread level parallelism available in user's code. Here is when FPGAs unique capabilities can help with both customized HW to leverage data level parallelism, and customized multi-core processor system to leverage thread level parallelism. Unlike other alternatives such as GPU boards that require the SW developer to rewrite the whole code in other languages like CUDA, there is no such hassle when it comes to heterogeneous multiprocessor systems on FPGA.

FPGA's complexity and reconfigurability comes with a price that if not addressed and abstracted away from the end users, will dramatically reduce their usage in SW developers community. In order to work with FPGAs, one should have some basic information about HW architecture and Digital design, and most SW developers lack this knowledge. The ideal case would be to have the same productivity and portability as a typical CPU, therefor three areas should be addressed:

- Generating the HW platform bitstream requires broad knowledge in HW architecture and digital design and familiarity with CAD tools. All of these complexities should be automated. This can be done by a HW tool-chain which receives a high level specification file (like the number and type of processors, accelerators,etc) and spits out the final bitstream. This high-level specification file is either manually written or automatically extracted from user's code. Fig 1.1.a shows this first side of abstraction.

- The application code should be compiled to run on any CHMPs platform without the need for the user to change the application. This compilation flow should be aware of the platform details to generate an optimized binary file. Also, during runtime the operating system should smartly auto-tune the application to fully leverage the resources available on this platform including general purpose processors, customized accelerators and partial reconfiguration slots. Fig 1.1.b illustrates this aspect of abstraction.

- In order to transparently take advantage of customized HW circuits (aka accelerators), the integration of new accelerators to the existing HW/SW flow should be fully automated. In other words, the SW developer should only provide the C code for the function they want to accelerate (or even the VHDL code, if available) and they should not be concerned about how this accelerator is going to be modeled, integrated and communicate with the rest of the system. Fig 1.1.c show this crucial part of abstraction.

In sum, abstraction is the key if FPGA's are going to be widely used and accepted in SW developers community. The HW/SW flow should be automated and optimized based on application

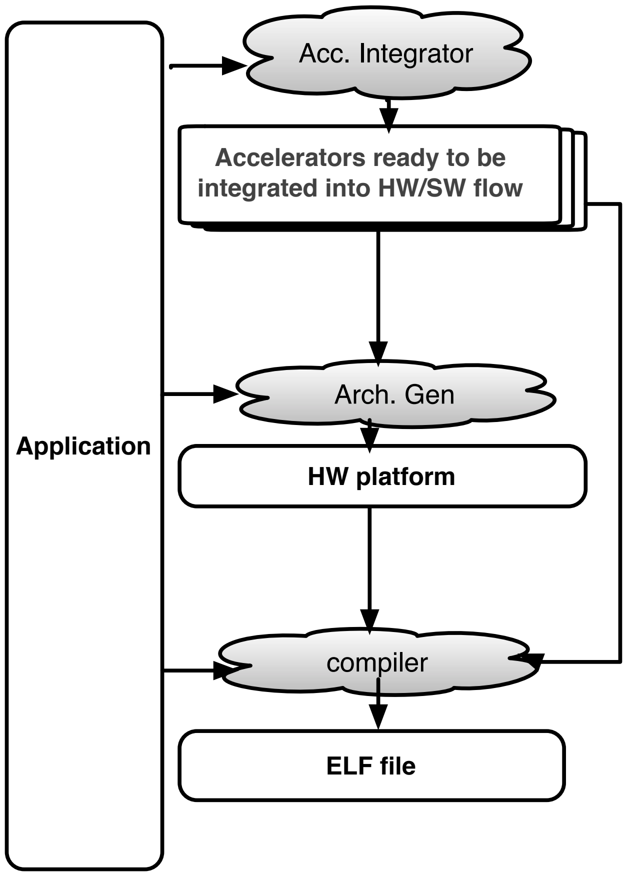


Figure 1.2: The entire flow from C code to final binary file

code to abstract away the details from SW developer. This thesis tries to investigate how this productivity and portability can be achieved in FPGA-based CHMPs. The entire flow is shown if Figure 1.2. First of all, the accelerators are extracted from the C code and then they are used to generate the HW platform. Finally, the HW platform along with the accelerator drivers is used to compile the application into final binary file.

In this thesis, I will argue that the formation of CHMPs system with extensible processor nodes, defined as a processor plus partial reconfigurable slot, can be automated and will cover a very broad range of accelerator use cases with minimal performance loss compared to a fully custom designed architecture. The complexities of different organizations of accelerators can be made transparent and portable through operating system and middleware abstractions. I believe that Heterogeneous Extensible Multiprocessor Systems (HEMPS) can address dark silicon without sacrificing portability and productivity. HEMPS systems are flexible and general enough to abstract the platform complexities from SW developers and also their generation can be automated. They provide the SW users with the HW architecture which supports different levels of parallelism and a transparent programming language.

Designers of real time and embedded systems are continually challenged to provide new system capabilities that can meet the expanding requirements and increased computational needs of each new proposed system, but at a decreasing price/performance ratio. FPGAs have become important components contributing to the creation of a family of Commercial Off the Shelf (COTS) hardware platforms for future real time and embedded systems. FPGAs densities are maturing to allow a complete CHMPS system. The system software for FPGAs must provide a fairly general set of capabilities to support the widening range of applications, but must also be capable of providing the specialized support required to satisfy a particular application's interfacing and performance needs. Creating such a capability for real time and embedded systems applications is a difficult challenge in part, because it requires the simultaneous satisfaction of apparently contradictory forces; generalization and specialization. However new architecture and run time systems support is needed to enable the model to scale within systems containing 10's to 100's of compute components. This thesis investigates the run time services and new architecture to enable developers to express applications that seamlessly scale across specialized large CHMPs systems using the generalized scalable and portable multi-threaded programming model.

Current HW/SW co-design in FPGA-based CHMP systems lack portability and productivity. These two issues has made SW developers community reluctant to switch to FPGAs from general purpose processors. To achieve portability the platform complexities like partial reconfiguration, custom HW circuits and soft and hard IPs should be abstracted away from SW developers . This makes the FPGAs capabilities transparent to the end users. There are also some challenges in the way of productivity. First off, dealing with the CAD tools require a fair amount understanding of HW architecture and digital design, which calls for automating the generation of HW platform . Secondly, as the system grows in complexity, fine/coarse grained hand partitioning of the tasks among different computational nodes gets more time consuming . Finally, there has not been much effort in providing accelerators with stack management and pointer support as SW programmers are used to C-like capabilities.

At the same time, lack of parallelism and power issues has led to Dark silicon problem, forcing general purpose community to consider heterogeneous systems. This great potential opportunity for FPGAs has not been exploited due to lack of both a transparent programming language to capture different levels of parallelism, and the HW architecture which supports different levels of parallelism. There has been a large efforts in how to seamlessly integrate accelerators in a heterogeneous system to combine thread level parallelism and data level parallelism. However, most often this has led to decreasing the productivity and portability. There are three challenges in addressing both efficiency and accessibility at the same time: Abstracting away the complexities of CAD tools, Abstracting away the details of HW platform and finally easy integration of accelerators to the existing system.

1.1 Abstracting away the complexities of CAD tools

One of the main advantages of a FPGA is it's ability to allow designers to create custom circuits at design time. This same advantage has also been one of its main disadvantages; the use of FPGAs requires hardware design skills and the use of hardware centric Computer Aided Design (CAD) tools. This makes FPGAs unaccessible for most software programmers and domain experts. One of the challenges in reconfigurable computing for over two decades has been focused on how FPGAs can be accessed and used through more generalized software development languages, tools, and development flows.

To begin with, one might ask whether generating a general HW platform be automated to spare users from dealing with CAD tools? In other words, based on a high-level specification file that specifies the number of processors, type of accelerators and the type of interconnect we want to generate the final bitstream that is ready to be download on FPGA. Ideally this specification file should be tailored for the SW developer's code. This spares users from dealing with CAD tools, which require a fair amount understanding of HW architecture and digital design. Automating this type of effort will enable researchers to address scientific questions quicker, and with an enhanced ability to increase experimentation across a broader range of configurations. Automating the creation of standard CHMPs architectures also makes it easier to port and maintain the standard operating systems and middleware that bring the fundamental software engineering tenets of portability and reuse onto FPGA based CHMPs system. In the absence of this automated flow, each CHMP system must be created by hand within a CAD tool. For such custom systems this results in long development times and limits the creation of parallel architectures to hardware designers who possess knowledge of digital design, HW/SW co-design, and parallel computer architectures. Additional efforts are required to either create a new, or modify and port an existing operating system and standard middleware run time libraries for each CHMP system. The level of effort to recreate base hardware architectures and run time protocol stacks by hand also makes it nearly impractical for different researchers to reproduce systems to make fair comparisons; a fundamental component of good scientific experimentation.

New design flows for CHMP systems will be required that are architecture centric and exist an abstraction level above High Level Synthesis. These types of architecture centric flows are now trending within System on Chip (SoC) design practices. These practices follow a two phase design process that minimizes hardware development time and effort. In the first phase a base platform is constructed with standard programmable processor cores, memory hierarchies and interconnects. This first phase minimizes design costs through IP reuse and replacement of custom components with re-programmable processor cores. These base platforms are then optimized in the second phase through tuning of extensible cores and finally the addition of a small number of custom components necessary to enable the system to meet all performance requirements. The malleability of the FPGA fabric combined with the increased transistor densities allow designers to follow this same approach for constructing CHMP systems. First a base system can be constructed with soft IP programmable processor cores, and support components available through vendor specific as well as free libraries such as opencores [36]. Once the base system is configured designers can optimize components and create any additional accelerators that would be required to meet a specific applications performance requirements. While current FPGA vendors are addressing IP reuse and component integration within their CAD tools automating the creation of a base system would be better performed outside of any specific vendor flow to maintain design portability. To achieve true vendor neutrality will require new intermediate representations of system architectures that can then be translated into multiple vendor specific formats. The new flow for generating hardware platform should provide following benefits:

- *Complexity/Productivity* by automating the generation of flexible CHMP systems automated
- *Portability* Should be able to provide HW architecture for ”write code once and run anywhere”.
- *Avoiding Dark Silicon* Transistor Utilization and Efficiency (or Dark Silicon), through partial reconfigurable accelerators and/or power aware thread scheduling

1.2 Abstracting away the details of HW platform: Portable code

The second challenge in FPGAs is lack of abstraction, productivity and accessibility. To achieve portability the platform complexities like partial reconfiguration, custom HW circuits and soft and hard IPs and HLS tools should be abstracted away from SW developers. This makes the FPGAs capabilities transparent to the end users. While the general purpose processors provide users with more productivity and more abstraction compared to FPGAs, the FPGAs capabilities are not transparent to the users. This leads to less productivity in FPGAs. Therefore, abstracting both HW/SW design in FPGAs is the key to popularize FPGAs among SW developers community. Emerging programming models for CHMP systems elevate architecture details up into the source code. This increases the complexity of the design process, eliminates portability and can result in inefficient designs.

One way to tackle this problem is to create libraries that abstract the need for programmers to be aware of processors/accelerators and how they are configured within a particular heterogeneous platform. These callable compute patterns should self tune for each system. These calls are running on extensible processor nodes and enable the reusing through high level architecture abstraction and runtime tuning. We call these*Polymorphic functions*.

Polymorphic functions remain portable across CHMP systems, and enable a transparent and autonomous fine-grained HW/SW partitioning. This approach reinstates portability through polymorphic functions, and provides infrastructure for an adaptive runtime system that can perform runtime profiling and dynamic scheduling across systems with different combinations of heterogeneous resources. The combination of polymorphic functions, runtime profiling and adaptive heterogeneous scheduling eliminates the need for designers to exhaustively explore a multidimensional search through static profiling and can result in better resource utilization and performance for CHMP systems with different combinations of processors, and static and partially reconfigurable accelerators. This new portable programming model based on Polymorphic functions can abstract all heterogeneous resource differences out of the source code. Also the run time system profiles and adaptively partitions the high level application onto any combination of available heterogeneous resources.

1.3 Easy integration of accelerators into the existing system

FPGA densities continue to follow Moore's law and are now sufficient to support large CHMP systems. These systems can be populated with tailored mixes of compute resources, such as scalar, single instruction multiple data (SIMD) and vector processors, as well as custom accelerators to meet the specialization needs of each application. This combination of heterogeneous computational nodes allow a complete CHMPS system on FPGAs. However, the side effect of this complexity is how to integrate the customized circuit into CHMPs. On one hand, the accelerator developer should not be concerned about low level details of how the accelerator is going to communicate and transfer data with the rest of the system. On the other hand, the accelerator developer should be given a flexibility in the number of BRAMs and FIFO interfaces needed. Also, how the accelerator is going to be accessed and harnessed should not left to the end user. The challenge is how to make the integration of the accelerator into the existing system as easy and efficient as possible. This helps with both productivity and performance.

1.3.1 Modeling accelerators

On-chip accelerators can be classified into two classes: 1)tightly coupled accelerators where the accelerator is attached to a particular core and can only be accessed by that core. 2) loosely coupled accelerators, which the accelerator is an independent entity which can be shared and accessed among multiple cores [24]. There have been many proposals on how to model, program, and integrate accelerators into a scalable multiprocessor framework. At one end of the spectrum are loosely coupled accelerators that can be viewed as shared system components. At the other end of the spectrum are tightly coupled accelerators that can be viewed as extensions of a single processors ISA. To date, these two ends of the spectrum have been viewed as distinct classes of accelerators, with separate requirements for how the interface into system services, how they should be abstracted for programmers, what granularity of parallelism each type can efficiently support.

Meanwhile, The multithreaded programming model has evolved to enable programmers to combine software threads running on a processor with hardware threads implemented as custom accelerators within the reconfigurable fabric. Enabling hardware accelerators to be represented as threads was a significant step in enabling software programmers to access the potential of an FPGA from a familiar scalable parallel programming model. Traditional approaches provided a finite state machine version of a Hardware Abstraction Layer (HAL) to allow hardware threads to interface into the multithreaded programming model. So the accelerators used a custom HW finite state machine version of a Hardware Abstraction Layer to interface into the multithreaded programming model.

1.3.2 The traditional solution: FSM based HAL for loosely coupled accelerators

While the policies of the multithreaded programming model were defined to explicitly allow the expression of scalable parallelism, HALs as supporting mechanisms make inefficient use of hardware resources and impose restrictions on the level of parallelism that can be exploited. HALs are defined to provide system services for accelerators that are modeled at the coarse grained thread level. This can lead designers to synthesize a thread that includes sequential instructions that occupy gates and provide little to no performance increases.

While FSM based HALs for loosely coupled accelerators were appropriate for platform FPGA's with small numbers of processors and hardware threads, the implementation approaches are not viable for todays larger CHMP systems. The model supported a thread as the finest granularity of concurrent component supported. This required FPGA resources to support the sequential portions of a thread, and limited the thread to contain only a subset of functionality that could be synthesized. Implementations used finite state machines within a custom hardware interface to provide each thread with the system functionality required to interface and interact with the operating system. The size of the custom hardware interface grew as more state machines were added to provide a full range of functionality. FSM based HALs was acceptable for interfacing a thread but has limitations for next generation systems that may require moving functionality across the HW/SW boundary, better support of languages such as C that require a stack, ease of expansion and modification.

An FSM based HALs for loosely coupled accelerators abstracted custom HW as a hardware resident thread that accessed key operating system services such as mutex operations, as well as providing access to a linear address space, through a series of hardware based finite state machines encapsulated within a virtual abstraction layer. This allowed a complete thread body to be implemented as a hardware thread and seamlessly interact with all other software and hardware resident threads throughout the system. This model proves inefficient in resource usage, restrictive in supporting different accelerator models and for dynamic scheduling and resource allocation. First, any change to the operating system services required a redesign and re synthesis of the hardware resident virtual abstraction layer. Further the size of the virtual abstraction layer quickly grows greater than the size of a standard processor such as MicroBlaze. Additionally resources are required to implement the complete, and not just the computationally intensive portions of the thread

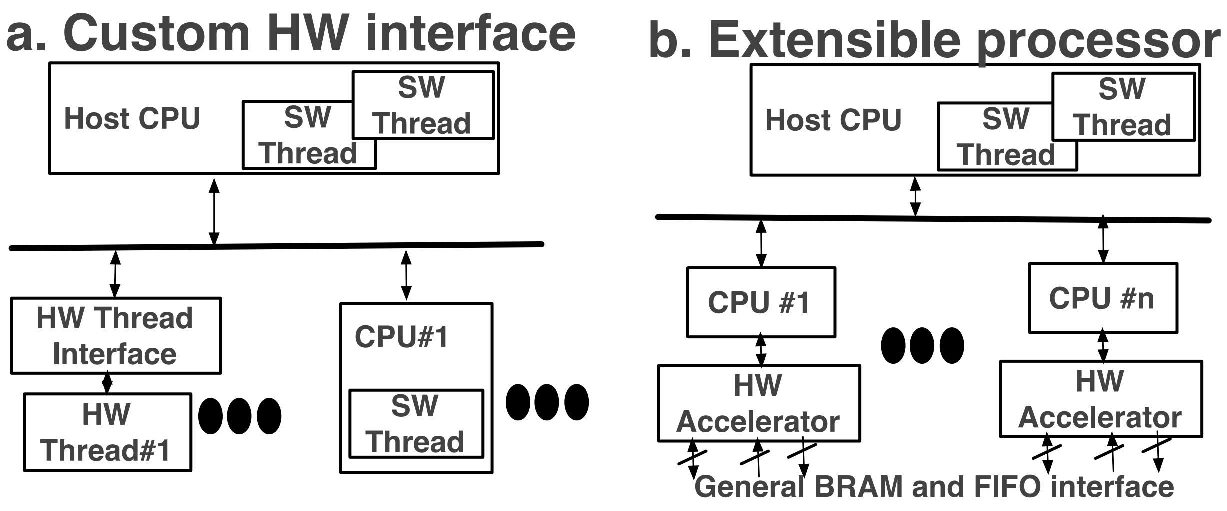


Figure 1.3: Loosely coupled accelerators Vs. Extensible processor.

yielding large accelerators. In sum, the drawbacks of HW threads implemented as loosely coupled accelerators with FSM based HAL approach are:

- changing or adding new features to HW thread interface requires HDL re-designing and re-synthesizing. It is not only time consuming, but also results in a bigger size Hardware.
- There is no fine grained HW/SW partitioning. The entire thread should be either run in HW or SW, which as a side effect results in a bigger HW threads with no added benefit for the sequential part of the thread.
- There is no support for Stack management for HW accelerator, which stops some algorithms like Quicksort to be implemented in hardware.
- Integrating new HW cores into system with different interfaces is hindered by what HW thread interface dictates.

1.4 Extensible processor

Can a single processor-accelerator combination unify the varying use cases of accelerators within FPGA based CHMPs systems? This node can work as the basic block of automatically generated

CHMPs systems, as well as autonomously executing the portable code targeting these systems. A processor-based HAL has the potential to unify hardware and software threads, provides stack management and pointer support plus distributed OS and RPC services for the accelerator. Implementing the HAL using a general purpose processor instead of a custom finite state machine provides increased flexibility and productivity. Also Using a general purpose processor as a front end in place of a custom circuit eliminates the need to distinguish between a hardware and software thread. All system components can be viewed as extensible processors and support any combination of hardware and software threads. Using a general purpose processor as a front end to the hardware accelerator turns a dedicated hardware thread into a much more flexible extensible processor. This extensible processor nodes can be combined to form a HEMPS system and directly ties to our first argument of automating the generation of CHMPS systems. A CHMP system can be built using extensible processors with static or partially reconfigurable accelerators, and DMA controllers for fast data transfers. It unifies both models of loosely coupled and tightly coupled accelerator in one architecture, as well as providing portability for applications. Figure1.3.b shows a high level view of an Extensible processor

Allowing the accelerator to be connected directly to a processor also extends the earlier model of loosely coupled accelerators across a system bus to also include tightly coupled access from the processor's register set. Each extensible processor can be used as both a loosely coupled or tightly coupled accelerator. This unifies the two models that have traditionally been treated separately. In short, an extensible processor model provides a more efficient implementation of a HW-based HAL, provides capabilities needed to standard programming models and languages, allows systems to scale, and extends the accelerator's use case.

Moreover, there has not been much effort in providing accelerators with stack management and pointer support as SW programmers are used to C-like capabilities. Extensible general purpose processors serves two purposes : Running the sequential part of the thread and autonomously assigning only the parallelizable part of it its attached HW accelerator. The general purpose processor in this node has the potential to provide stack management and other optimization services like RPC, overlapping data transfer with computation, etc for the HW accelerator.

Chapter 2

Introduction

The continued increase in fabrication densities of FPGAs are yielding devices that can support CHMPs architectures. Current generations of FPGA's such as the Xilinx VC707 can support over a hundred processors on a chip. As an example, the hthreads in the cloud project now makes bitstreams publicly downloadable for systems with up to 150 Microblaze processors. Bitstreams for accelerator rich MPSoPC's are available for systems with up to 36 Microblaze processors plus 36 partial reconfiguration slots are also available [21]. FPGA based CHMPs systems offer the advantage of allowing designers to include custom compute components such as accelerators into systems with scalable numbers of programmable processors to meet challenging application performance needs. To increase designer productivity commercially available C to HDL tools are available to translate portions of the application C code, such as loops, to be synthesized into accelerators [11]. Even though the transistor densities of commercially available FPGA's contain sufficient gates to support large and heterogeneously diverse MPSoPC systems, and C to gates capabilities are becoming common, how to form and program such large and heterogeneously diverse architectures is not fully understood. The question is *who and how* is going to use these resources available on FPGAs?

CHMP systems represent a new era in reconfigurable computing with system architectures that can bring the performance benefits of an application specific design but driven and accessible through familiar scalable programming models. Enabling these systems to be used by application developers will require new capabilities in processor centric design automation, more adaptive runtime systems, and new middleware abstractions within concurrent programming models.

To explore this thesis statement of this work, I want to investigate the following set of questions:

- Can the creation of CHMPS systems with programmable processors plus custom accelerators be automated without sacrificing the ability to customize systems ?

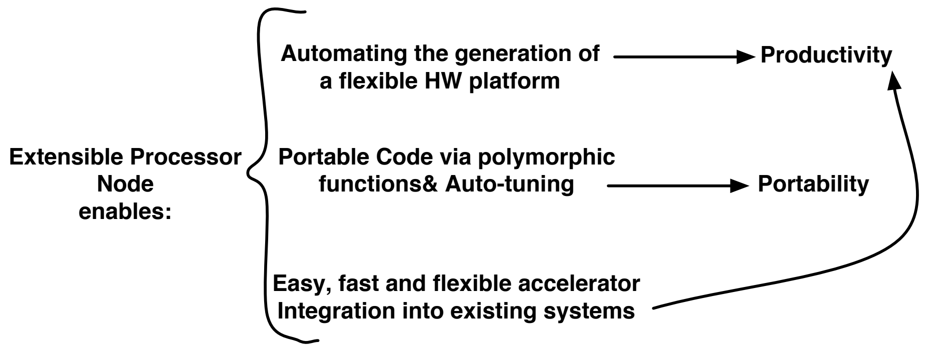

Extensible Processor Node enables:
Automating the generation of a flexible HW platform
Productivity
Portable Code via polymorphic functions& Auto-tuning
Portability
Easy, fast and flexible accelerator Integration into existing systems

Figure 2.1: Extensible processor node enables

- Can the platform complexities of an accelerator rich CHMPs system be abstracted to re-enable the notion of ”writing code once, and run everywhere”?.
- Can a single processor-accelerator combination unify the varying use cases of accelerators within FPGA based CHMPs systems ?

2.1 The need for making FPGA’s accessible to SW developers

The United States Bureau of Labor Statistics reported that in 2010 there were approximately 82,000 hardware engineers and over 1.2 M software programmers [10]. Although the densities and capabilities of FPGAs continue to grow, the lack of standard operating systems support and software centric programming models has continued to hinder their adoption by this large cadre of software programmers. FPGA based operating systems researchers have addressing the issue of FPGA accessibility for software programmers for the better part of two decades.

SW programmers should be spared of details of hardware platform in FPGAs. This is a multi-faceted problem. First off, there should be a model as to how to integrate custom HW into a multi-core system. Second, how to automate the generation of such a heterogeneous multi core system. And finally, how to compile the application for different heterogeneous multi core systems without the need to change the application or putting platform-specific details up into the application. Having said all of these, the basic question is what should a basic computational node look like in a heterogeneous multi node system? The organization of the computation node would be the basic block for generating the HW platform as well as the compile flow. Most recently the architecture community has been advocating accelerator rich heterogeneous multiprocessor architectures called chip heterogeneous multiprocessors (CHMPs). These types of systems are gaining interest within the FPGA community as well. One open research issue is how custom accelerators should be interfaced into CHMPS architectures and abstracted within a higher level programming model.

2.2 Dark Silicon challenge

The increased capabilities offered by a CHMP system does come at the costs of increased complexity for designers in constructing the base system, decreased code/design portability, and an increased potential to fall victim to ”dark silicon” [28]. Dark Silicon refers to transistors on a chip that are not used due to two inter-related issues; lack of available parallelism and fixed power budgets [28]. Dark silicon places limits on an applications ability to achieve desired speedups and prevents the application from scaling across generations of chips that can provide increased transistor densities. In response to dark silicon the general purpose computing community has already transitioned from homogeneous many-cores to *chip heterogeneous multiprocessors*. These systems contain scalable mixes of CPU’s, GPU’s, extensible processors and accelerators to better exploit all levels of parallelism and provide better energy efficiencies [42]. Reconfigurable computing bases, such as FPGA’s are not immune from dark silicon. FPGA densities are already sufficient to host a complete heterogeneous chip multiprocessor [12, 58]. This use case represents an interesting convergence between the general purpose and reconfigurable computing communities, albeit from opposite directions.

Current dark silicon concerns for FPGA’s stem more from a lack of available parallelism than fixed power budgets. Current process technologies are not yet violating Dennard scaling [26], where achievable transistor density levels would require turning off a portion of the chip to meet fixed power budgets. Chip vendors are now seeking to use modern process technologies such as Intel’s 14nm fabrication lines that may cause FPGA densities to violate Dennard scaling in the near future. Current dark silicon concerns for FPGA’s result from idle transistors that are not supporting parallelism. Dark silicon can appear even within heterogeneous chip multiprocessor systems. First, mismatches between the user visible higher level programming model and the actual capabilities of the heterogeneous resources will result in underutilization. Second, changes in runtime behaviors of the application or underlying system can lower utilization. Third, modifications or updates to the original application can effect the underlying architectures ability to exploit new behaviors. Thus a key challenge in keeping transistors at high utilization levels becomes a runtime issue.

FPGA components support the capability of allowing portions of the gates to be dynamically reconfigured during runtime. This promises to address the issue of "dark silicon" or portions of the chip that are not being used that now plague standard manycore architectures. Accelerators and components can be paged into the silicon on demand by the operating system during runtime. Reconfiguring transistors during runtime has the potential to increase transistor utilization. This is conceptually similar to current multithreading techniques that context switch multiple threads across a shared processor resource to increase utilization. The difference is primarily in what is swapped in and out; binary executables or bitstreams. The same application that has been partitioned into hardware and software components, profiled and tuned for maximum performance on a specific architecture will have to be re-tuned, partitioned, profiled and optimized if the configuration of the underlying CHMP changes. This is counter to the software worlds desire to write once, run anywhere.S Applications optimized for one particular architecture configuration may not run well on another. Architecture configurations can vary during runtime . A partitioning that is optimized based on the availability of accelerators or a vector processor would not perform efficiently if these resources were busy and the application was alternatively mapped onto a set of general purpose processors. Further programmers cannot know how the system will exhibit non-deterministic behaviors during runtime.

2.3 Modeling and abstracting the accelerators in a CHMPS architecture

One avenue of research has focused on using the multithreaded programming model as a unifying abstraction over Hardware/Software co-designed applications within an FPGA. This approach was proposed in 2003 by [49, 63] and has continued to gain in popularity [45, 19, 39, 65, 41]. The appeal of the model is it's ability to view physical accelerators as abstract threads. The model defines standard policies such as mutex and semaphore operations that allow multiple accelerators to run concurrently within the FPGA, synchronize with other threads (accelerators) and share data. The HAL was required to provide hardware threads with equivalent system call interfaces into standard Pthreads APIs. This includes as acquiring and releasing a mutex, transferring data, and support communications with other threads running throughout the system. One of the promises of FPGAs was the ability to deliver performance levels close to those associated with custom circuits, and this has to be harnessed by SW programmers.

The FPGA based operating systems research community has been investigating the use of the multithreaded programming model as a unifying framework. In this model custom hardware accelerators are abstracted as detached hardware threads. The flexibility of the model allows multiple accelerators to be defined and operate concurrently within the FPGA. This extends the earlier use case of FPGA as a single detached accelerator treated as as co-processor to an external CPU.

An important aspect of the model is the separation of policy and mechanism. Policies such as those defined in the Pthreads standard, are accessed through Application Programmer Interface (API) calls. System designers are free to implement the APIs using platform specific mechanisms. On commercial desktop systems, mechanisms are built using existing hardware and protocols, the processors Instruction Set Architecture (ISA) and Application Binary Interface (ABI) definitions, and standard software protocol stacks. FPGA operating systems researchers had no such set of predefined hardware components, protocol stacks, or ISA and ABI's. Thus a challenge for first generation efforts focused on defining these standard types of mechanisms encapsulated within an additional hardware component call a Hardware Abstraction Layer (HAL). The HAL served to provide a set of standard register interfaces to replace soft traps for access to the mechanisms, and sets of finite state machines to replace non-existent software system service libraries. These first generation efforts were successful in validating the approach. However this HAL suffers resource inefficiencies, limits the granularity of parallelism that can be supported, and presents difficulties in modifying and updating the system service interfaces and mechanisms.

This model extended the historical view of a FPGA as a single accelerator by allowing accelerators to operate concurrently with other software and hardware threads. Operating system services were provided for the hardware threads through HAL, which extend OS services to hardware accelerators connected on the system bus. The HAL Abstraction layers have evolved for supporting a standalone custom HW such as a thread. These HALs are implemented as a series of finite state machines that provide the hardware threads with the equivalent of libraries of linkable software system calls. The HAL allowed the hardware thread to request standard operating system services such as acquiring and releasing a mutex, or share data with other software and hardware threads running throughout the system. It also allowed the accelerator to be viewed as a schedulable detached thread by the runtime system. The HAL was fundamental in allowing programmers to use the multithreaded programming model to create multiple accelerators. Those accelerators can operate as concurrent threads running independently throughout the FPGA fabric.

The definition and use of the HAL has enabled accelerators to be viewed a threads, and has allowed the accelerator access to important policies such as requesting mutexes and semaphores that are critical to the model. However it does not remain faithful to the full separation of policy versus mechanism which is important for portability. The use of a HAL indirectly requires application developers to code and treat hardware threads differently from software threads. The run time system must also distinguish and schedule hardware and software threads differently. The size HAL will grows as additional functionality or system services are added. Finally, unlike system software that can be easily modified and quickly compiled, changes to the HW-based HAL required low level circuit design skills and resynthesis. In other words, while this approach successfully abstracted the HW/SW boundary from the programmer and enables multiple loosely coupled accelerators to interact with the runtime system, it suffered several disadvantages. First, the granularity of the computation mapped into the accelerator was at the coarse grained thread level. Sequential sections of code contained within the thread body became synthesized as part of the hardware accelerator. This results an inefficient use of the FPGA resources. Second, the size of a HW-based HAL can become large if sufficient OS service functionality is included to allow the accelerator to appear to the system as an independent stand alone detached thread. Additionally, unlike system software that can be easily modified and quickly compiled, changes to the HW-based HAL required low level circuit design and resynthesis. Each time more functionality is added the size of HAL increases.

In this thesis we try to see if a hardware HAL can be replaced with more flexible and programmable *extensible* processors. On our Xilinx based systems, we form an extensible processor with a Microblaze, set of standard FSL links, and three scratch pad BRAMs. The extensible processor provides the following additional capabilities over the first generation hardware based HAL. First, The size of a HAL will grow in proportion to the functionality placed into the HAL. Current HAL's routinely occupy more gates than a soft IP processor such as the Microblaze. Changes or additions to the system services requires hardware redesign and synthesis, a level of hardware design skill that is beyond most operating systems designers. The use of a hardware HAL has the secondary effect of imposing inefficient usage of transistors within the user code that was mapped and synthesized within the accelerator; even the sequential portion of the thread needs synthesized. The use of the HAL also required the accelerator house the complete thread. This places restrictions on the application code within the thread body to be synthesized. Most hardware HALs do not include the additional finite state machines and local memories to provide stack support for the hardware thread. This prevents designers from mapping thread code that contained function calls or allocated locate variables from a stack into hardware. Either the thread body must be re-written or the thread designated as a software thread.

We want to investigate whether the hardware HAL that sits in front of an accelerator can be replaced with a programmable processor to form a much more flexible and efficient *extensible processor*. The extensible processor allows the uniformity of the multithreaded programming model to be restored. Programmers no longer need to draw a distinction between threads that will run in software or hardware. All types of threads can run on an extensible processor. The run time scheduler can then map any thread onto any available extensible processor. An extensible processor allows a single thread to be split across the HW/SW boundary. The sequential portions of the thread run on the processor, and the computationally intensive portions of the thread can run on the accelerator to leverage data level parallelism. We investigate the possible resource savings this approach can provide over hardware based HALs, as well as the HW/SW partitioning which might provide interesting counter intuitive run time results. I think replacing custom hardware with a programmable processor can result in increased performance. Finally we try to see how the extensible processors stack can be used by the accelerator to allow code that contains function calls and local variables to be synthesized and mapped into the accelerator. This can increase the usage of the accelerator to threads that contain function calls, recursion, and local variables. We also explore performance results for systems built on extensible processor to see if it can scale into 100's of processors and accelerators, and can be programmed using concurrent threads.

As far as providing accelerators with operating system services goes, there are two approaches. One approach used a full Linux stack running on a master processor [63, 45, 39, 65]. Operating system services are executed on the Linux stack on behalf of the accelerators. This approach required the HAL to include RPC call support to communicate with wrapper functions that executed on the Linux stack running on the master. This approach introduced additional latencies for accelerators accessing the OS services running on the master node, and from sequentializing service requests. The second approach sought to reduce latencies by distributing more OS services into the HAL [19]. This approach reduced access latencies and eliminated the sequential server bottleneck. In the work, we follow the second approach as it is more scalable.

In this thesis, we try to show how a HW-based HAL can be replaced by a CPU-based HAL. Without any loss of functionality we show how a general purpose processor such as a MicroBlaze can be used as a plug-in replacement for a HW-based HAL. Replacing custom hardware with a programmable processor brings the obvious benefits of increased productivity and flexibility. Changes and updates to system services can be achieved through software compilation instead of hardware synthesis. Interestingly hardware savings are provided as the MicroBlaze processor requires fewer gates than a typical HW-based HAL [15]. Also the size of the accelerator can be reduced by moving the sequential portion of the thread out of hardware and into the MicroBlaze. Importantly combining a general purpose processor with an accelerator forms a very flexible *extensible processor*. Replacing the custom hardware based HAL with a programmable processor can increase performance, productivity and flexibility, while reducing overall area size. The extensible processor can still be used in it's original role of providing system services to a detached hardware accelerator thread. It also allows the accelerator to become more tightly coupled with the processor. This allows the size of the accelerator to be reduced by moving the sequential portion of the thread out of hardware and execute on the Microblaze.

An extensible processor is in a better position to support the needs of higher level programming languages as well as better support scalability within the system and multithreaded programming model. The standard processor's stack can be used to support function calls and recursion within the accelerator. The ability to migrate additional OS services into the HAL without increasing circuit size allows each extensible processor to assume more autonomous behavior. This eliminates the need for a HW-based HAL to provide remote procedural calls (RPC) to system services that are provided within a full OS stack running remotely on a master processor. An extensible processor also removes the need to draw distinction between a hardware and software thread; an extensible processor can execute any combination of software and hardware.

We try to remodel the accelerator as an extensible processor implemented through a tight coupling between the static or dynamic accelerator and a front end processor. This model has the potential to be resource efficient, portable, and better supports dynamic partitioning and allocation of resources. Without any loss of functionality the Microblaze can be simply viewed as a plug in replacement for the hardware virtual abstraction layer. Changes to the operating system can be realized through software compilation in place of hardware synthesis. Further resource savings are achieved as the sequential portion of a thread can be migrated out of hardware and into software executing within the Microblaze. An extensible processor model now widens the use of the accelerator to exploit a greater range of parallelism, from fine grained data level parallelism, through VLIW, and instruction fusion. Importantly this allows better resource utilization and support for dynamic tuning. Each extensible processor is now available to run any thread, with the choice of using an accelerator being made autonomously on each processor. Under dynamic reconfiguration each processor can make an independent decision to download a new bitstream into it's partial reconfiguration area. After introducing this notion of Extensible processor, we use it in a multiprocessor system with ICAP and DMA engines tailored to each node.

2.4 Need for a general heterogeneous multiprocessor system

We try to integrate multiple extensible processors to form a new class of multiprocessor architecture we call Heterogeneous Extensible Multiprocessor Systems (HEMPS). HEMPS systems support both thread level parallelism (TLP) through the multiple processors, as well as data level parallelism (DLP) within the accelerator extensions. The operating system for HEMPS should support the seamless use of static or partially reconfigurable accelerators to enable portability across different heterogeneous multiprocessor systems. Our standard HEMPS systems include multiple DMA controllers to support fast transfer of data between memory as well as partial bitstreams. HEMPS system unifies both models of loosely and tightly coupled accelerators. From an architecture perspective a HEMPS system is chip heterogeneous multiprocessor system built on extensible processor nodes. This allows the system to exploit a wide range of parallelism from coarse grained threads running across the scalable numbers of processors, to fine grain parallelism within the accelerator extensions to each processor. The HEMPS model allows accelerators to be included statically or dynamically by the operating system under partial reconfiguration rules. The static or dynamic accelerators are both visible by the operating system as schedulable sharable resources, or as dedicated extensions to a processors ISA. Within a HEMPS system any accelerator can be used in both roles. The HEMPS architecture provides portability and efficiency over CHMPS. In a HEMPS system we target an MIMD/accelerator model. The MIMD model provides thread level parallelism across the multiple processors, with the ability to accelerate execution of each thread by extending each processor with custom accelerators. However, required designer skills necessary to construct a complex heterogeneous multiprocessor systems change from digital logic to those associated with creating a complete parallel architecture.

The fact that FPGAs support a complete CHMP system within a single programmable chip [23], represents an interesting convergence with the general purpose computing community. General purpose computing community is pursuing heterogeneous manycore architectures to combat dark silicon [28]. The malleability of the FPGA fabric offers designers the advantage of tailoring a CHMP system with different numbers and types of processing elements and accelerators to meet the specific performance needs of each application. This allows *FPGA-based CHMPs* to be built that are tailored to the exact types of parallelism that may be available in each application. However, creating these systems would be too hard for a typical SW developer. So from the HW point of view, automating the generation of CHMP systems is a must. Moreover, From SW point of view, familiar concurrency models such as asynchronous threads can be used over scalable numbers of general purpose programmable processors. Data level parallelism can then be exploited on programmable vector and array processors as well as within co-processor accelerators. This type of automation allows the creation of complex heterogeneous chip multiprocessors by software designers and significantly reduces the time and complexity of creating complete Systems on chip (SoC) systems. As far as Hardware design goes for these CHMP systems, creating the base architecture within the FPGA is becoming easier. Through the availability of standard IP components, and tools that can automate the assembly of the IP components into a complete CHMP architecture. Vendors routinely supply soft IP components such as buses, interrupt and I/O components as well as programmable processors such as the MicroBlaze [4].Community efforts are providing additional soft IP components such as vector processors as well as libraries of open source accelerators [36].

