OPERATING SYSTEM INTERFACES TO RECONFIGURABLE SYSTEMS

BY

JOHN HENRY KELM

B.S.E., University of Connecticut, 2005

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Operating System Interfaces to Reconfigurable Systems

Approved by
Dr. Steven S. Lumetta

ABSTRACT

In this work, a hybrid CPU/accelerator platform, which runs a standard operating system, is prototyped using an FPGA. The goal is to provide consistent, protected, low overhead interfaces for accelerator developers to target. These interfaces enable software application developers to exploit the parallel processing power of hardware accelerators and reconfigurable computing. Furthermore, we aim to decouple the process of software development from accelerator design in a heterogeneous model of computation.

A switchable interconnect accelerator framework is developed to explore what resources should be embedded in future CPU/accelerator systems. We mate our accelerator and application-level interfaces to provide a seamless platform for developing hardware accelerated applications. Complexity of debugging and development using our framework is lowered by reducing the signal count by over 80%, compared to a stand-alone accelerator design. A hardware accelerated JPEG encoding application serves as a motivating example throughout this work.

A computation model is developed to allow standard applications running on our platform to access reconfigurable accelerators. The applications access accelerators, synthesized in the FPGA fabric, through an operating system layer. Applications running on the CPU interface with the accelerators using a library call like interface. We have found that system call and data transfer overhead can be well over 2000 cycles or made as low as 300 cycles for our example accelerators. Our model provides protection, encapsulation, and resource virtualization. Using our prototype platform we implement parts of our computation model and show the performance tradeoffs for four different access methods.

ACKNOWLEDGMENTS

I would like to thank both the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign and the MARCO/DARPA Gigascale Silicon Research Center (http://www.gigascale.org) for supporting this work. I would also like to thank Xilinx for donating the tools and development boards and Intel for donating workstations used in this work.

I would like to thank my advisor Professor Steve Lumetta and Professor Wen-mei Hwu for having been of great assistance in the development of this thesis, providing me with support, encouragement, and guidance throughout. I would like to also thank the members of IMPACT with whom I have shared much time discussing the ideas in this thesis; they include Shane Ryoo, Isaac Gelado, and Brian Watson. I would like to thank my friends Erica Lundgren and Brandon Swamy for their encouragement and for putting up with me for the past year while I worked on my thesis. Most importantly, I thank my mother for always supporting me in all my pursuits, whether academic, personal, or otherwise.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Contemporary microprocessors, based on complex microarchitectures, are reaching power and performance limits. At the same time, Moore's law continues to provide more and more transistors to architects. New methods must be developed for effectively exploiting the ever-increasing number of transistors integrated on a single die. One method explored by manufacturers is to add more cores to the same die. These manufacturers hope to bring to market a central processing unit (CPU) that has higher performance. However, simply adding more cores may not meet desired power, cost, and performance goals. Integrating reconfigurable hardware accelerators with general–purpose processor cores provides a viable alternative.

Contemporary applications are often single–threaded and therefore are not targeted at multicore architectures. Many desktop users are concurrently running only a small number of applications. Two- and four-core architectures have been able to provide gains for such users; however performance barriers will be reached as cores continue to multiply, while the number of programs run by a single user does not. Furthermore, performance will not scale with the power consumption. The added cost, in terms of power and price, of a whole second core on a single chip will not justify its limited utilization when running single–threaded applications. As such, methods that result in more efficient use of the added transistors must be developed.

The conventional microprocessor is adept at executing control-flow intensive code. The ability to execute flexible programs at high clock frequencies, comes at the expense of execution units available. Furthermore, the out-of-order superscalar processors must incur the cost of added die area dedicated to elaborate caching and branch prediction logic. Moreover, applications that exhibit large degrees of data parallelism are unnecessarily serialized on modern processors due to the limited number of functional units available. Such applications have their data–parallel nature lost when mapped into machine-readable binaries. The explicit parallel structure lost at compile time is only to be rediscovered by the processors at runtime in an effort to keep execution slots full. Rediscovery of lost parallelism comes at a great cost in terms of power consumption and die area. Data–parallel regions of code will benefit from technologies that deviate from the strict control-flow driven processing paradigm and integrate flexible dataflow computations using reconfigurable logic.

Application specific integrated circuits (ASIC) can be tailored to the needs of a single task providing high performance, low cost, and high efficiency. However, the nonrecurring engineering (NRE) costs of developing an ASIC and its application specific nature restrict its use. Due to these factors, ASICs fail as a medium for providing the benefits of custom–tailored logic to a host of applications that run on contemporary microprocessors. ASICs do find use as specific accelerators in the areas of video, audio, and physics engines, but static accelerators do not allow for a general computation platform. Reconfigurable hardware accelerators, based on field programmable gate array (FPGA) logic, can bridge the gap between custom ASIC accelerator designs and the flexibility of general–purpose processors.

FPGA technology, based on lookup tables (LUT) built into configurable logic blocks (CLB), provides the medium for dynamically matching the resource needs of applications with available functional units. FPGA avoids many of the NRE costs associated with ASIC technology, while also allowing for application-specific accelerators to be added to processors postproduction. FPGA devices allow for a large number of data–parallel operations to be executed simultaneously, without the overhead of scheduling and control logic. The overhead associated with control–driven processor architectures does and will continue to thwart efforts to extract more performance from contemporary superscalar designs. The data processing capabilities of FPGAs will be able to scale where single cores cannot, while doing so more efficiently than multicore designs.

We advocate a hybrid model of conventional microprocessor cores coupled with reconfigurable hardware accelerators, synthesized in an FPGA fabric. FPGAs provide a means to map the datapath of an application into a custom generated piece of hardware. Reconfigurable accelerators, based on FPGA technology, provide a means to exploit high levels of parallelism efficiently. They are able to do so by obviating the need for power and area hungry cache hierarchies and control structures (e.g., branch predictors). We have developed a test platform, using an FPGA with an embedded processor running a standard GNU/Linux distribution, to demonstrate the CPU/accelerator model we develop.

Mapping applications into a hybrid CPU/accelerator system is nontrivial. Many possibilities exist for accelerator access, mapping, resource allocation, and application interfacing. Without the necessary infrastructure for integrating FPGA-based accelerators with common applications, a hybrid CPU/FPGA platform will not be widely adopted. The focus of this work is to study a prototype of our hybrid model [1] and implement interfaces to the reconfigurable accelerators that are designed such that they become transparent to the application developer. The abstraction is made by encapsulating access to the accelerators in library calls. Furthermore, this work provides infrastructure in the form of a switchable interconnect framework for accelerator developers to target. Using our interface allows for reduced complexity by providing more of the supporting control logic and reducing signal count by more than 80% compared to a stand-alone accelerator. Moreover, the framework exports interfaces for the accelerators and operating system layer we develop, allowing software developers to integrate accelerators targeted at our platform in an easy and consistent fashion.

Current efforts for interfacing accelerators, reconfigurable or otherwise, with applications are not flexible. Reconfigurable systems available commercially make static decisions based on fixed rules. The rules for how data is transfered are codified in automatic code generation tools. Using the components developed in this work, future reconfigurable platforms will be able to adapt to their environment when accelerators are first accessed, or at runtime to changing conditions. Conditions that can change include data rates of the applications, the varying workload, priorities of contemporaneously executing tasks, and reconfigurable resource availability. We advocate pushing many of these scheduling decisions into the operating system, while still exporting an interface that is familiar to software developers for accessing accelerators. In doing so, the best choice of data transfer method and execution domain—hardware or software and in what form—can be made for accelerated applications run on our platform dynamically.

Image compression provides many levels of parallel granularity that can be mapped into reconfigurable accelerators. We use Joint Photographic Experts Group (JPEG) image compression as a motivating example throughout. On top of our GNU/Linux based platform, we use Independent JPEG Group (IJG) `cjpeg` [2] to demonstrate the capabilities of our interfaces and framework. We have accelerated `cjpeg` by encapsulating portions of the code in library calls that interface with reconfigurable logic via system calls made to the operating system. Multiple accelerator access implementations for JPEG are studied in the context of the switchable interconnect network developed.

1.1 Contributions

We develop a model for CPU/accelerator execution and implement a prototype on an FPGA with an embedded processor. Methods for integrating accelerators with the operating system and user applications are explored on our CPU/FPGA platform. We have developed an accelerator framework to integrate accelerators into the system, providing a consistent hardware interface. We describe a possible future model for computation integrating library call-like interfaces, the operating system, and shared hardware accelerators. The model allows for virtualization of the hardware resource, enabling the system to execute the accelerator tasks in hardware or software. An understanding of the tradeoffs inherent in our accelerator access methods is developed.

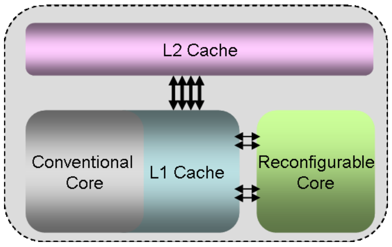

Figure 1.1 A possible future configuration of the reconfigurable interface with closer coupling.

The hybrid CPU/accelerator system, interfaces, and infrastructure developed are characterized. Current systems do not make use of accelerators that are swapped in and out on-the-fly, nor do they try to integrate the accelerators with applications and the operating system as presented here. The goal of such characterization is to provide guidance to those interested in developing applications incorporating hardware accelerators and accessing them using well-defined operating system and hardware interfaces. Furthermore, this serves as a stepping stone on the path to realizing a hybrid CPU/accelerator system, capable of outperforming contemporary computing platforms in terms of performance, cost, and power efficiency.

1.2 Related Work

One of the difficulties in developing future reconfigurable architectures is the lack of a general reconfigurable computer (RC) that can serve as a basis for ongoing work. However, there are systems built around FPGA devices with embedded processors that serve as development platforms [3,4]. These FPGA devices can be found in reconfigurable systems that can be integrated with a standard workstation computer [5]. Furthermore, numerous research efforts have developed reconfigurable platforms that serve as inspiration for this work [7–10].

We must first develop an understanding of the proximity of accelerators to the host processor, and the implications for accelerators being located close to or far from the processor, before we begin exploring other related reconfigurable platforms. In the context of this work, *close coupling* refers to the low latency and high bandwidth of channels between host and client, where the two share some form of high-speed connection. In our model, accelerators and the processor reside on the same die, such as the system shown in Figure 1.1. Close coupling, as is discussed in terms of current research efforts, often differs from the closeness of current commercial offerings. Research efforts have moved toward integrating conventional and reconfigurable logic, with data sharing accomplished via registers or the cache hierarchy, on the same die as the processor core. However, many commercial offerings are only accessible via system buses.