Vendor tools allow users to build systems with dynamically reconfigurable accelerators. All of these, care calling for automated tools to eliminate the need to hand assemble architectures within vendor CAD tools. From the application designers perspective, FPGA based CHMPs systems hold the promise of exploiting any and all levels of parallelism that may exist in their application. Familiar concurrency models such as asynchronous threads can be used over scalable numbers of general purpose programmable processors. Data level parallelism can then be exploited on programmable vector and array processors as well as within co-processor accelerators. Even Very Long Instruction Word (VLIW) and custom, or fused, instructions can be supported in hardware accelerators.

CHMPs systems hold the promise of providing a rich set of scalable general purpose and customizable processing components for designers to field systems that can meet challenging real time functionality, timing, and energy requirements. The malleability of the FPGA substrate allows designers to reduce design time by first adopting reusable base systems of scalable heterogeneous programmable processors and then augment these base systems with additional performance boosting custom accelerators or tuned processors. Partial reconfiguration, or the ability to swap accelerators into and out of a running system adds additional advantages for future FPGA based CHMP systems. The use of partial reconfiguration techniques can boost overall system efficiency through increasing the utilization of transistors. This may well become an important factor in addressing energy efficiency [44], and play an important role in offsetting ”dark silicon” or transistors that cannot be used due to lack of parallelism or chip power restrictions [28].

These CHMPs powerful platforms can be customized with scalable numbers and types of general purpose, extensible, and custom heterogeneous programmable processors, as well as custom hardware accelerators to meet the demanding needs of each application. Figure 2.2 shows the range of systems that can be realized within a modern FPGA. While these platforms offer designers significant flexibility in the types and numbers of components than can be integrated into an architecture this same flexibility introduces new challenges for designers when attempting to write *efficient* applications that are portable on, across and between different systems. Figure 2.2 shows the relationship between designer effort and and system complexity. FPGA’s can host simple scalar processor systems with linear address space models. Designs can be easily developed and modeled using historical software development approaches. Once the application is developed, profiled and optimized, it remains fully portable through simple recompilation if the processor within the FPGA is upgraded or changed. Researchers have successfully enabled CPU based SMP multiprocessor

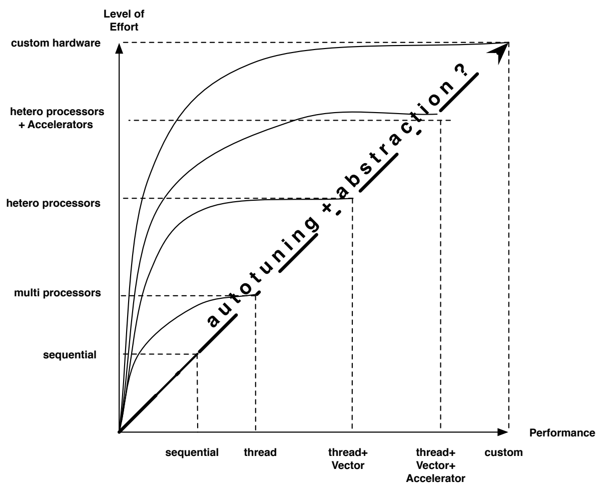


Figure 2.2: Performance Vs. complexity of the system

systems with linear address spaces within modern FPGAs. Familiar middleware such as Pthreads have been used to abstract the set of multiple homogeneous compute components into a single unified architecture continuing to offer the important notion of portability. Concurrency issues associated with the multithreaded or message passing programming models introduces additional complexity into the application development process. However these models extend the important notion of portability across multiprocessor architecture families with homogeneous compute resources.

The capabilities of CHMPs systems bring with them new challenges during architecture assembly as well as application design. Architecture assembly is focused on determining a proper set of heterogeneous resources to best meet the needs of an application. Wide selections of soft IP components are now available, ranging from general purpose processors, vector processors, and extensible processors along with the traditional approach of creating custom accelerators. This wide selection of disparate components coupled with the ability to integrate tens to hundreds of these components across different memory and interconnect configurations within a single FPGA makes this process extremely challenging even for experts in multiprocessor architecture design. Once chosen, the components must then be assembled into the complete system, debugged, and tested. Current design flows force this development process to occur by hand within vendor specific CAD flows. This development process is time consuming, error prone and prevents portability of the architecture between generations of vendor specific chips as well as across vendor platforms. Long synthesis times must be tolerated each time the architecture is updated or changed. All of these challenges call for automating the generation of HW platform, as we later discuss in the approach section.

2.5 Portability of the applications over different CHMPS systems

CHMPs systems increase the complexity of the application design process. Applications optimized for one particular architecture configuration may not run well on another. Architecture configurations can vary during runtime as the runtime scheduler changes the numbers and types of processors and accelerators available. A partitioning that is optimized based on the availability of accelerators or a vector processor would not perform efficiently if these resources were busy and the application was alternatively mapped onto a set of general purpose processors. The complexity offered by a CHMP system with Designers cannot cost effectively engage in exhaustive profiling of combinatorial numbers of code partitionings and mappings for systems with large numbers of heterogeneous processor/accelerator mixes, interconnect and memory hierarchy combinations. Further programmers cannot know how the system will exhibit non-deterministic behaviors during runtime for systems with 100s of mixed processor types and complex interconnects within a single, let alone across multiple unique platforms. Code should be written once by the application developer targeting a uniform abstract architecture and be portable across all vendor platforms.

Constructing a CHMP hardware platform is becoming easier, but developing and tuning applications to run efficiently across a multiprocessor system with heterogeneous resources is becoming harder. Once coded, profiled, and tuned a time critical application should be reusable and portable between systems and across generational platforms. Emerging programming models for heterogeneous systems such as OpenCL [6] prevent an application from being portable across processor specific boundaries. The introduction of heterogeneous compute resources prevents designers from developing applications that are portable and reusable, fundamental tenet's of best practice methods in software engineering. Using such programming models requires developers to decompose, evaluate and tune each application by hand for each unique configuration of heterogeneous resources. If the resources change the developer must reevaluate how the application was originally decomposed, coded, and mapped across the resources. This strict binding of application kernel to resource prevents the runtime system from attempting to increase overall system utilization by dynamically repartitioning and re-scheduling the application on available resources. Thus binding specific application kernels to specific heterogeneous resources can introduce performance bottlenecks, and impose substantial design times for developers to iteratively hand code, profile, and tune each application. As the complexity of future CHMPs systems increases the effort required to exhaustively develop, profile, and tune an application will quickly become prohibitive.

Several issues need to be resolved to bring portability and reuse to FPGA based CHMPs systems. First, our traditional compilation flow must be modified. Current compilation flows target producing executables for a single ISA that is portable across all processors within the system. CHMPs systems may split an application across groups of processors with different ISA's. This requires new approaches for generating multiple ISA versions of source code, cross linking functions, and resolving differences in Application Binary Interfaces (ABIs). Our ability to bring application portability through uniform software abstractions will significantly increase designer's productivity. The programming model abstractions and runtime systems that have brought portability and reuse for systems with homogeneous processor resources are not being pursued for CHMPs systems, and not yet available for systems with mixes of heterogeneous processors. Differences in processor ABI's have prevented the adoption of single unifying operating system images such as Linux on CHMP's systems. For example, different processor atomic operations such as load_linked, store_conditional are not compatible with test_and_set. This prevents standard operating systems from providing synchronization primitives across heterogeneous resources. Consequently, Differences in each processors and accelerators capabilities require architecture dependent coding practices that eliminate the runtime schedulers ability to map any application kernel to any available resource.

Interestingly the flexibility of CHMPs systems has introduced a paradox for the programming community. A new class of heterogeneous programming models evidenced by [6], has emerged to support statically configured CHMPs systems. A common trend within these heterogeneous programming models is to reverse the long established practice of increasing portability and productivity through higher levels of abstractions. and are only applicable thus far to systems with combinations of statically defined hardware, (i.e. CPUs, and GPUs). Programmers must once again study the underlying architecture and introduce architecture specific code to achieve near optimal performance. This is occurring as many of these programming models lack the automated mechanisms for profiling and tuning an application to the system's available resources. A lack of automated tuning capability is particularly detrimental for FPGA's that allow resources to be modified. Data must explicitly be marshaled between different memory partitions, and parallelism must be expressed using pragma's and processor specific data parallel constructs. Some of the platform specific requirements result from lessons learned during the prior parallel processing era that showed compilers could only achieve marginal success in automatically extracting data level parallelism from sequential code. This has an unfortunate effect on programmer productivity and application portability. One cannot rely on the programming model and the operating system (OS) to ensure their application remains portable across CHMP systems. Programmers must actively engage in hand partitioning and exhaustive static profiling of the application to find an optimal partitioning on a given CHMP system.

Systems radically increase the dimensionally of the design space in which designers must work to efficient applications upon which portability in homogeneous systems was achieved. Instead application developers must posses detailed knowledge of each architecture to write efficient applications. New heterogeneous compilation techniques as well as additional development time is required to explicitly decompose an application into unique code sections using the non-portable constructs and pragmas to expose parallelism in forms suitable for each heterogeneous processor resource. Differences in processor ABI's prevent the adoption of a single unifying operating system such as Linux to abstract the heterogeneous resources. Developers must possess detailed knowledge of the memory hierarchy and include explicit data marshaling commands within their application code. Thus code written for earlier homogeneous SMP systems is not only not portable between homogeneous and heterogeneous systems, but also not portable across different heterogeneous systems. This lack of portability also challenges the historical development approach of attempting to statically optimize the application through user profiling and tuning. The search space that a user would have to explore to experimentally decompose, profile and rework the application for a system with 10's to 100's of heterogeneous resources grows exponentially. The introduction of heterogeneous concurrency also introduces probabilistic operating behaviors across buses and interconnects that cannot be exhaustively evaluated using static profiling techniques.

This is a concern as the complexity of writing explicit data parallel code proved very complex for the average programmer and importantly, resulted in non-portable code. The parallel processing community moved away from this approach towards cheap clusters comprised of scalable numbers of homogeneous processors accessed through scalable and portable programming models. In particular the multithreaded programming model gained popularity as a unifying model and has been used within the reconfigurable computing community for enabling custom hardware threads within the FPGA fabric to seamlessly interact with software threads running on CPU's [16, 45, 63, 65]. These models are requiring programmers within their code to exploit the different levels of parallelism supported by the heterogeneous resources. History does show the necessity of explicitly exposing parallelism using data parallel constructs and pragmas within the source code. Programmers must once again learn low level details about each architecture and pursue mixed coding styles to meet the specific needs of each different processor or groups of processing elements. Emerging heterogeneous programming models are once again requiring Emerging CHMPs architectures are motivating a resurgence back to programming models that include explicit and processor specific parallel constructs. This has an unfortunate effect on programmer productivity; they cannot rely on the programming model (such as Pthreads) and operating system to enable their application code to be fully portable. They must now code to the machine, engage in tedious and exhaustive design space exploration and run exhaustive profile traces to optimize for each particular set of heterogeneous resources. History also shows that this approach eliminates portability and negatively effects designer productivity. Hand partitioning cannot account for the probabilistic run time behaviors that result from asynchronous concurrency models running threads across 100's of processors. As the number of available processors changes during run time how a function is decomposed into concurrent threads will also change. On todays CHMPs systems with relatively few numbers and types of heterogeneous processors, this is just an inconvenience that can be overcome with diligence. However for future systems with 100's of mixed types of processors, accelerators, multi-tiered memory hierarchies and complex interconnects the complexity of hand partitioning and profiling applications for all potential runtime eventualities becomes intractable.

Even verifying correct operation of a concurrent application is problematic as the observed behavior of a statically tuned concurrent application is not guaranteed repeatable under our traditional asynchronous concurrency models. Buses and interconnects can exhibit probabilistic behaviors as data transfer requirements throughout the system varies. Thus while this philosophical switch may help bridge the transition from homogeneous to heterogeneous systems, it may ultimately prove as only a stop gap measure until more traditional programming models emerge that can restore portability. Resolving heterogeneity at its root is a piece of a much bigger puzzle that will drive long term operating system research agendas. Creating such a new programming model is not an easy task. Processors with heterogeneous ISA's prevent the use of standard atomic operations and require adherence to different ABI's. Early programming models abstracted all resources. This allowed programmers to express thread level parallelism without regard to specific processors, interconnects or memory implementations. The runtime system was free to map any thread to any processor transparently to the application designer. Chip heterogeneous multiprocessors with different numbers and types of programmable processors and custom accelerators have broken this clean model. Emerging heterogeneous programming models require designers to be aware of and program for specific processor types, memory hierarchies and develop and decompose their source code for specific configurations. In other words, Programmers no longer have the convenience of a linear address space and must now explicitly marshall data between different memory tiers.

Thus a fundamental question is has the introduction of chip heterogeneous multiprocessors now eliminated our ability to write portable and scalable applications. This work investigates how a traditional unified machine model can be reinstated over chip heterogeneous multiprocessors based on extensible processor node. My approach is to first adopt and then minimally modify a traditional Pthreads compliant runtime system (based on Hthread [19]) to seamlessly run across FPGA based multiprocessors systems with differing numbers and types of programmable processors, as well static and dynamically reconfigurable accelerators.

As a simple example, processor specific atomic operations such as load_linked and store_conditional, and test_and_set are not compatible. Throughout the scalar processing and homogeneous manycore eras, operating systems and programming models abstracted all architecture details away from the programmer under a unified programming model. This was a fundamental capability that enabled portability and reuse; the very cornerstones of software productivity. The use of homogeneous resources then allowed the runtime system to perform dynamic load balancing. These new programming models still rely on traditional operating systems and Pthreads that still avoid and do not resolve heterogeneity. Developers must be aware of, and code to, specific numbers and types of processors. Control flow within require programmers to explicitly set up and control the execution of processor specific application code that will run on the heterogeneous slave processors. These models also requires programmers to explicitly marshal data between levels of the memory hierarchy from within their source code. These models do not resolve lower level heterogeneous ISA issues but instead avoid them by restricting the use of standard synchronization primitives such as mutexes between different processor types. In short, these types of heterogeneous programming models enforce a philosophical reverse away increasing abstraction and backwards towards promoting lower level machine details up into the application code. The optimal number of processors, static and dynamic accelerators, and combinations of these resources that should be used for an application cannot guaranteed to be known until runtime. Instead the optimal combinations and types of resources that should be used will vary during runtime. In some instances static mappings of the source code to static and dynamic accelerators or even multiple processors can result in poor utilization of resources, and poor overall performance. Further the source code would have to be statically profiled, configured and optimized for each different platform or composition of resources. To enable portability we have extended the notion of polymorphism to traditional functions and threads. The policy of a polymorphic thread create or function call remains consistent and independent of the numbers and types of resources. Machine specific information is encapsulated within the body of the function. This allows the higher level application program to be portable across any CHMP system. Resource scheduling decisions are built into each function that allows the operating system to select and schedule different combinations of implementation methods based on runtime information and available resources.

In sum, The complexity of designing and verifying systems with 100's of mixed types of processors within hierarchical memory structures and complex interconnect networks will soon be beyond the capabilities of most application developers and even digital hardware designers. The same application that has been has been partitioned into hardware and software components, profiled and tuned for maximum performance on a specific architecture will have to be re-tuned partitioned, profiled and optimized if the configuration of the underlying CHMP architecture changes. This is counter to the software worlds desire to write once, run anywhere.

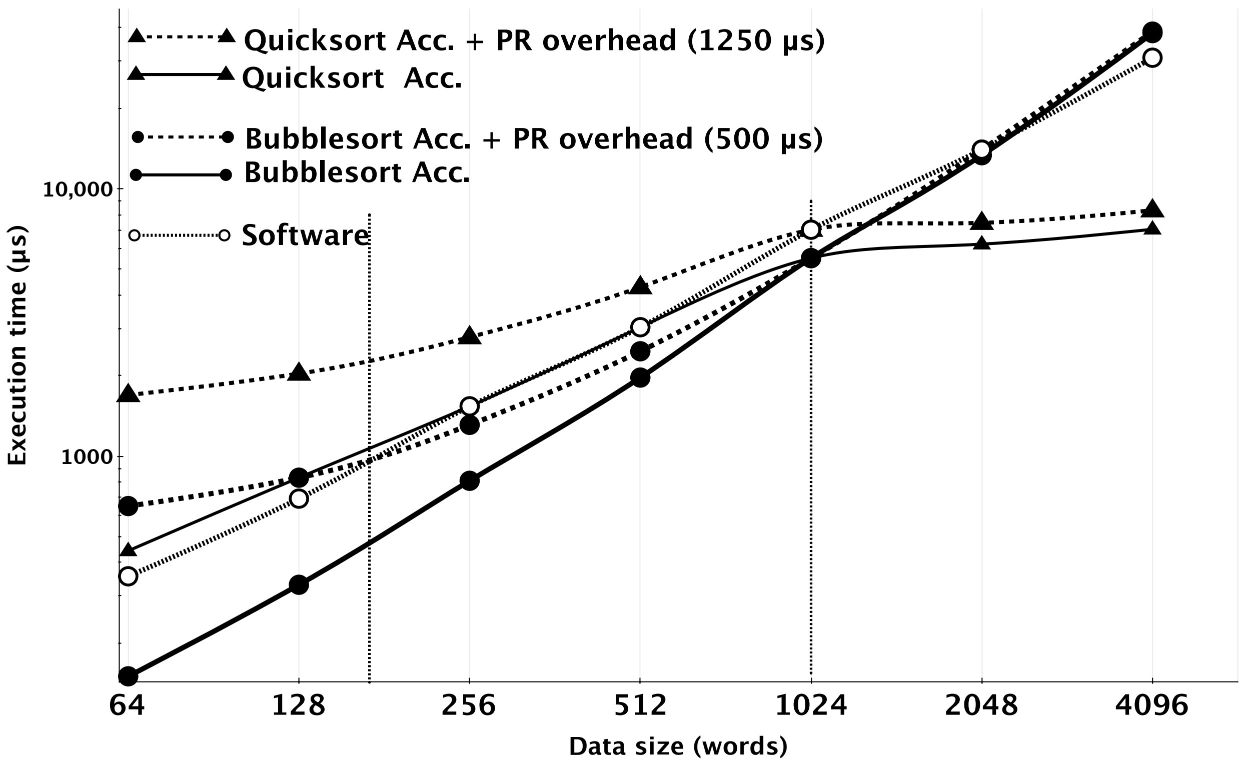
2.6 Example

The increasing flexibility offered by future generations of CHMPs architectures and the lack of a seamless and portable programming model will continue to negatively effect development time and cost, and developer productivity. Applications for earlier systems with homogeneous processors could be written, profiled, and optimized once, and then remain portable as device technologies provided generational systems with additional processor resources. Imposing the need for learning and mixing processor specific coding styles and specifics of each architecture will require the application be rewritten, profiled, and optimized each time a new generational offering is provided, or when a new component is added to an existing system. Designer effort may require multiple iterations of tedious by hand hardware/software partitioning's as the hardware resources change. As the number of heterogeneous resources that can be placed on a chip increases the search space that must be investigated for seeking an optimal partitioning of the application also increases. Even after extensive design space exploration, profiling, and benchmarking there is still no guarantee that the runtime behavior of the fielded system would be optimal, or match the desired behaviors observed on the bench. Just because a custom accelerator is available does not imply that it's use will always provide optimal performance. When to use an accelerator depends on the communication to computation ratio for the accelerator. In cases it might take more time to transfer data, setup and control the accelerator than simply executing the application in software. The decision to use or not use an accelerator will change dynamically during runtime. Consider an accelerator created to support a simple sort function. At times the function may be called on short lists of data that could be processed quicker in software on a single processor, or as a series of software threads. At other times the same sort function may called to process larger data sets where the overhead of data transfer and control would be amortized through pipelining or overlapping DMA transfers with the operation of the accelerator. Even this decision can vary based on the presence or absence of DMA engines and memory transfer times. Under partial reconfiguration the decision when to use the accelerator must also account for the additional overhead of transferring the bitstream if the accelerator has not previously been loaded.

Once optimized for a target CHMP architecture any change would require a rework of the application. runtime behaviors observed on the bench for concurrent applications in general can vary significantly from the deployed system. Asynchronous concurrency and conditional code introduce probabilistic timing behaviors. Probabilistic behaviors are implicit within our asynchronous concurrency models, and appear when communications patterns vary across shared interconnects. When the application is statically bound to specific resources the runtime system cannot adjust for these probabilistic behaviors.

Optimally partitioning an application into concurrent threads or tasks is one dimension of a design search space, determining under what conditions an existing accelerator should be invoked is a second, and determining if a portion of the FPGA should be reconfigured using partial reconfiguration is yet a third. Clearly the dimensionality of the search space grows as the heterogeneity and size of our CHMPs systems grow. Consider a simple case of sorting an array in a system consisting of general purpose processors (MicroBlaze [4]), and hardware accelerators . We have two different accelerator for sorting the data, the bubble sort(342 LUTs, 195 FFs with 500 us PR overhead), which is faster for smaller data sizes and the Quicksort(900 LUTs, 372 FFs with 1250 us PR overhead) which is faster for bigger data sizes. Let us assume in a CHMPS system, a host processor spawns threads to sort data on slave processors. Each slave might have either no accelerator, or a static or a partially reconfigurable sort accelerator (either Bubblesort or Quicksort Accelerator) attached to it. Figure 5.15 shows the results of sorting data in hardware (Bubblesort and Quicksort accelerators) and software (MicroBlaze running Quicksort algorithm). Also shown are the sorting times in cases where partial reconfiguration (PR) is necessary. As it can be seen in this figure, that the optimal decision of how to perform the sorting is dependent on the system configuration at the moment as well as the data size of the array. For example, if the size of the array is larger than 1k, then the Quicksort accelerator is faster. If it is less than 1k, then the Bubblesort accelerator is faster. However, if the size of the array is less then 256, sorting the data in software is faster than in Bubblesort hardware accelerator if partial reconfiguration overhead were to be included. Traversing this multidimensional search space continues to become more challenging as the heterogeneity and size of a CHMP's system grows.

Fig. 2.4 shows the execution time for adding two Vectors. Top: Thread level parallelism, in which the whole task is divided between 1-6 slave processors. Buttom: Data level parallelism in which the slave processor can either add the vectors in SW or using its vector_add HW accelerator (With/with partial reconfiguration overhead). This shows how complex the design exploration would be, consider the effort required to write and profile a simple vector addition operation on a system with six general purpose processors. A vector function can be written that divides work equally across processors for any vector length. Figure 2.6 on right, shows execution times taken on a simple SMP system built within a Xilinx Virtex 6 (ML605). These real results show the optimal number of processors that should be used varies based on the length of the vector. For a vector



t

Figure 2.3: Performance of different implemnation of Sort in HW and SW

length of 240 the optimal partitioning should be 3 and not 6 processors, and for vectors of lengths 960, 1920, and 3840, 5 processors. To know this, the programmer would have had to explore the design space through repetitive profiling runs and embed complex decision making code for this one particular system. This would not be intuitive for the programmer. To know this, the programmer would have had to explore the design space through repetitive profiling runs and then embed complex decision making code for this one particular system. The design space exploration and profiling would have to be repeated if the architecture changed. Finding the optimal number of processors will be affected by additional factors including specific processor types and bus organizations. Even an exhaustive profiling could not account for the probabilistic behaviors that do occur during run time. Probabilistic behaviors arise due to many factors; from dynamically changing bus traffic patterns to just the non-repeatable nature of the asynchronous concurrency programming model itself. Finding the optimal numbers of processors would be impossible for systems with 100's of processors.

And the final example, shows that attempting to hand profile and optimize code becomes more

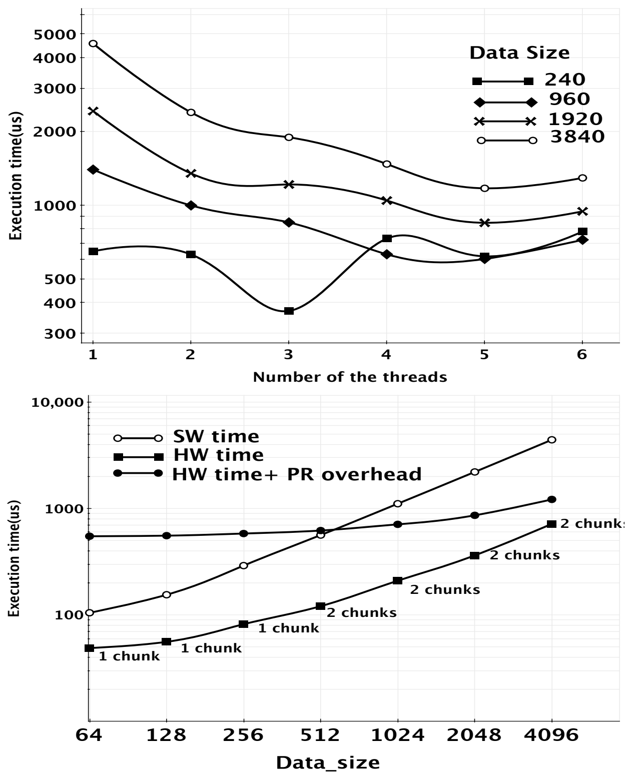


Figure 2.4: Data level parallelism Vs. Thread level parallelism

difficult when systems contain different numbers and types of accelerators. If an accelerator were present the programmer would be apt to always use it. Figure 5.14 on left shows the execution time for the vector operation on three systems; one with a processor, one with a processor plus static vector accelerator, and one with a processor plus a dynamically reconfigured vector accelerator. Clearly always invoking the accelerator if under partial reconfiguration rules would not give the best performance. The timing results also show that if the system possessed *self awareness* and learned that the bitstream had previously been loaded then the decision to use the accelerator changes. The "chunks" labels refer to the optimal block size of DMA transfers for pipelining data into this particular accelerator. The optimal block size does change and would require additional code to set up the DMA transfers based on the length of the vector. Optimal block sizes would naturally change for different accelerators as well as systems. The need for these types of trade-off analyses is well understood. However the knowledge of the need does not make the process of experimentally finding such types of crossover points through hand profiling any less difficult. The decision when to invoke this or any other partially reconfigurable accelerator will vary depending on the size of the data sets, bitstreams, and computation time. Finding the crossover point would have to be determined for every partially reconfigurable accelerator under many combinations of parameters on every system. Even if a system has partial reconfiguration capabilities it would be better to compute the vector operation in software for data set sizes less than 512 elements. When the data set size is greater than 512 performance can still be enhanced even when including the overhead of bitstream loading.

2.7 Need for new programming model and runtime tuning

As motivated by previous examples, we aim to create a special library calls that can be used inside the body of threads created by the user. The extensible processor provides the necessary infrastructure for these library calls to run. We call these library calls *polymorphic functions* ,as the actual execution of them is transparent to the user and will be determined during runtime. Basically, the idea is the programmer just calls SORT() or VECTORADD() in the body of the thread, and leaves the rest the runtime system. First of all, The runtime system tries to find a slave whose current accelerator matches to the first *polymorphic function* call in the body of the thread. Second, the slaves autonomously decide how to run *polymorphic function* calls in the thread assigned to them. We predict that polymorphism enables portability, while also allowing the run time system to transparently deliver increased performance as the complexity of the design space increases. We believe this programming model hold promise for programmers facing future CHMPs systems with 100's of heterogeneous resources.

I believe that extensible processor provides the necessary infrastructure to include self-awareness into a run time system, which in turn can effectively navigate this multidimensional search space and be used in place of hand profiling. In this theis work we try to show how building intelligence and learning capabilities into a run time system can effectively navigate a multidimensional search space and be used in place of hand profiling. The run time system can perform the partitioning and scheduling of an application across what it sees as available processors and invokes accelerators based on run time variables and learned system configurations. Any runtime system cannot decompose an application or migrate functions between heterogeneous resources if the programmer has statically bound particular sections of the source code to specific resources. To resolve this a co-ordination layer is needed where all resource specific information and methods can be encapsulated as linkable library routines. These routines can be invoked by the user through *polymorphic functions*. Our co-ordination layer is implemented using POSIX threads (Pthreads). I will have runtime results for standard and polymorphic calls running on a variety of experimental systems built with different numbers and combinations of processors, with/without static and dynamic accelerators. I think that the use of polymorphism enables portability, while also allowing the runtime system to transparently deliver increased performance as the complexity of the design space increases. Run time system can map portable polymorphic functions to allow the application to take better advantage of each systems set of available resources compared to by hand methods.

2.8 Contribution of this work

This thesis is aiming to make the following contributions to achieve portability of the applications across different HEMPS systems:

1. Extensible processor: Introducing a new way to interface with accelerators in a heterogeneous system using extensible processors which provide both fine-grained HW/SW partitioning of different portions of the code as well as Stack management and other services for accelerators.

2. Transparent Partial Reconfiguration: The runtime system allows all slave processors to autonomously make scheduling decisions and transfer bitstreams for partially reconfiguring local accelerators. The decision as to when and how to use a dynamic accelerator is based on run time profiling data and performed transparently by the runtime system. Our model seamlessly supports data level parallelism within static as well as dynamic accelerators using partial reconfiguration. We abstract the decision to use, the setup and transfer of bitstreams, and control of dynamic accelerators within the runtime system.

3. Automated generation of Heterogeneous Extensible Multiprocessor Systems: including extensible processor nodes which have the capability to autonomously and efficiently run portable applications. The application is portable on all different HEMPS systems, regardless of numbers of slave processors with dynamic or static Accelerators. The code is not only portable across different platforms, but also it will take advantage of any possible thread level parallelism and data level parallelism to achieve the best performance by using available accelerators, partially reconfigurable regions and slave processors.

4. Portable programming model: Extends the Pthreads programming model to support heterogeneous thread and data level parallelism. Polymorphic functions are introduced that are configured during runtime across heterogeneous resources such as standard processors, vector processors, and static and partially reconfigurable accelerators. These polymorphic functions provide transparent partial reconfigurations, fast DMA transfers and pipelining of data transfers with accelerator computation. The extended model still allows unaltered legacy applications to be run seamlessly across systems with different mixes of heterogeneous resources. This programming model unifies support for thread and data level parallelism allowing unaltered applications to be run seamlessly across systems with different mixes of heterogeneous resources. Our model seamlessly supports data level parallelism within static as well as dynamic accelerators using partial reconfiguration.

5. Polymorphic Functions: Functions that are both portable and can be dynamically tuned for any combination of software and hardware accelerators. The library functions tune the transfer of data sets into variable size data sets that can be DMA'ed into buffers and local scratch pad memories based on the applications dynamic parameters. Tuning considers the overhead of transferring dynamic accelerator bitstreams within a software/hardware partitioning. A library of polymorphic functions for computationally intensive parts of the application is provided for the user. This makes the code portable across any HEMPS systems.

6. Runtime tuning: A runtime system that supports our portable programming model to run

more effectively as the complexity of the system grows. It performs resource-aware scheduling and dynamic profiling of the application, and learns about the applications resource requirements to increase overall system performance compared to by hand static mappings. It also increases the utilization of all heterogeneous resources to provide a more efficient mapping of the application.

Chapter 3

Background

Field Programmable Gate Arrays (FPGAs) have long held the promise of delivering performance levels close to those associated with custom designed Application Specific Integrated Circuits (ASICs) but with productivity levels more closely associated with developing software. While the literature validates the performance side of the argument, the ability to deliver these performance levels with the ease of developing software remains an open area of research. One approach to resolving the general issues of designer productivity and accessibility for software programmers is to abstract the FPGA under a familiar modern parallel programming model and operating system. For example ReconOS [45] and hthreads [18] implemented a programming model based on Pthreads to abstract the hardware/software boundary.

The introduction of Platform FPGAs circa 2003 provided single chip FPGA components that contained diffused IP such as multipliers, BRAMs, and hard processor cores to compete in the growing system on chip market. These CHMP systems allow system designers to mix hard processors, soft processors, 3rd Party IP, or custom hardware cores all within a single FPGA. However achieving interkernel communication without sacrificing performance remained an issue. Software developed for these platform FPGAs sought to abstract both generalized software running on a processor and specialized hardware accelerators as threads under the visibility and control of a generalized run time system [49]. Under the multithreaded programming model any thread, including a custom hardware accelerator could be scheduled by the run time system, and synchronize and share data with all other threads. This thesis is addressing a significant advancement in resolving the contradictory forces of generalization and specialization within FPGAs.

The same thing is true in the realm of Real-Time and Embedded Control Systems (RTECS). Designers of RTECS are continually challenged to provide new system capabilities that can meet the expanding requirements and increased computational needs of each new proposed system, but at a decreasing price/performance ratio. RTEC designers continually struggle to balance these opposing forces of generalization and specialization. At one end of the spectrum CHMPs systems are already being created with standard programmable processors. These systems support the use of standard system software to provide a fairly general set of capabilities to support generalized use across widening ranges of applications, At the other end of the spectrum CHMPs systems support the historic role of FPGA's allowing designers to create specialized standalone custom accelerators controlled by an external master processor. Creating systems with combinations of programmable processors and specialized components bring the best of both worlds to RTEC designers.

Creating CHMPs systems with programmable processors bring designers generalization, whereas including custom components such as accelerators or tailoring the numbers and types of processors can bring specialization. Recently emerging CPU/FPGA hybrid chips are becoming extremely important components contributing to the creation of a family of Counter off the shelfs(COTS) hardware platforms for future RTEC systems. Creating such a capability for RTEC applications is a difficult challenge in part, because it requires the simultaneous satisfaction of apparently contradictory forces: generalization and specialization. FPGA's have always offered designers the advantage of specialization. Within CHMPs systems the numbers and types of processors as well as accelerators can be tailored to best match an applications set of unique and challenging requirements. If density increases continue to follow Moore's law we can realistically expect single chip FPGA's to comfortably host CHMP systems with up to 100 compute elements within a decade.

Multiple avenues of research have evolved to enable programmers and application designers to more easily adopt these powerful FPGA chips into a wide range of applications. The potential performance benefits brought by increased densities do come at the cost of increased design complexities. Configuring millions of gates into a multiprocessor architecture is significantly more complex than forming state machines and digital circuits within a few thousand gates. Research in tool flows are seeking to handle complexity through abstraction. Work such as architecture templates and hthreads in the cloud [21] have been investigating higher architecture level abstractions and automation capabilities to ease the design effort of creating a complete chip heterogeneous multiprocessor system. Work in programming models and runtime systems have been investigating how to support access to the gates through familiar programming abstractions instead of low level digital design and circuit synthesis techniques. The ability to host complete multiprocessor systems with scalable numbers of compute engines is also motivating new research in resource scheduling. New work is emerging in self aware scheduling that seeks to avoid creating Dark Silicon, which put an end to manycore scaling and is responsible for the shift to heterogeneous chip multiprocessors in the general computing domain [28]. Advancements have also been occurring along the more historical C to gates, custom hardware synthesis path allowing C like languages to drive the generation of custom gates [70, 50, 11].

The multithreaded programming model has proven effective for abstracting hardware and software computations running on an FPGA configured as a multiprocessor system on programmable chip [49, 63, 45, 39, 65, 57, 22]. The model allows hardware accelerators to be treated as detached threads that can synchronize and share data with all other threads. The importance of the model is also increasing with interest in porting newly evolving heterogeneous computation models such as OpenCL [6] over FPGAs. OpenCL implementations rely on standard multithreaded programming middleware packages such as pthreads to provide concurrency support.

Work in the general purpose computing communities for heterogeneous programming models and the supporting frameworks are all pointing towards new approaches to bring portability and scalability for systems with combinations of CPUs and GPUs [38, 13, 6, 48, 27]. In the reconfigurable communities, the better PR support for adaptable applications has led to a growing interest in self-aware and adaptive operating systems for FPGA-based heterogeneous multiprocessor systems. Projects such as SPREAD [65], Configuration Access Port OS (CAP-OS) [32], and ReconOS [33] typify approaches that seek to adaptively migrate threads (or tasks) created from a user's application onto available combinations of general purpose processors and specialized hardware. For example, during runtime the CAP-OS dynamically instantiates new hardware with application demand. Similarly to this work, both SPREAD and ReconOS unify software and hardware threads under a multithreaded programming model. This enables the OS to adaptively *switch* from software to hardware threads during runtime.

However, Much of these works perform all decisions and resource management services on a centralized (master) processor. Tasks or threads executing onto specialized hardware or other slave processors request OS services through remote procedural calls (RPC) to wrapper or delegate software threads running on the master. For the work presented here, scheduling decisions between hardware and software computation are localized to a lightweight extension of the OS running on each slave processor. This enables greater autonomy to be utilized and greater concurrency to be achieved, but most importantly it leads to more scalable designs. By augmenting reconfigurable regions to general purpose processors, the runtime system in this work provide greater application adaptability during runtime.

Self-adaptive systems also may require the load-balancing of software and hardware tasks in order to achieve an optimal solution for the given application. Some of these works, as in [65, 32, 53, 37], compute the hardware-software partitioning schemes partially offline during compile time. Many do not account for variable parameters that occur during runtime (e.g. resource contention) but simply perform hardware/software partitioning on estimated execution times. In contrast, the work proposed here utilizes an online profiling and partitioning scheme, where information is learned about previous execution history and used for more appropriate hardware/software partitioning. In this way, variable runtime parameters that may affect this profiling information can be used to more effectively partition the application.

3.1 Modeling accelerators in a CHMP system

FPGAs have a rich history in serving as application accelerators. Early work such as PRISM [66] pioneered the use of an FPGA based accelerator that was loosely coupled from the program control flow executing on a external CPU. On-chip accelerators can be classified into two classes: 1)tightly coupled accelerators where the accelerator is attached to a particular core and can only be accessed by that core. 2) loosely coupled accelerators, which the accelerator is an independent entity which can be shared and accessed among multiple cores [24]. In contrast to the loosely coupled approach, projects like DISC [68] as well as GARP [20] pursued modeling an FPGA based accelerator as being tightly coupled within a processors ISA. Pursuit of this approach largely ebbed for a decade within the reconfigurable computing community. However Tensilica continued to evolve this model as an extensible processor within the general purpose computing community. Tensilica showed the benefits of exploiting fused and VLIW (Very Long Instruction Word) instructions, as well as Single Instruction Multiple Data (SIMD) fine grained parallelism as extensions of a processors ISA. This approach is once again being pursued by work such as VENICE [73] or VESPA [72] that provides vector extensions to a RISC processor ISA. The advantage of tightly coupling an accelerator within a processors ISA is the ability to share register data and allow the general purpose processor to implement sequential program control. The disadvantage of this approach is the accelerator is not visible by the operating system as a schedulable and sharable resource.

In general, FPGAs have historically appealed to those wanting to integrate custom hardware into systems to boost performance. Heterogeneous multiprocessor systems employing a master-slave thread programming model can exploit thread-level parallelism (TLP) onto coarse-grained compute units such as heterogeneous processors and custom hardware. Extending the programming model for software application developers to abstract custom HW requires augmenting hardware abstraction layers (HAL) for custom HW. HAL allows the operating system (OS) to transparently assign threads to either processors(*software threads*) or custom HW (*hardware threads*).

Hardware Abstraction Layers (HALs) were created to provide state machine descriptions of system service libraries accessible by accelerators functioning as hardware threads [17]. HAL's are provided as wrappers that contain finite state machine circuit equivalents of software system services for accelerators. The system services within the HAL allow accelerators to invoke operating system services and perform memory operations. Two approaches have evolved to determine the types of system services that needed to be designed into a HAL. The first seeks to minimize the latency of system services for hard real time systems through hardware/software (HW/SW) co-design. The HAL is defined to replace software system service libraries with local and low latency hardware mechanisms [17, 15]. This approach achieves low latencies but at the cost of inefficient updates as the HAL has to be re-synthesized when the hardware system service libraries are modified.

The second approach seeks a better ease of update but at the cost of increased access latencies. Operating system services are provided within a single monolithic kernel, typically a variant of Linux, running on a centralized master processor [63, 45, 39, 65]. The HAL provides Remote Procedure Call (RPC) type mechanisms on behalf of the accelerator to issue requests for the software services running on the master node. The cost of this increased ease of update is the additional latencies of the RPC calls. Centralizing services on a single master node can also limit the approach's ability to scale. Most approaches to runtime adaptivity restrict themselves to a coarse-grained task level, and all OS requests are centrally served on a host processor [65, 32, 55]. While this simplifies load balancing, remote procedural calls (RPC) do not present a scalable solution. Enabling autonomous heterogeneous resources to make autonomous decisions combats this centralized scheme.

Overall, FSM-based HALs allow communication between the hardware threads running on custom HW accelerators and the OS. HAL enables a hardware thread to access system services. Also, it enables a hardware thread to be integrated into the multiprocessor system as a detached stand alone thread. The hardware based HAL allowed the loosely coupled accelerators to operate autonomously and invoke Pthreads equivalent system calls. This does not present a good solution for two reasons. First, further changes to the HAL requires redesign and recompilation for the targeted FPGA. This leads to decreased designer productivity. Additionally, area consumption is increased due to design complexity. Also supporting features such as recursion and memory allocation can be expensive and inefficient in hardware. Second, custom HW interfaces can not provide fine-grained HW/SW partitioning. Due to Amdahl's law, there always exists a percentage of a thread that is sequential and may not run best on hardware that is *tuned* for the parallel sections of the thread. therefor assigning the entire thread onto custom hardware may not provide efficient runtime performance. HALs support the synthesis of the complete thread including all sequential portions, as stand alone hardware accelerators. The size of the HAL along with the need to synthesize sequential portions of the application result in poor utilization of transistors.

Ideally, the functionality of a HAL needs to provide distributed low latency services that scale with the number of compute resources such as those provided by [15] but with updates through compilation and not synthesis as achieved by [63, 45, 39, 65]. This work shows how SW-based HALs running on a extensible processor achieves this goal.

Early modeling approaches abstracted an accelerator as a hardware resident thread that accessed key operating system services such as mutex operations, as well as providing access to a linear address space, through a series of hardware based finite state machines encapsulated within a virtual abstraction layer [39, 14, 45]. This allowed a complete thread body to be implemented as a hardware accelerator and seamlessly interact with all other software and hardware resident threads throughout the system. This model proves inefficient in resource requirements, restrictive in supporting different accelerator models and for dynamic scheduling and resource allocation. First, any change to the operating system services required a redesign and resynthesis of the hardware resident virtual abstraction layer. Further the size of the virtual abstraction layer quickly grows greater than the size of a standard processor such as our Microblaze. Additionally resources are required to implement the complete, and not just the computationally intensive portions of the thread yielding large accelerators. The model restricts the use of the accelerator to thread level parallelism. The virtual abstraction supports API's issued from the hardware thread resulting in the thread operating as a master and the virtual abstraction layer as a slave to the accelerator.

3.2 Chip Heterogeneous Multiprocessor system platforms

The unveiling of a new class of devices called Platform FPGAs, circa 2000, were introduced to compete in the growing systems-on-chip (SoC) market. Much bigger than their predecessors, they included diffused blocks of hardware, including blocks of static RAM (SRAM) memory, multipliers, and programmable processors. The programmable processor, either diffused within the substrate or loaded as a soft IP core, could now host an operating system (OS) within the chip. Research such as Hthreads [19], ReconOS [45], and FUSE [39] showed how the multithreaded programming model could be used on Platform FPGAs to abstract custom accelerators as hardware resident threads running within a multiprocessor system on chip for real time and embedded systems. In one sense, multiprocessor operating systems were being created, but with the OS providing abstraction and concurrency support for accelerators in place of standard programmable processors.

The general purpose computing community has already transitioned away from homogeneous manycores to heterogeneous manycores to address the issues of dark silicon. CHMPs architectures are generating interest within the general purpose computing community to address issues of ”dark silicon” [28, 42]. They can Provide a broader component base that can exploit multiple levels of parallelism can increase the utilization of transistor resources [35]. They also include more focused data processing components such as vector processors and custom accelerators which leads to a better energy efficiency and can address power issues [69, 44]. Advancements in FPGA semiconductor fabrication are yielding components that can host complete complete CHMPs. An FPGA offers additional advantages of customizing the mix of processor types, including custom accelerators, and supporting dynamic reconfiguration. Researchers have been investigating new methods for constructing CHMPS architectures, programming models and run time systems, and the use of such heterogeneous components to enable systems to self configure and operate more autonomously.

As FPGAs have grown is size and speed, providing software application developers with a general and flexible HW platform consisting of processors and accelerators have been a matter of interest. The idea is to spare both SW and HW developers from dealing with low level details of a CHMP system like bus interconnect, memory communications,etc. Ideally, the developers should provide their HDL code for accelerators, and leave the rest to automation. For example, Redsharc [41] provides A HW/SW platform for HW/SW kernels which communicate via streams or blocks. Its goal is to Simplify development on FPGAs for stream based computing applications. Other works like SPREAD [65] or FUSE [39], provide user with a flexible HW platform which is capable of running switchable HW/SW threads.

The design of a reusable CHMP base system is becoming more efficient through the availability of IP system level components and evolving architecture level design automation capabilities [21]. A growing selection of system level IP components are being offered by vendors as well as community efforts. Available vendor specific system level IP components include programmable processors, standard buses, and system I/O components. The community has been providing additional IP components to augment the vendor offerings. Additional system IP components such as vector and extensible processors, robust interconnects, and tuned accelerators are freely available through repositories maintained by community efforts such as OpenCores [36] as well as research institutions [9]. The base system components can be incrementally tuned or augmented with additional accelerators to meet each new applications set of uniques requirements at a fraction of the cost required to design each new system from scratch. Advancements are occurring to automate the assembly of the system IP components into a base CHMP architecture. This type of automated assembly can eliminate the time consuming process of integrating, testing, and debugging components by hand within a vendor specific CAD tool. Access to these tools is being made easier through cloud delivery [21] in Hthread project. This works tries to extend Arcgen project in Hthread to include custom hardware and partial reconfiguration, as well as compile flow for the application code to run on these complex system. This type of automation allows the creation of complex heterogeneous chip multiprocessors by software designers and significantly reduces the time and complexity of creating complete Systems on chip (SoC) systems. Additionally advances in high level synthesis have occurred that allow system designers to analyze and synthesize critical sections of code into efficient custom accelerators [11].

3.3 Programming models in a CHMP system

In a CHMP system, Heterogeneity has two aspects: Heterogeneous processors, and accelerators. Historically, Within standard compilation flows, the application and operating system are treated separately. The application code is compiled for a particular processor ISA without knowledge of the overall system architecture. Operating system code is linked after compilation to provide the policies and mechanisms of the higher level programming model. As code is compiled for a particular ISA and the operating system cannot abstract over ABI differences, programmers must develop and tune their applications as non-portable code. Programming models for heterogeneous systems such as Cell [51] and Exochi [64] as well as the currently popular OpenCL [6] and OpenMP [47] enforce this non-portable, architecture specific view up into the code of the application developer. Architecture specific coding requirements manifest in requiring designers to understand hierarchical memory organizations and explicitly marshal data between levels of the hierarchy. Designers must also be aware of which specific processor resources are available and code directly to those resources. This is contrasted to the the classic programmer abstraction of a linear address space and ubiquity of processor resources.

When it comes to including accelerators into a programming model in CHMP system, the multithreaded programming model has gained in popularity as a unifying framework to abstract hardware accelerators in hybrid FPGA/CPUs [49, 63, 45, 19, 39, 65, 41]. These first efforts included the definition of a HW-based HAL to extend OS services to the loosely coupled hardware accelerators. Additional research has continued to extend the multithreaded programming model [45, 19, 39, 65, 41]. Two broad approaches have been followed to integrate hardware accelerators as threads into this programming model. One popular approach is to use a full Linux stack running on a master processor and have the hardware accelerator threads request services through remote procedure calls (RPCs) [63, 45, 39, 65]. The second approach follows more of a microkernel design that distributes operating system services into independent concurrent components. This approach is typified by the hthreads [19] system that migrated synchronization primitives, thread management, and scheduling into independent hardware cores. This approach also distributes more operating system services into each HAL to break the bottleneck that can occur using a full Linux stack on a master processor [15]. In sum, works such as FUSE [39] and ReconOS [45] focused on how to extend Linux over hardware-software components by adopting a standard Linux kernel running on a host processor that provides operating systems through RPC mechanisms to threads running on slave processors. The hthreads hardware microkernel addressed incompatibility issues that arise due to ABI differences between processors [19].

My thesis focuses on restoring the idealistic notion that code should be written once by the application developer and remain portable across any combination of heterogeneous resources. Our approach is to view all heterogeneous resources as runtime schedulable components under the uniform multithreaded programming model. This is in direct contrast to evolving programming models for heterogeneous manycores such as OpenCL [6] that enforce static partitioning of each application across unique sets of heterogeneous resources, and raise platform specific details of the memory hierarchy up into the source code. These models eliminate portability, and impose high levels of effort onto to developers to customize, profile and optimize each application for every combination of heterogeneous resources. These models further prevent the operating system to view a systems resources as a pool of dynamically available heterogeneous computing units .

Hthreads and ReconOS allow hardware and software threads to be scheduled and synchronize under a unified programming model. These approaches provided performance increases through the use of standard programming models that allowed the parallelism within the application to increase in relation to the scalable numbers of processors that could be mapped into each new generation of FPGA's. Work such as [63, 61, 19] have explored approaches to abstract the differences between threads running in hardware and software.

Early work on software based high level portable programming models for FPGA's focused on abstracting the hardware-software boundary. Work such as [63, 19, 45, 65] are representative of popular approaches that abstract hardware accelerators as threads that can synchronize and communicate using typical operating system services. This model allows hardware and software based threads to synchronize and share data under the Pthreads multithreaded programming model. Within the general purpose computing community OpenCL [6] has evolved as the standard programming model for heterogeneous manycores. Researchers [52] as well as chip vendors [60] are investigating how to bring FPGA based accelerators under the OpenCL umbrella. OpenCL forces static partitioning of the application across unique sets of heterogeneous resources, and does not allow applications to synchronize across heterogeneous compute resources. Platform specific details of the memory hierarchy are also raised up into the source code. The OpenCL model thus eliminates portability across heterogeneous compute components, and imposes additional design effort for developers to customize, profile and optimize each application for every combination of heterogeneous resources.

Another aspect of FPGAs that effect programming model is the option of partial reconfiguration. There has long been historical interest in partial reconfiguration [55] and its use in systems that can dynamically adopt regions based on online task requirements [59]. Partial reconfiguration is a service provided by the OS and like other OS services, commonly provided on a centralized processor. Bitstream loading is similarly controlled by the OS on the centralized processor. Both [65] and [32] followed this approach incorporating runtime adaptivity through seamless hardware/software task switching. Research efforts break user applications into individual tasks where additional high-level information such as control/data dependencies, deadlines, estimated execution times, etc. are used to map them onto the reconfigurable fabric [55, 32]. There, decisions about reconfiguring or re-using current logic (processors, accelerators, both) on the dynamically reconfigurable region can be exploited.

Many approaches have taken on the task of supporting run time adaptivity through dynamic hardware reconfiguration. Many recognize that in order to get best performance per unit of energy, software applications (or some part of it such as individual tasks in embedded or real time systems) needs to run on hardware that is *tuned* for such tasks. They also acknowledge that in order to build a computing system capable of supporting any task thrown at it, they must battle flexibility with performance with general purpose processors capable of executing same software applications. The introduction of dynamic reconfiguration has allowed this group of researches to *adapt* both practices in order to combine both flexibility and performance under a single computer architecture.

In sum, the challenge of fielding such systems now lies in developing appropriate higher level programming model abstractions and runtime scheduling capabilities. The higher level programming model must balance generality with customization, providing a generalized set of abstractions that allow programmers to express scalable heterogeneous computations. The runtime system then needs to efficiently map the heterogeneous computations across available resources, including general purpose processors, extensible processors and partially reconfigurable accelerators.

3.4 Heterogeneous Run Time Systems

Self-adaptable runtime systems on a reconfigurable platform has traditionally been limited to the scope of real-time tasks with hard/soft deadlines [59], and streaming applications such as sensor networks and data collection systems. This work is focused on providing an adaptable runtime system for undefined software applications intended to run on a heterogeneous SoC platform. Therefore, the run time should allow software application portability through generality over an already dynamic platform. Our work attempts to not favor a particular application domain, allowing the user to write at a more flexible and granular level (threads). A higher level abstraction is achieved through this allowing the system to easily adapt to any heterogeneous system.

Heterogeneous systems are challenging our existing run time systems, including binary incompatibilities and changes in how the scheduler cannot move threads across resources that are brought on by heterogeneous executables.Run time systems within homogeneous systems were designed to interface with a single type of architecture. Resources of these systems interacted with and executed on the system all in a similar fashion. Therefore, any resource management and task scheduling only had to resolve for one particular architecture. In consequence, run time flexibility of enhancing the system's performance further is limited due to similar resources. Introducing a mix of heterogeneous resources such as processors of different ISAs and ABI's can add further improvement in performance and power over homogeneous systems, as certain tasks can be optimized for particular types of architectures. However, run time systems accounting for such heterogeneous systems also are burden with juggling across interfacing and translating across many architectures. This is a result of a system where heterogeneous resources can no longer communicate directly amongst themselves. Hence, the run time system must account for this inability.

When designing a runtime system to include coordination of task/thread scheduling and management, MPSoPCs have traditionally considered resource availability and task priority alone. Designing MPSoPCs atop of a reconfigurable fabric introduces additional considerations that should be made in order for near optimal system efficiency. With a system that is reconfigurable during runtime, the run time system can take advantage of alternative task mapping/scheduling and resource allocation. The run time can examine where best to schedule a task as opposed to naively finding a first available resource(s). This is particularly beneficial for similar tasks scheduled in the future to reuse these resources that may have been previously reconfigured (and tuned) for such tasks. In contrast, resource allocation allows the run time system to introduce new logic into the system during its runtime. Thus, allowing parts of the system to shape to a program's needs that best fits it.

3.5 Related Work

3.5.1 RedSharc

The Reconfigurable Data-Stream Hardware Software Architecture (Redsharc) [41] is a programming model designed to meet the performance needs of multi-core systems on a programmable chip (MCSoPC). Redsharc wraps HW accelerators in a thin HW kernel interface. The HW kernels and SW kernels communicate via Streams or blocks by custom designed Networks on chips. The Redsharc API and infrastructure makes it easy for fast kernel integration. Redsharc uses an abstract API to allow programmers to develop systems of simultaneous kernels, in software or hardware.

Redsharc is A HW/SW platform for HW/SW kernels which communicate via streams or blocks. Its goal is to Simplify development on FPGAs for stream based computing applications. Our works is different in that we create threads that can be run on either host processor or on extensible slave processors. In Redsharc, there are both software and hardware computational nodes which are called ”kernels”. Kernels are implemented as either software threads running on a processor, or hardware custom circuits running on the FPGA fabric. Regardless of their type, kernels communicate via the Redsharcs abstract API, which is based both on a streaming model to pass data and a block model to exchange index based data. This API have been implemented as a software library for software kernels and some VHDL components with generics for HW kernels.

Basically ,the goal of Redsharc is to implement as light-weight as possible. That is why they chose a streaming system over and MPI or Pthreads based system.

Their API knows two types of kernels, worker and control. There is only one control kernel in the system which is in charge of creating and managing streams, blocks, and worker kernels. However, multiple worker kernels can simultaneously exit and process the data elements presented in streams or blocks.What is important is that because both hardware and software kernels use the same API, the first stages of application development can be agnostic of where a a particular kernel is being implemented. The whole system can be viewed as kernels which communicate with each other as shown in Fig 3.1. Later, the kernels can be implemented either in hardware or software, depending on the which one is more suitable for this particular kernels task as it is shown in Fig 3.1.

The Hardware Kernel Interface (HWKI) is a thin wrapper that connects hardware kernels and it implements the Redsharc stream API. This leads to significant productivity, as it enables rapid kernel integration into new and existing systems. Through the use of HWKI, a kernel developer can focus on the design of the kernel instead of wasting his time on low level implementation details of how to access streams and blocks of data. As long as the HW developer sticks to FIFO and BRAM interfaces, the system designer can easily integrate the new kernels to the system. Fig 3.1 shows an implementation of a Redsharc system with one Microblaze processor and 8 BLAST HW kernels.

3.5.2 FUSE

In FUSE[39] the custom hardware circuits are integrated into a SoC as a memory mapped IO device peripheral. This is achieved via a customized HW interface abstracted away by a corresponding Loadable Kernel Module in the kernel space. Transparent to the user, *thread_create()* will either runs the entire thread in SW or HW based on resource availability. In other words, the operation system checks to see if there is a free accelerator for that function. If so, it runs it in HW. The overhead of loading/unloading the LKM as well as calling OS services to communicate with the

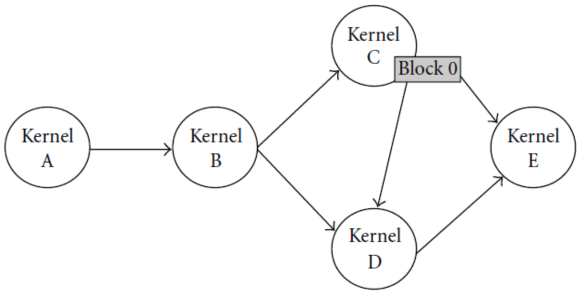
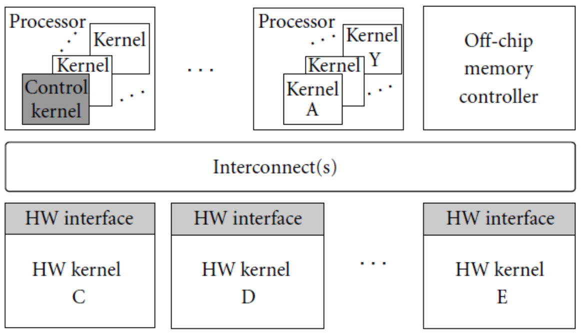


Figure 2: Logical view.



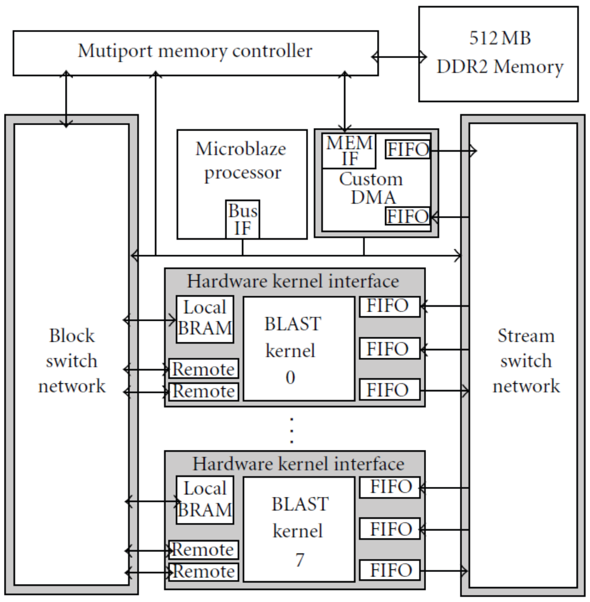


Figure 8: MicroBlaze BLAST system.

Figure 3.1: RedSharc System [41]

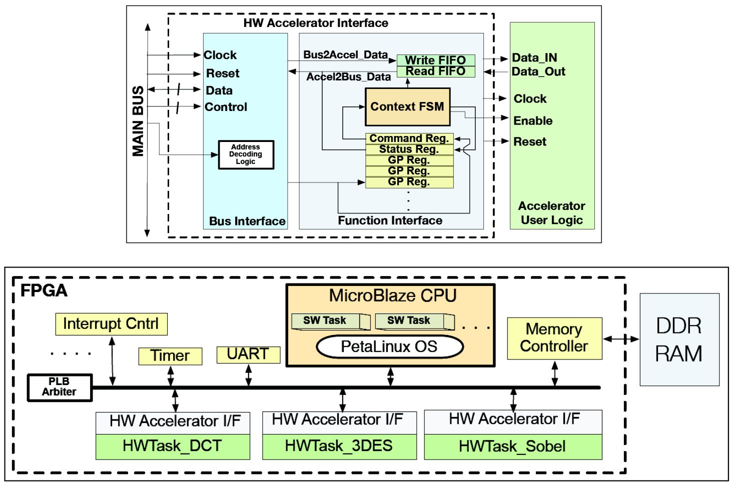


Figure 3.2: FUSE [39]

HW accelerators are among the drawbacks of this approach. Plus, there is no fine grained HW/SW dynamic partitioning. In other words, the entire thread is either running on SW or HW.

Figure 3.2 Left shows the HW interface in FUSE which is more aimed for streaming data. Figure 3.2 right, shows a typical implementation of a FUSE system with three HW accelerators, while the Microblaze CPU is running PetaLinux OS. Basically, FUSE provides an API for how accelerators interface with with and embedded Linux OS with POSIX threads running on a general purpose processor. Updates made to a HW accelerator's design results in changes to both its interfaces and its LKM, so any change to the HW accelerator requires re-synthesizing the HW accelerator interface on top of changes made in LKM.

3.5.3 SPREAD

SPREAD [65], introduces the idea of switchable HW/SW thread. A thread upon creation can run on either host processor or on any of the reconfigurable processing units, and some threads can migrate between HW or SW during their execution time. There is no smart tuning for when a switchable thread which is running on SW can be migrated to a free Reconfigurable programming unit(RPU). For example, the PR overhead might nullify the performance improvement resulting from running the rest of thread in HW. To avoid thrashing, they just had a simple *one time max*

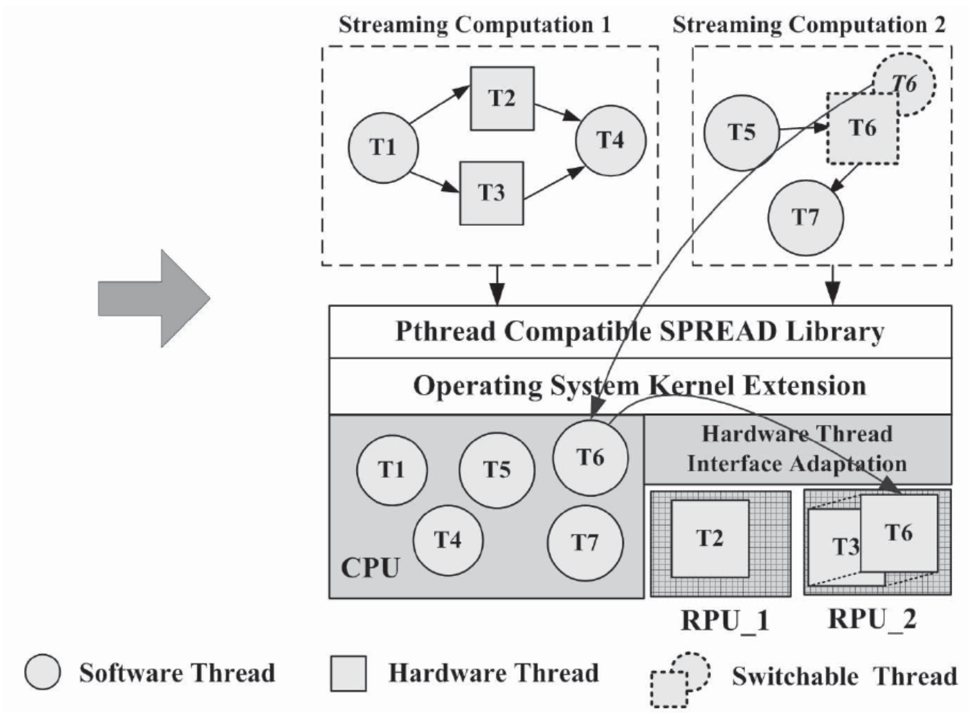


Figure 3.3: SPREAD [65]

rule as how many times a thread can be migrated between HW and SW.

There are two interesting ideas in SPREAD. First, A thread can be switched to HW during its runtime using stub thread. Whereas, in most approaches like FUSE or ReconOS this decision can only be made once at the start of the thread. Second, Non switchable HW threads can preempt the switchable ones based on priority to meet the real time constraints.

Basically, the operating system will transparently make following decisions when thread_create() is called in user's program: If the attributes of the thread is switchable and there is no RPU available then it is run on SW. But when a RPU is available, it is assigned to a free RPU (PR is performed if the current bitstream loaded does not match to what the thread needs). It can be run to completion or it can be preempted by a non switchable HW thread and the rest is run in SW. If the thread was non-switchable HW thread then it will be run on an available RPU. If there is none, then it will preempt a switchable thread that is running on HW or a HW thread with lower priority.

Figure 3.3 shows there are three kind of threads in SPREAD: HW hthreads running on Reconfigurable programming unit(RPU), SW threads running on CPU and switchable threads that can

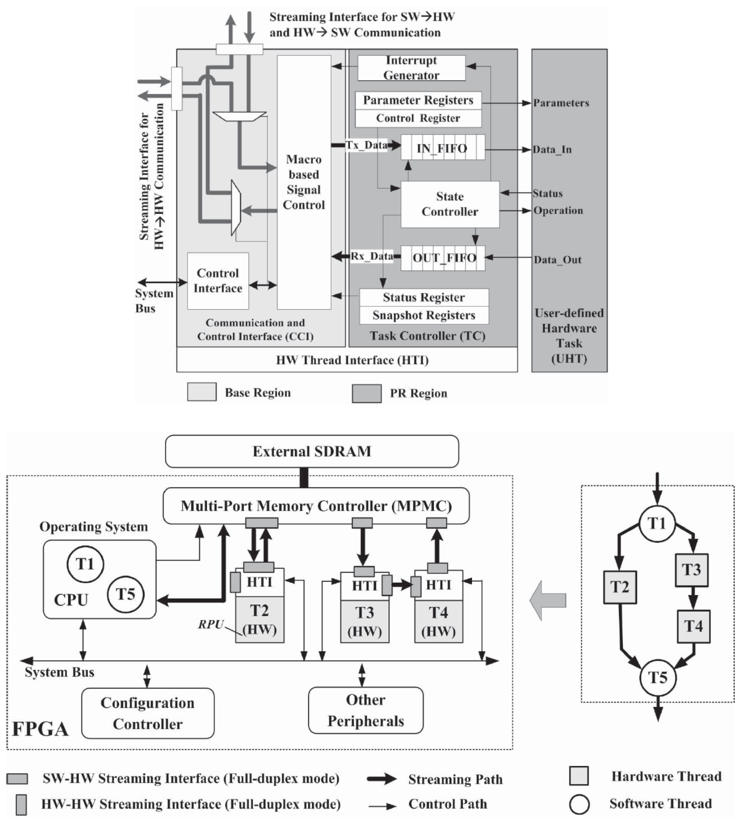


Figure 3.4: SPREAD system with three HW accelerators [65]

run on either SW or HW.

Figure 3.4 Left shows the SPREAD HW interface. As the name of SPREAD indicates, the HW interface is tailored for Streaming-based communications. There are two fixed streaming interfaces for each RPU. The one on the side is for HW-¿HW communication and the one on top is for HW -¿ SW communication. There is no BRAM interface for the accelerators. This is one caveat of this system, which makes the accelerator developers limited to two pair of FIFO interfaces. This is the opposite of RedSharc approach with flexible FIFO/BRAM interfaces which makes HW accelerator integration to the system easy. Figure 3.4 Right shows and implementation of a SPREAD system with three HW accelerators with fixed connections which can not be changed during runtime, and therefor not only free RPUs can be used for any HW/SW switchable thread.

They have a separate interface for point-to-point connections with hardware threads. They keep track of what PR regions have been reconfigured into runtime and reconfigured into what (what hardware resides there) using a configuration cache, which is referred to as resource allocation table. However, there are some problems with SPREAD. First of all, Having HW threads instead of HW accelerators is not resource efficient, as the entire thread needs to be mapped in gates, while only a portion of the code will take advantage of parallelism and the rest is sequential with no added benefit. Secondly, They use a customized HW Thread interface aka HW-based HAL, which is inflexible and is big as half of the size of a typical Microblaze.

SPREAD presents a partially reconfigurable system, incorporating runtime adaptivity through seamless hardware/software task switching. Although the project targets streaming applications only such as multimedia or cryptographic applications, SPREAD shares a few similarities in runtime adaptivity support with this Work. They cache runtime partial reconfigurations in order to minimize the overhead and increase the reuse of runtime allocated resources. Their work distinguishes several threads that echo similar behavior HEMPS, namely software, hardware, and switchable threads. Switchable threads contain both hardware and software implementations of the given thread logic in order to allow for runtime adaptivity due to the needs of the application/environment. A thread upon creation can run on either a processor or on the reconfigurable processing units, or migrate between HW or SW during its execution if it is switchable. However, runtime tuning is absent from these works as they employ static hardware/software partitioning during compile time. Therefore, the decision of determining whether partial reconfiguration or switching a thread between HW and SW brings any performance gain may be not be fully explored. This becomes a more important runtime decision as the system becomes more complex and can lead to possible performance degradation. From what was presented, all software threads execute on a single processor that also executes the OS. As a result of this, software threads do not operate concurrently and parallelism can only be exploited to the extent of hardware threads and one software thread. This is in contrast to our approach where software threads can execute concurrently on several processors within the system, and seamlessly migrate computation to an attached accelerator or perform computation with it in parallel.

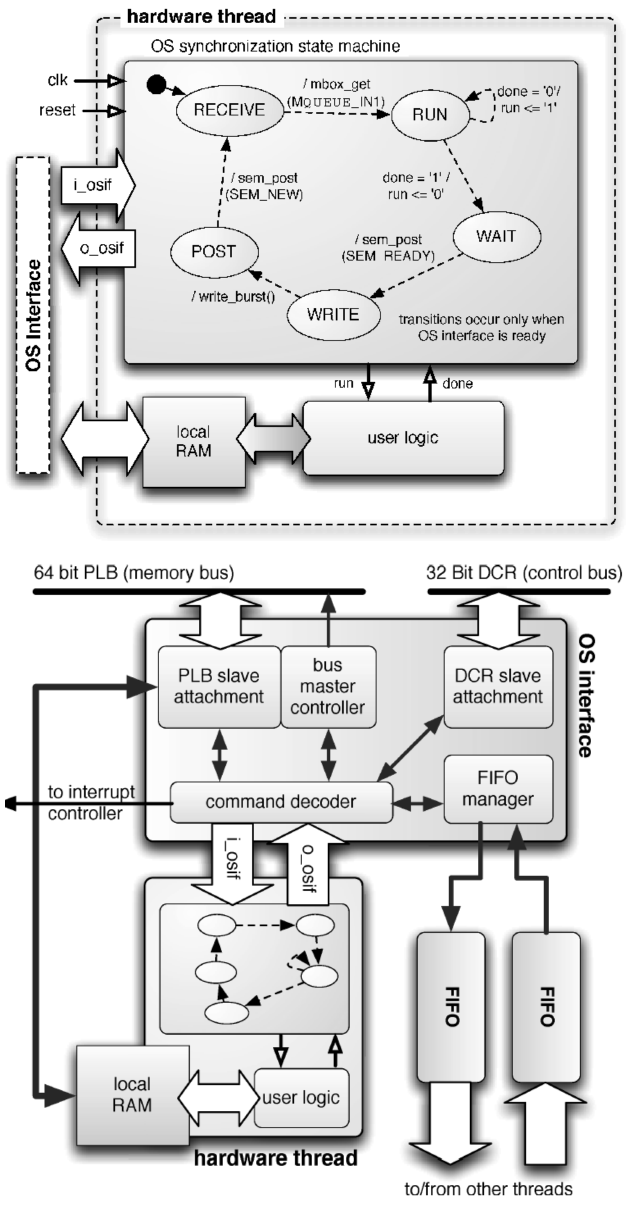


Figure 3.5: Left: A HW thread in ReconOS . Right:OSIF interface [45]

3.5.4 ReconOS

Hthreads[19] and ReconOS [45] allow hardware and software threads to co-exist and seamlessly be scheduled and synchronized under a unified programming model. These approaches provided performance increases through the use of standard programming models that allowed the parallelism within the application to increase in relation to the scalable numbers of processors that could be mapped into each new generation of FPGA's.

ReconOS [45] adopt a standard Linux kernel running on a host processor that provides operating systems through RPC mechanisms to threads running on slave processors. Basically, ReconOS uses the traditional approach of HW-based HALs which provide accelerators with Remote procedure calls (RPC) and other operating system services. On the data side, each HW thread has one BRAM interface and one pair of FIFO interfaces to exchange data with other HW threads, as shown in Fig 3.5

Figure 3.6 shows two different implementation of ReconOS. on the left, there are two hardware threads running on Custom HW and two SW threads running on CPU, talking to the rest of the system thanks to the OS interfaces. On the right, it shows how the HW thread can exchange streaming data via their OS interfaces. Using threads and common OS services as an abstraction layer, ReconOS extends the multithreaded programming model of software domain to reconfigurable domain.

3.5.5 OpenCL

OpenCL [6] (Open Computing Language) is a standard for cross-platform, parallel programming of modern processors , servers and embedded devices. It provides a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs and FPGAs. It includes a language for programming these devices, and APIs to control the platform and execute different task on the available compute devices. There are both using task-based and data-based parallelism in using OpenCL. It has been adopted by a lot of big names such as Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Altera,etc.

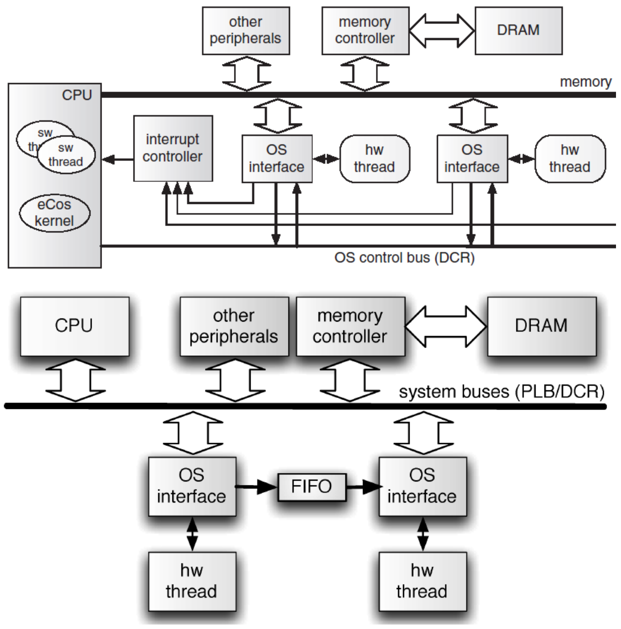


Figure 3.6: Left: ReconOS Two HW threads. Right: thread-to-thread interface [45]

For instance, OpenCL can give an application access to a GPU even when the application is not the non-graphical computing type. [40] has tried to automatically compile OpenCL programs into application-specific processors which are running on FPGAs. Also Altera [1] has developed some tools to translate OpenCL to run on their FPGA devices.

The memory management is explicit in OpenCL. Applications must explicitly transfer data between different memories. OpenCL does not support recursion, function pointers, etc. OpenCL is portable across different platforms. However, it runs best when the application is tailored specifically for that specific platform . Programs written in OpenCL does not always maximize a systems performance when moving from system to system. This is because the programmer has to explicitly specify kernel mapping in their program. Projects such as Bolt have tried to address this by making more smart run-time decisions.

OpenCL 2.0 spec has recently allowed kernels (threads) to further schedule additional work on other parts of the system. The OpenCL 2.0 standard has just recently allowed computation to be redirected to other compute devices. However, the programmer is responsible for making sure computation migration is the best choice during runtime for a given thread.

3.5.6 Bolt

Bolt [27] is a library of high level constructs for creating accelerated data parallel applications. With Bolt, kernel code to be accelerated is written in-line in the C++ source file. Bolt runs on top of other supporting languages such as OpenCL, but it does not require explicit OpenCL code within the source file . All initialization and communication with the OpenCL or C++ AMP device is handled by the library. Bolt also includes common compute-optimized routines such as sort, scan, transform, and reduce operations.

Bolt Dynamically queries the platform capabilities at startup and selects the accelerated path (accelerators) if possible. More importantly, it will also run on multi-core CPUS when accelerated path is not available. In sum, projects such as [48, 27] aim to provide the abstractions for transparently scheduling work across heterogeneous resources.

3.5.7 Cap-OS

Hübner et. al. [32] reported on a runtime adaptive OS referred to as Configuration Access Port OS (CAP-OS). They argue that the traditional software-tasks scheduling based 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, co-processors, and accelerators. During design time, applications are defined as a collection of tasks whereby each are described through a control-data flow graph (CDFG). These applications are profiled offline for execution time and possible suggestions for hardware implementation opportunities are presented to the user. The user can choose to provide hardware implementations of suggested code blocks enabling the runtime system for alternative task placement and scheduling. Similar to our work, Hübner classifies tasks into three categories according to where execution occurs: software tasks (processors), codesign tasks (processor+hardware accelerator), and hardware tasks (hardware accelerator). Scheduling of all such tasks occur through the main processor where CAP-OS executes. Similarly, OS services such as additional hardware resources occur through this single processor. This is a potential bottleneck, as services are ultimately serialized through

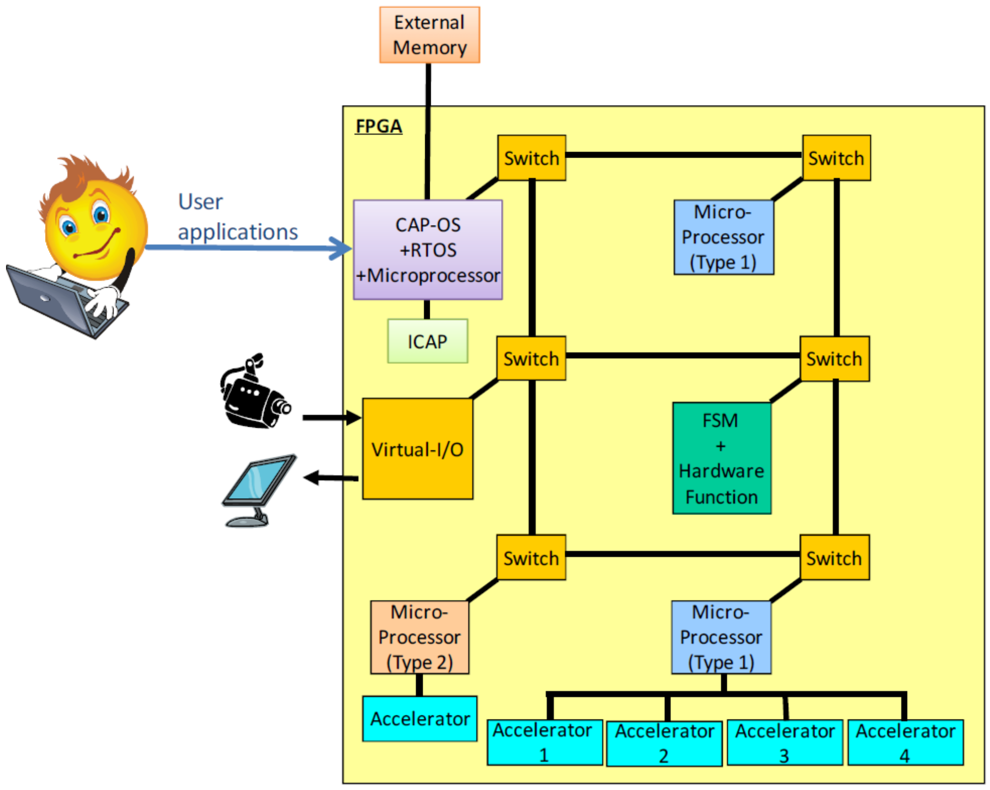


Figure 3.7: Left: High level view of CAP-OS [32]

the CAP-OS. In contrast, HEMPS does not follow similar remote procedural calls (RPC) to the OS. Processors within a HEMP system are autonomous such that they can decide whether to run a given task on itself in software and/or on the attached hardware accelerator. In the case for processors having an attached partially reconfigurable (PR) region, processors within a HEMP system can also decide to reconfigure this region independent of the main processor and other processing elements within the system.