Loosely coupled systems may exist on a peripheral bus (e.g., [5]), and may execute an entire application locally, making little use of the host system. Such a model requires a compute-intense, latency-insensitive operation to be identified and mapped into hardware to make effective use of the reconfigurable substrate. We aim to take advantage of finer-grained accelerators by integrating them more closely with the CPU.

Nearly all of the commercial examples have a host-client relationship, with the conventional microprocessor serving as host to the FPGA-based system. The reconfigurable systems can be closely coupled to the host by means of high-speed, dedicated links (e.g., HyperTransport) or by using an FPGA with an embedded processor core to play host to the FPGA's reconfigurable fabric (e.g., the Virtex-II Pro used in our experiments). Few CPU/FPGA platforms have been empowered with operating systems, able to support common user applications, that incorporate native accelerator support. A model is developed in this work in which portions of applications running on the embedded processor are accelerated in the

FPGA fabric. Doing so pushes reconfigurable computing past some of the limitations of current CPU with coprocessor models, where only certain application domains are targeted and accelerated.

Stream-based computation is found in a wide variety of media applications and is the target of both conventional and reconfigurable systems research. Efforts have focused on capturing the inherent parallelism and data flow of streaming applications using new languages, while also trying to take advantage of the functional unit needs of these applications using reconfigurable hardware accelerators. Languages that assist the developer in mapping applications into parallel architectures are key to the widespread acceptance of reconfigurable platforms, such as the CPU/accelerator model we develop. StreamIt [6] is a language framework for developing stream-based applications that attempts to take advantage of the parallelism inherent in streaming codes. Recent efforts [9, 10] in reconfigurable computing architecture have focused on mapping applications to hardware resources, compiler initiatives to map applications to the reconfigurable system, and a stream-based data model. We incorporate the stream-based model for interaccelerator communication and as a model for application-to-accelerator mapping, but focus also on adding protection, operating system interfaces, and providing a more general platform for reconfigurable accelerator development.

Earlier projects [7, 8] had the goal of developing an optimized reconfigurable architecture where accelerators are accessed at the instruction set architecture (ISA) level. Current soft-core platforms [11] allow for accelerator access via specific instructions. Our application interface has the goal of developing the infrastructure necessary to move research efforts forward without breaking compatibility. We do so by encapsulating accelerator access in a library call-like interface. Our goal is to free the high-level applications developers from many of the system interface issues present in contemporary accelerator design.

Earlier work, such as [7, 12], integrated the accelerators into the processor pipeline with special purpose instructions. It is the ultimate goal of this work to show that tighter coupling is necessary to broaden the reach of reconfigurable hardware acceleration; however, the model posited by earlier work, attempting to integrate accelerators into the processor pipeline, may be too close for optimal acceleration. Furthermore, such tight coupling unnecessarily breaks compatibility with nonaccelerated applications and requires CPU manufacturers to commit to accelerator integration at an early design stage, possibly delaying the wider acceptance of a CPU/accelerator model.

The speed of memory access and the protection model employed are key to accelerator performance. Earlier work [10] discusses isolating the accelerators from the system via a stream lookaside buffer. It is the goal of this work to investigate finer-grained, but still protected, random and stream data accesses to enable a more general model of computation for the reconfigurable accelerators. Other research efforts have focused on the memory interface issues with respect to reconfigurable accelerators [13], but do not fully investigate protection. Furthermore, independent accelerator access using the virtual address space of applications has been explored in [14], but do not make use of finer-grained accelerators. No single previous work has focused on all of the system level details concurrently, including access methods, resource virtualization, and resource sharing.

Time multiplexing the reconfigurable substrate was explored in [15]. In their design, the reconfigurable resources are partitioned into regions that are occupied by pipeline stages. PipeRench serves as an example of accelerator modules chained together, via a defined interface. The data flows from one accelerator to the next—essentially pipelining—allowing for a small amount of reconfigurable fabric to be shared among multiple accelerators that cannot all be made resident concurrently. One of the main motivations for [10] is the ability to virtualize the hardware resources. Virtualization in that context allows for mapping an arbitrarily sized, tiled reconfigurable hardware specification into the available reconfigurable tiles. Essentially, the DFG for the application is mapped into the hardware resources. Our model assumes that the entire hardware application is resident in the FPGA fabric, but could use such techniques to enable future hardware virtualization. Furthermore, [15] introduces, and [10] extends, the idea of a staged execution in reconfigurable hardware with communication protocols between temporally and spatially adjacent accelerators similar to the framework we develop.

The notion of giving hardware direct access to virtual memory is developed in [16, 17]. In [16], the authors investigate the performance benefit of providing a network interface card with a translation unit capable of addressing virtual memory. The accesses to memory and the translation are independent of the processor. Enabling physical-to-virtual address translation allows their method to provide direct access to user space buffers. Providing direct access to internal device buffers relieves the processor of doing unnecessary copies and managing additional buffers for devices. The model developed in [17] provides low-latency access to network interfaces by providing direct mappings to I/O devices and support for user-level direct memory access (DMA). In [17], the authors leverage user space I/O and direct mappings to allow for low overhead message passing. The SHRIMP-I device is very similar to the access methods developed here where data is accumulated and then explicitly sent by executing a system call. Furthermore, current commercial offerings, such as Infiniband [18], allow for mapping device resources directly into applications to provide low-latency access for data transfers.

In [14], the authors develop an embedded CPU plus FPGA coprocessor model. In their model, the embedded CPU is running a general-purpose operating system. The goal of [14] is to provide a seamless integration of coprocessors resident in the FPGA fabric with applications running on the embedded processor. They achieve this goal by providing the coprocessors with access to the virtual address space of the host application. A major difference between [14] and our work is how and when data is transfered to the accelerators. Unlike [14], which always accesses memory, on behalf of the accelerators, across the bus using their built-in translation mechanism *every access*, we use a variety of methods, not all of which use our translation mechanism to access data, and we employ local buffering to reduce bus traffic. Providing multiple access paths allows our model to meet the needs of a wider variety of applications by hiding latencies due to data transfers and only providing protection and random access when needed.

In [19], the authors present a set of operating system and FPGA communication primitives for placing hardware tasks on a reconfigurable substrate. Our work attempts to accelerate common applications that would otherwise run on a general-purpose CPU using reconfigurable logic while still allowing those applications to take advantage of the services provided by a standard operating system. On the other hand, the model explored in [19] focuses on scheduling purely hardware tasks into the reconfigurable logic. Furthermore, the model of [19] uses a separate CPU and FPGA platform whereas we have an embedded CPU/FPGA model that allows for closer coupling and reduced overhead.

1.3 Example Applications

We have experience with a set of three sample applications: motion estimation, filtered backprojection, and JPEG image compression. We survey these examples to provide the motivation for our framework and the need for a consistent interface. Each of these examples has different buffering, data transfer, control, and timing requirements. Each application exhibits varying levels of parallelism that can be exploited by a hybrid CPU/accelerator system. To provide background for this thesis, we explore our sample application, JPEG image compression, in greater depth.

1.3.1 JPEG compression

JPEG exhibits many levels of parallelism, from high levels of instruction level parallelism (ILP) when processing a small portion of the image, medium-grained parallelism, or loop-level parallelism, when doing the discrete cosine transform (DCT) and quantization stages, and coarse-grained parallelism if the image compression is partitioned into multiple threads of execution. As most applications are some mix of parallel and sequential execution, JPEG serves as a relevant example for its many levels of parallelism, but also for its more sequential components such as run-length encoding and Huffman coding phases.

The basic unit of computation for JPEG is the *macroblock*. A macroblock is an 8 x 8 pixels portion of the image to be compressed. The phased behavior and data pipelining of macroblock computations allow for exploration of data flow in the computation more easily than is possible in many other applications.

At a high level, as illustrated in Figure 1.2, JPEG encoding of a single image proceeds in the following steps:

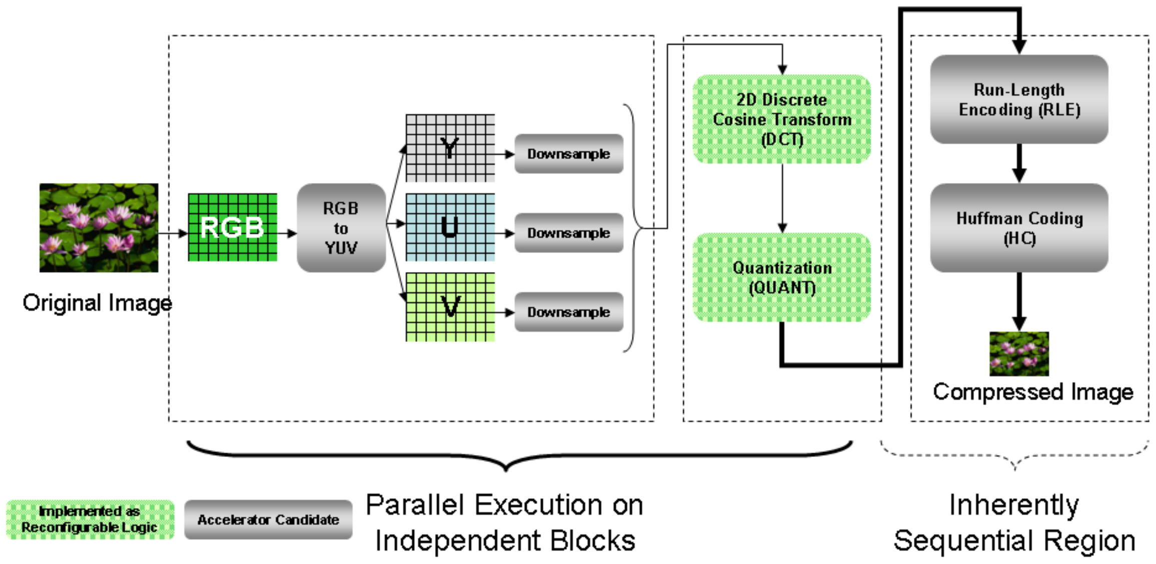
Original Image
RGB
RGB to YUV
Y
U
V
Downsample
Downsample
Downsample
2D Discrete Cosine Transform (DCT)
Quantization (QUANT)
Run-Length Encoding (RLE)
Huffman Coding (HC)
Compressed Image
Implemented as Reconfigurable Logic
Accelerator Candidate
Parallel Execution on Independent Blocks
Inherently Sequential Region

Figure 1.2 The logical flow of an image being compressed into a JPEG.

1. Partition a bitmap (uncompressed) representation of the image into 8 x 8 macroblocks representing 64-pixel tiles. Each of these blocks is independent for a majority of the processing.

2. Convert the image into luminance and two chrominance channels (YUV representation).

3. Independently perform a 2D DCT on each of the 8 x 8 macroblocks, yielding 64 coefficient values. The image is now in a frequency representation that is suitable for removing extraneous information while minimizing perceived quality loss to a human observer.