A high level view of CAP-OS system is shown in Figure 3.7. In sum, CAP-OS [32] requires user profiling to build up a flow graph (CDFG) that is used for run time scheduling on the main processor. They use remote procedure calls (RPC) to invoke centralized scheduling decisions that occur on the main processor. Also software threads are limited to execution on a single processor. As a result, software threads do not operate concurrently and parallelism only exists through multiple hardware threads and one software thread. This thesis differs in which it supports true software concurrency by running software threads in parallel on scalable numbers of heterogeneous slave processors, as opposed to time slicing on a single central processor. My work is different than theirs in following aspects:

- They account for heterogeneous reconfigurable regions as opposed to homogeneous regions, however there is no indication of heterogeneous processors .
- They assume this is a sequential program vs. we already target a multithreaded application. Also they target C/C++ programs (with MPI) vs Pthreads for us.
- HEMPS is a system that works in reality, however a lot of their work is not yet fully implemented and it is at the theory stage. For instance, they have not implemented a way for reconfiguring a accelerator on demand during runtime.
- The decision of mapping tasks on either hardware and software appears to be limited to the main processor that is running their CAP-OS. We allow this level of decision making on top of independent slave processors to decide allowing for higher levels of concurrency and in the future, faster aggregate learning.
- They allocate resources given a need. If a software task is ready to be scheduled, and if no processor is available and won't be available soon, a new processor is added to the system during runtime. This is in contrast to our approach where we define during the design stages, how many processors are present. We consider the cost of reconfiguring such a complex device, albeit the MicroBlaze is really well optimized .
- They have 3 types of tasks: software tasks (processors), codesign tasks (software+hardware), and "pure" hardware tasks (echoes previous hardware accelerator + HWTI in Hthreads). An extensible processor is well tailored to run all three of these tasks, not only because an extensible processor can do fine grained HW/SW partitioning(for software and codesign tasks) but also it can serve as SW-based HAL for a hardware task running on its attached PR region.

- To reconfigure through the ICAP for software tasks, they achieve speeds of 28MB/s (FSL-ICAP) and 13 MB/s (FSL-FSL). For our codesign tasks we achieve speeds up to 96 MB/s (DMA-ICAP) for Virtex6 and 380 MB/s for Virtex 7.

3.5.8 Elastic Computing

Elastic Computing, [67] is a heterogeneous programming model which provides transparent, portable and adaptable computing model over heterogeneous resources and targets the FPGA as an accelerator on the whole.

Multi-core heterogeneous systems are becoming increasingly common in domains such as power embedded systems, high-performance embedded computing and high-performance computing (HPC) systems. Different approaches has been taken to reduce design complexity such as High level synthesis or new languages to ease parallel programming. The problem that they are tying to address goes back to the fact that usage of multi-core heterogeneous systems has largely been limited to device experts, due to significantly increased complexity. Their solution is to provide a library of specialized elastic functions that separates functionality from implementation details. Their thesis statement is that elastic functions allow designers to execute the same application code efficiently on potentially any architecture and for different runtime parameters such as input size, battery life, etc. Unlike the work in this thesis, they target HPC systems (including FPGA, GPU and CPUs) and try to provide Transparency, Portability, Adaptability. A high level view of Elastic functions is shown in 3.8. It shows an overview of elastic computing, which is enabled by (a) elastic functions that enable implementation planning to explore and even generate different implementations specialized for parameters such as input size, available resources, etc. (b) When an executing application calls an elastic function, the elastic computing system selects the quickest implementation based on the current runtime parameters and available resources. Note that no changes to the application code are required to use different resources.

Their work assumes the use case model of running applications on a desktop CPU and using heterogeneous resources such as FPGA GPUs to further boost the performance. They introduce

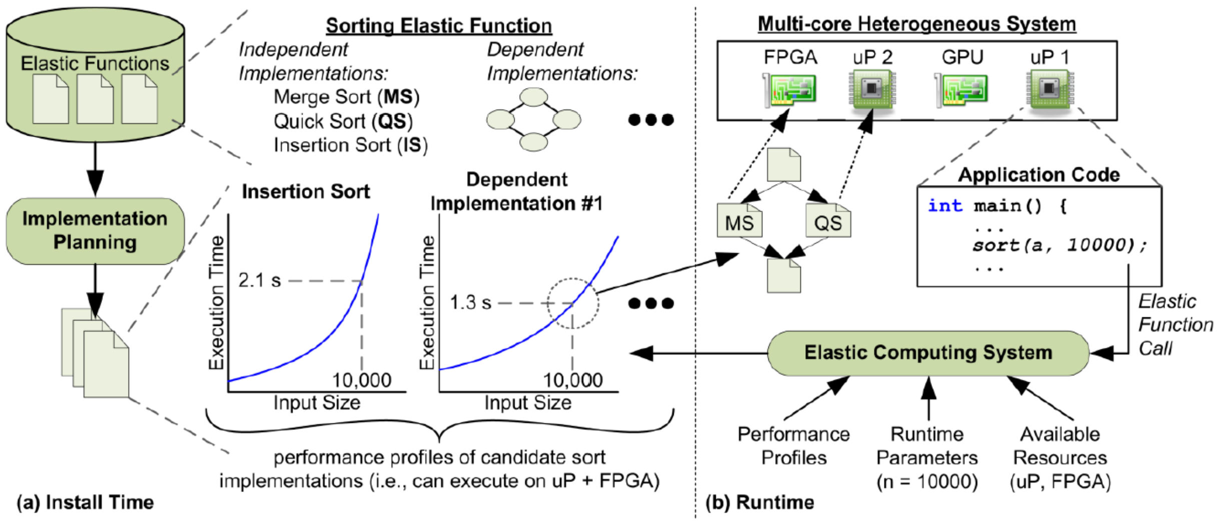


Figure 3.8: Elastic Computing [67]

the idea of elastic function libraries. These functions contain one or more implementations varying on their algorithmic implementation and/or different resources they require (i.e. GPU implementation only or FPGA). The Authors create a performance profile for all combinations of resources and implementations based on one elastic function before application start. That way, no overhead is incurred during application runtime. When it comes to scheduling, they don't take into consideration work queuing. Instead the resource with the best performance profile is selected. This Greedy approach can eventually lead to cases where all the work is assigned to only one processor/computing resource.

Each elastic functions can have multiple implementation. The implementations can be divided in two different categories: Independent implementations and dependent implementation. The first ones are binary executables for a specific combination of resources. For example, a sorting elastic function may have independent implementations like quick-sort, insertion-sort or merge-sort running different resource combinations like microprocessor, FPGA, GPU, microprocessor+FPGA and microprocessor+GPU. On the other hand, Dependent implementations internally call one or more elastic functions . For example, a dependent implementation of a Sort elastic function may internally rely on elastic functions for Split, Merge, and Sort. Dependent implementations create some sort of freedom to create new implementations for different runtime situations. Also, as the authors mentioned in the paper :"The main limitation of elastic computing is that improvement in design productivity depends on the percentage of code that can be defined using elastic functions." [67].

Although, both my work and the work presented in Elastic computing, mainly target the same notion of "Writing the code once, and running it everywhere", However The differences between the work in this thesis and Elastic computing are as follows :

- They target HPC, where FPGA itself is considered as a stand-alone accelerator, whereas in our approach the FPGA hosts the whole many-core heterogeneous system.
- There is no partial reconfiguration discussed in their work, while we take advantage of this capability in our Extensible processor.
- All the decisions of choosing the most efficient implementation of a elastic function are being centralized which leads to bottleneck and runtime overhead. In HEMPS system, this decisions are being delegated to each extensible processor. This distributed OS makes the system more scalable when the number of resources grow.

3.5.9 ARC

Jason,et al [24] presents a framework to support accelerator rich CMPS (ARC). ARC tries to address the following problems:

- How to share loosely coupled accelerators among cores in a many-core system on chip?
- How to service interrupts from accelerators to the cores?
- How to create bigger accelerator out of smaller ones during runtime (Virtualization)?

They refer to growing use of on-chip accelerators in many-core designs, since it helps in:Performance, Power and Utilization wall(dark silicon). However they claim OS-based management of loosely coupled accelerators in CMPs is not effective. This is only partially true, as in reality what makes

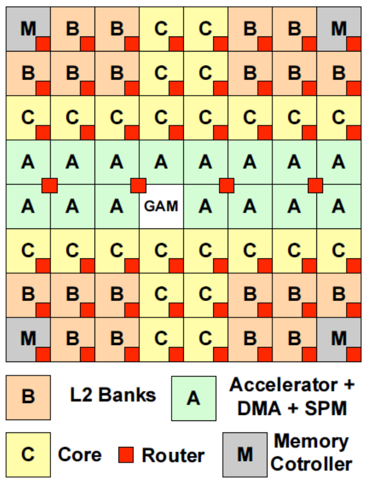


Figure 1: **Overall architecture of ARC**

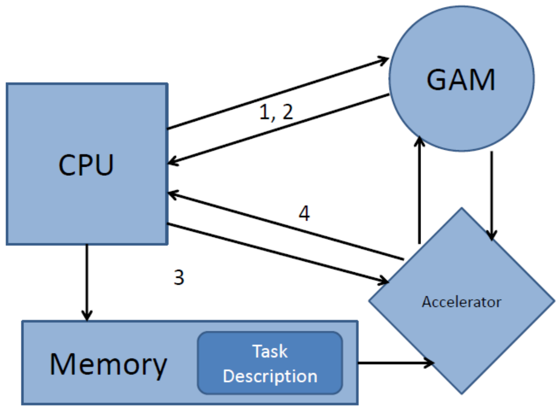


Figure 2: **Communication between core, GAM, and accelerator**

Figure 3.9: ARC approach [24]

it ineffective is the centralized decision making in the host processor which creates a bottleneck and stops scalability. Their solution is to propose HW-based management of loosely coupled accelerators.

They propose an accelerator-rich CMP architecture framework, named ARC,with a low-overhead resource management scheme that allows accelerators to be shared and virtualized in flexible ways, is minimally invasive to core designs and finally is friendly for application programs to use.

Figure 3.9 shows the overall architecture of ARC which is composed of cores, accelerators, the Global Accelerator Manager (GAM), shared L2 cache banks and shared NoC routers between multiple accelerators. It shows 1. The core requests an enumeration of all accelerators it may potentially need from the GAM (lcacc-req). The GAM responds with a list of accelerator IDs and associated estimated wait times. 2. The core sends a sequences of reservations (lcacc-rsv) for specific accelerators to the GAM. The core waits for the GAM to give it permission to use these accelerators. The GAM also configures the reserved accelerators for use by the core. 3. The core writes a task description detailing the computation to be performed to the shared memory. It then sends a command to the accelerator (lcacc-cmd) identifying the memory address of the task description. The accelerator loads this task description, and begins working. 4. When the accelerator finishes working, it notifies the core. The core then sends a message to the GAM

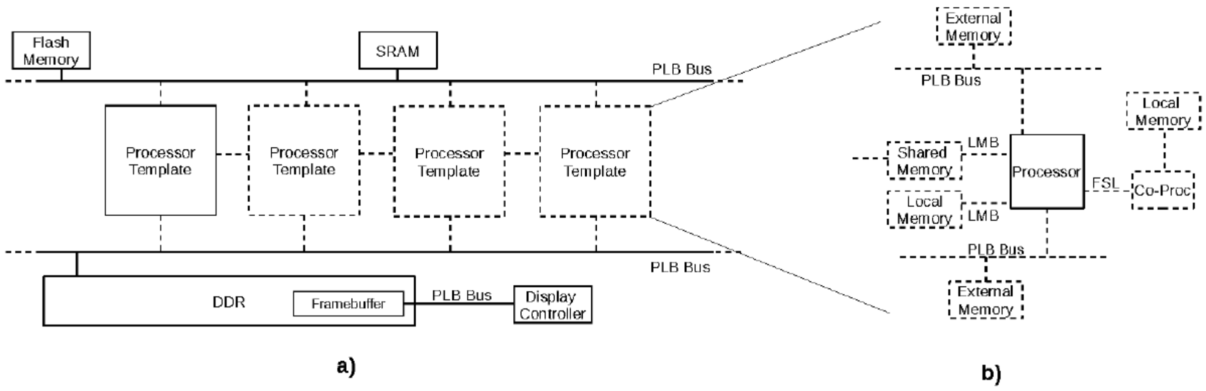


Figure 3.10: The processor template in TBES [25]

```
for (i = 0; i < 1138; i++) {
    for (j = 0; j < 42; j++) {
            fetchTask(&iqzz_d);
            iqzzTask(&iqzz_d, &block_YCbCr);
            idctTask(&block_YCbCr, &Idct_YCbC);
            yuvTask(&Idct_YCbC, &pix);
            dispatchTask(&pix);
    }
}
```

Figure 3.11: TBES tool chain [25]

freeing the accelerator (lcacc-free). All of the mentioned components are connected by the NoC. GAM is introduced to handle accelerator sharing and arbitration. GAM is performing a sharing and management scheme which can dynamically determine whether the core should wait to use an accelerator or should instead choose a software path, based on an estimated waiting time. The GAM tracks: 1) the types of available accelerators and the number of accelerators of each type; 2) the jobs currently running or waiting to run on accelerators, their starting time and estimated execution time.

3.5.10 TBES

TBES :Template-based exploration and synthesis of heterogeneous multiprocessor architecture on FPGA [25] is another ESL tool to allow designers with software programming skills to easily implement their ideas in programmable hardware (like FPGAs) without having to learn traditional hardware design techniques. The inputs to their tool is user's SW code which is strictly limited to explicit task-based coding like the one shown in figure 3.11, and a HW template that includes: static part, design space exploration boundaries, and finer HW details as shown in Figure 3.10. The hardware template has up to four Microblazes, with each Microblaze can have a HW accelerator via FSL links.

In TBES, Design space exploration is the cornerstone of the tools. It statically analyzes different combination of resource allocation, data mapping on memories and task mapping on processors to reduce the overall cost and increase performance. They use HLS tools to generate different types of accelerators and their performance and also use some rough estimation for SW execution times. They use these execution times to decide which accelerator should be attached to each processor. Although they claim that the HW platform can be flexible, but the result section is only showing a limited case of the architecture shown in Figure 3.10.

Our HEMPs approach is different than TBES in many ways, both HW and SW:

- The SW application in our approach is a general multi-threaded application, whereas in TBES the code is limited to a very specific task-oriented coding style shown in 3.11.
- In TBES The mapping of tasks to processors is static, whereas in our approach we have runtime coarse/fine grained partitioning based on available resources and runtime variables .
- The accelerators are static, and there is no partial reconfiguration option available in their flow, which decreases the flexibility and the performance of the design.

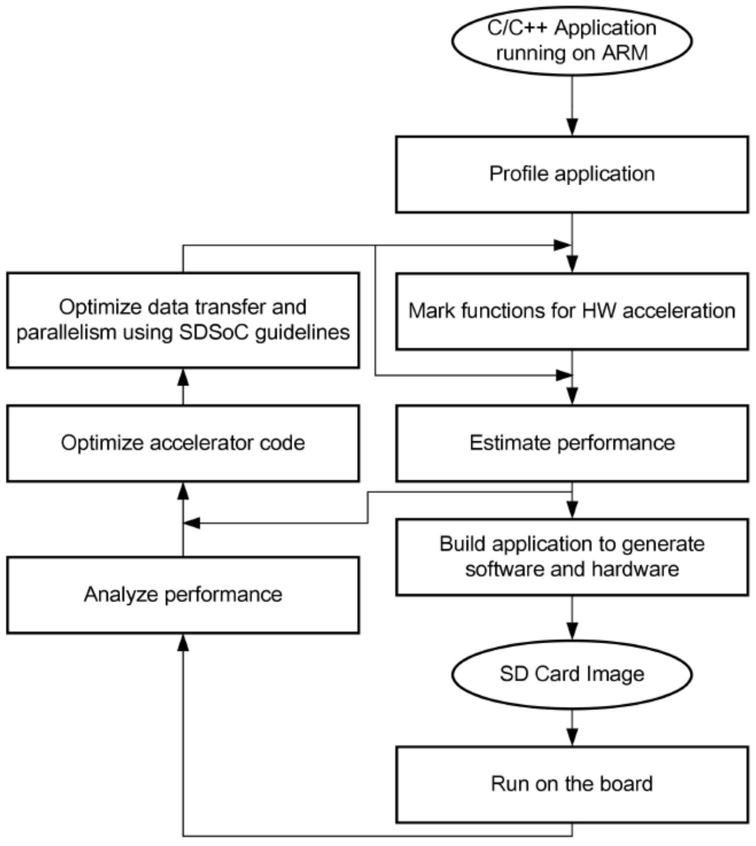

C/C++ Application running on ARM
Profile application
Mark functions for HW acceleration
Optimize data transfer and parallelism using SDSoC guidelines
Optimize accelerator code
Estimate performance
Build application to generate software and hardware
Analyze performance
SD Card Image
Run on the board

Figure 3.12: SDSoC environmental flow [71]

3.5.11 SDSoC

Sotware Defined System On Chip (SDSoC [71]) is a single point entry and fully automated approach for HW/SW codesign provided by Xilinx on Zynq platforms. The high level view is shown in Fig. 3.12.

The user provides a C/C++ application, as well as marking the functions that needs to be implemented on HW. The tools then takes care of the rest by running Vivado HLS under the hood, making the interfaces and software package libraries. As a result, the SW developer does not have to need anything about the details of hardware and FPGA fabric. However, the supported HW platforms is limited to Zynq board with two ARM cores to run SW threads. Therefore, it lacks the heterogeneity of enabling user to take advantage of higher thread level parallelism by providing more cores, or different type of general purpose processors other than ARM cores.

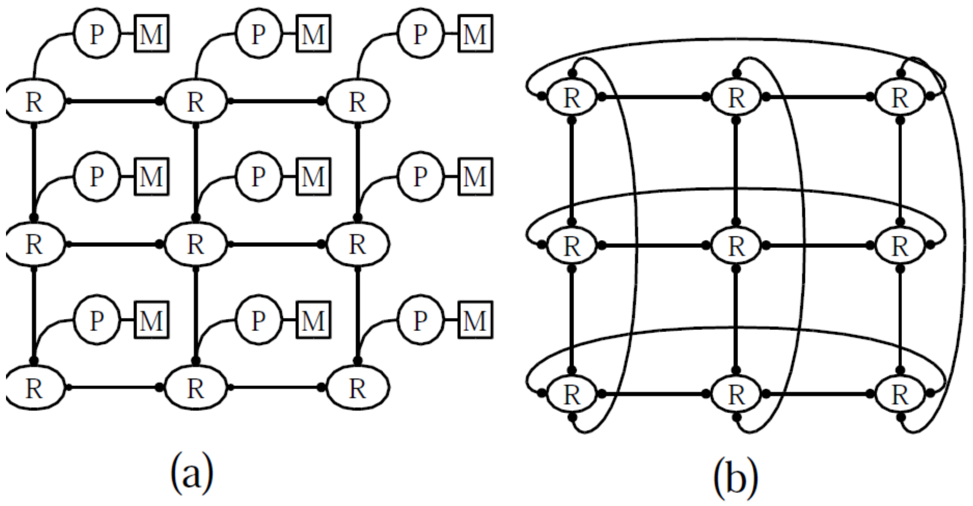


Figure 3.13: ICN approach [46]

3.5.12 Other works

Authors in [46] presented a system where an operating system executes on a general purpose processor, and tasks are scheduled onto dynamically reconfigurable tiles arranged in a 2D array network on an FPGA. Tiles are composed of custom hardware, but the authors mentioned that they can also be reconfigured to include soft processors. Decoupling task computation within tiles and the rest of the system allows the authors to concurrently adapt other parts of the system with ongoing running tasks. The overall architecture of the system is shown in Fig. 3.13. It shows in an ICN (a), each processor (P) is connected to a router (R). Each processor has access to local memory (M). In a 2D torus (b), each row and column of routers is connected in a ring, reducing router complexity with respect to a 2D mesh, in [30]

The work presented in [30] addresses issues with mapping the critical sections of source code onto hardware, either on coarse-grained or fine-grained components. They propose a methodology for partitioning and mapping computationally intensive parts of the code when it runs on top of reconfigurable hardware blocks of different granularity. Mapping more computational-intensive parts of the original program is reserved for coarse-grained components, whereas fine-grained components are swapped in and out of the FPGA. As shown in Fig. 3.14, they designed a self-adaptable

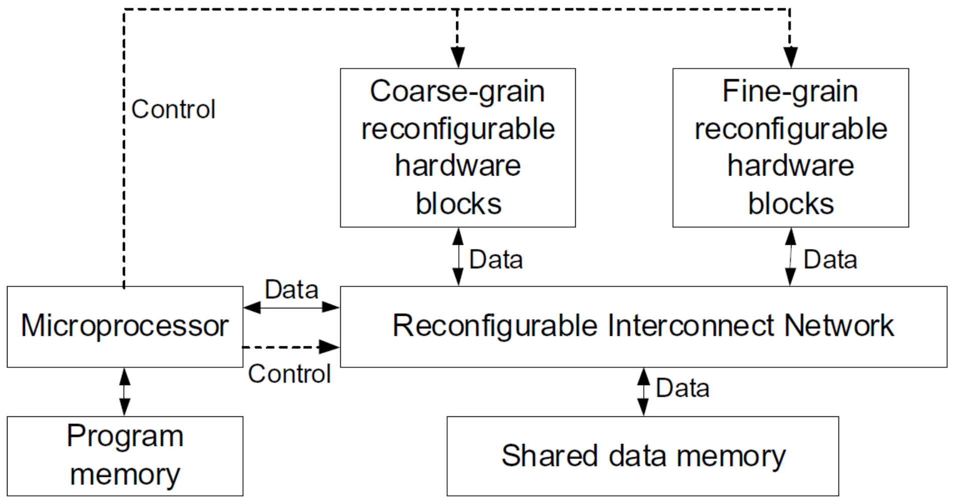

Control
Coarse-grain reconfigurable hardware blocks
Fine-grain reconfigurable hardware blocks
Data
Data
Data
Microprocessor
Reconfigurable Interconnect Network
Control
Data
Program memory
Shared data memory

Figure 3.14: Generic reconfigurable platform architecture in [30]

task manager on an FPGA for handling real-time tasks and scheduling them accordingly onto more coarse-grained components (i.e. hard/soft processors, ASIC devices, etc.) and fine-grained components (hardware IP that are dynamically reconfigured onto the device during runtime). The authors attempted to schedule incoming tasks based on priority, the amount of partially reconfigurable regions on the device a task needs, and the current state of other parallel tasks in the system. Their system is also capable of online learning, logging performance and power in order to improve future scheduling decisions.

Also, there are other interesting works in HW/SW co-design of loosely and tightly coupled accelerators:

- Off-chip loosely-coupled Accelerators (Convey [2], Nallatech [5])
- On-chip Tightly coupled accelerators(Garp [34], UltraSPARC [8], Intel's Larrabee [56], IBM's WSP processors [29])
- OS support for accelerator sharing and scheduling (P. Garcia, et al [31])
- Heterogeneous Architectures (EXOCHI [64], SARC [54], HiPPAI [62]), they use SW-based methodologies to access Accelerators.

Chapter 4

System Design

This work is orthogonal to techniques that treat the FPGA as a big accelerator controlled by a separate processor. We try to provide the abstractions, tool flows, and runtime systems to allow designers to define and program CHMP systems all under more familiar software centric programming models. We base this work on our original hthreads runtime system. A system level thread scheduler runs on a host processor and maps threads onto the available system resources, including processors with different ISA's and hardware threads with HW-based HAL (called HWTI).

This thesis presents HEMPs: Heterogeneous Extensible Multiprocessor system. At the architecture level HEMPS starts with the automation of the construction of a special CHMP system build upon extensible processors. At the software level, a compilation and runtime system is provided that allows the use of the standard Pthreads programming model. System designers with no hardware design experience can use these tools standalone to create heterogeneous chip multiprocessor systems, and program the system using scalable numbers of asynchronous threads. Beside thread level parallelism, HEMPs then provides standard interfaces and automated compilation flows to allow each processor within the multiprocessor system to support custom accelerators. These two capabilities support an integrated MIMD/accelerator model. The MIMD model provides thread level parallelism across the multiple processors, with the ability to accelerate execution of each thread by extending each processor with custom accelerators. Designers can allow the runtime system to dynamically tune an application based on runtime state. To support dynamic tuning the HEMPS compilation flow and runtime system supports the inclusion of operating system support libraries that allow functions to be encapsulated and scheduled during runtime based on performance tuning objectives as well as time varying system resource loading. Each library can contain both a software and hardware method of a function. The runtime system then determines if the function should be executed on the processor in software, as a static accelerator, or loaded into a

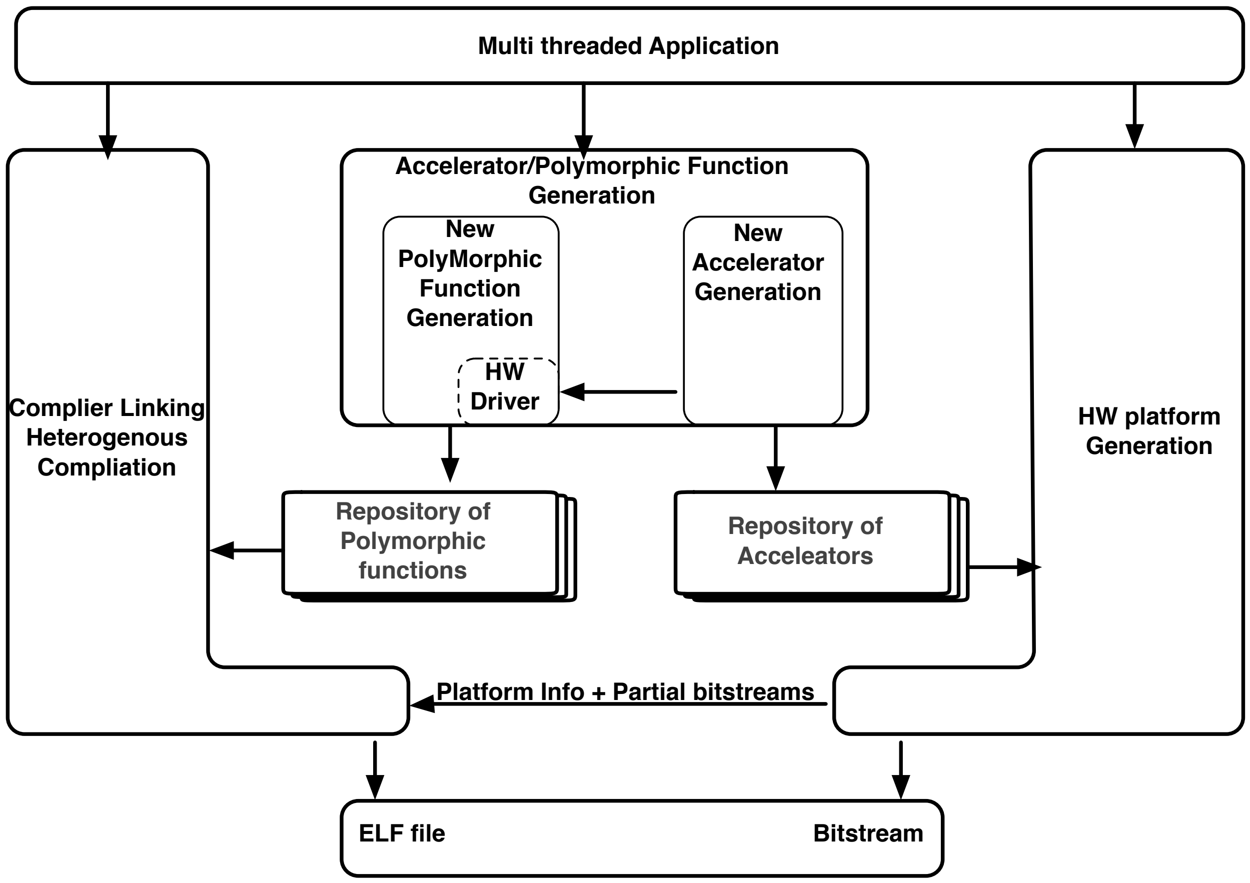

Multi threaded Application
Accelerator/Polymorphic Function Generation
New PolyMorphic Function Generation
HW Driver
New Accelerator Generation
Complier Linking Heterogenous Compliation
HW platform Generation
Repository of Polymorphic functions
Repository of Acceleators
Platform Info + Partial bitstreams
ELF file
Bitstream

Figure 4.1: General flow of HW/SW codesign in a HEMPS system

reconfigurable slot and run after doing partial reconfiguration. Figure 4.1 shows the general flow of HW/SW codesign in a HEMPS system.

This work extends Hthread system to run on a HEMPS system and support autonomous partial reconfiguration and software/hardware co-designed library functions. In a HEMPS system, hardware accelerators exists statically or dynamically behind a processor, and the runtime system should be modified to account for accelerators directly attached to slave processors, extending its resource-driven scheduling. Accelerator utility is driven by simple function calls linked in with a library. These functions allow execution to continue in software, or if better suited, on an attached hardware accelerator.

This work tries to create new abstractions and runtime capabilities to self tune applications as they run across FPGA based heterogeneous chip multiprocessors. This brings the following benefits. First we enable portability and efficiency for system designers and programmers. Once written the application program can be tuned by the runtime system to achieve better performance and energy efficiency across different mixes of processors and accelerators. Second, this eliminates the time consuming requirement for application developers to deal with CAD tools to build the HW platform from HW point of view, and spares them from performing exhaustive static profiling and partitioning from SW point of view. For the first part, the generation of a HEMPS system tailored for the application is automated. For the second part, based on the resources available the runtime system identifies and maps the application across the best available combination of processors, accelerators and program mappings, resulting in better performance and energy efficiency compared to static profiling and mapping.

In this work, the Hthread OS running on HEMPS system allows Software threads to run on scalable numbers of heterogeneous slave processors. All threads perform service requests locally to a lightweight extension of the operating system that runs on each extensible processor. Further scheduling decisions between hardware and software computations for all threads are made autonomously on each extensible processor node. Our approach can perform the same migration of a complete thread body from software to hardware, but with the additional flexibility of allowing a sequential portion of a thread to execute on the slave processor, and dynamically migrating only the data parallel portion of the thread into hardware. This decision is learned from profiling information.

4.1 Extensible processor node

4.1.1 Base Hthread system platform

Figure 4.2 illustrates a typical HybridThreads (Hthreads) platform, with Hthread cores being a HW/SW co-designed microkernel that supports the POSIX threads standard. Some of the OS core services such as thread scheduling and synchronization are implemented within hardware for fast, distributed access across both HW and SW threads. The host processor communicates with Hthread cores to create/join software/hardware threads, as well as synchronization among the

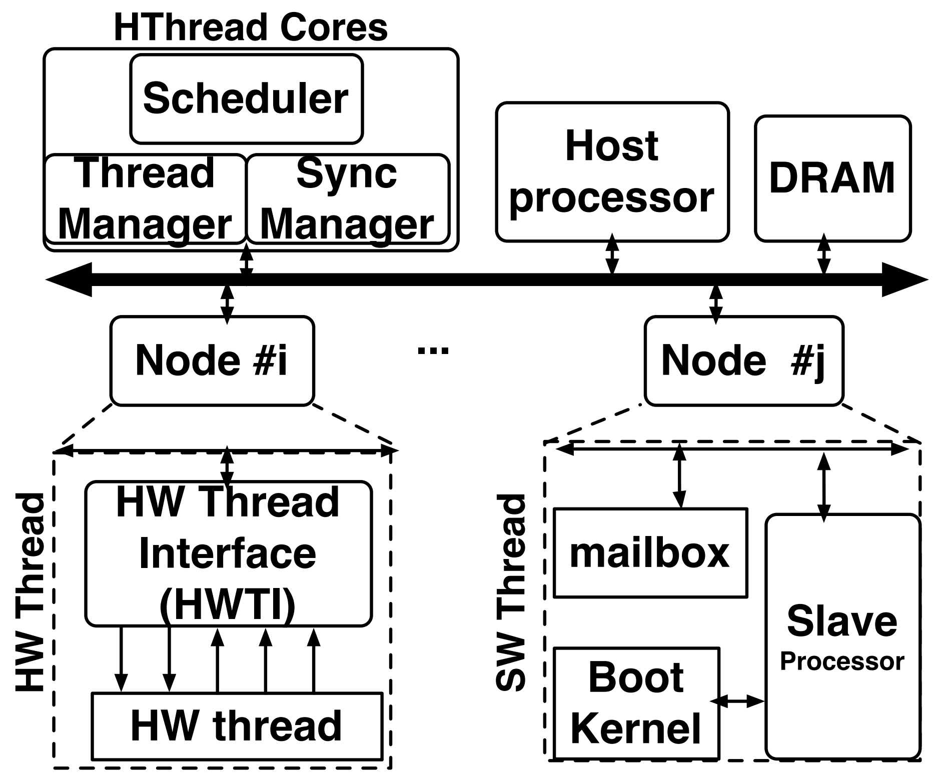


Figure 4.2: High level view of Hthread system

threads.

The host can also assign threads to run on general purpose slave processors. During compilation flow, each thread function is compiled for both host and slave processors, which can be of different types. During runtime, based on which processor is going to execute the thread the corresponding image will be passed to the processor. There is a boot kernel code which runs on slaves during boot up, and a BRAM called Virtual HWTI BRAM (VHWTI) which both slave and Hthread core have access to it, and communicate via this BRAM. Hthread cores create a thread by writing thread information into pre-defined entries in VHWTI , which is constantly checked by the slaves when it is idle. Then, the slave starts executing the SW thread stored in DRAM (Each slave has a ICache to boost the performance), and upon exiting the thread informs the rest of the system by writing into pre-defined entries in VHWTI.

Figure 4.3 provides an example of an Hthread system when Extensible processor node is added. As shown in this figure, HW threads execute on custom hardware and are able to access OS ser-

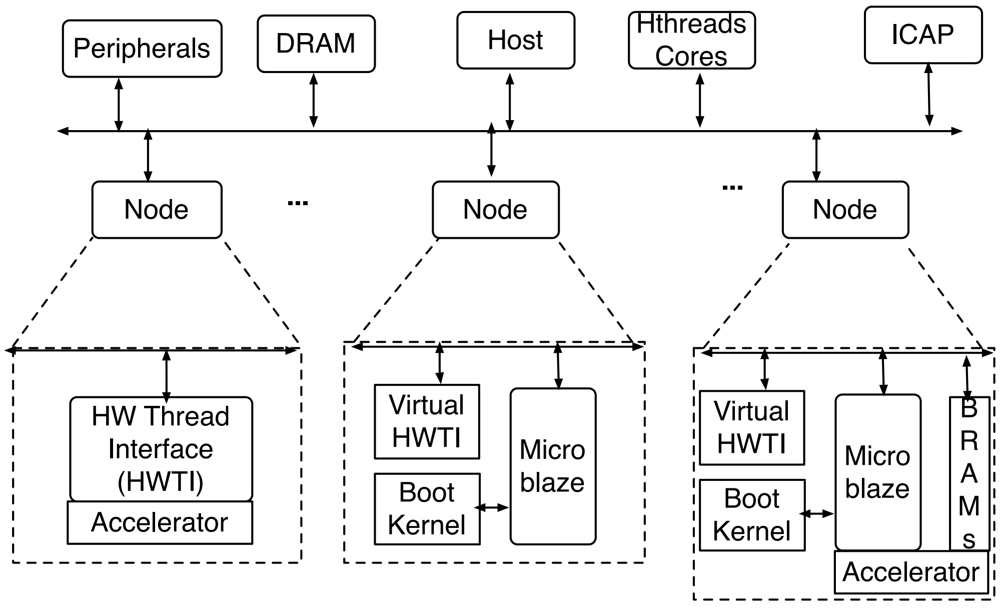


Figure 4.3: A Typical Hthread system with different computational nodes

vices through a local FSM-based HAL referred to as the Hardware Thread Interface (HWTI). On the other hand, SW threads execute on slave processors through a mailbox and request OS services through software library calls. These software library calls translates to simple load/store operations to the Hthread cores enabling any heterogeneous processor supporting atomic memory calls to be supported by Hthreads. Further details can be found at [19]. The first two node templates, HWTI+Accelerator and General purpose MicroBlaze was already tested and working. The contribution of this thesis is the last node, which is referred to as Extensible processor node.

This work, unifies both loosely coupled and tightly coupled accelerators as an extensible processor implemented through a tight coupling between the static or dynamic accelerator and a front end processor. There is a standard interface using AXIS links between the processor and accelerator for control flow. This model proves resource efficient, portable, and better supports dynamic partitioning and allocation of resources. As shown in Fig 4.4, Without any loss of functionality the Microblaze can be simply viewed as a plug in replacement for the hardware virtual abstraction layer. Changes to the operating system can be realized through software compilation in place of hardware synthesis. Further resource savings are achieved as the sequential portion of a thread can be migrated out of hardware and into software executing within the Microblaze. An extensible

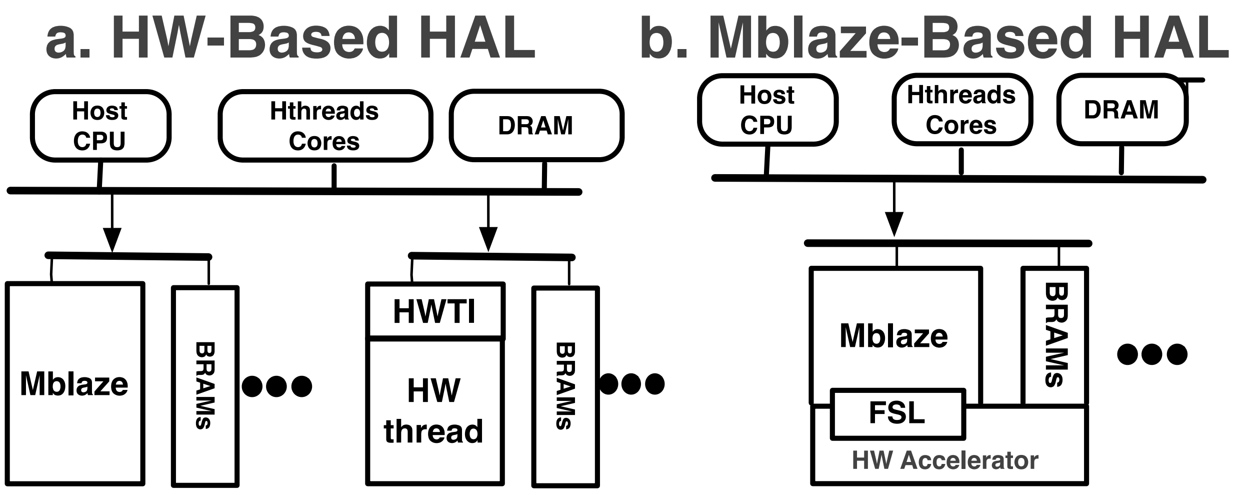


Figure 4.4: a. HWTI b.Micoblaze-based HWTI.

processor model widens the use of the accelerator to exploit a greater range of parallelism especially fine grained data level parallelism. This allows better resource utilization and support for dynamic tuning. Each extensible processor is now available to run any thread, with the choice of using an accelerator being made autonomously on each processor. Under dynamic reconfiguration each processor can make an independent decision to download a new bitstream into it's partial reconfiguration area.

4.1.2 Microblaze-based HAL and API

As shown in Fig. 4.5 Extensible processor is built on the approach followed in the Hthreads system with hardware threads. These steps were taken to design the extensible processor node

1. The hardware circuitry of the HWTI is replaced by a small operating system kernel stored in the Microblaze's local memory. The MicroBlaze replaces the original HWTI's command, control, and status circuitry for an accelerator. The Hthreads system provided a traditional software HAL as part of the operating system that ran on all slave processors. The existing software HAL performed the exact same functions as the System State Machine in HWTI and can be adopted with minimal modifications. The following modifications was performed to completely replace the HWTI VHDL wrapper with the software based HAL running on

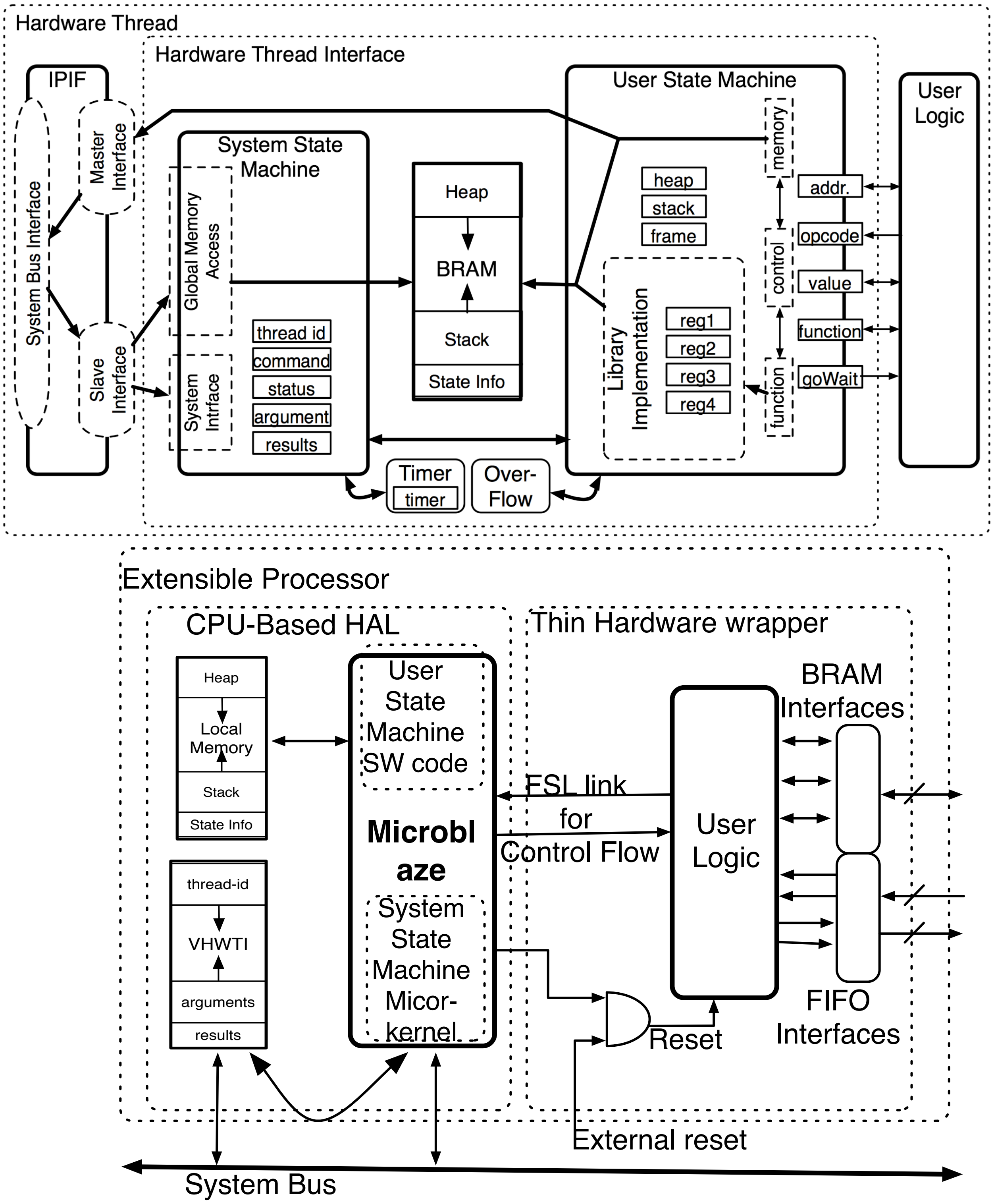


Figure 4.5: HWTI Implementation [15] Vs. Extensible processor

Microblaze:

- First, the five user visible registers shown in Figure 4.5 was replaced by the two Fast Simplex Links (FSLs) [3] shown in Figure 4.5. This requires rewriting the interface and modifying the protocol that was provided within the user logic to request services. The API's, or system service policies accessible by the accelerator did not change. However, the invocation protocols was re-defined to use the FSL links.
- Second, the system services implemented as finite state machines within the User State Machine shown in Figure 4.5 was rewritten in C. The MicroBlaze then monitors the FSL links, decodes requests, and performs the appropriate system calls on behalf of the accelerator.

2. One pair of FSL/AXIS links is used for communication between the accelerator and Microblaze. Control, configuration, and status commands are transferred using these FSL/AXIS links between the MicroBlaze's register file and the accelerator. These links are useful for exchanging single data items but are not appropriate for supporting fast block transfers of data into and out of the accelerator. So, it should be mainly used for control flow.

3. Additional dual port BRAMs is provided between the accelerator and the extensible processor's local bus. These BRAMs are made visible within the system's global address space and accessible by any bus master in the system. The operating system as well as the application code running on the MicroBlaze can directly access these memories, or set up DMA transfers into and out of the memories on behalf of the accelerator. Providing a local DMA per extensible processor node also gives each MicroBlaze the flexibility to create unique access patterns per accelerator, such as strided transfers for matrix operations. The numbers and length of the BRAMs can be tuned by the system designer for any application running within an accelerator. The local DMA can also used by the operating system to load bitstreams into the partially reconfigurable accelerator slot. The data interface of HW accelerator is general and flexible since it is independent from MicroBlaze. The accelerator

can have as many FIFOs and BRAM interfaces without MicroBlaze intervention.

4. A set of APIs and a standard interface protocol was defined between Microblaze and accelerator. These can be used by accelerators to perform system calls and memory operations, as well as the run time system on the MicroBlaze to control the the accelerator. The stack of Microblaze is made accessible to the accelerator through these standard APIs. This allows accelerators to be written that include functions calls and recursion. Figure 4.6 lists a few of the more common APIs. The first category of APIs allows the accelerator to declare and access local variables on the stack. The second category provides APIs for managing the stack, and returning from function calls. The third category provides APIs for performing system calls. The complete set of APIs supported are the same as those implemented in the hthreads HAL.The operating system on the MicroBlaze receives APIs from the accelerator across standard FSL/AXIS links, processes the request, and returns status back to the accelerator on an FSL link.

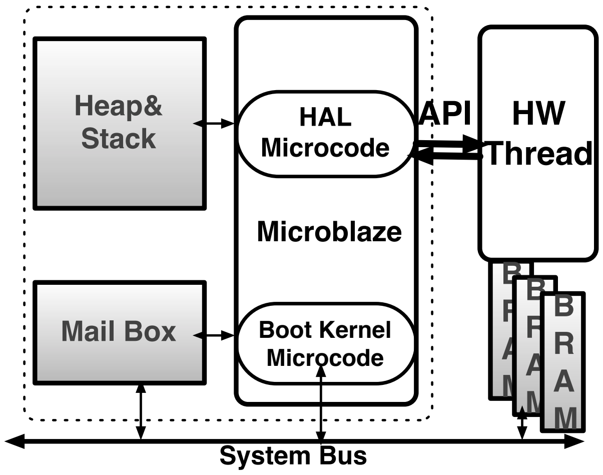
5. A VHDL wrapper code was written that will be included in the user_logic code of each accelerator. The wrapper includes states for issuing our standard APIs. New APIs are added by creating new states. The operating system running on the MicroBlaze identifies requests from accelerator. System service processing is contained with the case statements. New APIs and system services are added by writing additional case statements.

Using MicroBlaze instead of HWTI gives us the ability to add more functionalities via software code way much easier than if we had to do it in HWTI using HDL languages. New functionality can be added through software modifications compared to earlier requirements of redesigning and re-synthesizing hardware state machines. MicroBlaze provides RPC, stack management, pointer support, true recursion and function call for HW circuits like Quicksort accelerator. It also provides optimizations like overlapping the data transfer to/from BRAMs with data processing by the accelerator. This means we can have more flexible accelerator with function calls and pointer support. Sorting is a good example for this case. Quicksort have significant performance compared to bubble sort when the size of the array increases, and Quicksort needs stack manager since it has recursive calls. The only caveat of implementing HAL using flexible processor instead of Custom HW interface is having more delay in servicing the requests of HW accelerator.

The Top part of Figure 4.6 shows how the operating system on the MicroBlaze receives APIs from the accelerator across standard FSL links, processes the request, and returns status back to the accelerator on an FSL link. In this example, the operating system service call performs three consecutive reads of the FSL, performs the appropriate processing, and returns a single status response.

The numbers of reads and writes across the FSL link and the processing performed is determined by the type of service call being implemented. Figures 4.7 and 4.8 show the wrapper code for constructing API requests in the accelerator and the operating system code running on the MicroBlaze to process the API requests. Figure 4.7 shows VHDL wrapper code that should be included in the user_logic code of each accelerator. The format of the API request has been standardized, and includes an op_code and up to two arguments. The wrapper includes states for issuing our standard APIs. New APIs are added by creating new states. The *send_request* state at the bottom of the wrapper code is a pre-defined state that issues the API request across the FSL link . This eliminates the need to completely re-design the HW interface. The operating system running on the MicroBlaze identifies requests through the op_code in the *sw_based_hal* function shown in 4.8. System service processing is contained with the case statements. New APIs and system services are added by writing additional case statements.

Figures 4.8 and 4.7 provide an example of how an accelerator (quicksort in this example) would be written, called, and executed. The application code shown in the top of Figure 4.8 calls the quicksort function within the body of a thread. The middle part of the code shows how the call would be implemented using system service calls running on the MicroBlaze. Note that the implementation is fully portable and can be run on any extensible processor within the MPSoPC. The data to be sorted is first DMA'ed from DRAM to the local BRAM using *dma_data (&data, &LOCAL_BRAM, size)*. The run time system resolves the physical address location of the local



Category	HW thread Request	Description	MicroBlaze Response
Local Vars	declare *n*	declare *n* local vars on stack	0
	read *i*	read value of local var *i*	stack[i]
	write *j val*	write *val* into local var *j*	0
Function Call Support	push *x*	push *x* on stack	0
	pop	pop top of the stack	stack[top]
	return	return from function call	*ret_st*
	call *ret_st*	push *ret_st* on stack	0
System calls	sys_call(*x*)	perform sys_call *x*	0

Figure 4.6: API protocol Across FSL links between MicroBlaze and HW thread

```
-- FSM logic
case (current_state) is
   .......
   when write_local_var =>
      opcode_next  <= WRITE_OPCODE
      param1_next  <= j
      param2_next  <= val                     User
      return_state <= state_2;                Logic
      next_state <= send_request;             Code
   .......
   when accelerator_exit =>
      opcode_next  <= OPCODE_SYSCALL
      param1_next  <= EXIT_THREAD;
      param2_next  <= (others=>'0');
      return_state <= idle;
      next_state <= send_request;
------------------------------------------------------
   when send_request=>
      --Send opcode, param1 and param2 via FSL link
      FSL_M_data <= opcode &param1;
      FSL_M_data <= param2;
      --Receive response from Mblaze
      mblaze_response <= FSL_S_data;        Hardware
      --Resume sequencing through FSM .     Abstraction
      if (opcode =  OPCODE_RETURN) then     Layer
         next_state <= mblaze_response;
      else
         next_state <= return_state;
      end if;
end case;
```

Figure 4.7: Snippet of VHDL Wrapper Code

BRAM. The accelerator is provided the address of the array and number of elements to be sorted in BRAM using *putfsl(&LOCAL_BRAM); putfsl(size)*. The start command is then issued using *putfsl(GO_CMD)*. After the start command is issued control is transferred to the software based hardware abstraction layer portion of the operating system by calling the *sw_based_hal()* function. It is worth noting that all of this processing occurs locally and autonomously on each MicroBlaze processor within the MPSoPC. Thus no centralized bottleneck is present that would restrict scalability.

The accelerator initiates processing when the *putfsl(GO_CMD)* command is received. The accelerator can request additional system services within the user state machine description as shown in the top of Figure 4.7. The *send_request* state at the bottom of Figure 4.7 communicates system service requests to the *sw_based_hal* in the bottom of Figure 4.8. When the accelerator is finished it transfers a done command to the *sw_based_hal* which returns control back to the function body. The results are DMA'ed back from BRAM to DRAM and control returns to the calling

```
void * foo_thread(void* arg)            Application Code
{  //Some sequtial work....
    quicksort(data[],size);
   //Some sequtial work....
 }//---------------------------------------------------------
void  quicksort(data[],size){        Function Implementation
  dma_data (&data, &LOCAL_BRAM, size);
  //send start command to accelerator
  putfsl(&LOCAL_BRAM); putfsl(size);
  putfsl(GO_CMD);
  sw_based_hal();
  dma_data (&lOCAL_BRAM, &data,  size);
} //---------------------------------------------------------
void sw_based_hal(){
do {
 getfsl(opcode&param1);  getfsl(param2);
 switch(opcode)
 {  case(OPCODE_WRITE):
         stack[frameptr+param1] = param2;
         putfslx( 0);
     break;                                  Hardware
     .......                                 Abstraction
     case (OPCODE_SYSCALL) :                 Layer
         switch(param1):
             case (EXIT_THREAD):
                return (void*)0;
             break;
             ......
     break;
  } } while (1);}
```

Figure 4.8: Snippet of Operating System Interface Code

application. used to resume the control back after the request is serviced, unless it is a *return* request. In that case, the MicroBlaze retrieves the return_state from the stack which was sent to MicroBlaze during the previous *call* request. There are three categories of accelerator requests as shown in Figure 4.6. For the first two ones, MicroBlaze uses its scratch-pad memory as a stack to enable the accelerator calls different states in its FSM, declare local variables and pointers. Also, MicroBlaze performs system_call requests on behalf of the accelerator. These system_calls are either expensive or time-consuming to implement or mostly sequential, for example mutex_lock, etc. Extensible processor HAL provides the following benefits:

- Enables users to more efficiently exploit finer grained parallelism within a thread body.
- Enables compilation to replace synthesis for HAL based services.
- Reduces the gate requirements of HAL and accelerator circuits.
- Providing accelerators with stack management, pointer support, RPC services and other op-

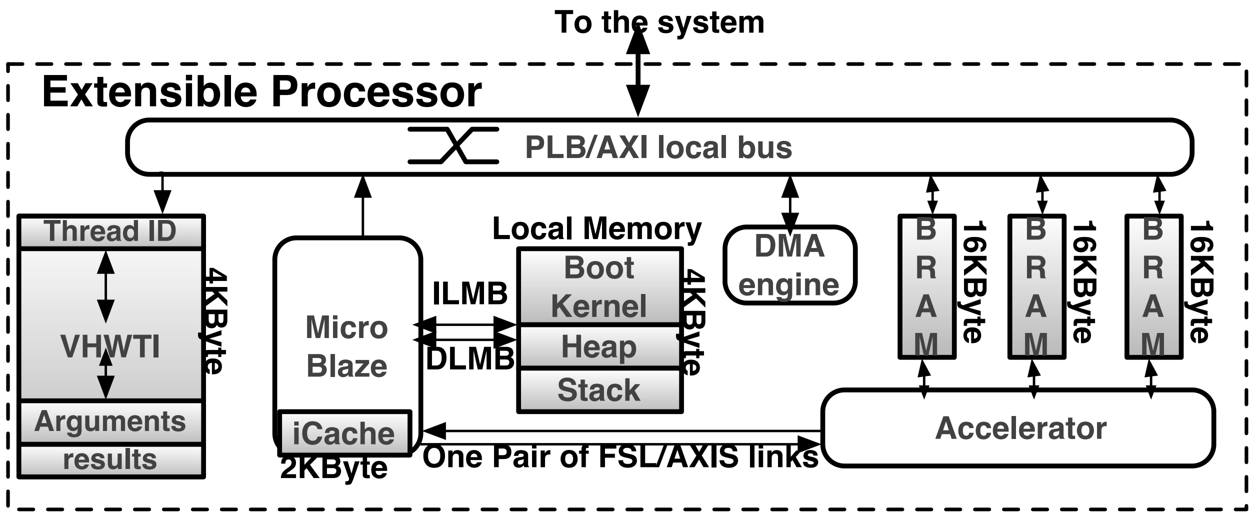


Figure 4.9: Extensible processor node with only one accelerator

timization like overlapping data transfer with computation.

- Providing distributed low latency operating system services that can transparently scale with growing numbers of processors and accelerators. Eliminating the overhead of RPC calls, the bottleneck of running all services on a centralized master node. As such this approach scales better for next generation MPSoPC systems.

4.1.3 Data path

Figure 4.9 shows the detailed architecture of an extensible processor node with only one accelerator. The standard interface provides three BRAMs. This interface should remain the same for all accelerators since it is necessary for partial reconfiguration. Each extensible processor is responsible for managing the flow of data into and out of the BRAMs through a local DMA Engine. For slave processors equipped with dynamic accelerators, the DMA engine is also used for transferring new bitstreams to the ICAP for low- latency partial reconfiguration (PR). A wrapper is provided around the accelerator region that allows the slave processor to query the presence and type of accelerator, and enables bidirectional communication between the accelerator and slave processor through the AXIS. This supports autonomous *tuning* of a thread execution to occur, independent of other slave processors.

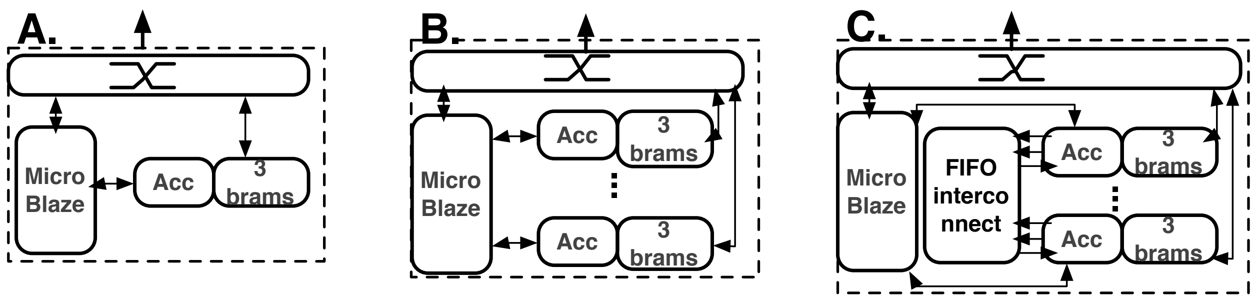


Figure 4.10: The three potential architectures of an extensible processor node.

The three potential architectures of an extensible processor is shown in Fig 4.10. From Left going to the right, each architecture provides more performance, at the cost of complexity. However, no matter which one I choose the details should be abstracted away from SW developer. The first one on the left shows a node that each Microblaze has only one partially reconfigurable accelerator with BRAMS shared between the two. The one in the middle, shows that each Microblaze can have more than one accelerators with their BRAMS shared, however the accelerators can not communicated directly with each other. The advantage of this over the first one is reducing the possible number of partial reconfigurations overhead during runtime. Finally the last one shows the architecture in which the accelerators can communicate with each other via a FIFO interconnect. This leads to better performance over the middle one as it reduces the number of data transfer back and forth between BRAMS and external DRAM.

Figure 4.9 shows the use of either a PLB or AXI bus for the extensible processor's local bus, and FSL/AXIS links for communication with the HW thread. Control, configuration, and status commands are transferred using the FSL/AXIS links between the MicroBlaze's register file and the HW thread. These links are useful for exchanging single data items but are not appropriate for supporting fast block transfers of data into and out of the accelerator. Figure 4.9 shows how dual port BRAMs provided are shared between the HW thread and the extensible processor. The processor uses its local DMA engine to transfer data between DRAM and these BRAMs. These BRAMs are made visible within the system's global address space and accessible by any bus master in the system. As these memories exist within the systems global address space they can also be

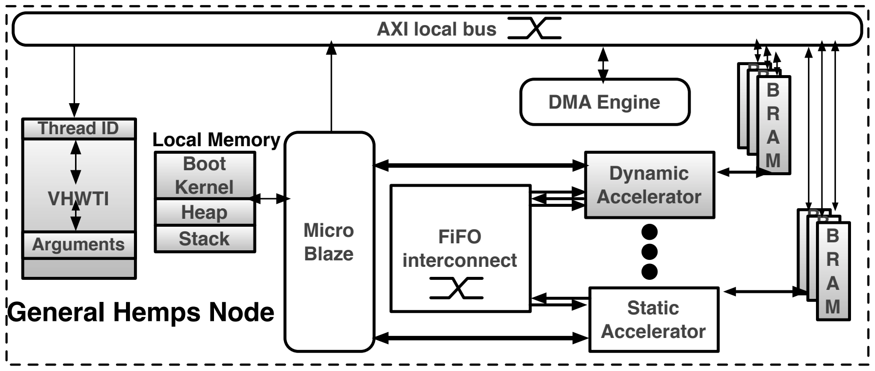


Figure 4.11: Architecture of a Simple HEMPs node

used as FIFO buffers between extensible processors for streaming purposes. The numbers and length of the BRAMs can be tuned by the system designer for any application running within an HW thread. The OS as well as the application code running on the MicroBlaze can directly access these memories, or set up DMA transfers into and out of the memories on behalf of the HW thread. Providing a local DMA controller per extensible processor node also gives each MicroBlaze the flexibility to create unique access patterns per HW thread, such as strided transfers for matrix operations and prefetching data to overlap data transfer with computations . It is also used to load bitstreams in case HW threads are partially reconfigurable during runtime, and eliminates the central bottleneck that existed within earlier systems that only allowed a single master processor to load bitstreams [65]. We have measured transferred rates of 96 Mbytes/sec to transfer bitstreams into the ICAP on Virtex 6 with ISE tools.

Fig 4.11 shows the general structure of a HEMPS node. Each node consists of a general purpose processor, which can be soft IP like Microblaze, BRAMs and accelerators, DMA engine, a FIFO interconnect for accelerator streaming data and local memory and mail box for the Microblaze. The Microblaze is communicating via a pair of AXIS links with each accelerators for control flow. The number of the AXIS interface on Microblaze is limited to 16, so is the number of the accelerator in each node. Accelerators can be either dynamic or static. Each accelerator can have arbitrary number of BRAMS which is connected to local bus of the node. This gives the Microblaze the ability to access the data in those BRAMs plus DMAing data from/to DRAM on behalf of the accelerator. Also each accelerators can have arbitrary number of FIFO ports to enable direct streaming between them. This saves the time to DMA data back and forth to DRAM in case more than one accelerator is processing the same data.

The operating system running on the MicroBlaze controls the transfer of bitstreams into the accelerator region and data between the DRAM and BRAM's. This is an advantage over systems that use hardware HAL's. In these earlier systems all bitstream and data transfers were controlled by the operating system running on a master node.

Each Microblaze has 3 local BRAMs. The first one is VHWTI bram which serves as a mailbox to synchronize the threads assigned to this node. It is accessible by both host processor and Hthread cores, further details can be found [18]. The second BRAM is a local memory which stores both the small operating system running on slave Microblaze. It has both the code for a simple bootkernel to check the VHWTI for thread operations, plus a SW-based HAL code to serve the custom HW threads assigned to the accelerators. Extensible processor node provide the HAL for custom HW threads. This unifies the two models of loosely and tightly coupled accelerators. It provides Stack management, fine grained HW/SW partitioning and distributed OS services. The final BRAMs are the ones shared between accelerators and Microblaze which can serve as fast access data for Microblaze since the Data cashe in not enabled (there is no cache coherency protocol).

The architecture of a Extensible processor node is flexible in a way that only the Microblaze and it's local memory and VHWTI brams are needed. Each node can arbitrarily have any number of dynamic or static accelerators (if any), any number and size of BRAMs for each accelerator, an optional FIFO interconnect in case more than of the accelerators have FIFO ports. This flexible structure makes it easy for HW developers to integrate new accelerators into the system, not only from the interface point of view, but also any stack or pointer management or recursive calls can be handled by HAL implemented on Microblaze.

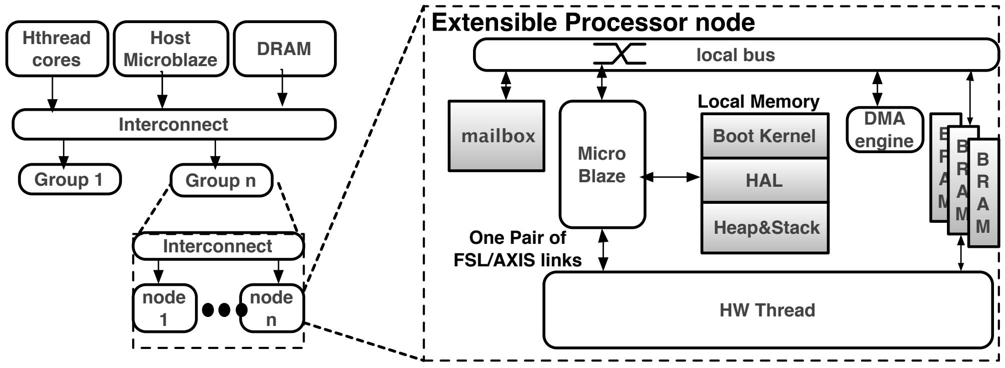


Figure 4.12: Right: The HEMPS platform. Left: Extensible Processor Node

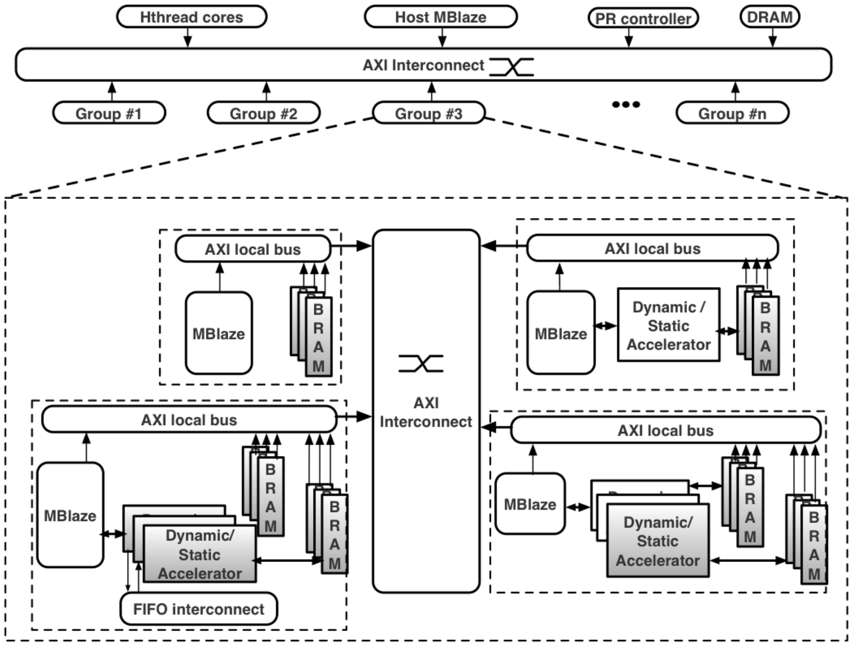


Figure 4.13: The overall architecture of a HEMPS system

4.2 Automating the generation of a HEMPs system

Fig. 4.12 shows how HEMPS system is built upon an Extensible processor node. Fig. 4.13 shows the overall architecture of a complex HEMPS system. The general architecture of the system is an Central AXI interconnect with the host processor, Hthread cores, DRAMs and other peripherals are connected. The nodes are connected to the rest of system by a two-tier AXI interconnect hierarchy. Each node can have its unique configurations. Fig 4.13 shows nodes with no accelerators, one accelerator, three accelerators with or without a FIFO interconnect.

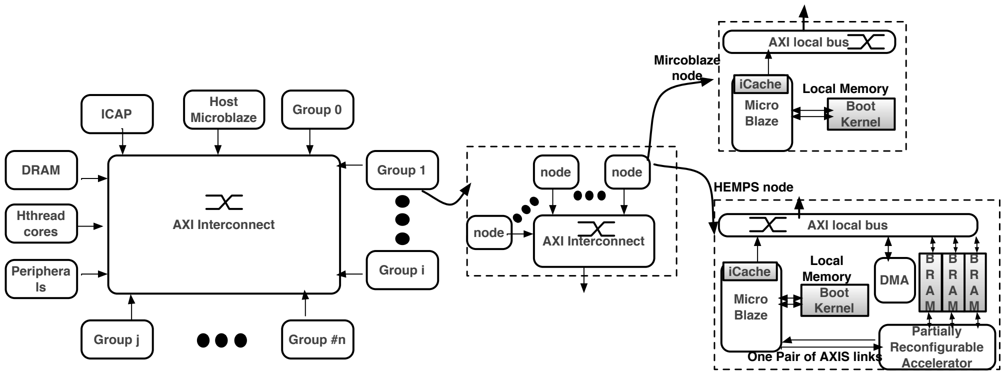


Figure 4.14: The high level view architecture of a HEMPS HW platform

Fig. 4.14 shows a general view of the HEMPS platform. There are some points worth mentioning here:

- There is two level of hierarchy to connect nodes. This is because AXI interconnect will not accept more than 16 masters/slaves.
- each node can be configurable as either a simple Microblaze with no accelerator, or a Microblaze with an attached accelerator and BRAMS.
- The top level AXI interconnect consists of Host processor, ICAP and Hthread cores. Hthread cores provide a low-jitter OS support for a multi-threaded application running on host processor. ICAP provides the support for dynamic reconfiguration within a HEMPs system which leads to the added flexibility of accelerating different portions of a thread executing on slave extensible processors. There is a two-tier hierarchy of AXI interconnect which supports up 16 groups with 16 nodes per group. The nodes in each group have faster access to each other's local memory and BRAMS. We currently build HEMPS systems with up to 36 extensible processor nodes on a Virtex VC707.

Figure 4.15 provides further details on how we included DMA engines for marshaling data between DRAM and BRAMs or ICAP. Each extensible processor can support a static or partially reconfigurable accelerator attached via standard FSL links. We defined a standard interface

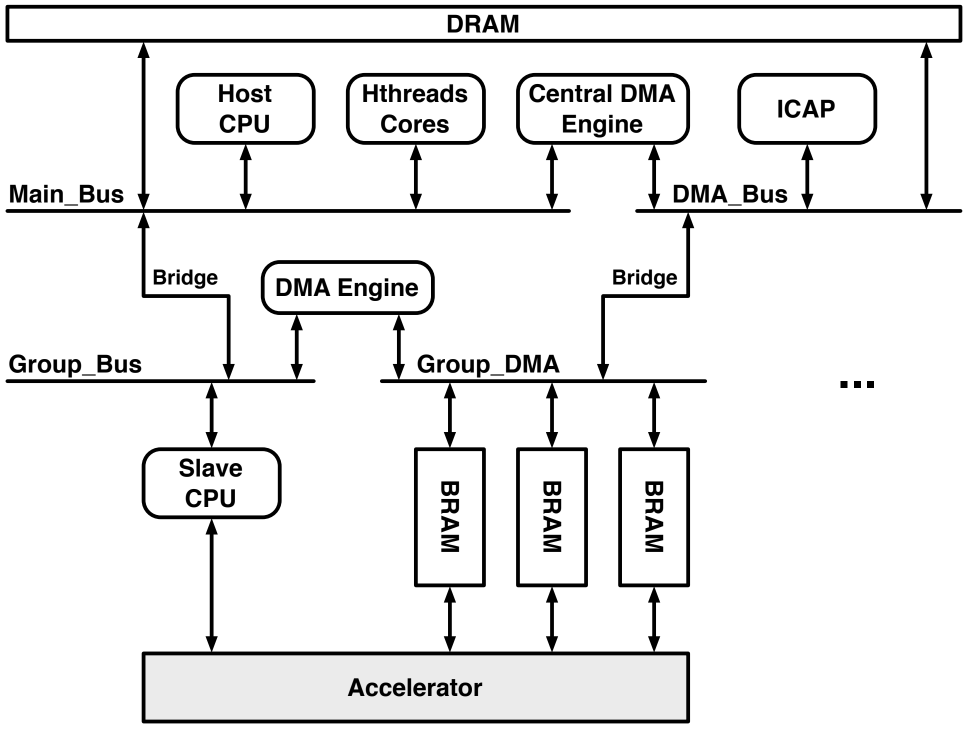


Figure 4.15: Node Architecture

connection for up to three dual port BRAM blocks (two data input, one data output) for the an accelerator, as well as a dedicated DMA engine for transferring data between BRAMs and DRAM. Our current HEMPS system contained a central DMA engine that can be directly accessed by all extensible processors. This can be used for marshaling data into and out of the accelerator as well as for transferring partial bitstreams into the ICAP at the speed of 96 Mbytes/sec. If the ICAP is directly mastering the bus, the transfer rate can go up to 232 Mbytes/sec on Xilinx Virtex 6 [43]. Feeding the accelerator the data they need has a big impact on performance of the system, since DMAing data is the sequential part of the code that can not be parallelized, and therefore Amdahl's law limits the speed up we can achieve through TLP or DLP.

Fig. 4.16 shows an example of a HEMPS system with 6 nodes, and each node has a partially reconfigurable region. Each HEMPs system provides Internal Configuration Access Port (ICAP) driver support for PR. Theoretically, the ICAP can consume bitstream data up to 100Mhz with data width of 32 bytes resulting in 400MBs transfers rate if the data is stored into BRAMs, which

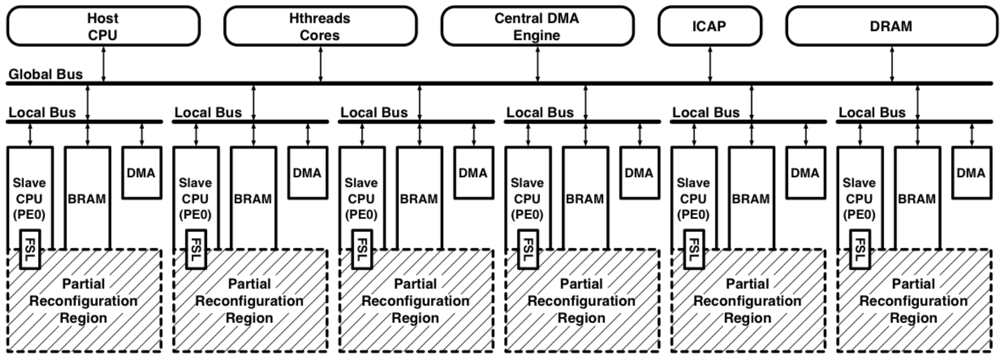


Figure 4.16: A HEMPS system with 6 nodes

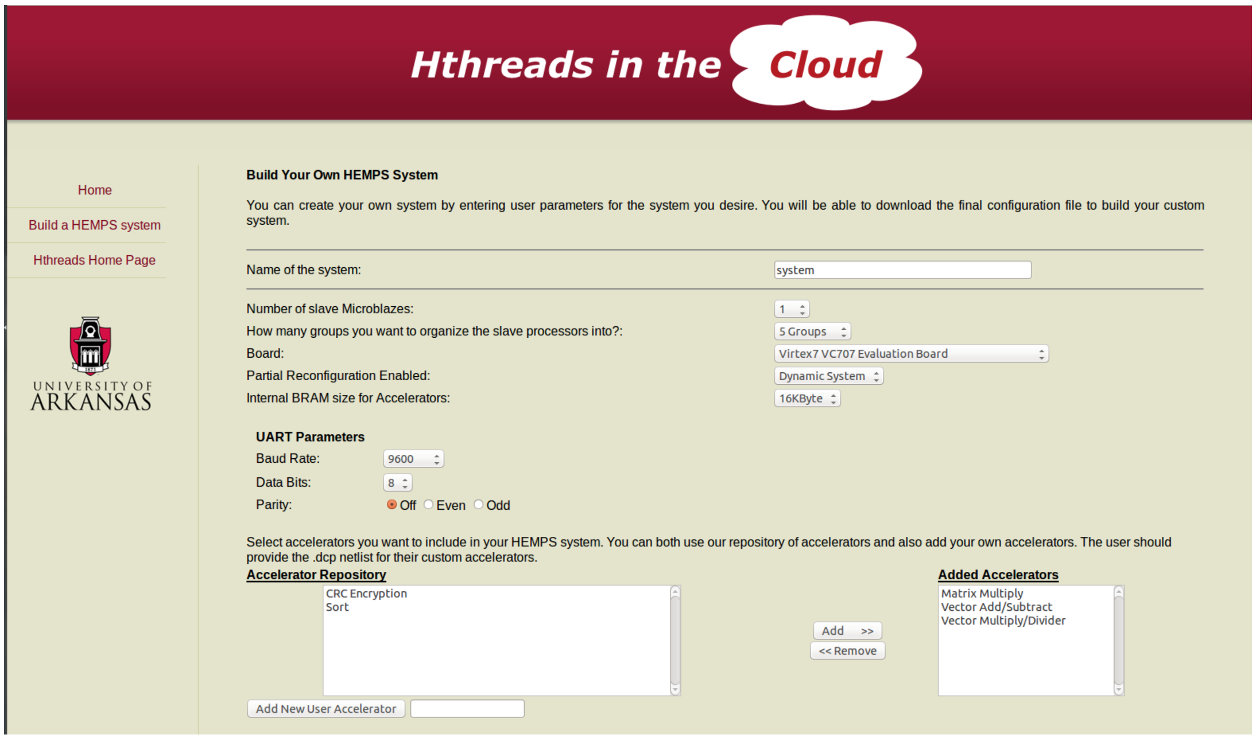
is not feasible. Partial reconfiguration speed can achieve speeds up to 232 MB/s when the ICAP is directly attached to the Multi-Port Memory Controller (MPMC) on a Xilinx Virtex 6 board [43].However, we just used a DMA engine to transfer partial bitstreams into ICAP, reaching 96 Mbytes/Sec.