4. Quantize the 8 x 8 matrix of coefficients using a quantizing table chosen such that high frequency coefficients (i.e., those that are less noticeable to the human eye) are minimized. Quantization is simply a per-element divide of the result matrix of DCT by a specially chosen table. The same table can be used for each channel, a standard table can be chosen, or quantization values of all ones can be chosen to provide lossless compression (of course rendering quantization an unnecessary stage). The goal of quantization is to make as many as possible of the entries in the transformed matrix become zero in preparation for the next stage.

5. The high number of zeros now present in the 8 x 8 block can be efficiently reduced in size using run-length encoding (RLE). RLE is a sequential process as each block is run-length encoded in order. Data is no longer in a tiled form and cannot be processed independently as was possible earlier. However, more parallelism could be uncovered by partitioning RLE input and stitching together the results.

6. Further compression is realized by taking the output data from RLE and applying Huffman coding to it.

7. The image can now be wrapped in a JPEG header that combines the code table and quantize tables (if necessary), completing the encoding.

Many of the details of the JPEG standard are glossed over, but the levels of parallelism should be apparent. The contributions of all the major phases of JPEG encoding, in terms of processor cycles

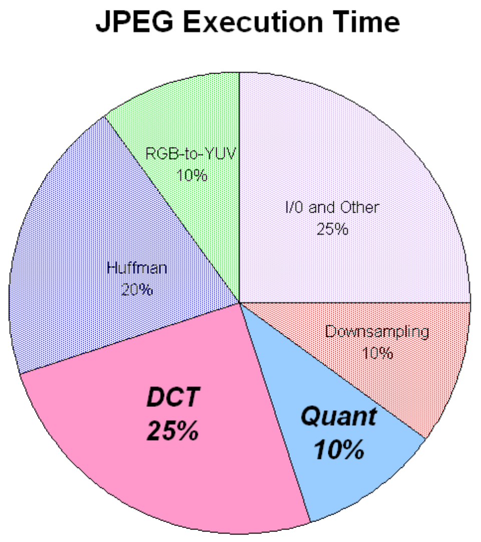

Figure 1.3 Portion of time spent in various parts of JPEG execution.

on a PowerPC processor, are depicted in Figure 1.3. The ILP is available due to how the macroblocks are processed. The process-level parallelism (PLP) could be realized by having multiple threads of execution with separate partitions of the image data and joining prior to the sequential stages. Exploiting PLP, however, has the high overhead of process/thread creation. The LLP is available in each of the steps (e.g., quantization and DCT) along with ILP. The flexibility of the standard and the number of permutations of input parameters make implementation more difficult for a general version. For this work, only the default compression quality setting for JPEG is chosen and studied, without loss of generality.

1.3.2 Motion estimation

A motion estimation (ME) accelerator for use in H.263 video encoding was developed as a motivating example for reconfigurable accelerators due to its inherent data parallelism. The accelerator was developed for the Xilinx ML310 board with a Virtex-II Pro FPGA and was later ported to the XUP platform used in this work. ME attempts to find the subframe of the previous full frame that most closely matches the subframe in the current frame that is having motion estimation performed on it. ME must be performed for each 16 x 16 pixel subframe of each frame of the video to be encoded. The execution time of the accelerator is an order of magnitude better that contemporary x86 machines. However, if the data movement overhead is considered, the accelerator is unable to provide such high levels of speedup.

The interface to the accelerator allows for easy hardware accelerator and accelerated application development. The ME accelerator presents itself to the system organizationally as an embedded Block RAM (BRAM) memory; architecturally as an address space mapped onto an auxiliary, single-master bus; and, from the point of view of the system software, as a memory-mapped region to be issued standard load and store instructions. However, this interface model does not fully utilize the interconnects and support services provided by the PowerPC and FPGA. The hardware-accelerated version of the video encoding application is unable to provide high levels of speedup on the XUP platform when compared to a software-only version, run on the PPC core embedded in the FPGA.

Motion estimation serves as a node in the taxonomy of reconfigurable accelerators examined by the level of parallel granularity they incorporate. Accelerating a full application, or an algorithm constituting the majority of the computation for the application, represents the most coarse-grained type of accelerator. ME is one level more fine-grained in terms of parallelism as it is one part of a larger application—in this case MPEG or H.263 video compression. Video encoding has a multitude of parallelism that can be exploited using reconfigurable accelerators and is explored as a motivating example, as both an application amenable to coarse granularity acceleration such as motion estimation and to finer-grained, loop-level parallelism.

1.3.3 Filtered back-projection

At a very coarse-grain level, there is full application or algorithm acceleration. One example where an entire algorithm constitutes the majority of the computation is filtered back-projection [20] (FBP). The entire application considered is medical imaging, where a majority of the computation is performed in a single section of the code. The algorithm for FBP is separated out and implemented in hardware. FBP is a single-use accelerator and has little utility for application developers and therefore does not lend itself well to a library-like interface. The FBP accelerator has high throughput requirements and is latency insensitive. FBP provides an example of very coarse-grained acceleration and stands in contrast to the work developed in this thesis where finer-grained accelerators are shown as a viable component of future computing systems.

CHAPTER 2

COMPUTATION MODELS

In this section a model for computation in a hybrid CPU/accelerator platform is developed. A model that considers all possible combinations of implementations and interfaces exacerbates the difficulty of the mapping process. A fundamental issue for such a hybrid system is that the mapping between software defined algorithms and low-level hardware constructs can be a daunting task. Without a strict protocol for accelerator-to-application interaction, programmers cannot be certain that their applications will remain correct across platforms or versions. Where data exist, what entities own those data, and when the ownership of those data is transfered between entities are all key concerns in a multiprogramming environment. A restricted view of the reconfigurable system, with many of the architecture-specific interactions abstracted away, facilitates an environment appropriate for developing reconfigurable accelerators and corresponding applications. The set of interfaces provided by this work, coupled with the interaction models explored in this chapter, provides the infrastructure necessary for a hybrid CPU/accelerator computing model.

The stream-based computation model [21] for reconfigurable systems and FPGAs as a platform for building dataflow architectures [22] influence the accelerator framework and interaction models we develop. Independent of reconfigurable computing, dataflow machines have been studied extensively [23]. Although dataflow architectures showed promise in certain application domains, they were unsuccessful in attaining wide acceptance. Many of the techniques developed for dataflow can be mapped into our model, including programming language support [24]. In our model, data dependence information representing the data flow in accelerated applications is coupled with the logical data movement mechanisms provided by streams, for data transfer between applications and the operating system. Furthermore, the interaccelerator communication mechanism is streamlike, but extends that model by also providing mechanisms to stall computation and to randomly access memory when needed.

Hardware-software codesign has long been an interest of researchers [25]. Mapping an application into our hybrid model requires parts of it to run in software on the embedded CPU and parts to run in hardware as accelerators. Our contributions do not solve how one decides what part of an application should be placed in either domain. However, before guided hardware-software partitioning can take place, many architectural specifications to enable codesign must be defined for a CPU/accelerator hybrid computing environment including: coherence between the memory of the accelerators, CPU caches, and system memory; consistency models for ordering of accesses to different locations in the same data set shared by both CPU and accelerator; and lastly, the necessary control semantics and dataflow used within the reconfigurable accelerator framework. With codesign requirements in mind, we develop a model to assist developers in partitioning their applications across the CPU and the accelerator framework we develop.

Certain program codes exhibit a high degree of data parallelism, and are therefore more able to exploit the spatial computation abilities of hardware accelerators. Our model can provide better scalability for data-parallel applications than simply making wider execution core processors. We are able to provide better scaling by maintaining and exploiting the otherwise lost data parallelism present in many codes. A CPU/accelerator system removes the memory bottleneck associated with the load-store architecture found in general-purpose processors. The wide array of memory available in FPGA architectures including distributed memories, look up tables (LUT), BRAMs, and large off-chip bandwidth to static random access memory (SRAM) and synchronous dynamic random access memory (SDRAM), allows for the constricted, single path to system memory present in conventional microprocessors to be avoided. Furthermore, FPGA-based accelerators allow for arbitrary composition of execution units, memory, and interconnects that are custom tailored to the current computation. The flexibility of FPGA-based accelerators, and their application-specific nature, helps to address the claims of inefficiency leveled at stand-alone dataflow architectures.

We introduce an application-to-accelerator data transfer and execution model for developing applications using our interfaces and accelerator framework. The models developed provide the means to systematically map applications, straddling both the software and hardware domains, into our platform. The presence of concurrent execution in both software and hardware complicates the process of hardware-software partitioning. Our computation model aims to reduce the complexity of developing concurrent hardware and software applications by providing simple, consistent interfaces for integrating accelerators and software applications. This chapter examines the fundamental application interface model and system structure underpinning our CPU/accelerator hybrid computing platform.

2.1 Application Partitioning

Determining what parts of an application should be synthesized in hardware and those suitable for software execution is an area of research know as software-hardware codesign. The codesign process consists of using some means to map parts of an application into, possibly many, hardware units while other parts remain in software, running on the host processor or processors. The designer must optimize over the conflicting domains of power, area, and performance. The computing model presented in this chapter is aimed at applications predominately run on the embedded processor. We do not try to solve the hardware-software codesign problem, but instead to provide the infrastructure necessary to enable guided application partitioning, and to ease software/hardware interfacing, by encapsulation of the hardware units, or accelerators in our model, inside library calls. We partition JPEG image encoding across both hardware accelerators and the embedded processor as a case study in hardware-software codesign for a CPU/accelerator platform.

Frameworks exist for developing applications targeted at heterogeneous environments [26, 27]. Design space exploration and design frameworks are able to assist developers in reaching their system-level design goals through modeling and analysis. Rapid system space exploration allows for myriad design choices to be analyzed and the optimal or near-optimal selected. Before such tools can become a utility for designers who wish to target our hybrid CPU/accelerator platform, the model we espouse must be throughly understood. We characterize different access methods and data transfer modes that can be incorporated into such system space exploration frameworks, to enable accurate modeling of heterogeneous system designs that incorporate accelerators.

It may not be necessary to use extensive design space exploration to achieve desired performance goals. Developers may understand their applications well enough to pinpoint which sections of code would be best mapped to reconfigurable accelerators. Insight into how best to partition the application can be gained empirically through profiling and timing measurements taken on a full software version of the application. Furthermore, studying the data flow graphs (DFG) and control flow graphs (CFG) provides better resolution for understanding what parts of the application have the data movement and processing patterns amenable to FPGA acceleration. As compute intensive, data-parallel regions are found, they can be mapped into accelerators. In the future, common routines will be available as libraries, which can be loaded at runtime, and are easily accessed by developers.

Software analysis tools and methods are already well understood by software developers who seek highly optimized code. Therefore, it is possible for software developers to partition applications, developing a portion of the code for CPU execution, and isolating regions for implementation as a set of accelerators that the developers can integrate into their design as hardware modules. Using this partition flow reduces the number of new tools developers must learn, easing the transition to our hybrid CPU/accelerator model.

The application developer does not need to be concerned with designing a functional block in a hardware description language (HDL) in our model. Many common components would already be available in a stock library. Likewise, hardware developers could build highly tuned accelerators for commonly used functions just as there are standard synthesis macrocells for ASIC design and IP libraries available today. By providing a framework that remains consistent across all domains, the accelerators need only be developed once and can be accessed by any application developer running our platform.

Should areas of code be identified as candidates for acceleration, and suitable off-the-shelf library components are not available, the software developer can hand off the task of accelerator design to a hardware developer. The interfaces our platform provides allow for these two tasks, software application development and hardware accelerator design, to be mostly decoupled. The well-defined abstraction layer provides the software and hardware developers with the communication interfaces necessary to design their components in isolation, decoupling one design process from the other. The applications developers and the accelerator designers need only understand the data flow and data representations that are transfered between the two domains using the abstraction layer provided by our model.

Custom accelerators for applications targeted at our hybrid model provide optimal results. In the future it may be possible to develop tools, such as profiling compilers with integrated, high-level language synthesis capabilities, that take arbitrary C code and map it into our platform without any additional input from the developer. However, it is possible to realize there are a common set of tools that have been in programmers' repertoires for many years, such as the standard C library. Using such an approach, accelerators can be distributed as hardware/software libraries and incorporated into designs without the complexity of using such automatic code generation tools. Furthermore, in certain domains there are oft used algorithms that are taken off the shelf when a new application is developed. Examples of recurring algorithms include transform methods and filters for DSP applications. Combining the notion of code reuse with the need for defined interfaces for reconfigurable systems results in the library call interface model that we present.

The "accelerator as a library call" interface method we have developed for our heterogeneous system lends itself to programming with recurring functional blocks. Just as sine (`sin()`) is part of the standard C library, we advocate adding a set of hardware accelerator libraries to enhance the performance of commonly used library calls. Hardware accelerator-based libraries promote the same code reuse patterns seen in software design today. Not all procedures wrapped as library calls found in standard libraries today would be mapped into accelerators, but many could be. Furthermore, by incorporating hardware and software versions of the accelerator into the library, source-level compatibility is maintained. Dynamically linked accelerator libraries that also include software implementations allow for binary-level compatibility by enabling platforms that lack accelerators to still run the same executables, while those that have the necessary hardware resources can take advantage of faster, more efficient accelerators. The key point is that regardless of the implementation method, the basic application interface remains consistent to the application developer across implementations since the call made has similar (or the same) syntax and semantics as any other library call.

Portability is maintained as both the software and hardware versions of the procedure coexist in the library object file, with the operating system mapping the appropriate version at runtime. A more subtle portability issue is performance degradation and semantic differences between platforms. We have isolated application development and accelerator development from the system, allowing for optimization for their respective interfaces. Taking advantage of evolving hardware platforms is a valid concern explored in previous work such as [10], but is not examined further here.

2.2 Operating System Layer

Our model sets the stage for future work to explore operating systems integrating hardware accelerators, capable of making runtime decisions regarding scheduling and resource allocation. In our model, the operating system is used as an abstraction layer to provide protection between accelerators and surrounding hardware in the system. Furthermore, the operating system provides protection and isolation between users accessing accelerators. The operating system manipulates logical streams of data

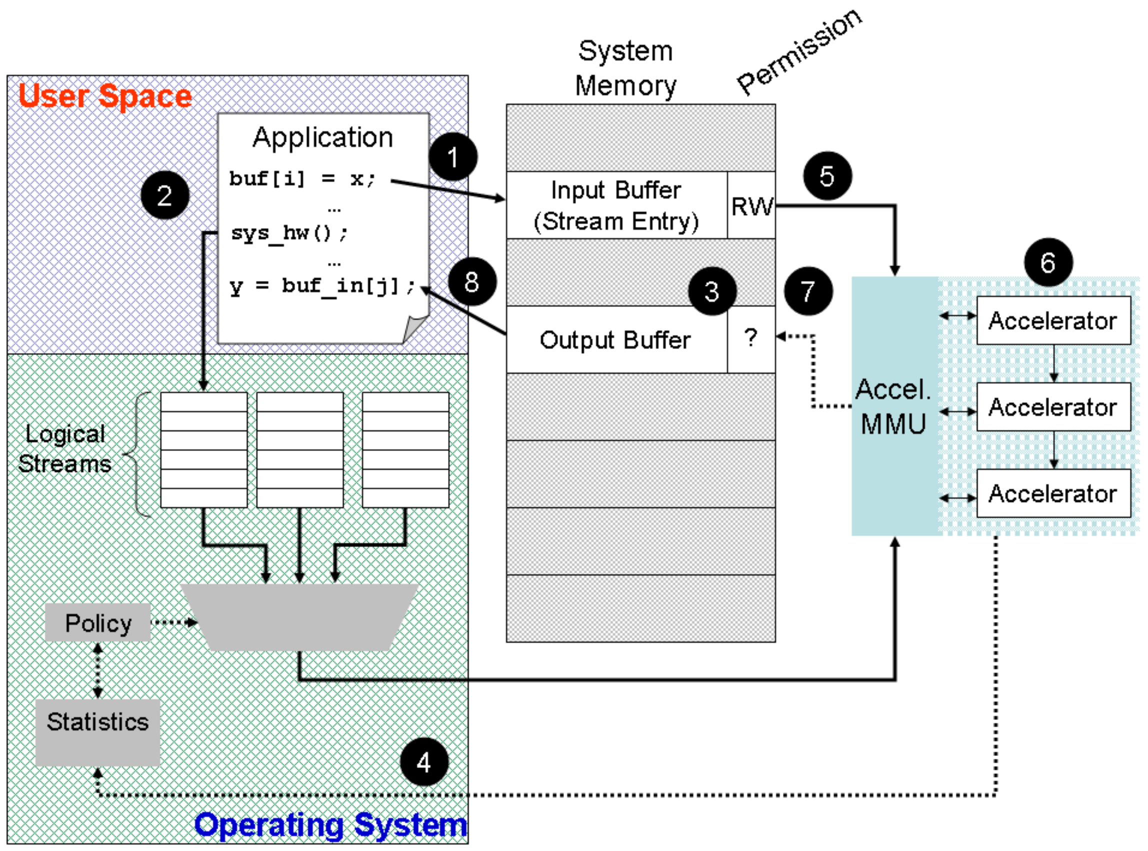

Figure 2.1 Access model with logical streams.

blocks that are then moved to the accelerator using the concept of a transaction. Each data block is then allowed to stream through the accelerators. The operating system layer for reconfigurable accelerators provides the "glue" necessary for joining the application compute model, using a DLL approach, to the accelerator compute model of streaming dataflow execution.

A great deal of flexibility exists in how the operating system layer is implemented and many avenues for future work exist. Discussion of the complete compute model is discussed in this section while the details of our implementation of a subset of our model is left to a later chapter.

2.2.1 Overview: Operating system layer

Operating system integration is the cornerstone of our model for future CPU/accelerator hybrid computing systems. The model provides interfaces to the applications and to the accelerators. The intermediate layer, encapsulated in the operating system, handles the mapping of user requests to accelerator resources. Figure 2.1 shows a high level view of how an application accesses the accelerator framework in our computation model. The application begins accelerator access by setting up a user output buffer, and a user input buffer and filling that buffer, which is mapped into its virtual address space (1), with the data to be sent to the accelerator. The application has no knowledge of where this virtual address range is physically mapped. Once all the data is ready for the accelerator, the user space application passes control over to the operating system through a system call (2). At this point, the operating system is given the user input buffer by the user and the location of the user output buffer for the results from accelerator execution. The operating system marks the user output buffer region as inaccessible by the application (3) to allow the application and accelerator to run concurrently, but to halt the application should it attempt to access data before it is ready.

The operating system is now in control of the user's data and must decide whether to execute the requested accelerator library function in software or hardware. The accelerator may be busy, or the application may have extinguished its resource allocation for the given time interval. The operating system buffers the input data logically in a stream. A logical stream connects an application to an accelerator. Once the data is recorded and buffers locked appropriately, the system call can return allowing the application to continue running while the accelerators in our framework execute. The application can continue to add stream entries into the logical stream by making repeated calls to the accelerator library.

The operating system continually tracks system behavior and records statistics. These statistics, when combined with system policy, are used to determine when and how to execute the data present in the logical input streams (4). The operating system scheduler will eventually allow a data block to be moved from the logical input stream entries into the accelerator (5) to begin execution. The accelerators are then allowed to execute (6), consuming the input data from the application. The results are eventually placed back in the output buffer and the presence of the data is made known to the system (7). The application will eventually access the results yielding one of two outcomes: (a) the data are not yet ready, and the access traps to the operating system, or (b) the accelerator has completed, the data is available, and execution continues as normal for the application.

2.2.2 Scheduler integration

A multitasking operating system has the responsibility of providing fair resource allocation, meeting execution deadlines for real-time processes, and delivering on quality of service (QoS) requirements for its users. In our platform, each process is allowed to execute in a time multiplexed fashion, while respecting a set of administrator-defined priorities. Hardware resources are partition based on a similar priority scheme. The goal is to provide an environment where many applications can run without any single application taking more than its share of the resources as dictated by policy set by the administrator of the machine. The need to enforce system policy and resource limits using the accelerator and reconfigurable resources in our system is no different.

To accommodate the requirements of reconfigurable accelerators, the base scheduling model must be extended. To that end, the operating system must provide "fair" allocation of accelerator resources to applications requesting access to the accelerators. Resource allocation must take into account accelerator utilization, reconfiguration overhead, buffering requirements, and system policy in determining a fair allocation schedule. Reconfiguration time and area allocation could be compared with the gain in performance provided by hardware accelerators. Moreover, our platform can avoid reconfiguration by having multiple, smaller parts of independent applications resident in the accelerator as depicted in Figure 2.2. With multiple applications using the same framework concurrently, the scheduler need only switch between them on context switches and avoids costly runtime reconfiguration.

Previous work studied scheduling of hardware accelerators mapped into reconfigurable systems. These works focus on a slightly different accelerator access model than what is developed in this thesis. They are focused on reconfigurable platforms in which the majority of the application is implemented in hardware or do not consider the interface between the applications running on the CPU and the execution kernels running in the reconfigurable fabric. However, these previous works provide insight into what considerations must be made when developing scheduling algorithms for reconfigurable systems.