The next step was to write a TCL script called Archgen that inputs a high level specification file and spits out the final bitstream of a HEMPS system defined by the parameters in that file like: The number of nodes, the type of accelerators, the size of internal BRAMs, etc. The Archgen is based on a HEMPs node, which is a flexible extensible processor node. The final HEMPS systems is built by specifying how many of these nodes needed and the customization for each node.

Our tool-chain takes the user provided configuration file shown in Fig 4.17, and invokes the Vivado tools to build the HEMPS platform. Also, users can use our GUI interface in the cloud at http://hthreads.csce.uark.edu/ARCHlang/prPages/hemps.html to generate this configuration file. There are some general parameters like the number of the nodes, the target board and the frequency of the system. Then all the nodes in each group are being customized. This includes the general specifications like the type of the processor, the configuration of the processor like Barrel shifter,

```
No. Nodes                    :2
Board                        :vc707
Frequency:                   :100 MHz
======================Group0===============================
            Type   Bsh/Mul/Div/FPU   ICACHE-Size   No-Accelerators  Fifo-interconnect
Node0:   MB     1/0/0/0               8k              3                Y

                    PR   Default-ACC     BRAMS:No/size  FIFO_ports:In/Out        PR-moudules
    ACC0:           Y                           3/4k                1/1
['FFT', 'bubblesort']
    ACC1:           N     'matrix-m'            3/16k               3/3
[                   ]
    ACC1:           Y                           2/8k                0/0
['crc', 'bubblesort']
----------------------------------------------------------------------------------
Node1:      Type   Bsh/Mul/Div/FPU   ICACHE-Size   No-Accelerators  Fifo-interconnect
             MB       0/0/0/0            4k              0              0

             PR   Default-ACC     BRAMS:No/size  FIFO_ports:In/Out        PR-moudules
    ACC0:    N      'Blank'            3 /16k           0/0               [       ]

----------------------------------------------------------------------------------
            Type   Bsh/Mul/Div/FPU   ICACHE-Size   No-Accelerators  Fifo-interconnect
Node2:     MB       1/1/1/1            16k              1              0

                    PR   Default-ACC     BRAMS:No/size  FIFO_ports:In/Out        PR-moudules
    ACC0:           Y                          3 /16k               0/0
['matrix-m','crc', 'vectoradd','vectormul']
.........

======================Group1===============================
........
```

Figure 4.17: The high level specification config file


Hthreads in the Cloud

Build Your Own HEMPS System

You can create your own system by entering user parameters for the system you desire. You will be able to download the final configuration file to build your custom system.

Name of the system:	system
Number of slave Microblazes:	1
How many groups you want to organize the slave processors into?:	5 Groups
Board:	Virtex7 VC707 Evaluation Board
Partial Reconfiguration Enabled:	Dynamic System
Internal BRAM size for Accelerators:	16KByte

UART Parameters

Baud Rate:	9600
Data Bits:	8
Parity:	[✓ Off] Even Odd

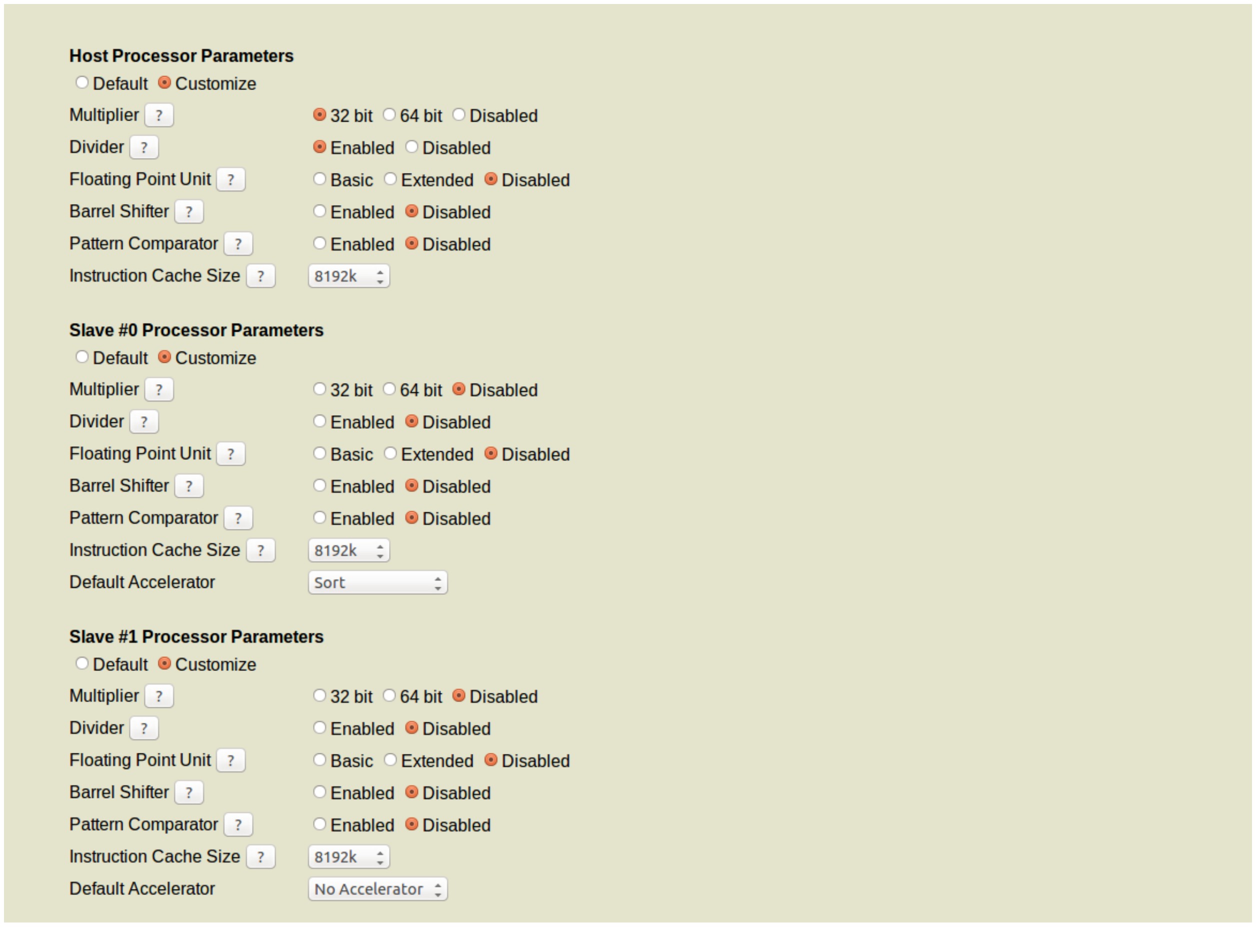
Select accelerators you want to include in your HEMPS system. You can both use our repository of accelerators and also add your own accelerators. The user should provide the .dcp netlist for their custom accelerators.

Accelerator Repository	Added Accelerators
CRC Encryption	Matrix Multiply
Sort	Vector Add/Subtract
	Vector Multiply/Divider

Figure 4.18: Archgen in the cloud

multiplier, divider and floating point unit and size of ICache. Also the number of the accelerators and if there is FIFO interconnect needs to be instantiated. After this, each accelerator can be customized to determine if it is partially reconfigurable, what is the size and number of its BRAMS and the number of in/out FIFO ports. If the accelerator is static then the default accelerator should be specified, otherwise the PR-modules should be listed so that during the PR flow process, the partial bitstreams for this region gets generated. If the Microblaze does not have any accelerator, then a blank accelerator will be instantiated with the desired number of BRAMs. This is necessary because the Microblaze does not have the Dcache enabled and therefor without having local BRAMS the performance is very low. Some screenshots of th GUI interface are shown in Fig. 4.18 and Fig. 4.19, which will result in the final the config.txt file.

We use PR controller IP which centralizes the bitstream transformations and it can reach up to 400 Mbyte/Sec for PR data transfer in Virtex 7 tools using PRC IP provided by Xilinx. The toolchain generates the final full bitstream plus a header file containing of all the partial bitstreams for the SW application to do partial reconfiguration.



Host Processor Parameters

Default [✓ Customize]

Multiplier ?	[✓ 32 bit] 64 bit Disabled
Divider ?	[✓ Enabled] Disabled
Floating Point Unit ?	Basic Extended [✓ Disabled]
Barrel Shifter ?	Enabled [✓ Disabled]
Pattern Comparator ?	Enabled [✓ Disabled]
Instruction Cache Size ?	8192k

Slave #0 Processor Parameters

Default [✓ Customize]

Multiplier ?	32 bit 64 bit [✓ Disabled]
Divider ?	Enabled [✓ Disabled]
Floating Point Unit ?	Basic Extended [✓ Disabled]
Barrel Shifter ?	Enabled [✓ Disabled]
Pattern Comparator ?	Enabled [✓ Disabled]
Instruction Cache Size ?	8192k
Default Accelerator	Sort

Slave #1 Processor Parameters

Default [✓ Customize]

Multiplier ?	32 bit 64 bit [✓ Disabled]
Divider ?	Enabled [✓ Disabled]
Floating Point Unit ?	Basic Extended [✓ Disabled]
Barrel Shifter ?	Enabled [✓ Disabled]
Pattern Comparator ?	Enabled [✓ Disabled]
Instruction Cache Size ?	8192k
Default Accelerator	No Accelerator

Figure 4.19: Archgen in the cloud: Customizing the nodes

The tool will chose the PR region based on the biggest accelerator assigned to that region. For each PR regions, there are 3 PR region candidates: Small, medium, and big. For example, Matrix Multiply and Quicksort are bigger than other accelerators like CRC or VectorADD. This helps with the final resource usage and partial reconfiguration overhead.

4.3 Polymorphic functions library

Polymorphic functions provide the needed separation of policy and mechanisms for next generation chip heterogeneous multiprocessors. Polymorphic functions run on Extensible processors. We try to reinstate application portability over CHMPs architectures. Reinstating application portability over CHMPs systems cannot be achieved through incremental advancements in just programming languages, or programming models, or runtime systems, or architecture support. Reinstating portability will require new methods and interactions between abstractions and runtime mechanisms. From the users perspective we retain the semantics of the familiar multithreaded programming model. Figure 4.20 shows the SW toolchain for polymorphic functions.

Previously, runtime tuning within Hthreads was limited to the scope of processor and/or standalone custom hardware availability. With the introduction of a tunable accelerator library, I am expanding this scope to include a scheduled thread to execute in both hardware and software. This has the benefit for a slave processor to adapt to its hardware features when executing an accelerator library call. This allows the runtime system tune the program based on the availability of software and hardware heterogeneous resources. It is worth mentioning that the learning occurs transparently to the programmer.

From the application developers perspective, polymorphic functions are simply linkable libraries of self tuning functions. Application designers no longer need to hand partition, profile and tune the functions for each platform or variances in runtime behavior. The functions are tuned and scheduled by the operating system across optimal sets of heterogeneous resources and accelerators. A small lightweight operating system running on each slave processor can independently determine if an attached accelerator is present or dynamically loadable, and if the accelerator should be

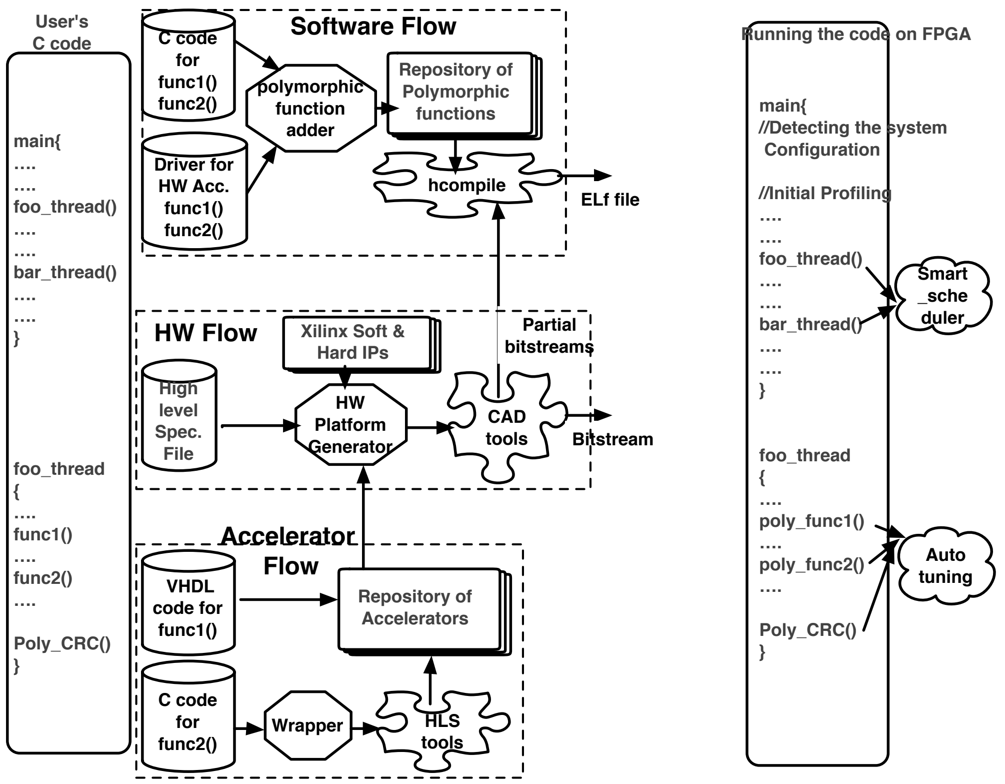


Figure 4.20: SW toolchain for polymorphic functions

used. Accelerator presence and usage information is maintained on each slave processor. Fig. 4.20 shows how the user's C code (on left) will go through the chain flow to change to the code running on FPGA which uses polymorphic functions(on the right)

Polymorphic functions allow *data-level parallelism* (DLP) to occur within the existing multi-threaded programming model. They are partitioned by the Polymorphic Function Partitioner that runs autonomously on all slave processors. Whereas the dispatching of threads is centralized on the host, the decision on how to partition a polymorphic function is made autonomously on each slave processor. The operating system running on each slave processor makes localized optimization decisions thus eliminating the bottleneck that can occur when all decisions must be made in a centralized fashion on the host processor.

We create libraries that abstract the need for programmers to be aware of processors/accelerators and how they are configured within a particular heterogeneous platform. Each library contain both a software and hardware method of a function. A small and efficient runtime adaptive resource aware scheduler running on extensible processor will tune this polymorphic functions for each node. That means during runtime these functions can run in different ways based on available resources and profiling information. The small operating system can dynamically load and run accelerators during the execution of the thread, transparent to the user.

Profiling describes the collection and utilization of performance data during runtime. Profiling is needed here to provide more efficient use of the hardware, as well as to provide more adaptable runtime decisions. To address this, a shared table was created that stores profiling data. The runtime system stores profiled information in the *profiling table*. This table is used by any polymorphic call to *tune* the system to fit an application's needs. The profiling table captures prior execution timing information for polymorphic functions given a certain data size. When invoked, polymorphic functions utilize this information to make better runtime decisions on whether to execute in hardware or in software. Figure 4.21 shows a 2D structure of the profiling table. For each polymorphic function, the table stores an entry for data sizes (order of 2) between 64 - 4k bytes. Parameter sizes passed to polymorphic functions are rounded to the nearest data size within this table. Each entry in the table contains software and hardware execution times. Additionally, an optimal division factor (chunks) is recorded that allows processors to pipeline DMA transfers with accelerator computation. This table is initialized by the host at system boot and might be updated by slave processors during runtime to increase accuracy. Polymorphic functions are generically written, using runtime profiling information to decide whether to bootstrap to the user provided function source code or its hardware variant.

An example of a multi-threaded application which uses polymorphic functions is shown in Fig. 4.22. The polymorphic function *poly_vector()*, in Figure 4.22 shows how slave processors utilize the profiling table. Initially, slave processors determine whether to use an accelerator attached to it through the function, *isHW()*. There, software and hardware execution times for the given polymorphic function are compared. If the slave processor has a dynamic accelerator, the overhead of PR is also included into the comparison. A boolean value is returned indicating whether to perform the function in either software or hardware. For dynamically attached accelerators,

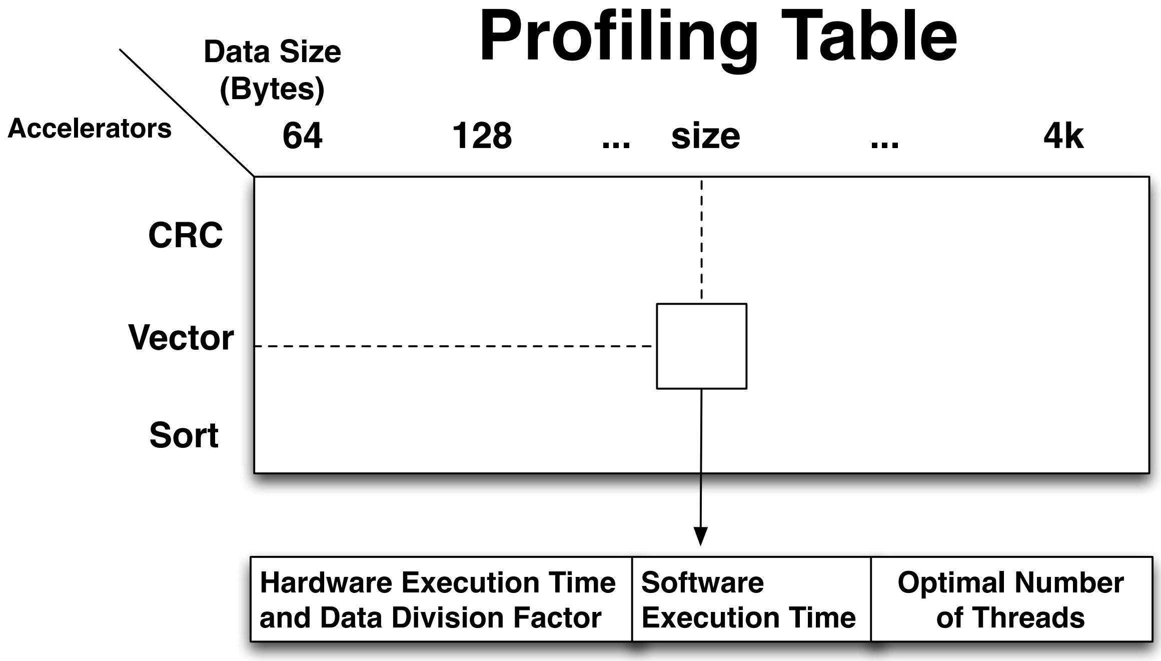


Figure 4.21: Profiling table

isHW() also performs PR if returning true. If the decision to execute in hardware is determined, processors additionally inquire whether to divide the passed data set into chunks as defined by the data division factor, to overlap data transfer in chunks with accelerator computation to increase performance.

Fig. 4.23 shows how the thread running on an extensible processor executes a polymorphic function. For dynamically reconfigurable systems, slave processors decide whether to swap out the existing attached accelerator for the appropriate one by comparing software execution time and the sum of both hardware execution time and PR overhead for the given accelerator library call. If the slave processor can swap out the existing attached accelerator, it additionally quantifies whether software execution time is greater than the sum of both hardware execution time and PR overhead for the given accelerator library call. If so, the slave processor can perform PR autonomously without any remote procedural calls (RPC) to the host processor. Access to the PR hardware (ICAP) is synchronized through a global mutex all processors share in the system. When BRAM space is limited,the division factor field in the profiling table suggests to slave processors

```
int main() {
  init_profiling_table();
  thread_create(foo_thread);
  thread_create(bar_thread);
  ....
}
void * foo_thread() {
   poly_vector(&a,&b,&c,size,ADD);
   poly_crc(&data,size);
   poly_sort(&data,size);
}
void * bar_thread() {
   poly_sort(&data,size);
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Polymorphic Function
void poly_vector( ... ){
   bool use_accelerator=isHW(); //Performs PR as well;

   if (use_accelerator) {
      divide_data_into_chunks();
      for ( i < # of chunks ) {
         dma_data(chunks[i]);
         hw_vector(chunks[i]);
      }
   }
   else
      sw_vector();
}
```

Figure 4.22: A Multi-threaded application with polymorphic functions

an optimal size it can overlap data argument marshalling with hardware accelerator computation through the use of a DMA engine. This autonomous control increases data-level parallelism in the system and introduces other opportunities for parallelism and computation efficiency.

Polymorphic functions extend the semantics of a thread body to also support finer grained data level parallelism that can be exploited on custom hardware accelerators. We preserve the Pthreads notion of a linear address space . We provide the definition of a polymorphic function that can be used with the body of a thread function which can then be decomposed into different combinations of software and hardware components based on run time parameters. The approach also supports backwards compatibility and will run standard Pthreads programs. Thus our approach brings portability by allowing users to write threads as if they were running on a traditional system with homogeneous processors and a linear address space. These steps were taken to implement this new programming model:

1. Both HW version and SW version of functions like:SORT,VECTOR and CRC, Matirx Mul-

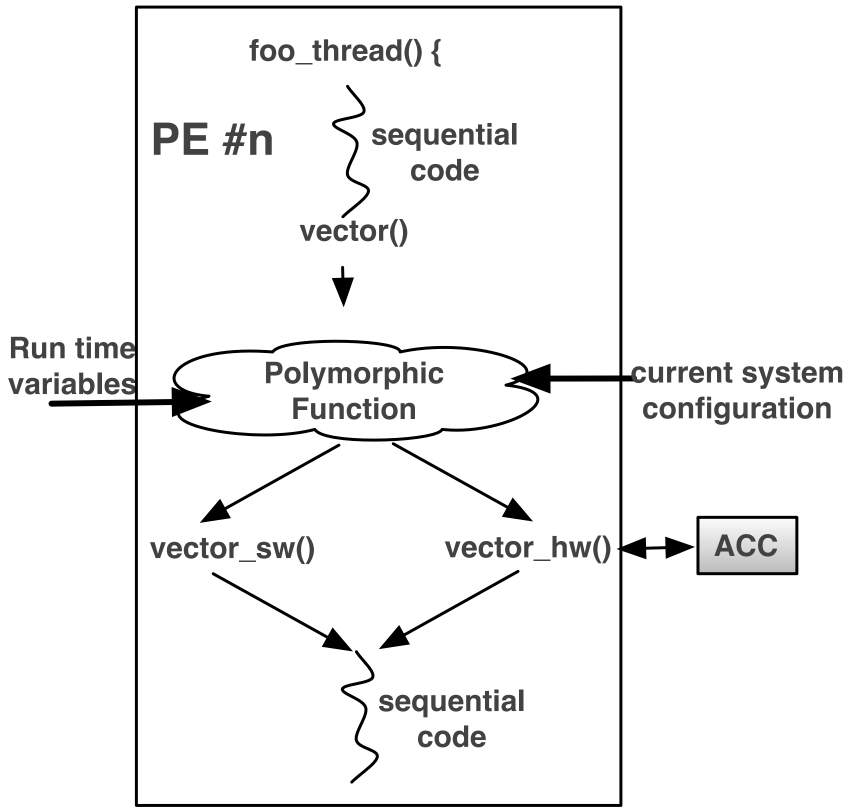


Figure 4.23: The thread running on PE #n calls a polymorphic function.

tiply, IDEA encryption were written. For HW version, It's either written in FSM like the one shown in Fig.4.24 or I used Vivado HLs like Fig.4.25.

2. The number of the library calls that we can add to our repository is not limited by anything. So, the user has the option to add their own polymorphic functions (both HW or SW), or find the candidate functions in their application and run it through HLS to add new polymorphic functions to the repository.

3. *Polymorphic functions who call other polymorphic functions*: One of the advantages of having standard polymorphic function is that they can be used to build more complex functions. So, more complex polymorphic functions will call the simpler ones. This increases productivity.

4. Some new entries were added to Virtual Hardware Thread Interface (VHWTI) BRAM, so that during boot time, the extensible processors should indicate whether the attached acceler-

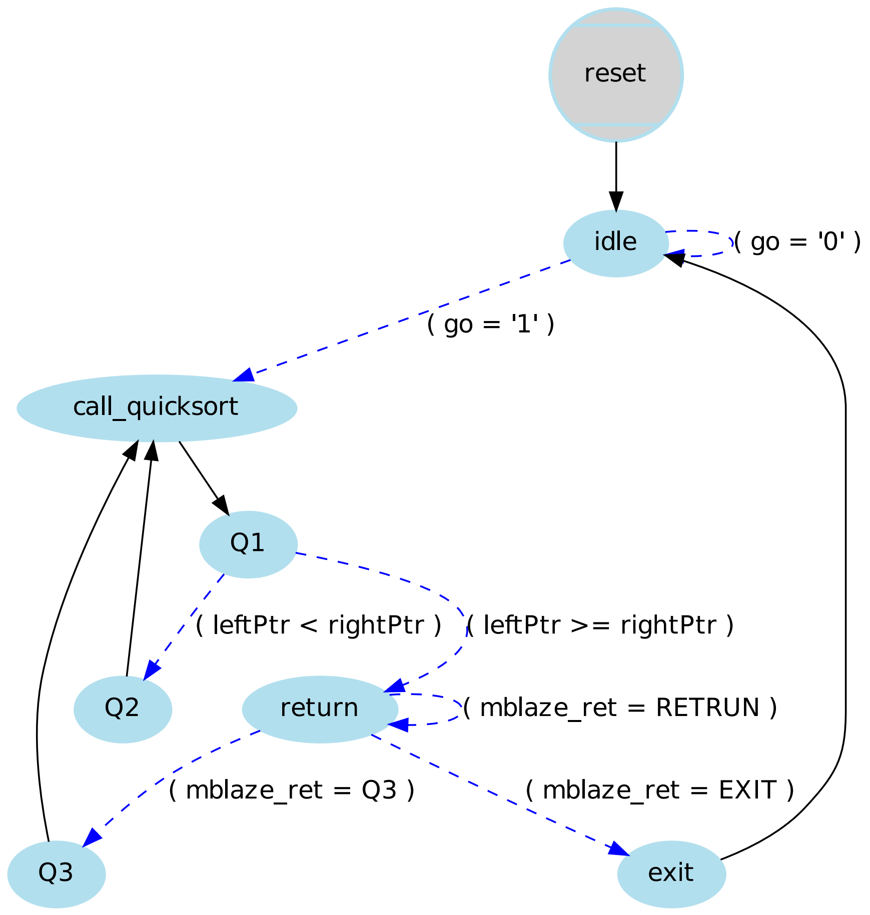
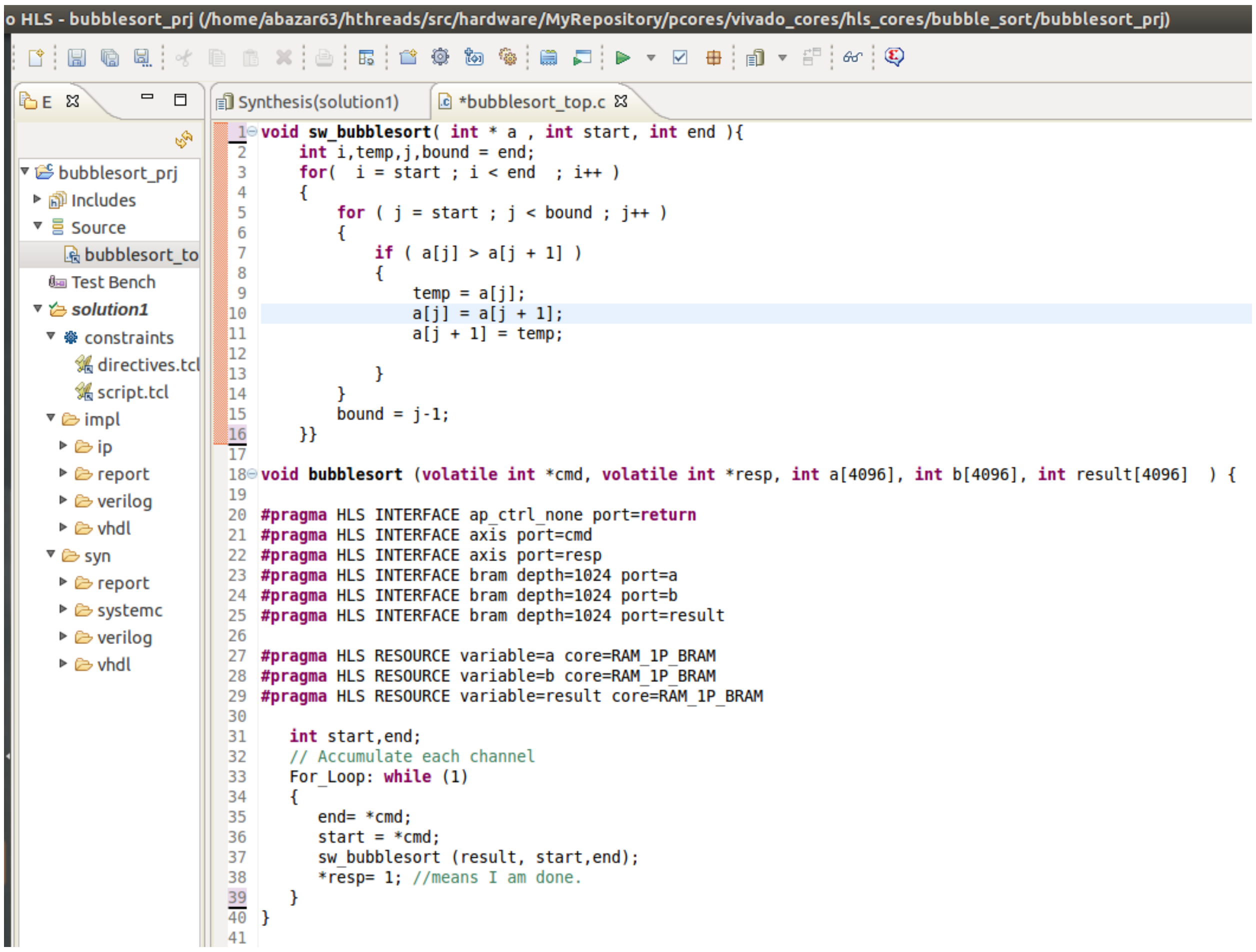


Figure 4.24: Designing an Accelerator using FSM

ator is partially reconfigurable or not (if any). This information will be written into VHWTI to be read by host processor. Also, Each slave processor is equipped with a local allowing it to profile accelerator function calls, whether it be in software or hardware.

5. Before the host starts executing the user's code, the profiling table which contains SW and HW execution times for each polymorphic function will be filled . For each polymorphic function, it runs the function with various data sizes to gets the SW time and HW time for different chunks (to find the best DMA overlapping factor).

6. The compilation flow was modified to produce multiple implementation of each polymor-



```
void sw_bubblesort( int * a , int start, int end ){
    int i,temp,j,bound = end;
    for(  i = start ; i < end  ; i++ )
    {
        for ( j = start ; j < bound ; j++ )
        {
            if ( a[j] > a[j + 1] )
            {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;

            }
        }
        bound = j-1;
    }}

void bubblesort (volatile int *cmd, volatile int *resp, int a[4096], int b[4096], int result[4096]  ) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=cmd
#pragma HLS INTERFACE axis port=resp
#pragma HLS INTERFACE bram depth=1024 port=a
#pragma HLS INTERFACE bram depth=1024 port=b
#pragma HLS INTERFACE bram depth=1024 port=result

#pragma HLS RESOURCE variable=a core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=b core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=result core=RAM_1P_BRAM

   int start,end;
   // Accumulate each channel
   For_Loop: while (1)
   {
      end= *cmd;
      start = *cmd;
      sw_bubblesort (result, start,end);
      *resp= 1; //means I am done.
   }
}

```

Figure 4.25: Designing an Accelerator using Vivado HLS

phic function. A SW version which runs on Microblaze, and a HW driver to control the accelerator implementing that function.

7. A light weight operating system running on slaves was developed which decomposes each thread into components that run across the software/hardware boundary. This light weight OS running on the slave processors exercise additional autonomy, making further localized scheduling decisions. It autonomously refers to the profiling table to choose the efficient implementation of the polymorphic function within the thread, based on runtime variables and system configuration at the moment.

8. Extensible processors store the information of the thread they are running as well as their attached accelerator to help host processor make better decisions when assigning the threads

to slaves. For this, entries in Virtual Hardware Thread Interface (VHWTI) BRAM is used, to record the polymorphic functions call made within the thread , as well as currently attached accelerator (if any). All of these updates occur concurrently with other slave processors and allows the host processor to use this updated information immediately when scheduling threads onto slave processors. During runtime, slave processors update those entries in VHWTI in order to reflect any of its resource changes. In the case of PR, processors that swap in a new accelerator will update the corresponding VHWTI entry. Unlike a processor with a reconfigurable slot, for a processor with the static accelerator the currently attached accelerator will not change.

9. The first time a thread or library function is executed nothing is known about their resource requirements. When the thread or function executes, the HAL running on the slave processor should start the process of cataloging key information about execution behavior and resource usage, specifically the type and frequency of the polymorphic functions called within the thread. When the function is called again (typically within a loop) the system level thread manager walks through all free processors checking their HAL to see if this was the last and hence still resident accelerator connected to any processor.

Chapter 5

Results

This section provides the results to verify the ability of our HW/SW co-design to re-enable the notion of writing the code once and run it on any HEMPS platform with extensible processor nodes. I also show how profiling can increase resource utilization and provide increased performance. The same program without modification can be efficiently scheduled and run on different HEMPS systems that contain only general purpose processors, mixes of processors and static accelerators, and mixes of processors and partial reconfiguration slots.

5.1 Extensible processor node Verification

The extensible processor node was originally implemented using Xilinx 12.3 ISE tools on a ML605 using the older PLB bus for interconnect and FSL bus for point-to-point connection between Microblaze and HW thread. The current version has been updated using the Vivado 2014.4 tools on a VC707 using the AXI interconnect and AXIS streaming interface. The measured data transfer rates between DRAM and BRAMS were 94 Mbytes/sec and 395 Mbytes/sec across the PLB and AXI4 buses respectively (at 100MHz). The higher transfer rate on the AXI4 bus is a result of separate read/write data buses, and a higher external memory bandwidth for 7 series Xilinx FPGAs. Also the partial reconfiguration speed is 96 Mbyte/Sec on Virtex 6 and 380 Mbyte/Sec on Virtex 7. The higher bandwidth on Virtex 7 goes back to both AXI bus protocol that enables reading and writing at the same time on the bus, as well as a new soft IP named Partial Reconfiguration Controller (PRC for short) that bursts the data from DRAM to ICAP memory directly.

5.1.1 HAL comparison

Table 5.1 shows a qualitative comparison of the HAL functionality between our extensible processor approach and other research efforts in this area including FUSE [39], Redsharc [41], SPREAD [65],

Project	HW circuit	Data Interface	HAL implementation	Stack mgt	OS and RPC services	HW/SW partitioning
FUSE [39] ReconOs[45] SPREAD[65]	HW thread	Fixed	FSM HW-based	No	Centralized	Coarse
HWTI[15]	HW thread	Fixed	FSM HW-based	Yes	Distributed	Coarse
Redsharc [41]	Accelerator	General	Thin Wrapper	No	Centralized	NA
Extensible processor	Accelerator	General	Software MicroBlaze-Based	Yes	Distributed	Coarse Fine

Table 5.1: Comparison of the HAL functionality

Name	Number of FF	Number of LUT	Number of BRAM
FUSE	–	–	–
HTI in SPREAD[1]	723	706	2
OSIF in ReconOS	1430	3043	3
HWTI	947	4399	4
Mblaze[2]	1395	1100	3

Table 5.2: Comparison between resource utilization of different implementation of HAL.

ReconOS [45], and the Hthreads Hardware Thread Interface (HWTI) [15]. In RedSharc [41] the accelerators are being accessed by a single processor, and there are no hardware threads. Among all HALs, Extensible processor is the only one with the ability to have both Coarse and fine grained HW/SW partitioning. Moreover, FUSE, ReconOS and SPREAD do not provide stack management and distributed OS and RPC services for the accelerator, which is available in both HWTI and extensible processor.

Table 5.2 shows a comparison of resources used for several common HAL implementations. On average, replacing the HAL with a Microblaze resulted in a 60 reduction in LUTs . Table 5.3 from [19] provides insight into the resource requirements for implementing several key system calls as state machines. Adding additional calls would increase the size of the HAL. Additional system calls can be added into the processor with no additional hardware resource penalties.

Hthreads Call	Slice Count
create	401
join	356
self	13
equal	69
exit	0
mutex_lock	73
mutex_trylock	66
mutex_unlock	54
cond_signal	115
cond_broadcast	117
cond_wait	197

Table 5.3: Size of selected HWTI system calls [19]

Service Call Latencies

Table 5.4 compares the individual latencies of key service calls. Latencies for reading and writing local variables and manipulating the stack increased between 10 and 19 clock cycles with an average increase of 13 clock cycles. This is to be expected when services are implemented in general purpose software routines instead of dedicated hardware. The system calls shown in Table 5.4 increased between 12 and 208 clock cycles, with an average increase of 72 clock cycles. The large variance results from different levels of processing complexities necessary to implement each system call. The mutex_lock and mutex_unlock calls as well as thread_exit averaged an increase of 15 clock cycles. These calls, specifically mutex_lock and mutex_unlock would be heavily used if the HW thread needed to form a critical region and exchange data with other processors and HW threads. Clearly, how these increased latencies effect overall performance depends on their frequency of use. The 15 clock cycle increase represents a 10 increase in latency overhead compared to the HWTI. The cond_signal and cond_wait averaged an increase of 156 clock cycles.

Accelerator Request	HWTI cycles	Mblaze cycles	CRC& VectorADD	QuickSort
Read	5	15	0	4072
Write	3	17	0	4076
Pop	7	21	0	8147
Push	3	22	0	8147
Declare	3	15	0	4074
Call	5	20	0	4073
Return	9	20	0	4073
mutex_lock	144	161	0	0
mutex_unlock	144	156	0	0
cond_signal	134	235	0	0
cond_wait	342	550	0	0
thread_exit	3	21	1	1

Table 5.4: Comparison of system services latencies.

5.1.2 Performance

Figure 5.1 shows the two test platforms were built to evaluate the functionality, size, and performance of the extensible processor node Vs. HWTI. The first platform contained the hthreads HWTI to service custom HW Quicksort). The second platform replaced the HWTI with an extensible processor node. Both systems were built using the Xilinx ISE Design Suite v12.3, synthesized, and run on a Xilinx Virtex6 (ML605) Evaluation Board. Both systems included the hthreads microkernel OS.

To make fair comparisons in performance and size, three BRAM's were connected to the accelerator in both systems. A dedicated DMA engine was provided for performing data transfers between DRAM and the BRAMs. Performance and area comparisons of the interfaces as well as the accelerators were then made between the two systems.