Other fully dynamic allocation methods in [28] proved to have high overhead. However, the system model for reconfiguration required many more decisions to be made regarding accelerator placement than we consider, resulting in added computation at the end of each reconfiguration epoch. In [29], the author reexamines the same platform and develops a quasi-static method for scheduling. The quasi-static method generates a schedule statically and then allows it to adapt at runtime, providing reduced overhead, while maintaining many benefits of dynamic scheduling. In a more recent effort, runtime

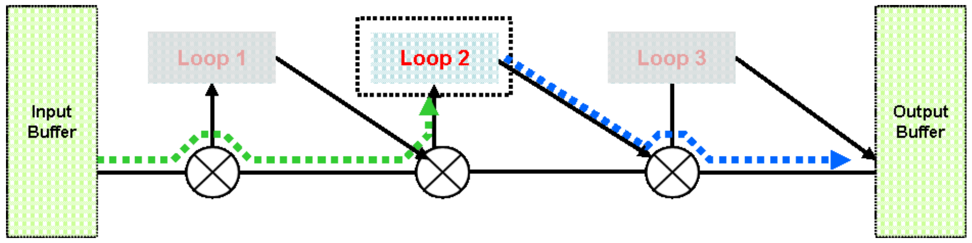

Figure 2.2 Loop bodies can be mapped into reconfigurable blocks and time multiplexed.

allocation of reconfigurable resources, with both software and hardware implementations available in a library, is explored in [30]. The authors use a multiconstraint knapsack algorithm to determine the best division of resources between hardware and software.

For our model, runtime statistics are kept, similar to the scoreboard used in [30], and analyzed by the operating system. Using these statistics, the operating system varies the access method based on system policy and the parameters developed in this work. Reconfiguration decisions would be based on the runtime requirements of the system and system policy. The reduced complexity of scheduling decisions in our platform may allow for fully dynamic scheduling, but depending on usage patterns, statically placed hints may be necessary to allow for fast convergence of resource allocation.

2.2.3 Stream model maintenance

Streaming as a general model for computation breaks down for applications that make use of even a moderate amount of control-flow or require a large number of random accesses to memory. For data parallel computations mapped into hardware accelerators, streaming may be a fitting model. However, models that allow for streams into and between reconfigurable blocks need to account for buffering due to rate mismatch and nonresident hardware. Our model removes the burden of internal buffer management by using an interaccelerator communication protocol that does give the accelerators a notion of unbounded external buffers. Furthermore, transactions with the accelerator framework are run to completion—there is no interleaving of accesses to the accelerators. Our concept of a logical stream consists of multiple transactions, only one of which is allowed access to the accelerators at a time. The data block from a single transaction is then allowed to stream through the accelerators.

In pushing the level of atomicity and the need to manage buffers up to the software/hardware interface, application-to-accelerator streams need not be managed by the hardware accelerator framework. However, many of the applications that will make use of reconfigurable accelerators have streaming data. Not using any notion of streams and forcing accelerator calls to block on every access leads to reduced concurrency, degrading performance. Process swapping, even when multiple processes are runnable, may not be a viable option as the cost of context switching negates accelerator benefits. A method where streams can be built to accept data from applications and buffer results is necessary to avoid time wasted blocking. For full utilization of both the CPU and FPGA resources in our model, data must be accepted from the user and processed concurrently with CPU operation.

We build a logical stream between applications and the accelerators into the operating system layer of our computation model. If streams were built into the accelerator framework, the fact that buffer management in FPGA hardware is difficult due to placement and routing constraints would become a severe limitation. General-purpose processors have developed highly efficient memory hierarchies. The operating system running on the CPU is able to dynamically reallocate buffers for running processes on the fly, using the mature paging facilities of contemporary CPUs. The interface between the accelerator and the application is a fitting place to insert the notion of a stream. Each application-to-accelerator input mapping is abstracted via a stream. When an application passes data to the operating system during an accelerator call, that data is placed in the corresponding stream. The application is then able to continue running immediately using nonblocking call semantics.

Input streams to the accelerator enforce in-order execution of data blocks for a given application-to-accelerator mapping. However, the application may not access the results in the same order that the corresponding input was produced. Our model allows access to results without undue delay by having the operating system place results in independent buffers. Doing so enables the application to access any data that is available without having to respect ordering, further increasing the flexibility of the model. Access to the output buffers is handled similar to presence bits found in dataflow machines. The presence bits are realized by exploiting the memory translation and protection facilities of the CPU.

The logical streams that exist between applications and accelerators are a powerful abstraction. The access models presented in this work, along with other possible interconnection and system access methods, can be mapped in such a way that data elements (1) come out of the logical input streams, (2) are written to the framework input buffer, (3) are processed, and (4) are then written into the logical output buffer for the accelerator-to-application mapping. The logical mapping can be a direct mapping between the application's virtual address space and the physical address space of the framework's input buffer. Should another application have the framework mapped, the operating system could place the application's input data into the stream buffered in system memory. The operating system could then process the data in the logical stream when it allocates accelerator access to the application or the operating system is able to process the data using a software thread, virtualizing the accelerator resource as needed.

There are other implementation-level benefits for accelerator resource management and allocation integrated into the operating system. The buffers present in our model are generated using a finite, shared resource (i.e., system memory). The software application has the illusion of infinite buffers, but the operating system cannot allocate even large, finite buffers for the application when available memory is low. Should an application exhaust its resource allocation, the operating system must block the execution of an accelerator call. All accesses to the accelerators go through a system call, providing normally nonblocking semantics for increased concurrency, but may block, if necessary, without impacting the programming model. At these well-defined points, the operating system can disallow further buffer resource allocation to the application until software virtualized or physical hardware accelerators are able to process some of the process' associated data resident in its streams. A system call interface thus provides protection and resource management, at well-defined points in the execution of accelerated applications, using our model.

Nonblocking call semantics lead to asynchronous execution of the application partitioned across the CPU and accelerator. Our model must handle the case where the application tries to access the output of the accelerated call before the results are ready. If the application is forced to make system calls on every data access, the overhead would negate any performance gained by using reconfigurable accelerators. If blocking call semantics are allowed for accelerator access, then the call could put the process to sleep until all the data are ready. However, such a model ruins concurrency in applications that have data available to be processed, but do not need the results of the pending call they are blocking on immediately. To handle this case, we logically tag each output buffer with a presence bit. Our model makes the output buffers unreadable until the data is available using the memory management facilities of the CPU. As a simple means to provide these capabilities, the framework can manipulate the page table entries corresponding to the buffers when data is written. If the application tries to access data early, it will cause an exception which will cause the operating system to put the application to sleep until a time when the data is ready.

The performance implications of using page table management to avoid premature data access without hardware support can be immense. Having tagged translation lookaside buffer (TLB) entries that can be injected into the TLB when a change is required mitigates the cost of flushing the TLB on every page table change (i.e., accelerator output buffer presence being set or cleared). Keeping the TLB consistent with the accelerator subsystem through TLB injection, as opposed to using flushing as a means to ensure consistency, allows our model to only suffer performance degradation in the (hopefully) less common case, where the application attempts to prematurely access data. A TLB miss could take hundreds of cycles or more in a software managed TLB architecture. A means to avoid the TLB overhead must be provided if page table entries and their respective entries in the TLB are going to be used as a means to guard against premature data access.

The tricky issue of stream management has been pushed to a level inside our model, the operating system layer, that can provide buffer management effectively. In our model, the operating system is in control of resource allocation and accelerator management. The system call and library interface method provides the necessary abstraction to allow the operating system to allocate reconfigurable resources fairly while remaining transparent to the application. One down side to this method is a restriction on how the accelerators can be accessed. For applications that are inherently streaming or those amenable to some form of data structure windowing, a streaming model provides the necessary flexibility. For some workloads, the stream model may be too restrictive to fully exploit the encapsulated accelerators. In such situations, other means must be used to access accelerators should performance be the only concern, but our model is still be capable of providing correct execution and a consistent interface.

Streams offer a natural model for expressing the independent blocks of data sent to the accelerators by many inherently parallel applications. Having only a limited number of accelerators, but numerous applications trying to access said accelerators, requires some form of buffer management. Our model is able to handle the many-to-few mapping situation by providing streams for each application-to-accelerator mapping and output buffers that block computation until data is ready. The abstraction provided by the streaming model we have incorporated into our hybrid system also enables seamless virtualization of the hardware accelerators: whether the accessor is software or hardware, as long as the rules governing the input and output buffers are respected, the application has no concern for how that data is processed. Using our model, concurrency is maximized, protection is ensured, and resource sharing is implemented efficiently.

2.2.4 Accelerator virtualization

A goal for future operating systems based on this work is full accelerator *virtualization*. Virtualization refers to the ability to execute a hardware-accelerated kernel of an application in software should some condition exist not allowing for execution in the reconfigurable fabric. The way in which the application developer accesses the accelerators remains functionally unchanged when virtualization takes place. Furthermore, protection models and protection boundaries when accessing the library must be maintained in both the virtualized version running on the CPU and the accelerated version running on the FPGA.

Virtualization has many benefits including added concurrency, backward and forward compatibility when well-defined semantics are defined and obeyed by the developer, partial hardware acceleration, priority-based resource sharing, and a consistent interface for application developers. There are two features of the switchable interconnect accelerator framework interface our computing model targets that enable virtualization. The first is through operating system management of users' access to the framework. The second is the explicit operating system-to-user level transfers that define transaction boundaries.

One issue often overlooked by reconfigurable systems embedded in applications is the execution cycles wasted during a blocking call to a hardware accelerator. In [7], the semantics of a call to the reconfigurable resource dictate blocking calls and thus limit concurrency. Previous work has also looked at the embedded processor as a controller—merely orchestrating data movement in the system and not participating in computation. Our computation model assumes a programming paradigm where the microprocessor is the focus of the CPU/accelerator platform; the CPU is the main executer of the application. The hardware accelerator resources are only a method for accelerating segments of the code wrapped as library calls that may be reused throughout execution.

Blocking while independent action is taken by the accelerator framework (e.g., data movement from main memory or accelerator execution) is a wasteful approach. By making forward progress on the portion of the application running on the CPU while the accelerator progresses, concurrency is increased and in turn performance is increased as well. The bulk of execution can take place while DMA transfers to the framework's local store and execution on the accelerators occurs, while some amount of computation can take place on the host system that would otherwise be wasted cycles spent polling or waiting on an interrupt. However, virtualization to increase concurrency may not yield any performance gains and may in fact degrade application performance, occupying the processor while it could otherwise be attending to the control actions needed by the accelerator. Furthermore, such an approach must weigh the interference caused by accessing the same set of data or other shared resources as the accelerator versus the potential gain provided by concurrent execution.

Performance is only one small part of reconfigurable virtualization; other virtualization techniques are critical for the acceptance and the proliferation of applications incorporating reconfigurable hardware. A library call approach to accelerator access allows for seamless movement between software and hardware execution. Providing both hardware and software versions of the library allows for systems without reconfigurable fabric, or embedded accelerators, to still execute the code, albeit at possibly reduced performance. Moreover, not adding additional instructions to the architecture maintains backward compatibility with other machines of the same architecture family.

Software virtualization has not yet been fully implemented in our platform. There are multiple avenues for future work to demonstrate further software virtualization. It is possible to run the virtualized accelerator code inside the operating system, but this method may prove intractable or impossible if certain functionality exported to applications by the operating system is not available within the operating system itself, such as system calls. The operating system kernel may also be executing with a reduced memory space that could preclude operating system execution. A more viable means to execute accelerator code in software would be to spawn off a new thread or tasklet that could execute the code at a later time. In symmetric multithreaded (SMT) environments, a helper thread could be kept resident and swapped in with little overhead should software virtualization be necessary, similar to what was studied by Keckler et al. in [31] for use in the M-Machine. Although we provide insight into the application to operating system interface, we place no restraints on how the operating system handles software virtualization. Without these restraints in our model, future systems designers will have the freedom to chose an appropriate method for software virtualization.

The operating system interfaces presented in this work allow the operating system to virtualize the accelerator resources by providing a consistent view of buffers to the application, regardless of the data movement technique that was employed. If resources are unavailable, or the mapping must be changed in the case of direct mapped access, the operating system can take the necessary actions at call boundaries. Pushing these corrective actions into the system call allows the application to execute accelerated code without regard to what hardware resources are physically available, providing complete virtualization of the accelerator access.

2.3 Stream-Based Computation

A *stream* is both a data structure containing blocks of data and a set of methods for accessing those blocks. Conceptually, a stream is a collection of independent data items flowing between processing stages. An example of such a data block is an 8 x 8 macroblock for JPEG. A stream is different from a scalar or a vector quantity in that it is multiple values, of possibly differ types, in some conglomeration. Streams are accessed with the understanding that not all values of the stream may be available for the computation initially. A stream of data can be thought of as a first-in, first-out (FIFO) structure that is implemented as a queue. In our model, the stream has values (scalars or blocks) inserted into it by the data producing *source* entity, which are consumed by the the *sink* entity, one at a time.

The semantics of the stream allow for not all data to be available when accessed by the sink. If no data are in the queue, the stream can implement a blocking read which will cause the accelerator to wait until data are available. Likewise, if the queue is full, the producer application must implement a blocking write or expect its data to be lost. Fundamentally, the streaming model is applied at many levels in the interaction model developed here as a means to connect components the CPU/accelerator

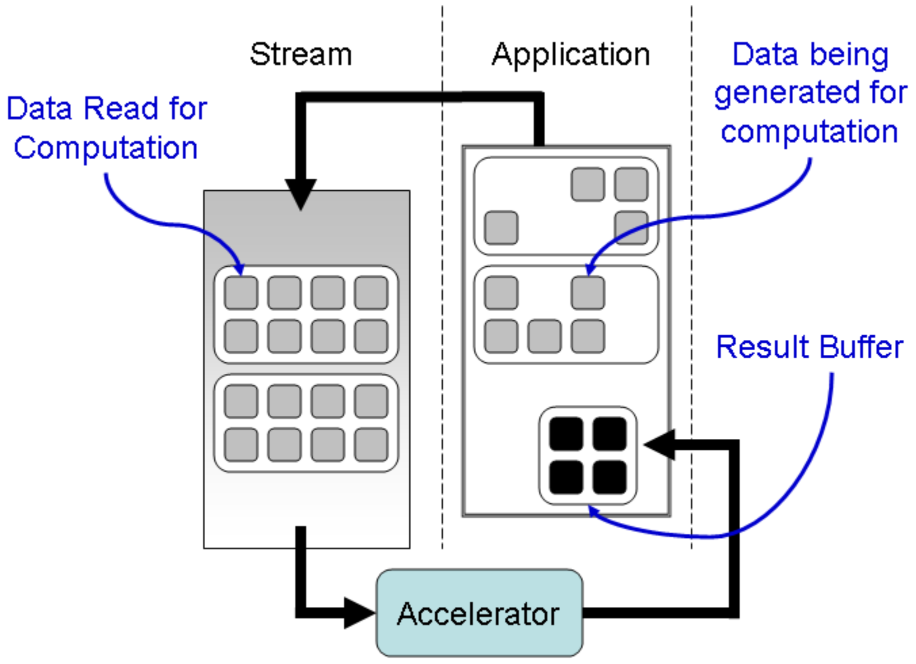
Stream
Application
Data Read for Computation
Data being generated for computation
Result Buffer
Accelerator

Figure 2.3 Stream transactions with explicit control transfers represented by dashed lines.

system. The major use of streams in this work is as a means to collect logical units of ready data, managed by the operating system, that are awaiting processing as depicted in Figure 2.3.

2.3.1 Application to accelerator streaming

Streaming in our model differs slightly from previous stream designs. Other reconfigurable systems have taken a strictly stream-based computation model [21]. As in other models, our model has all data flowing through the computation units as they become available. Data is manipulated at each accelerator and the results are then placed in an output stream, completing the transaction. Conceptually, these streams can be made infinite, but in practice infinite streams are not possible. Therefore, the system designers must consider blocking reads and blocking writes. The way our model handles disruptions in the flow of data is one of the differences between our model and other stream designs. Once the application has placed a value into the stream, the application may or may not have a reference to the transferred data. Furthermore, systems that allow for runtime reconfiguration while the accelerator is in use must treat these streams as a form of state, to be maintained across accelerator context switches. We specify distinct points where control of data and of accelerator resources is transfered between entities in our model. Understanding the semantics and state associated with streams in our model is key to programming our platform, while also having implications for extensions of our work such as accelerator sharing in a multiprocessing environment.

Accelerator resource sharing between applications requires extra care be taken to ensure memory consistency and coherence between memories and caches in the system. When resource sharing is allowed, proper state maintenance is an issue. One example of the need for state maintenance is the point when a source has added a value to an input stream, but before the sink has retrieved the result, at which point reconfiguration is triggered. State maintenance could require a costly saving of all the data in a stream and saving of the partially computed values in the accelerator. An example demonstrating the overhead of state maintenance is SCORE [21]. Streams in the SCORE model are blocks of data that can be swapped out when hardware is reconfigured, maintaining the state. In an effort to reduce accelerator context switching overhead, and the difficult task of extracting partially computed values from accelerators, our model avoids maintaining a large amount of internal state.

We define a *transaction* as an explicit transfer of data and control from the operating system interface to the accelerator subsystem, with the guarantee that the result will be computed and provided at a later time. Once the transaction is completed, one entity or the other has control of the object, but never both. State is never maintained in the framework memories, accelerators, or interconnect links between transaction boundaries. Instead, the state is maintained by the operating system, making the stream resident in memory. Our model simplifies state maintenance by reducing the number of points where accelerator context can be swapped to transaction boundaries. With this added constraint, system memory-based streams obviate the need to save any partial state between reconfiguration or process-to-accelerator remappings, such as in the event of a priority-based context switch.

A preemptive system allows for an executing thread of an application to be stopped, its context saved, and another thread allowed to run. When preemption is allowed, direct mappings between user applications and hardware resources require buffering of input and output streams on process context switch boundaries. An example where buffering is required occurs when a user is given direct access to accelerator buffers with a data block (which may be incomplete) in the local memory of the framework and the application is put to sleep. While the application sleeps, reconfiguration is triggered. Such a situation can be handled in our model as all accelerator initiations begin at well-defined entry points to the operating system. The operating system knows the state of the accelerator, and which processes it belongs to, upon reconfiguration points or at accelerator access call boundaries. The operating system takes the necessary action, such as buffering the input to the accelerator in system memory and restarting the transaction at a later time.

Using streams to move the majority of data consumed by the accelerators allows the implementation to exploit high-throughput transfers at the expense of latency, relative to block-by-block transfers. Results in this work show buffering into streams followed by explicit flushes to accelerator memory more fully and efficiently utilize the framework and interconnection resources. To that end, the switchable framework, expanded upon in a later chapter, makes use of cache buffering for direct mapping methods and DMA transfers across the system bus from system memory for other access methods.

The framework does not enforce a policy with regard to stream block size or value type. Conceptually, the application places data in a stream in one of three ways: into a user-level buffer, passing a pointer to the operating system via a system call; into a DMA enabled region directly from user space; or into the hardware buffer mapped directly into the application's virtual address space. The last two methods of inserting data into a stream take effect when the system call is executed, but less work needs to be done inside kernel space compared to calls using kernel controlled data transfers. Further explanation of various access methods and framework is left to a later chapter.

For many applications used to show the efficacy of a reconfigurable system (e.g., transform methods, encryption, media encoding), there is a large amount of coarse-grained data-parallelism inherent in the codes. A streaming model of computation is amenable to applications that have blocks of data that can be sent off for hardware acceleration. However, not all applications are able to produce large blocks of data due to data dependences and control constraints. The inability to dereference a pointer inside accelerators would force many codes to be refactored at large expense, or would be made outright intractable to port to a CPU/accelerator system. The marshaling that would be needed to consolidate data from pointer-based structures into contiguous data regions for accelerators can exact a high cost due to added copies. As an extension to the streaming model we employ, we add the ability to randomly access system memory during otherwise streaming computation. Incorporating protected side accesses into our model allows for a wider range of applications to make use of the computational power of hardware accelerators. The details of our protected access method to system memory is left to a later chapter.

There are limitations to the stream-based computation model. Data must be accessed in stream block-sized chunks and the data are not randomly accessible from within the stream. Another possible model aggregates the necessary data and passes the data by value to the reconfigurable hardware. The marshaling of data done to allow for blocks to be passed to accelerators results in additional overhead from extraneous copies. Models that guarantee logically unbounded streams allow for hardware independence, but at the expense of hardware and software management simplicity. The limitations imposed by purely streaming models serve as the impetus for developing our switchable interconnect network and the computation model discussed in this chapter. We adapt a streaming approach to a model that allows for aggregating and accessing the streams in a way that allows for reduced buffering requirements for accelerators and opens the possibility for random access to application data.

2.3.2 Interaccelerator streaming and data flow

The strengths of the accelerator framework we develop are its ability to interconnect multiple accelerators seamlessly, to allow for modular hardware accelerators, and to enable resource virtualization. Entry and exit points are defined by the application-to-framework transactions described in the previous section, but the internal communication protocol must still be addressed. Our model for interaccelerator communication is explored in the context of streaming input and dataflow computation at a high level. The implementation details of interaccelerator communication are left to a later section.