Benchmark Accelerators

We adopted the same three accelerators originally used in Anderson [15] to evaluate the hthreads HWTI. The three accelerators implemented were quicksort, CRC encryption, and vector add. Our objective was to perform fair and unbiased comparisons between the HWTI and our extensible

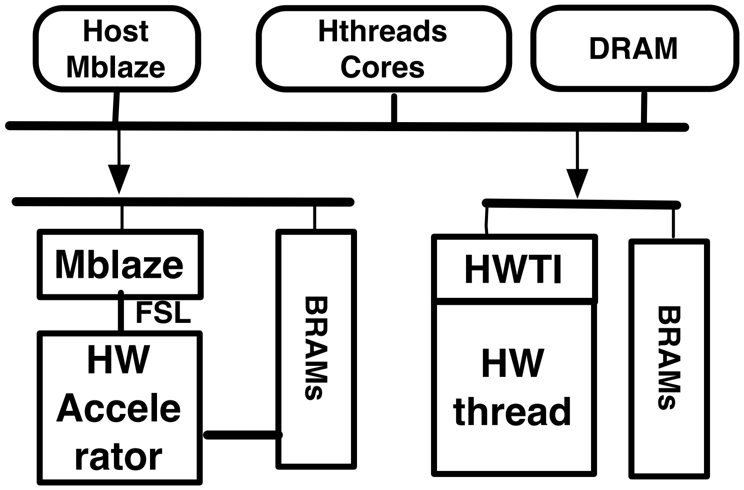


Figure 5.1: The test platform built for evaluation

processor replacement and not the optimality of the accelerator implementations. Thus we did not alter or optimize the descriptions of the accelerators. The only required changes were to the interfaces as discussed below.

The hthreads HWTI contained interface registers (op_code and argument registers) written to by the user_logic to request services. The VHDL signal assignment statements used to write these registers were retargeted to write into our FSL link replacements. No change was made to the order or number of system service requests.

An interesting and potential advantage of using an extensible processor to replace a hardware based HAL is the ability to reduce the resources synthesized into the accelerator. This can be accomplished by moving sequential code out of hardware and into software. We also experimented with this type of migration to evaluate both the positive and negative effects on the size and performance of the accelerator.

Next, we compared the performance and area usage of Extensible processor compared to HWTI [15].

Accelerator	HAL	FFs	LUTs
CRC	HWTI	246	492
	MBlaze	92	237
VectorAdd	HWTI	331	507
	MBlaze	210	260
Quicksort	HWTI	304	1054
	MBlaze	378	915

Table 5.5: Resource Utilization of Accelerators

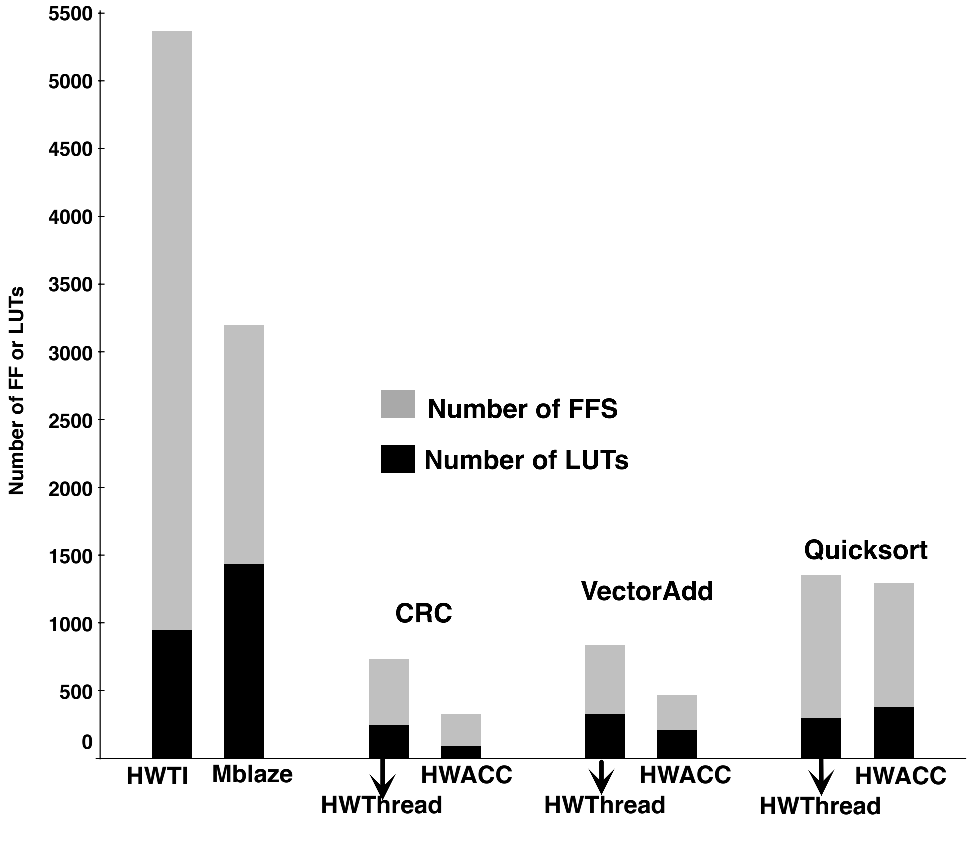


Figure 5.2: Resource utilization comparison .

Size and Area Comparison

Table 5.5 and Figure 5.2 show the size of the accelerators was reduced using the extensible processor. The accelerator resources were reduced on average by 110 FFs and 200 LUTs. Resource savings result from the lighter interface of FSLs and a ”thinning” of the hardware wrapper for the extensible processor compared with HAL. Moving the sequential portions of the thread into software only resulted in an average reduction of 22 LUTs, and 3 FFs for these accelerators. This modest reduction is a result of the parallel structure of the accelerators; only a small section of sequential code was executed at the startup of the accelerators. This size reduction is really important factor if one wants to do Partial Reconfiguration during runtime. In the case of our CRC, Vector_add and sort HW circuits, The small HW accelerators which are ripped off of their sequential parts and moved to the MicroBlaze takes about 500 us to swap in and out, while this time is around 1300 us for HW threads.

Effects on Accelerator Performance

Figures 5.3, 5.5, 5.4 and 5.6 compare the execution times of each of the three benchmark accelerators. We recoded each accelerator in software and measured the execution time on the MicroBlaze for comparative purposes.

The CRC and Vector add yielded interesting results. Both accelerators did not use any stack operations and only a single system call (thread_exit). As expected the system call latencies shown in Table 5.4 had no negative effect on performance. What was not expected was the 40 performance increase observed. Analysis showed these performance increases resulted from the ability to overlap, or pipeline, data transfers issued within the portion of code running on the MicroBlaze with the computations running within the accelerator. The state machine controller in the HWTI could only idle the accelerator during the transfer of data.

As a proof of concept, Quicksort was implemented as a HW thread to show use of the stack and state machine versions of push, pop, call and return built into the hthreads HWTI. For example,for data size of 2k words, 4072 reads, 4076 writes, 4042 declares, 8147 push and pops, and 4074

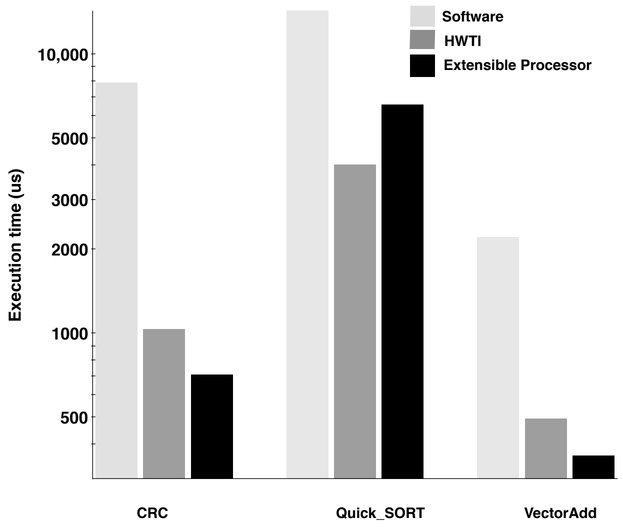


Figure 5.3: Execution time for SW threads, HW threads and extensible processor

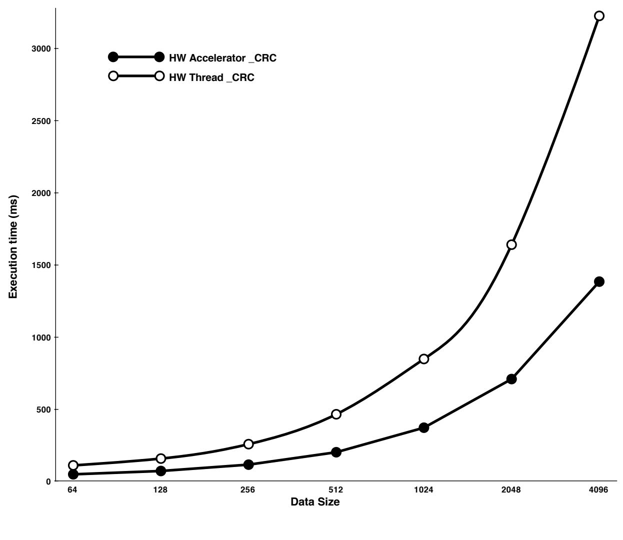


Figure 5.4: CRC HW thread Vs. CRC hw accelrator

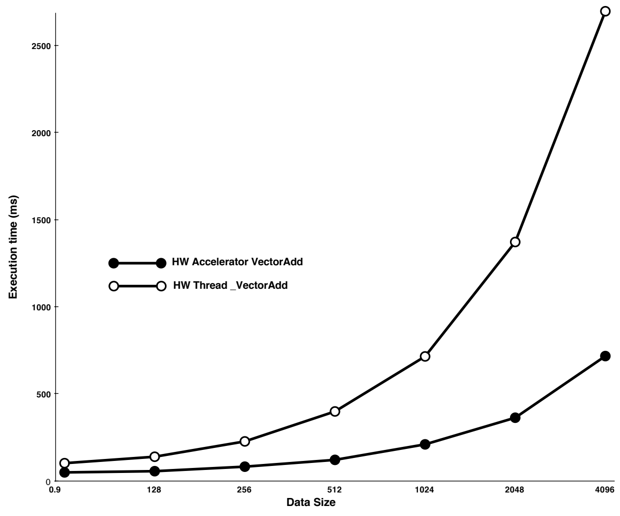


Figure 5.5: Vector HW thread Vs. CRC Vector accelrator

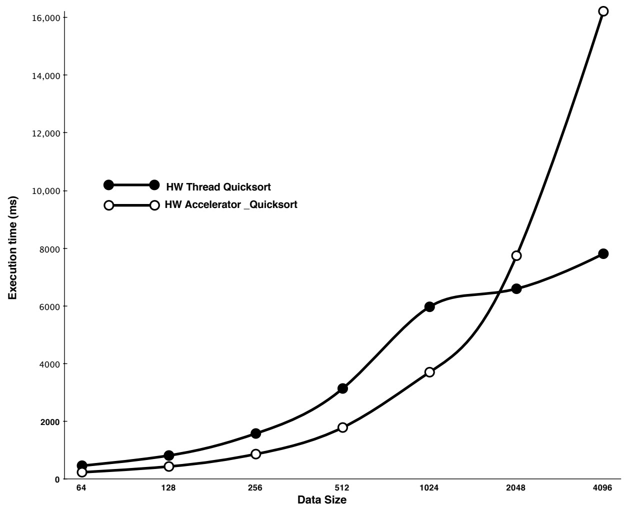


Figure 5.6: Quicksort HW thread Vs. Quicksort hw accelrator

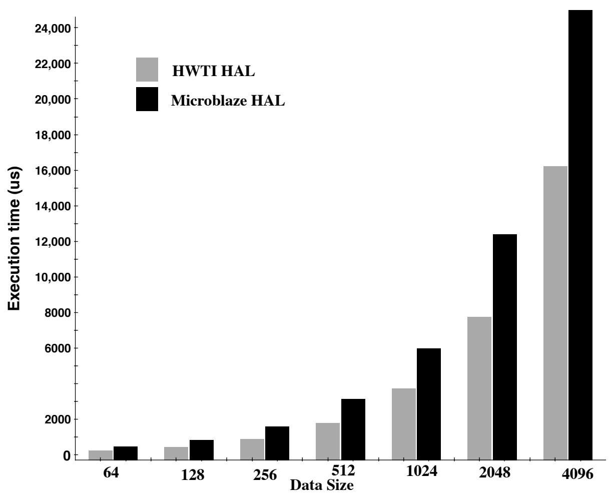


Figure 5.7: Performance comparison for Quicksort HW thread

calls and returns were made by Quicksort HW thread. The average latency difference of 13 clock cycles shown in Table 5.4 for these operations clearly decreased overall performance of Microblaze HAL , as shown in Fig. 5.7. This represents the real cost of the programmability provided by the extensible processor.

Analysis Summary

The analysis yielded both expected and unexpected results. The objective of replacing the hardware HAL with a programmable processor was to increase the flexibility and productivity of updating system services, and reduce resource requirements by allowing sw/hw partitioning to occur within and not at the thread level of granularity. It was fully expected that some level of performance would be lost by replacing custom circuits with a programmable general purpose processor. The results showed cases where the programmable processor approach yielded higher performance than the hthreads custom hardware HWTI. In extreme cases as evidenced by quicksort, performance degradation can be expected.

Table 5.6 list the optimization methods provided to three different HW circuits (used in this paper) by MicroBlaze.

HW ciruit	Stack Management	Pipelining of Operations
CRC	–	yes
VectorAdd	–	Yes
Quicksort	Yes	–

Table 5.6: the optimization methods provided to three different HW circuits by MicroBlaze

5.2 HEMPS platform results

We are using Vivado 2015.2 toolset, Xilinx VC707 development board which has a Virtex 7 xc7vx485t FPGA. We are builing the following systems using our automated scripts. Systems with up to 16 nodes, with following accelerators that are build using Vivado HLS 2015.2: Matrix Multipy, CRC, VectorADD, VectorMultiply and Bubblesort. For simplicity, we only build systems with nodes that either do not have any accelerator (A simple node) or a static accelerator or dynamic accelerator. When then nodes are dynamic, each node has all of the five above netlists during PR flow.

5.2.1 Scalability

We verified our ability to combine extensible processors into a functioning HEMPS system. Here, we built HEMPS systems with extensible processor nodes . Each node can be viewed as either *processor + co-processor* (with CRC accelerator) or *SW-based HAL + HW thread* (with Quicksort HW thread). We show the HEMPS system is scalable with the number of the nodes ranging from one to 32 nodes.

Test platform: HEMPS systems with up to 32 extensible processors were built using Xilinx Vivado 2014.4, synthesized, and run on a Xilinx Virtex7 (VC7047) Evaluation Board. A base system with six MicroBlaze slave processors was first built using the open-source automated architecture generation tool accessed at [21]. The base architecture was modified using the Xilinx PlanAhead tool to include reconfigurable slots for each MicroBlaze to form the extensible processors.

Benchmark: We used CRC accelerator and Quicksort HW thread to show the flexibility of a HEMPS system. The first application creates SW threads performing CRC encryption. Each

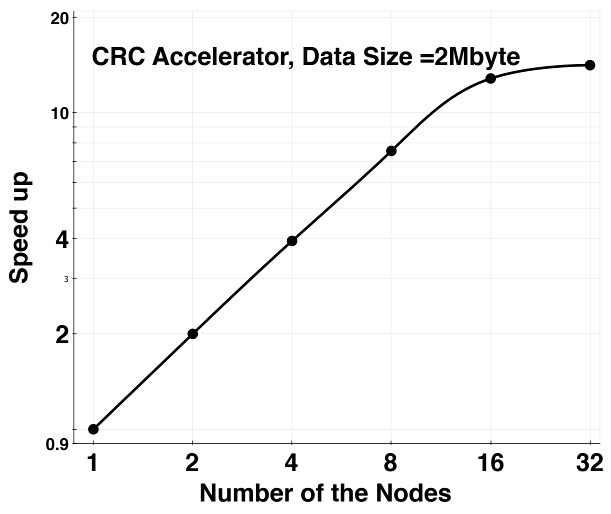


Figure 5.8: CRC, strong scalability

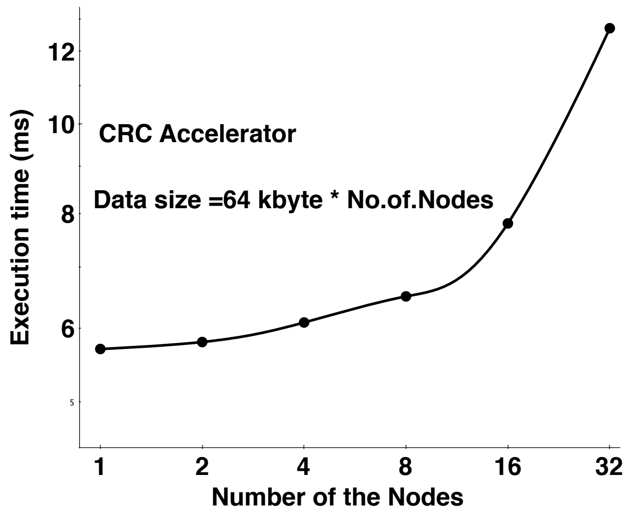


Figure 5.9: CRC, Weak scalability

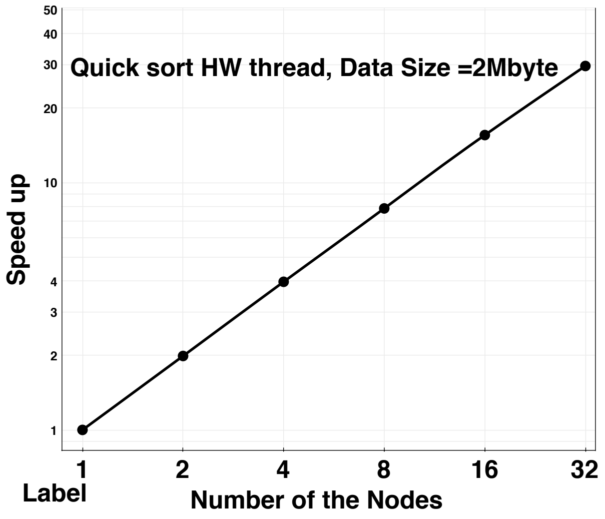


Figure 5.10: Quicksort, strong scalability

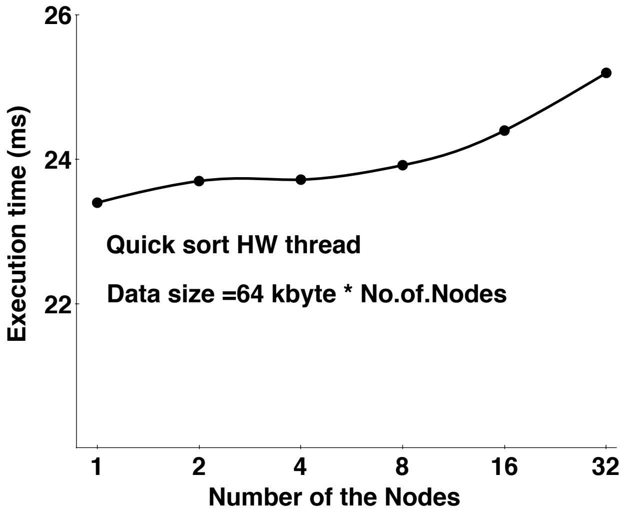


Figure 5.11: Quicksort, Weak scalability

No. Nodes	Acc. Type	Synthesis	PR flow	Total
1	NO Acc	21 mins	—	21 mins
4	NO Acc	40 mins	—	40 mins
8	NO Acc	71 mins	—	71 mins
16	NO Acc	116 mins	—	116 mins
1	Dynmic	25 mins	3.5 hours	4 hours
4	Dynimc	49 mins	7 hours	8 hours
8	Dynimc	83 mins	12.5 hours	14 hours
16	Dynimc	140 mins	18 hours	20 hours

Table 5.7: Synthesis time

extensible processor uses its CRC co-processor to boost the performance. Fig. 5.8 and Fig. 5.9 show the results of strong and weak scalability test for encryption of up to 2Mbytes of data using CRC algorithm by creating SW threads. Fig. 5.10 and Fig. 5.11 show the result of strong and weak scalability test of an applications sorting an array up to 2Mbytes size by creating HW threads. First off, the results show the scalability of both systems up to 32 nodes and data sizes of 2 Mbyte. The sequential part of DMAing data from DRAM to BRAMS is degrading the scalability when the number of the nodes or the size of data increases beyond the bandwidth of AXI4 bus and external DRAM. Second off, it shows how extensible processor node unifies both models of loosely and tightly coupled accelerators under one model.

5.2.2 Synthesis time and resource usage

The PC we are running the tools on is a Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz with 32 GB memory running Ubuntu 14.04. For Systems with Dynamic accelerators, the initial netlist for accelerators is vectoradd. Then, for the PR flow, all the 5 modules have been added to that region. During PR flow, we create threads to simultaneously do th PR flow for all netlists. Tables 5.7 and 5.8 show the synthesis times and resource utilization for systems built using Archgen upto 16 nodes with different configurations.

5.2.3 Partial reconfiguration

Figure 5.12 shows the floorplan of the final PR flow for a system with 16 nodes.

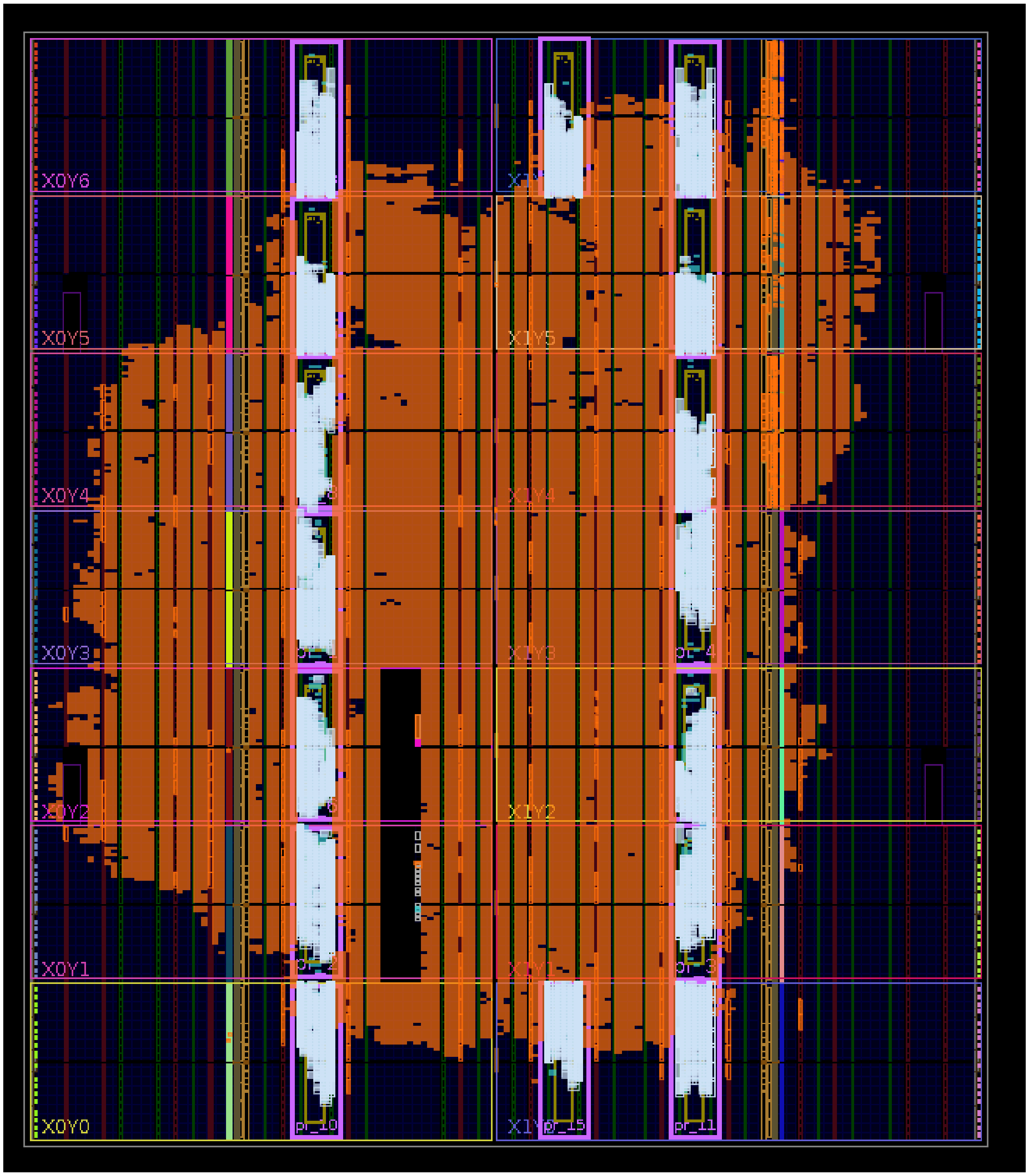


Figure 5.12: Floorplan of a PR system with 16 nodes

No. Nodes	Acc. Type	FF	LUT	BRAMs
1	Static	45816/7.5	39691/13	44/4
4	Static	72893/12	56152/18	100/9
6	Static	89032/14	64286/21	137/13
8	Static	120785/19	85714/28	180/17
16	Static	181676/30	121014/40	322/31

Table 5.8: resource utilization

Storage	Transfer Time	Bandwidth (MByte/S)
Compact Flash	185 ms	0.24
DRAM	13 ms	3.27
DRAM + DMA	460 us	96

Table 5.9: PR overhead

Table 5.9 shows the Partial Reconfiguration on Virtex 6 time based on where to save the bit-streams (Bit stream Size is 45Kbyte).

5.3 Polymorphic functions

This section evaluates the portability, productivity and performance of multi-threaded applications, which use polymorphic functions, running on HEMPS systems. For this part, we show our approach reinstates portability or the ability for an application developer to write the code once and run anywhere, and also how dynamic partitioning can result in better resource utilization and better overall system performance. Tables 5.10 and 5.11 show the performance and resource usage of two different hardware accelerator for sorting. This shows that choosing the right implementation of the polymorphic function can and will effect the performance. Figures 5.14 and 5.13 and 5.15 shows the detailed result of our three polymorphic functions with different data sizes.

5.3.1 Portability proof

To show the portability, I ran a simple program shown at Fig. 5.16 on different Hemps systems with 6 nodes. The program just simply creates threads that run crc polymorphic function. The program first self identifies the number and type of the nodes and uses these information later during runtime.

Data_size	bubble_sort	quick_sort	software
64	150	442	355
128	330	784	693
256	808	1535	1468
512	1966	3034	3228
1024	5263	5775	6759
2048	13345	6206	14253
4096	38204	7057	30908

Table 5.10: Sorting time comparison

accelerator	LUT	FF
bubble_sort	342	195
quick_sort	900	372

Table 5.11: resource usage for two different sorting accelerators

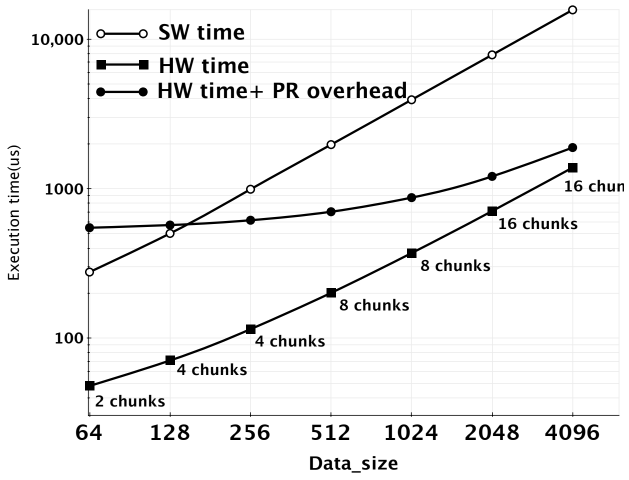


Figure 5.13: Polymorphic CRC

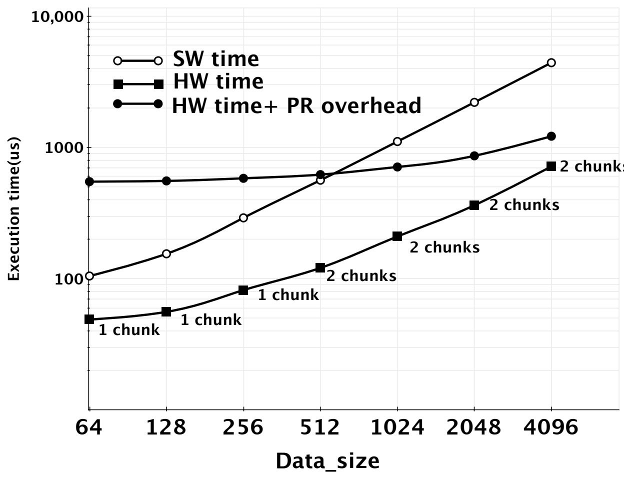


Figure 5.14: Polymorphic VECTOR

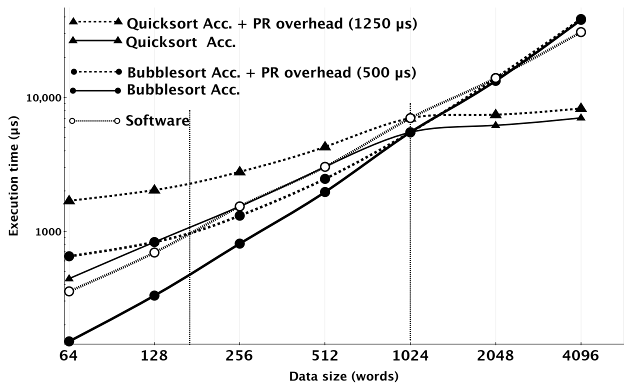


Figure 5.15: Polymorphic SORT

Fig. 5.17 shows the result of the program running on a the mixed system with 6 nodes. The system is a mixed one with 6 slaves, two of them don't have any accelerators, two of them have static accelerators and the last two ones have dynamic accelerators.

Fig. 5.18 shows the result of the program running on a system with 6 nodes and no accelerators. The system has 6 nodes, with none of the nodes have accelerators.

Fig. 5.19 shows the result of the program running on a static system with 6 nodes. The system has 6 nodes, with all of them have static accelerators.

Fig. 5.20 shows the result of the program running on a dynamic system with 6 nodes. The system has 6 nodes, with all of them have dynamic accelerators initially loaded with different accelerators.

5.3.2 Performance analysis

Test Configurations The following three classes of platforms were created to evaluate the portability of the programming model and ability of the runtime system to efficiently map the application across different combinations of heterogeneous resources. Figures 5.21 and 5.22 show the block diagram of the platforms being tested. The platforms are as follows:

- $Platform_{xNo}$: An SMP multiprocessor system with x slave processors.
- $Platform_{xFixed}$: A heterogeneous multiprocessor system with x slave processors each having a static hardware accelerator attached via Fast Simplex Links (FSL). Three types of

```
void * crc_thread(void * arg)
{
        data * package = (data *) arg;
        poly_crc(package->dataA,package->size);
        return (void *) 0;
}
//==================================================

int main{
...

for (k =0; k <NUM_AVAILABLE_HETERO_CPUS; k++)
{
   //creating Thread on slave CPU #k
   thread_create ( &tid, &attr , crc_thread_FUNC_ID , (void * )&package,k+2,0);
   hthread_join(tid, (void *) &ret);

   //printing info about how many times the polymorphic functions are done in HW or SW.
   print_finer_info();
 }
...
}
```

Figure 5.16: A simple program creating threads that invoke poly_crc function.

```
-----Begin Program-----
There are 6 slave processors in this HEMPS system
Slave #0: Mblaze, With  No Accelerator
Slave #1: Mblaze, With  No Accelerator
Slave #2: Mblaze, With  Static Accelerator: crc
Slave #3: Mblaze, With  Static Accelerator: VectorAdd
Slave #4: Mblaze, With  Dynamic Accelerator, loaded: crc
Slave #5: Mblaze, With  Dynamic Accelerator, loaded: VectorAdd

Creating Hetero Thread (CPU#0)!
Total HW Counter: 0   Total SW Counter: 1  Total PR Counter: 0

Creating Hetero Thread (CPU#1)!
Total HW Counter: 0   Total SW Counter: 2  Total PR Counter: 0

Creating Hetero Thread (CPU#2)!
Total HW Counter: 1   Total SW Counter: 2  Total PR Counter: 0

Creating Hetero Thread (CPU#3)!
Total HW Counter: 1   Total SW Counter: 3  Total PR Counter: 0

Creating Hetero Thread (CPU#4)!
Total HW Counter: 2   Total SW Counter: 3  Total PR Counter: 0

Creating Hetero Thread (CPU#5)!
Total HW Counter: 3   Total SW Counter: 3  Total PR Counter: 1


FINISH
```

Figure 5.17: A mixed system with 6 nodes

```
-----Begin Program-----
There are 6 slave processors in this HEMPS system
Slave #0: Mblaze, With  No Accelerator
Slave #1: Mblaze, With  No Accelerator
Slave #2: Mblaze, With  No Accelerator
Slave #3: Mblaze, With  No Accelerator
Slave #4: Mblaze, With  No Accelerator
Slave #5: Mblaze, With  No Accelerator

Creating Hetero Thread (CPU#0)!
CRC , MB 0,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 1
Total PR Counter: 0

Creating Hetero Thread (CPU#1)!
CRC , MB 1,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 2
Total PR Counter: 0


Creating Hetero Thread (CPU#2)!
CRC , MB 2,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 3
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#3)!
CRC , MB 3,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 4
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#4)!
CRC , MB 4,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 5
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#5)!
CRC , MB 5,  size: 2048*********************************
Total HW Counter: 0
Total SW Counter: 6
Total PR Counter: 0
-----------------------

FINISH
```

Figure 5.18: A system with 6 nodes and no accelerators

```
-----Begin Program-----
There are 6 slave processors in this HEMPS system
Slave #0: Mblaze, With  Static Accelerator : crc
Slave #1: Mblaze, With  Static Accelerator : VectorAdd
Slave #2: Mblaze, With  Static Accelerator : crc
Slave #3: Mblaze, With  Static Accelerator : bubblesort
Slave #4: Mblaze, With  Static Accelerator : crc
Slave #5: Mblaze, With  Staic Accelerator  : matrix_m

Creating Hetero Thread (CPU#0)!
CRC , MB 0,  size: 2048*********************************
Total HW Counter: 1
Total SW Counter: 0
Total PR Counter: 0

Creating Hetero Thread (CPU#1)!
CRC , MB 1,  size: 2048*********************************
Total HW Counter: 1
Total SW Counter: 1
Total PR Counter: 0


Creating Hetero Thread (CPU#2)!
CRC , MB 2,  size: 2048*********************************
Total HW Counter: 2
Total SW Counter: 1
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#3)!
CRC , MB 3,  size: 2048*********************************
Total HW Counter: 2
Total SW Counter: 2
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#4)!
CRC , MB 4,  size: 2048*********************************
Total HW Counter: 3
Total SW Counter: 2
Total PR Counter: 0
-----------------------

Creating Hetero Thread (CPU#5)!
CRC , MB 5,  size: 2048*********************************
Total HW Counter: 3
Total SW Counter: 3
Total PR Counter: 0
-----------------------

FINISH
```

Figure 5.19:   A static system with 6 nodes

```
-----Begin Program-----
There are 6 slave processors in this HEMPS system
Slave #0: Mblaze, With  Dynamic Accelerator, loaded: bubblesort
Slave #1: Mblaze, With  Dynamic Accelerator, loaded: crc
Slave #2: Mblaze, With  Dynamic Accelerator, loaded: VectorAdd
Slave #3: Mblaze, With  Dynamic Accelerator, loaded: matrix_m
Slave #4: Mblaze, With  Dynamic Accelerator, loaded: crc
Slave #5: Mblaze, With  Dynamic Accelerator, loaded: vectormul

Creating Hetero Thread (CPU#0)!
CRC , MB 0,  size: 2048*********************************
Total HW Counter: 1
Total SW Counter: 0
Total PR Counter: 1

Creating Hetero Thread (CPU#1)!
CRC , MB 1,  size: 2048*********************************
Total HW Counter: 2
Total SW Counter: 0
Total PR Counter: 1


Creating Hetero Thread (CPU#2)!
CRC , MB 2,  size: 2048*********************************
Total HW Counter: 3
Total SW Counter: 0
Total PR Counter: 2
-----------------------

Creating Hetero Thread (CPU#3)!
CRC , MB 3,  size: 2048*********************************
Total HW Counter: 4
Total SW Counter: 0
Total PR Counter: 3
-----------------------

Creating Hetero Thread (CPU#4)!
CRC , MB 4,  size: 2048*********************************
Total HW Counter: 5
Total SW Counter: 0
Total PR Counter: 3
-----------------------

Creating Hetero Thread (CPU#5)!
CRC , MB 5,  size: 2048*********************************
Total HW Counter: 6
Total SW Counter: 0
Total PR Counter: 4
-----------------------

FINISH
```

Figure 5.20: A dynamic system with 6 nodes

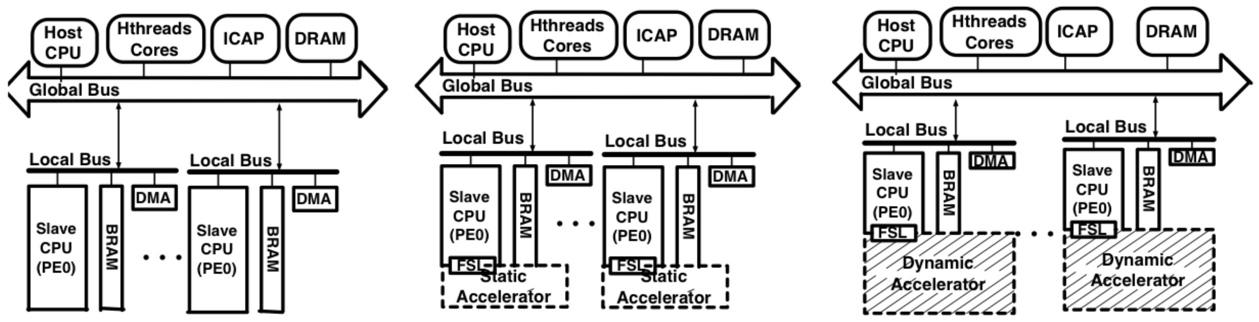


Figure 5.21: Platfrom configuration 1

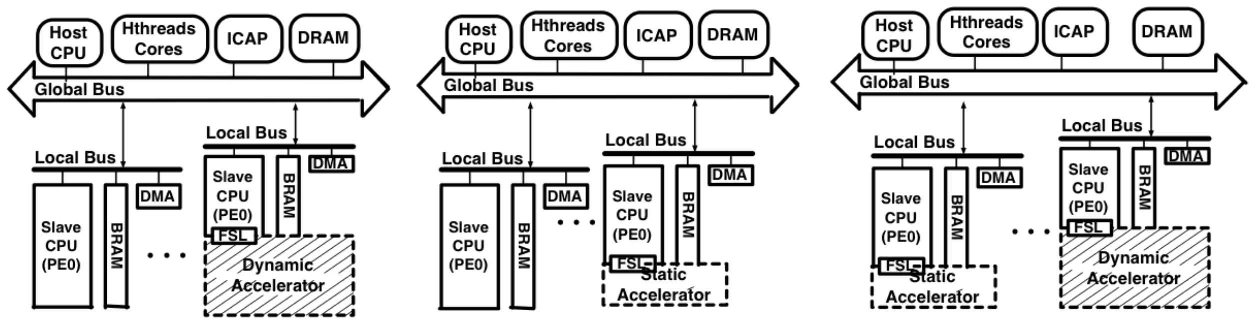


Figure 5.22: Platfrom configuration 2

accelerators were created: CRC, Sort, and Vector. Each processor had a single accelerator instantiated. Two processors had CRC, two had Sort, and two had vector accelerators.