Dataflow architectures [23] attempt to exploit data parallelism by removing the restraints of control-directed execution and the memory bottlenecks present in conventional von Neumann processor architectures. Dataflow architectures maintain only the data dependences in their code, enabling large amounts of concurrent computation where data parallelism allows. However, dataflow architectures have not overtaken control-flow driven processors as a computing paradigm due to their programming models and scheduling inefficiencies. Furthermore, the functional programming languages that are a natural fit for expressing the parallelism inherent in many codes, and used to develop applications for dataflow machines [24], are not widely employed, while current application development trends toward procedural and object-oriented programming models [32]. Learning from the dataflow community, many of the techniques cited in [24] could provide a means for software developers to express portions of their applications in a way that could more easily be mapped into hardware accelerators.

Dataflow machines suffer the inefficiencies brought on by fine-grained instruction scheduling. Dataflow architectures, however, are able to take advantage of the natural parallelism in certain applications deftly, but have yet to be proved feasible as a general-purpose computing model. Hardware accelerators represent a means to statically map data flow subgraphs, within the DFG of a portion of an application, into a custom dataflow processor (hardware accelerators in our model). In the context of a dataflow processor, the subgraph would otherwise be mapped into the processor's instruction stream. Hardware accelerators allow for the most appropriate regions of application code to be mapped into data-parallel processing engines. Our model provides the conceptual framework and the interfaces necessary to achieve dataflow-based accelerators.

An additional connection to dataflow machines is the ability to access variable latency data via the accelerator memory management unit we develop. The ability to access variable latency data is analogous to the input arcs being armed with tokens in dynamic dataflow machines—when the value is present, the computation can fire. Another connection to the token-driven execution of dataflow machines is at the core of the model we espouse for our framework. A dataflow-like communication protocol between accelerator blocks is provided to stall interaccelerator communication. The means to stall data flowing through the accelerators is necessary to enable random accesses that have nondeterministic timing. Providing methods to access data outside of pure streams, and to stall computation in the event data is not ready, allows for a more flexible programming model than a deterministic, purely stream-based computation model.

Even though no commercial dataflow machines exist, the dataflow model is apparent in the execution core of contemporary high-end microprocessors. Out-of-order (OOO), superscalar microprocessors [33] have execution cores that rely on finding and tracking data dependences dynamically. OOO execution provides speedup to a wide array of applications, but suffers from the cost of honoring proper control dependences and the cost of maintaining correct program execution during speculative execution. Moreover, statically identifiable dataflow relationships are lost in inherently sequential machine codes for von Neumann architectures. The lost parallelism is only to be rediscovered by the power-hungry, OOO execution engine. Our inter-accelerator model attempts to incorporate dataflow similar to what is done using OOO execution, but without the prohibitively large overhead needed to rediscover lost parallelism and without the restriction of limited functional units found in contemporary superscalar processors.

The split CPU/accelerator design allows our model to overcome the disappointing execution of control-intense applications on dataflow architectures, but still exploits its parallel processing abilities. We avoid the restriction imposed on general-purpose processors of having a limited number of execution units available. The general-purpose processor is adept at executing codes that make heavy use of branch prediction, control speculation, and small data sets (i.e., those that fit into the processor's physical register file and low-level caches). Hardware accelerators are capable of running many computations in parallel with a set of functional units tailored to applications' needs using distributed memory. The framework and interfaces developed in this work integrate these two design philosophies in a consistent, easily programmable way.

2.4 Summary

Integrating reconfigurable accelerators with applications requires creating defined protocols, standards, and communication models prior to their acceptance as a mainstream computing tool. The streaming model for computation is a foundation of this work, but it has drawbacks and is not a complete model for accelerator access. Even after choosing such a model, there are many remaining details that must be explored. This chapter lays the groundwork necessary for understanding interactions between applications, accelerators, memory and the operating system in the hybrid CPU/accelerator platform presented in this work. Subsequent chapters will build upon the concepts developed in this chapter to examine processing in a mixed CPU and accelerator environment, using a set of interfaces we have constructed, that are demonstrated using our CPU/FPGA prototype platform.

CHAPTER 3

RECONFIGURABLE DEVELOPMENT PLATFORM

The Xilinx University Program development board, based on the Virtex-II Pro [3] FPGA, provides a development platform for embedded systems and reconfigurable architectures. The Virtex-II Pro FPGA has two PowerPC (PPC) 405 processors embedded in the reconfigurable logic of the device. The embedded processors allow the XUP board to serve as a system on a chip (SOC) prototyping platform. A widely used operating system can be run on one of the embedded processors of the FPGA. Our development environment runs a PowerPC implementation of the Linux 2.4.26 kernel. Only the processor cores are hardwired; all other related and supporting logic—including system buses, memory controllers, interrupt controllers, peripherals, and reset services—must be synthesized in the reconfigurable fabric. The reconfigurable substrate further allows computer architects to explore the design space of future systems. One possible configuration is depicted in Figure 3.1. In this work, we make use of our CPU/FPGA platform to prototype our hybrid CPU/accelerator model. We demonstrate that our platform can be used as a prototyping environment for exploring future architectures.

3.1 XUP System Architecture

The system setup used throughout this work includes one of the embedded processors of the Virtex-II Pro FPGA clocked at 300 MHz. The second processor is always completely disconnected from the system.[1] The processor has separate 16 kB data and instruction caches, no second level cache, and 256 MB of double data rate (DDR) system memory. The main system bus is a synthesized processor local bus (PLB) running at 100 MHz. The PLB attaches the processor, the system memory, the on-chip peripheral bus (OPB) and the reconfigurable system developed in this work. Figure 3.1 shows the DDR memory and the PPC attached to the PLB. The OPB is shown bridged to the PLB providing access to slower, more latency-tolerant peripherals. In the test platform, many of the OPB peripherals including the VGA frame buffer, PS/2 ports, and Ethernet support have been removed to increase the available slices for accelerator use.

There are many ways in which accelerators can be interfaced with the processor in our test platform. In our model, accelerators can be attached to the 32-bit data side on-chip memory (OCM) bus as shown in Figure 3.1. The purpose of the OCM bus is to provide the processor with a fast, deterministic scratch pad memory. The address ranges mapped to the OCM bus are noncacheable, and the bus is accessed by standard load and store instructions with a lower bound on the latency of two processor cycles. The ME accelerator shown in the figure receives both control and data values via the OCM bus interface that is connected to the accelerator through a dual-ported block of memory or BRAM.

We utilize the PLB exclusively for data transfers and make use of device control registers (DCR) for the majority of control messages exchanged between the accelerator framework and the processor.[2] The

[1]The Xilinx tools require the unused processor to remain connected to the JTAG chain and clock signals, but it is not connected to the system.

[2]The DMA controller used in this work has PLB memory mapped registers for exchanging control messages. If complete separation of data and control were desired, it would be possible to develop a DMA controller that used DCRs, relegating all control traffic to the DCR bus and all data traffic to the PLB.

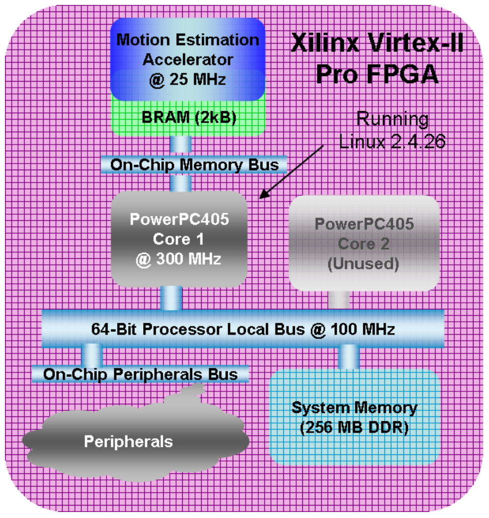
Motion Estimation Accelerator @ 25 MHz
Xilinx Virtex-II Pro FPGA
BRAM (2kB)
Running Linux 2.4.26
On-Chip Memory Bus
PowerPC405 Core 1 @ 300 MHz
PowerPC405 Core 2 (Unused)
64-Bit Processor Local Bus @ 100 MHz
On-Chip Peripherals Bus
System Memory (256 MB DDR)
Peripherals

Figure 3.1 The XUP system architecture.

PLB was chosen over the OCM because transferring areas of memory from the main system memory (on the PLB) to the OCM bus requires the processor to oversee the transfer. Furthermore, using the OCM bus disallows DMA transfers. Accesses to the PLB incur the cost of arbitration, possible cache pollution, and startup overhead associated with standard loads/stores and DMA transactions. Accessing the OCM from the PPC for small data regions has the advantage of removing arbitration and cache pollution while providing deterministic timing for memory access. However, for data transfers that use the DDR system memory as an endpoint, the PLB is already going to be accessed. If the other endpoint for the transfer from or to DDR memory is the OCM bus, synchronization overhead and serialization delay will result in greater latency and reduced throughput than if both endpoints were on the PLB. Although the OCM can and has been used for accelerator development, we choose to explore the PLB exclusively for data movement.

3.2 FPGAs as a Reconfigurable Prototyping Platform

In terms of computing technologies, FPGA-based computing is still a nascent field. FPGAs have long served as the “glue logic” in low production systems and as a prototyping platform for systems that would inevitably be implemented as application specific integrated circuits. FPGA devices have advanced from a slow, low-density test and interface logic platform to a technology that can integrate millions of synthesized gates, megabytes of on-chip memory, and gigabit I/O on and off chip, while providing embedded, high-speed functional units. We leverage the flexibility and support for high-speed functional units and processors that FPGAs currently provide to develop the prototype for our computational model.

Contemporary reconfigurable platforms, such as FPGAs, provide researchers with the ability to implement reconfigurable systems, but at clock speeds that are much slower than contemporary conventional CPUs. Speedup on reconfigurable platforms can still be achieved by taking advantage of the parallel nature of hardware designs. In FPGA, high clock rates are exchanged for the arbitrary reconfiguration capability of its logic blocks. Fixed logic blocks such as DSP slices, multipliers, and BRAM have been embedded into FPGAs catering to the applications that currently drive the FPGA industry. Domain-specific additions show that the reconfigurable speed barrier is not immutable by providing some amount of fast, fixed logic wedded to flexible reconfigurable logic on the a single chip. However, even with high-speed functional units integrated into the reconfigurable fabric, pure reconfigurable logic still suffers from an order of magnitude slowdown in terms of clock frequency when compared to hardwired digital logic, such as ASIC. Therefore, any hope of showing tangible speedup with reconfigurable logic, running at reduced clock speeds, requires that the hardware accelerated version exploit an order of magnitude more parallelism than a software-only version.