- $Platform_{xPR}$: The same as $Platform_{xFixed}$, but all slave processors had slots created for partial reconfiguration to load any of the three accelerators.

- $Platform_{3No\_3PR}$, $Platform_{3Fixed\_3PR}$, $Platform_{3No\_3Fixed}$: Mixed heterogeneous multiprocessor systems with 6 slave processors each having either a partially reconfigurable region, static hardware accelerator or no accelerator.

All systems were built using Xilinx ISE Design Suite v12.3 and all tests was performed on a Xilinx Virtex6 (ML605) Evaluation Board. All code was compiled using the heterogeneous compilation flow previously reported in [7]. [12]. Although the runtime system used in this work supports the use of heterogeneous processors, only MicroBlaze processors were used due to board selection. For convenience in this work we used MicroBlaze [4] processors.

Synthetic Benchmark: A synthetic benchmark program was written and executed on all platforms. The synthetic benchmark created approximately 150 threads. Each polymorphic thread creates a variable number of joinable (on average three) worker threads based on run time information. All threads made between one and three polymorphic function calls (*poly_crc()*, *poly_sort()*, *poly_vector()*). As explained previously, each polymorphic function contained both software and hardware accelerator implementations. The same hardware version of each accelerator was used in both systems with fixed or Partially reconfigurable accelerators. The order of the polymorphic function calls and their data input sizes were randomized. A much simplified pseudocode version of the benchmark is provided in Figure 5.23.

We use these terms in this section:

Best match found ratio: The percentage of the available accelerator needed for a thread mapped to a slave processor being present. Higher is better.

HW accelerator usage ratio: The percentage of how many times the polymorphic functions ran in Hardware accelerators instead of running in Software. Higher is better.

```
int main() {
  init_profiling_table();
  thread_create(foo_thread);
  thread_create(bar_thread);
  ....
}
void * foo_thread() {
   poly_vector(&a,&b,&c,size,ADD);
   poly_crc(&data,size);
   poly_sort(&data,size);
}
void * bar_thread() {
   poly_sort(&data,size);
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Polymorphic Function
void poly_vector( ... ){
   bool use_accelerator=isHW(); //Performs PR as well;

   if (use_accelerator) {
      divide_data_into_chunks();
      for ( i < # of chunks ) {
         dma_data(chunks[i]);
         hw_vector(chunks[i]);
      }
   }
   else
      sw_vector();
}
```

Figure 5.23: Pseudo code for the synthetic benchmark.

PR ratio: The percentage of how many times times partial reconfiguration is done on slave processors. Lower is better.

Also, We refer to the combination of both Profiling and self-aware scheduling as *Runtime Tuning*.

- Data arguments given to accelerator functions are less than 2kB. This avoids the execution time over bigger sizes of data from dominating the improvements provided by our runtime system.

- Software threads execute on the host processor, and are currently time-sliced in a round-robin manner.

- We chose a target of 50 percent slave processor utilization in our system of 6 when creating threads manually using thread_create. Host accelerator calls are blocking, as they can create an optimal number threads (=¡ 6) within the call.

- There are three examples of accelerator functions implemented in both software and hardware: CRC, SORT, & VECTOR functions.
- In the beginning, the main function calls online_tuning() function which fills the tuning_table();
- As mentioned before, once we compile it for any of the platforms, we then use the same elf file for all other two platforms.
- Since it is synthetic benchmark,we have a average load of keeping just three MicroBlazes busy. So,before creating any new threads we first wait until at least three slaves are free.

Portability was validated by running the synthetic benchmark *unaltered* on all platforms. Figure 5.24 shows the execution times of the synthetic benchmark run on eleven unique platforms. Each platform contained varying numbers of processors and different combinations of static and partially reconfigurable accelerators. The correctness of the results were verified using an automated script that configured and executed each test case, read the runtime results back onto a workstation and compared against expected results. All results passed. Figure 5.24 demonstrates how the runtime system transparently tuned the application on all combinations of system resources with no changes to the program. The next section provides a quantitative analysis of the results. Figure 5.24 shows the more systems gets complex, the more performance is achieved .

Moreover, Runtime Tuning has virtually no effect in simple systems like $\text{Platform}_{\text{xNo}}$ and $\text{Platform}_{\text{3Fixed}}$, while it has a great effect in $\text{Platform}_{\text{xPR}}$ with the $\text{Platform}_{\text{6PR}}$ benefiting more than all other cases since it is the most complex system among all 9 tested platforms. Figure 5.24 shows that enabling Runtime Tuning in $\text{Platform}_{\text{6No}}$ only leads to 3 improvement, 11 In $\text{Platform}_{\text{6Fixed}}$ but 61 in $\text{Platform}_{\text{6PR}}$.

As would be expected, the results show relative performance gains as the number of processors was increased and additional performance boosts provided when accelerators are added. With portability verified we next sought to understand if our runtime partitioning scheduling approach would compare favorably with what would be achieved by statically profiling and partitioning the application on the bench for each for systems with different particular sets of resources.

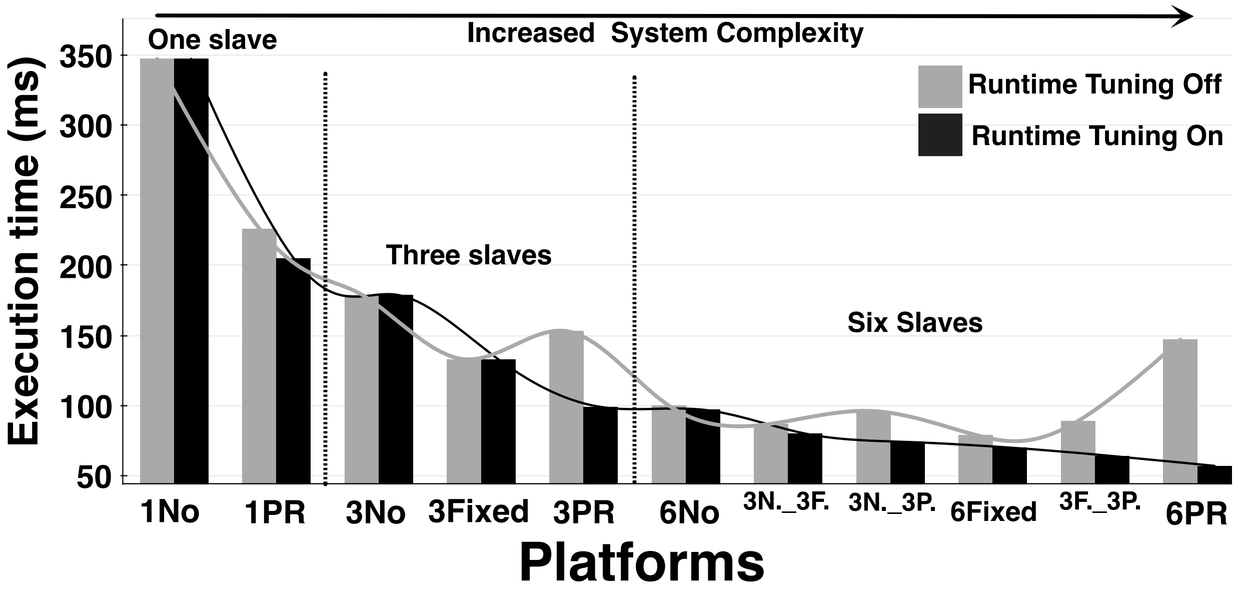


Figure 5.24: Execution time for the Synthetic benchmark

The results show that for small systems hand partitioning is as good or better than dynamic partitioning. As systems grow in complexity dynamic partitioning yields better results. Runtime tuning consists of self-aware scheduling and runtime profiling. Figure 5.24 shows the performance gains when these two capabilities were active. The shape of the curves in Figure 5.24 provide insight into the effectiveness of runtime tuning. For systems with processors that have no accelerators (Platforms$_{\text{1No,3No,6No}}$), exploiting only TLP provides no significant performance differences when runtime tuning is off or on. This indicates that explicitly hand tuning in one's application to make use of all processors matches the same performance gains as runtime tuning. However, naively using all processors may cause the system to expend unnecessary energy when using less processors in the same system provides the same benefit. This is what runtime tuning achieves through the use of profiling information. However a measurable difference can be seen in achieved efficiency of resource utilization and speedup when the diversity and numbers of resources starts to increase. With runtime tuning enabled, the execution time monotonically decreased, even as the complexity of the system grew through the addition of processors, static accelerators, and partially reconfigurable accelerators (Platform$_{\text{1No}}$ to Platform$_{\text{6PR}}$). as the number of resources increases and static

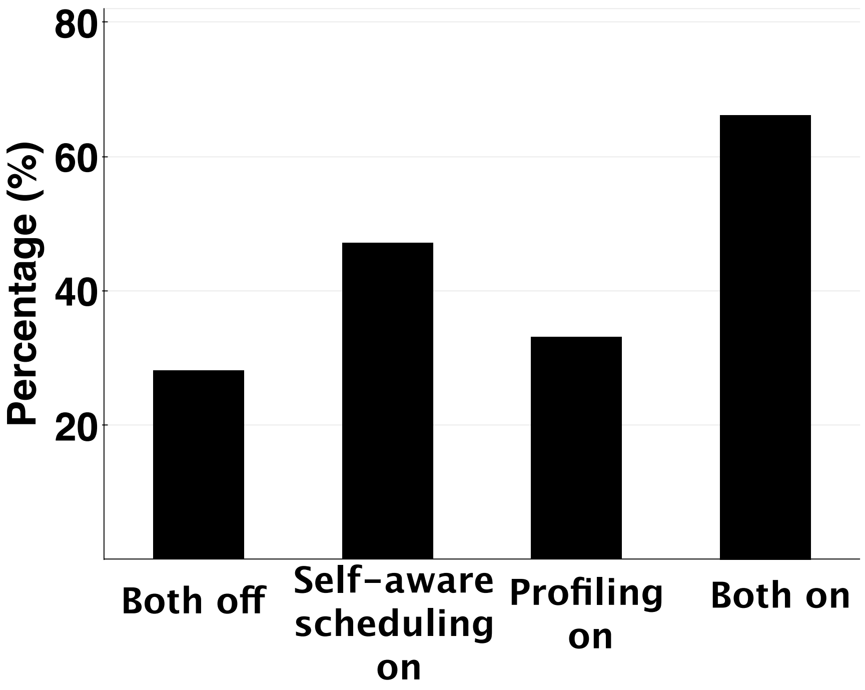


Figure 5.25: Synthetic benchmark on Platform$_{6PR}$: First accelerator hit ratio

groups of accelerators are replaced by partial reconfiguration regions (Platform$_{1No}$ to Platform$_{6PR}$). This shows that runtime tuning continually enabled the system to deliver performance gains as resources became more diversified and complex during runtime. This trend holds as systems become larger through the addition of processors, processors with static accelerators, and finally processors with partially reconfigurable accelerators. The runtime Tuning enables the polymorphic calls to wisely choose when and how they use the resources and options available. When runtime tuning was disabled, some performance increases were observed but still the more complex systems in average run faster, thanks to the portable programming model which transparently takes advantage of any available resources. However the shape of the curve was no longer monotonic. For example, from Platform$_{3Fixed}$ to Platform$_{3PR}$ and from Platform$_{6Fixed}$ to Platform$_{6PR}$, performance degraded as the diversity of the hardware increased. This shows that as the number and types of resources within a CHMP system increases, the difficulty of producing near optimal solutions for all combinations of resources by hand tuning also increases. The following section focuses on the individual effects of self-aware scheduling and runtime profiling.

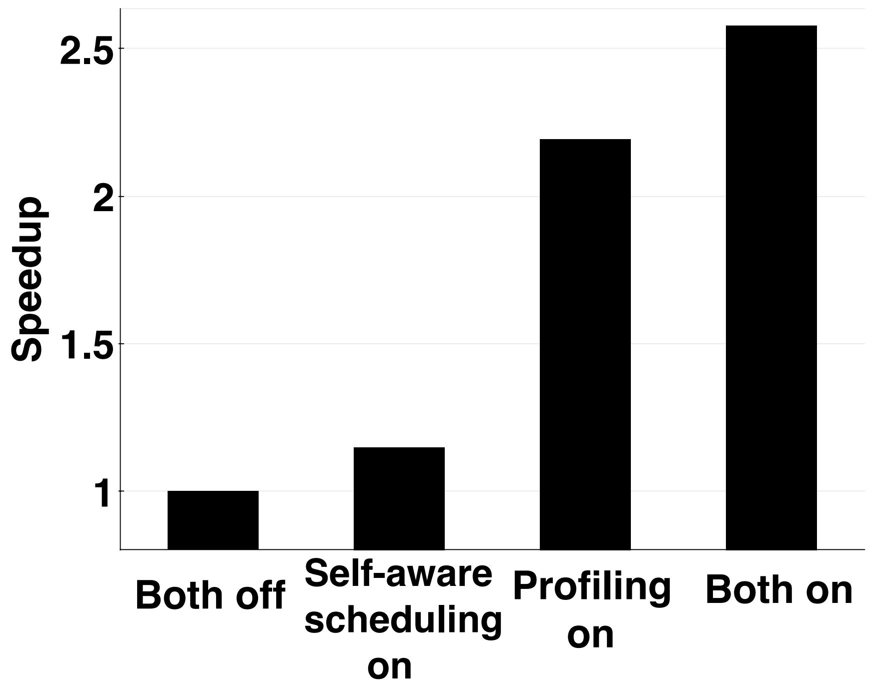


Figure 5.26: Synthetic benchmark on Platform$_{6PR}$: Speedup

Figure 5.26 shows speedups for the synthetic benchmark running on Platform$_{6PR}$ using the different runtime tuning options. Figure 5.27 shows the percentage of time the system had to load an accelerator before it could be used. Figure 5.25 shows the percentage of time that the system scheduler was able to find the first accelerator needed by a thread already loaded to an available processor in the system. Turning on either self-aware scheduling or profiling leads to more speedup, less partial reconfiguration ratio and higher *first accelerator hit ratio*. As it can be predicted, enabling both of them leads to better results.

Figure 5.26 shows profiling provides greater speedup than self-aware scheduling. This results from self-aware scheduling making its scheduling decision based on the first polymorphic function called in a thread. Conversely, runtime profiling includes decisions for *every* polymorphic function called in a thread. that are contained within a thread leads to significantly better speedup and less partial reconfiguration ratio since it *tunes* all polymorphic calls in a thread rather than only the first one.

Figure 5.25 shows that enabling self-aware scheduling learns which accelerator is first used in each thread. It learns by maintaining a priori history of what accelerators are already loaded when

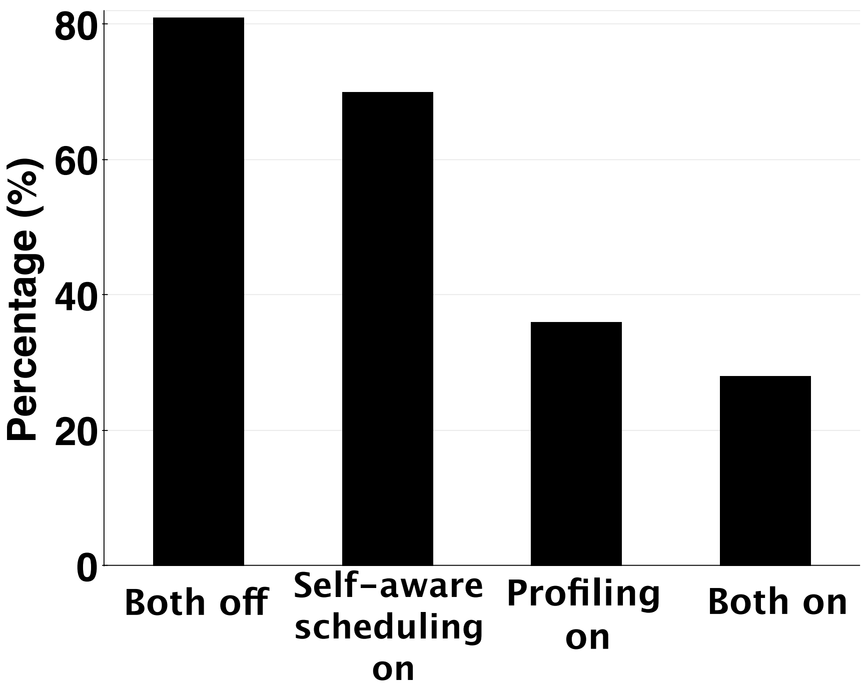


Figure 5.27: Synthetic benchmark on Platform$_{6PR}$: PR ratio

scheduling a new thread. achieves a 20 increase in *first accelerator hit ratio*. This increase is due to the self-aware scheduler storing first polymorphic function calls within threads and using this information to reschedule threads at a later time. This provided two benefits. First, slave processors did not have to decide if the cost of loading a bitstream offset the benefits of using the accelerator compared to simply executing the function in software. Second, the system was able to reduce the PR ratio by 11 which reduced the overhead of loading additional bitstreams. The combination resulted in a 1.15*x* speedup.

Figure 5.27 shows that enabling profiling reduces the PR ratio by 45 percent. Profiling information was used to make a more fine-grained decision as to when to use an accelerator. Figure 5.28 shows the result of a program with 120 threads , Each thread randomly can have up to 3 polymorphic function calls. It shows that the PR ratio decreases while speedup increases. In some instances, the additional overhead of loading the bitstream would have resulted in a slower execution time using the accelerator compared to simply executing the polymorphic function in software. The reduced PR ratio showed that this was the case. Reducing the PR ratio also provides the system with important secondary benefits. Reducing the number of times bitstreams are

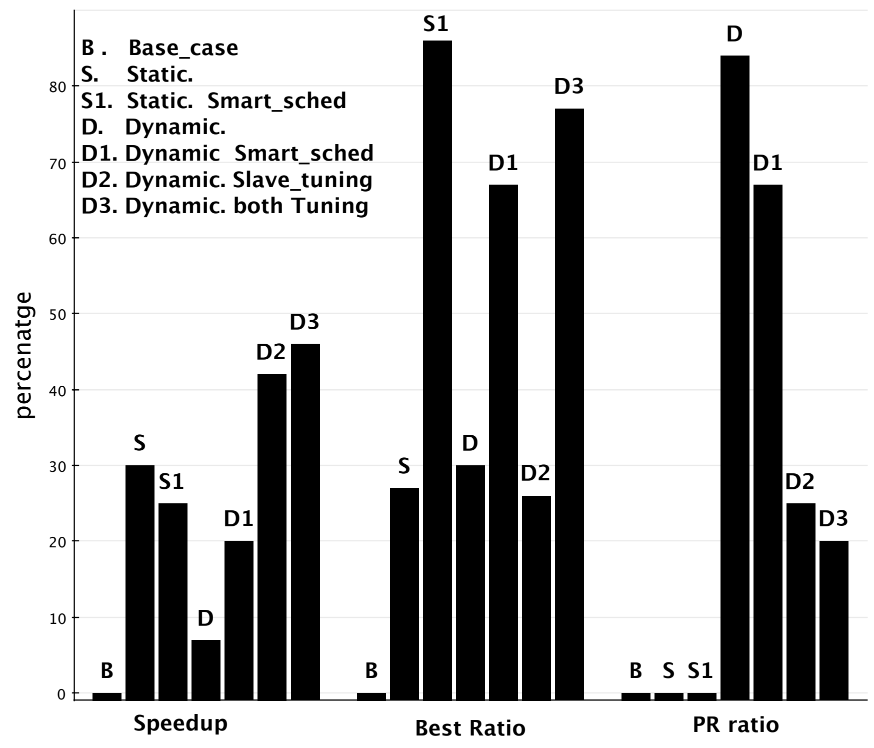


Figure 5.28: 120 Threads

transferred eases the load on key system resources such as buses, DRAM, and the ICAP. These resources that otherwise would have used are now available for other processing requests. The net effect of profiling resulted in a significant 2.2x speedup on Platform$_{\text{6PR}}$.

5.3.3 Evaluating Run Time Profiling

The polymorphic functions need the Training data stored in the tuning table to determine the best version to be called. The performance gain comes from reducing the number of threads created to an optimum corresponding to the current platform, as well as significantly reducing the overhead of Partial reconfiguration by avoiding unnecessary swapping in and out accelerators. Profiling helps polymorphic calls to Autonomously run in the most efficient way, transparent to the user. Figure 5.29 isolates the effects of run time profiling. On Platforms$_{\text{6No,6Fixed,6PR}}$, Platform$_{\text{6Fixed}}$ and Platform$_{\text{6PR}}$ enabling Profiling information was used effectively by the polymorphic function partitioner when determining if bitstreams needed to be loaded. Profiling shows negligible impact

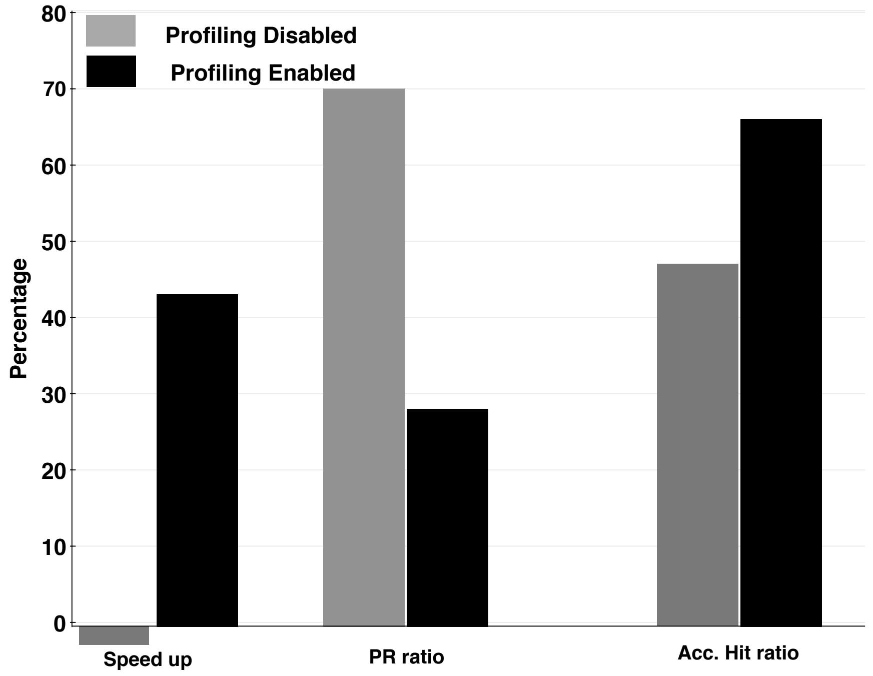


Figure 5.29: Evaluating profiling effects on polymorphic calls

on $\text{Platforms}_{\text{6No,6Fixed}}$. Conversely profiling shows significant impact in $\text{Platform}_{\text{6PR}}$. The profiling information enables polymorphic functions to justify swapping partially reconfigurable accelerators, instead of blindly reloading a bitstream every time an accelerator was invoked. As expected reducing bitstream transfers leads to better overall performance. Conversely Figure 5.29 clearly shows with profiling disabled, the performance of $\text{Platform}_{\text{6PR}}$ is worse than $\text{Platform}_{\text{6Fixed}}$. Even more concerning the performance becomes worse than $\text{Platform}_{\text{6No}}$, the system with no accelerators. obtain an increase in performance. When profiling reduces the number of processors used to increase performance, it also reduces the number of processors generating data transfers and synchronization requests. Reducing unnecessary reloading of bitstreams also eases bus contention. These resources that otherwise would have used are now available for other processing requests.

Training data stored in the tuning table is being used by both host and slave polymorphic functions:

1. Using training data for slave polymorphic functions means evaluating software execution time, and hardware execution time and PR overhead, to decide where computation should

occur for a specific polymorphic function call. Therefore, it is limited to systems with dynamic accelerators. That is why in Figure 5.29 D3 is significantly better than D1, compared to negligible improvement going from B to B1 and S1 to S3.

2. Using training data for host polymorphic thread create functions means creating an Optimal number of threads based on runtime variables compared to maximum number of threads. Although one might think that creating more threads always boosts up performance, Figure 5.29 shows by using the Training data, the performance increases while significantly creating smaller number of threads. The online Tuning function allows more threads to be created if there is at least 10 percent performance is gained. This not only leads to a faster execution time, but also will save power if we turn off the MicroBlazes that we are not scheduling a thread on them. This idea is more discussed in our other paper. The best Optimal number of threads is a function of system configuration , and therefor it is different in each of these three platforms. However,it was hard to get that information for system with Dynamic accelerators since the Opt.No.Threads is also a function of configuration of the system at the moment. Therefore, we used the results of static system an estimate for Dynamic system.

Chapter 6

Conclusion

The multithreaded programming model has been effective in enabling programmers to model accelerators as hardware threads that can synchronize and exchange data with software threads under the control of the operating system. HW threads need to be provided with an analogous set of software library middleware written for SW threads to interface with the system. This Hardware Abstraction Layer (HAL) has been traditionally written via FSM in hardware. The model relies on the use of a custom HAL to provide the operating system services and data access interfaces necessary for the hardware thread to seamlessly interact with the rest of the system. Traditional approaches provided a finite state machine version of a HAL to allow hardware threads to interface into the multithreaded programming model.

In this paper we argued that CPU-based HAL provides increased flexibility and productivity, less area usage and better performance compared to traditional HW-based HALs. Replacing the custom hardware HAL with a general purpose processor software enables the model to better support the unique types of data and loop level parallelism that may exist within each thread. Importantly, the combination of a programmable processor and accelerator form a Heterogeneous Extensible Multiprocessor (HEMP) node that can be replicated to form large Chip Heterogeneous Multiprocessor Systems. The results showed how a HEMPs system can be used to seamlessly exploit different types of parallelism within familiar programming patterns.

HEMPS unifies both models of loosely coupled and tightly coupled accelerators in one architecture, and provides a portable platform for applications. We provided comparison between our approach and other research efforts in this area. We built HEMPS systems with up to 32 nodes to support different use cases of custom HW in FPGAs. We showed how we can build scalable and portable MPSoC systems and provided a detailed analysis of the area and performance comparison between different HEMPS systems.

Our original pragmatic objective was to replace custom hardware with a programmable controller to increase productivity and flexibility. However, this pragmatic objective resulted in unforeseen changes in how we model, build and program hybrid HW/SW multithreaded processor systems. Following is the list of benefits of an extensible processor node which provides the basic structure for HEMPS and polymorphic functions:

- *performance*: Having a MPSoC system with Extensible processor nodes not only enables both coarse grained HW/SW partitioning through Thread Level Parallelism (TLP), but also it enables users to more efficiently exploit finer grained parallelism within a thread body. It also provides transparent and distributed partial reconfiguration of the accelerator/HW-thread during the execution of the thread.

- *resource usage*: The by-product of fine grained HW/SW partitioning is having smaller custom HW threads, as the sequential part of the thread can be run on the processor. This will Reduce the gate requirements of accelerator/HW thread circuits.

- *Portability*: The flexibility of the Extensible processor node provides the infrastructure for the threads to be run *portably* via a library of functions with implementations in both HW and SW.

- *Supporting Polymorphic Functions and Fine Grained HW/SW Partitioning*: The Extensible processor enables a fine grained HW/SW partitioning of the thread assigned to it.

Form the hardware point of view, the average SW developers can not deal with the CAD tools to build a complicated HEMPS system, not to mention the complexities of compiling the code for a heterogeneous platform. In this work we addressed the problem of FPGA's lack of accessibility to SW developers by presenting Archgen script to generate the complete system on chip which can be accessed in the cloud. We show the results for generating different HEMPs systems using our toolchain.

From the software point of view, we presented a library of polymorphic functions that restores application portability over HEMPS systems. The programming model eliminates the need for programmers to exhaustively profile their machine specific applications. The programming model enables our runtime system to profile and efficiently create and schedule concurrent threads on heterogeneous resources. It also allows data level parallelism within each thread. This can significantly enhance productivity as it moves the burden of profiling and optimizing from the programmer to the run time system. We also presented run time profiling of the application. Runtime results gathered from a synthetic benchmark on different system configurations of processors, processors and static accelerators, and processor and dynamically reconfigurable accelerators showed better performance results when making partitioning and SW/HW scheduling decisions during run time.

6.1 Research Contributions

Engineering Artifacts:

- Extensible processor node
- Transparent partial reconfiguration
- Automated generation of HEMPS systems
- Portable programming model via polymorphic functions
- Runtime tuning

Scientific outcomes:

- New paradigm for accelerator integration into CHMP systems.
- Better performance on FPGAs without sacrificing productivity and portability on FPGAs
- Enable software programmers to use FPGA unique capabilities without the help from hardware engineers
- Reduce impact of Dark Silicon on next generation heterogeneous multiprocessors

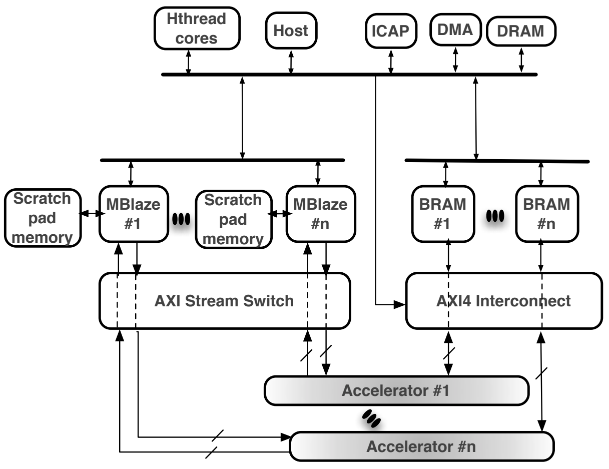


Figure 6.1: HEMPS with loosely coupled accelerators

6.2 Future Work

Future work includes:

- building up libraries of linkable polymorphic representations of common programming patterns.
- Investigating on systems that include additional heterogeneous resources such as soft core programmable vector processors.
- Creating a set of tools and automated design environment for system programmers to create and add additional polymorphic functions into our library.
- Investigating on new machine learning approaches to increase the efficiency of our scheduling decisions.
- making all the accelerators accessible by all slaves via a Crossbar switches as shown in Fig. 6.1. This will further reduce unnecessary Partial reconfiguration, and provides better resource management. Moreover, it gives us the infrastructure for extending our API to support chaining of Accelerators and eliminate unnecessary DMAing data back and forth, as

well as giving us the option of having different sizes of Partial Reconfiguration regions, so that we efficiently map both small and big accelerators.

- Improving our current runtime tuning by gathering more extensive HW/SW information of the running system like the bus traffic, ICAP queue status, the static profiling of the software application and use some machine learning techniques to make more efficient decisions.
- Creating two Hardware co-processors, one to service host processor in smart scheduling, and the other one for the slaves for slave_tuning. Both of them will get the information from current traffic of ICAP resource, available accelerators, Profiling information of the C program running, and they might use some machine learning techniques to be able to provide more accurate information needed by host or slave processors.

References

[1] Altera SDK for OpenCL. http://www.altera.co.uk/products/software/opencl/opencl-index.html. Last accessed May 3, 2016.

[2] Convey Computer. http://www.conveycomputer.com/. Last accessed May 3, 2016.

[3] FSL Bus Product Specification. http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20. Last accessed May 3, 2016.

[4] Microblaze Soft Processor. http://www.xilinx.com/tools/microblaze.htm. Last accessed May 3, 2016.

[5] Nallatech FSB- development systems. http://www.nallatech.com/. Last accessed May 3, 2016.

[6] OpenCL - The open standard for parallel programming of heterogeneous systems. The Khronos Group. Last accessed May 3, 2016.

[7] Removed for blind review.

[8] Spark project.

[9] UCLA Open-Source Accelerator Store. http://cadlab.cs.ucla.edu/accelerator_store.html. Last accessed May 3, 2016.

[10] U.S. Bureau of Labor Statistics. http://www.bls.gov/data/. Last accessed May 3, 2016.

[11] Vivado High-Level Synthesis. www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Last accessed May 3, 2016.

[12] Jason Agron and David Andrews. Building Heterogeneous Reconfigurable Systems With a Hardware Microkernel. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, New York, NY, USA, 2009. ACM.

[13] AMD. Heterogeneous System Architecture: A Technical Review. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf. Last accessed May 3, 2016.

[14] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *Proceedings of the 14th Annual Conference on Field-Porgrammable Custom Computing Machines*, 2006.

[15] Erik Anderson, Wesley Peck, Jim Stevens, Jason Agron, Fabrice Baijot, Seth Warn, and David Andrews. Supporting High-Level Language Semantics Within Hardware Resident Threads. *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, August 2007.

[16] David Andrews, Douglas Niehaus, and Peter J. Ashenden. Programming Models for Hybrid CPU/FPGA Chips. *IEEE Computer*, 37(1):118–120, 2004.

[17] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbee, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming Models for Hybrid FPGA-CPU Computatoinal Components: A Missing Link. *IEEE Micro*, 24(4):42–53, 2004.

[18] David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, and Ron Sass. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 19-22, 2005.

[19] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving Programming Model Abstractions For Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, January 2008.

[20] Timothy Callahan, John R. Hauser, and John Wawrzynek. The GARP Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, 2000.

[21] E. Cartwright, A. Fahkari, S. Ma, C. Smith, M. Huang, D. Andrews, and J. Agron. Automating the design of mlut mpsopc fpgas in the cloud. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 231 –236, aug. 2012.

[22] Jongsok Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277, Dec 2013.

[23] 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.

[24] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich cmps. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 843–849. ACM, 2012.

[25] Youenn Corre, Jean-Philippe Diguet, Loïc Lagadec, Dominique Heller, and Dominique Blouin. Fast template-based heterogeneous mpsoc synthesis on fpga. In *Proceedings of the 9th International Conference on Reconfigurable Computing: Architectures, Tools, and Applications*, ARC'13, pages 154–166, Berlin, Heidelberg, 2013. Springer-Verlag.

[26] R. H. Denard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-state Circuits*, 98, 1974.

[27] Advance Micro Devices. Bolt C++ Template Library. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/bolt-c-template-library. Last accessed May 3, 2016.

[28] 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.

[29] H. Franke, J. Xenidis, C. Basso, B.M. Bass, S.S. Woodward, J.D. Brown, and C.L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1–3:11, January 2010.

[30] M. D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris, and C. E. Goutis. A partitioning methodology for accelerating applications in hybrid reconfigurable platforms. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 3*, DATE '05, pages 247–252, Washington, DC, USA, 2005. IEEE Computer Society.

[31] P. Garcia and K. Compton. Kernel sharing on reconfigurable multiprocessor systems. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 225–232, Dec 2008.

[32] 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.

[33] Markus Happe, Enno Lübbers, and Marco Platzner. A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking. *J. Real-Time Image Processing*, 8(1):95–110, 2013.

[34] J. Hauser and J. Wawrzynek. GARP: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1997.

[35] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41:33–38, 2008.

[36] http://opencores.org/. OpenCores.

[37] Chen Huang and Frank Vahid. Dynamic coprocessor management for fpga-enhanced compute platforms. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 71–78, New York, NY, USA, 2008. ACM.

[38] Intel. Intel Threading Building Blocks Documentation. www.threadingbuildingblocks.org/docs/help/index.htm. Last accessed May 3, 2016.

[39] Aws Ismail and Lesley Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *FCCM*, pages 170–177, 2011.

[40] P.O. Jaaskelainen, C.S. de la Lama, P. Huerta, and J.H. Takala. Opencl-based design methodology for application-specific processors. pages 223–230, July 2010.

[41] William V. Kritikos, Andrew G. Schmidt, Ron Sass, Erik K. Anderson, and Matthew French. Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip. *International Journal of Reconfigurable Computing*, 2012, 2012.

[42] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, nov. 2005.

[43] Ming Liu, W. Kuehn, Zhonghai Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502, 2009.

[44] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Energy reduction with run-time partial reconfiguration (abstract only). In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 292–292, New York, NY, USA, 2010. ACM.

[45] 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.

[46] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on fpgas. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, pages 795–805, London, UK, UK, 2002. Springer-Verlag.

[47] Tim Mattson and Larry Meadows. A Hands-on Introduction to OpenMP. Super Computing (SC) Tutorial, November 2008. http://openmp.org/mp-documents/omp-hands-on-SC08.pdf.

[48] Microsoft. C++ AMP (C++ Accelerated Massive Parallelism). http://msdn.microsoft.com/en-us/library/hh265137.aspx. Last accessed May 3, 2016.

[49] Douglas Niehaus and David Andrews. Using the Multi-Threaded Computation Model as a Unifying Framework for Hardware-Software Co-Design and Implementation. In *9th International Workshop on Object-oriented Real-time Dependable Systems*, Isle of Capri, Italy, Sept. 2003.

[50] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[51] Albert Noll, Andreas Gal, and Michael Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, 2008.

[52] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. Synthesis of platform architectures from opencl programs. In Paul Chow and Michael J. Wirthlin, editors, *FCCM*, pages 186–193. IEEE Computer Society, 2011.

[53] H. Quinn, L.A.S. King, M. Leeser, and W. Meleis. Runtime assignment of reconfigurable hardware components for image processing pipelines. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 173–182, April 2003.

[54] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The sarc architecture. *Micro, IEEE*, 30(5):16–29, Sept 2010.

[55] Javier Resano and Daniel Mozos. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 119–124, New York, NY, USA, 2004. ACM.

[56] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *Micro, IEEE*, 29(1):10–21, Jan 2009.

[57] B. Senouci, A.M. Kouadri M, F. Rousseau, and F. Petrot. Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers. *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008. RSP '08*, pages 41–47, June 2008.

[58] Lesley Shannon and Paul Chow. Leveraging reconfigurability in the hardware/software codesign process. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):28:1–28:27, August 2011.

[59] Y. Shiyanovskii, F. Wolff, C. Papachristou, and D. Weyer. An adaptable task manager for reconfigurable architecture kernels. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 132–137, 2009.

[60] Desh Singh. Higher level Programming Abstractions for FPGAs using OpenCL. Presentation at FPGA 2011.

[61] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1392 thru 1407, November 2004.

[62] P.M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell. Hippai: High performance portable accelerator interface for socs. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 109–118, Dec 2009.

[63] M. Vuletic, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design & Test of Computers, IEEE*, 22(2):102–113, March-April 2005.

[64] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System. *ACM SIGPLAN Notices*, 42(6):156–166, 2007.

[65] 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.

[66] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, 1993.

[67] 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.

[68] M. J. Wirthlin and B. L. Hutchings. DISC: The Dynamic Instruction Set Computer. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, volume SPIE 2607, pages 92–103, 1995.

[69] Dong Hyuk Woo and H.-H.S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[70] www.celoxica.com. Celoxica. Last accessed May 3, 2016.

[71] www.xilinx.com/sdsoc. Sdsoc.

[72] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[73] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector Processing as a Soft-Core CPU Accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 222–232, New York, NY, USA, 2008. ACM.