A fully reconfigurable substrate only serves as a means to prototype novel architectures. In future implementations, any component we develop that is common to all accelerators can be implemented in fixed logic that is faster and more dense, in terms of transistors, than the reconfigurable version in the FPGA development platform. The reconfigurable fabric of the FPGA provides an (almost) completely unrestricted environment for developing accelerators (e.g., DCT and loop bodies) and the infrastructure (e.g., DMA and interconnects) necessary to integrate the accelerators with the CPU, memory, and peripherals. However, with the increase in flexibility, there is a cost in terms of packing density, speed, and power dissipation that must be paid for using reconfigurable logic. When evaluating a reconfigurable system we must be able to extricate the area and slowdown attributable to reconfigurable logic, from the components that would be hardwired logic in a real implementation.

The interconnections necessary for attaching accelerators to the CPU may become as commonplace in reconfigurable designs as memories and multipliers once were, leading to their being embedded into the reconfigurable logic. Our framework and computing models provide some of the prerequisite knowledge that will guide the development of the interconnection components implemented as fixed logic, as BRAMs and multipliers are today, providing many of the speed and power benefits of ASIC to reconfigurable systems.

3.3 Future Architecture Exploration

We examine a hybrid CPU plus accelerator model, exploiting data parallelism in hardware accelerators, to speed up standard applications. The generality afforded by using FPGAs can be exploited beyond the "FPGA as a coprocessor" model. It is possible to survey many choices and decide upon a highly tuned architecture prototyped on the FPGA. Cycle accurate simulators allow for arbitrary system organization, but require assumptions to be made regarding timing and size requirements. Furthermore, full system modeling using simulation can be prohibitively time consuming.

FPGAs allow architects to build novel systems into the reconfigurable fabric of our CPU/accelerator model to enable future systems to be more effectively prototyped. No longer are designers bound by the constraints of software simulation, nor by the high cost and long latency associated with ASIC prototypes. Our model allows for designers to explore system configurations that utilize most of the features found in contemporary microprocessors, such as those found on the PPC405, while running the Linux operating system.

Our model allows designers to explore different organizations for buses, memories, interconnects, peripherals, accelerators, and coprocessors with a level of flexibility and speed not found in simulators or in the reconfigurable fabric of the FPGA alone. To enable fully unrestricted architectural exploration, soft-core processors can be implemented that allow arbitrary changes to the processors' microarchitecture.

3.4 Summary

The continually growing number of transistors provided by Moore's Law has allowed FPGAs that were once relegated to menial "glue logic" tasks to take center stage as computing platforms. Moreover, the reconfigurability of FPGAs allows for exploration of the architectural design space at speeds and costs not previously seen. Our platform serves as an intermediate between conventional CPU architectures and fully reconfigurable systems. With the bridge in place, future work can go beyond simulation and prototype reconfigurable systems and novel computer architectures with the support of a widely used operating system.

CHAPTER 4

HARDWARE ACCELERATOR FRAMEWORK

In this chapter, a hardware framework for implementing and interconnecting hardware accelerators is described. The complexity of designing and debugging hardware accelerators has been reduced by eliminating over 80% of the signals that must be considered by developers. Further benefit is achieved by providing a target for software interfaces, allowing the accelerator developer to concentrate on hardware design.

The motivations and goals for our accelerator framework are developed. A stand-alone accelerator is examined as a brief case study to demonstrate the complexity that must be navigated by a system designer when using an unrestricted system interface. The features and limitations of our framework are briefly discussed. A means to provide protected random-access to memory, the accelerator memory management unit, is introduced. As a means to demonstrate the reduced design and debugging complexity our platform offers, the accelerator developed in the brief case study is retargeted at the accelerator framework.

4.1 Goals for the Accelerator Framework

Mapping accelerators into the general platform constituted by our CPU/accelerator system requires many design choices. Such choices include what bus configuration to use, how to transfer data, what sorts of memory to include, and how to interface hardware accelerators with software applications. Figure 4.1 depicts the choices an accelerator developer must make with darker regions indicating less flexibility, wider regions representing longer access latency, and taller blocks indicating relative size for memories. The subset of design choices we use is highlighted in blue in the figure. The application-level interfaces are intended to abstract away the implementation details of the accelerator and the reconfigurable system. However, this still leaves the problem of mapping accelerators into a general platform with design tradeoffs for the operating system, the application software, and the system architecture. A software-level interface, to complement the hardware-level interfaces developed in this chapter, is left to the next chapter. The framework is one part of attaining the goal of providing a consistent interface that decouples hardware accelerator design from software application development.

Having a general platform offers the greatest degree of flexibility for accelerator designers to choose how to design and interface their accelerators with the software applications. However, the flexibility to pick and choose bus protocols, bus widths, control interfaces, memory hierarchies, addressing schemes, synchronizations mechanisms, and accelerator-to-accelerator protocols is at odds with providing consistent software interfaces. Most accelerators do not need a level of complete flexibility, nor do their designers desire debugging such an unrestricted implementation. Furthermore, the large number of signals that must be managed when supporting a general bus architecture, DMA controller, I/O ports, and their associated interfaces can be cumbersome. Full system simulation is slow, potentially difficult to set up properly, and can be impossible to implement if simulation models are not available. Using these facts as a guide, we present a framework for developing reconfigurable accelerators and incorporating the

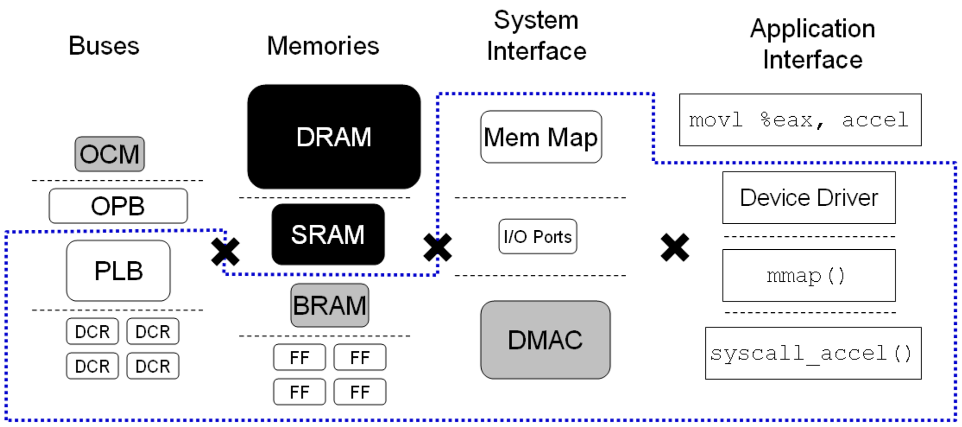

Figure 4.1 Possible design and interface choices for our target platform.

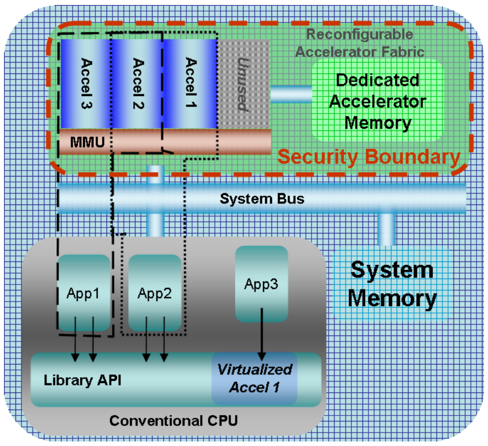

Figure 4.2 System model for the XUP board with protection barriers shown.

accelerators with a conventional processor system. The high-level organization of our CPU/accelerator system is depicted in Figure 4.2.

The goals of the switchable interconnect accelerator framework are:

- To remove the burden of understanding platform-specific bus interfaces from the application developer by reducing signal count. A platform-independent interface allows developers to design accelerators once, and then take advantage of them on any platform where our framework is available.
- To reduce the overhead of communication and interface logic by taking advantage of embedded logic, and sharing it among multiple accelerators. Examples of embedded logic in FPGAs today include: memories such as BRAM, arithmetic units such as multipliers, and communication mechanisms such as Rocket I/O [34].
- To ease the debugging process by reducing the number of signals in the top-level HDL design of the accelerators.

- To enable efficient runtime reconfiguration by electronically isolating separate accelerators. Even though tools for runtime reconfiguration of accelerators [35] are becoming available, many of these tools and the interfaces are still maturing, but when they are in widespread use, such isolation will still be required and is supported by our framework.
- To provide a vehicle for future work in operating system interfaces and compiler generated accelerators.
- To motivate alternative data access models and programming paradigms for reconfigurable hardware accelerators (i.e., break the stream and dataflow models to provide a more general computing platform).
- To allow both coarse- and fine-grained accelerators to coexist, utilizing the same interfaces so that many levels of parallelism can be efficiently mapped into our computing model. We use DCT as an example of a more coarse-grained accelerator and quantize as an example of a more fine-grained accelerator in this work.

The goals outlined above are reflected in our accelerator framework. The accelerator framework is explored in detail in Section 4.4. However, first we will discuss accelerators that will target our framework to further develop the motivation for the accelerator framework.

4.2 Reconfigurable Accelerators

We developed simple accelerators to help guide our thinking. Here we present a 2D DCT and quantize accelerator for our motivating application, JPEG encoding. The JPEG encoder software has a variety of components with high levels of parallel granularity that can be implemented as hardware accelerators in FPGA logic. The developed accelerators were tested as stand-alone accelerators and within the context of the accelerator framework we present here.

The software version of JPEG encoding used in this work was developed by the Independent JPEG Group. The freely available, open source library provides a JPEG encoding implementation that is optimized for single processor performance. The JPEG encoder utility included with the package is `cjpeg` . The `cjpeg` utility can be configured to only use integer representation for all values computed, obviating need for floating-point hardware (or emulation of floating-point on the PPC405, which lacks an FPU). All modifications made to the compression utility included with the library were made to all versions of the code for x86 (used for comparison), for PPC software (use as a baseline), and for hardware-accelerated implementations (used to test our platform).

4.3 Stand-alone DCT Accelerator

The stand-alone DCT implementation makes use of a customizable intellectual property (IP) core from Xilinx [36]. The core was wrapped with the necessary support structures to integrate it directly with the PLB, to provide buffering, and to control the flow of data into the Xilinx-provided core. The wrapped version will be referred to as the stand-alone DCT accelerator. The number of signals required for debugging the wrapper that provides access to the bus, control logic, and necessary buffers is in the hundreds. The developer must ensure that the proper inputs are provided to the bus and the signals coming from the bus are handled by the control logic in the accelerator properly. Furthermore, debugging a stand-alone interface attached to the bus and to an IP core, while being controlled by a software application running on the CPU, possibly under the control of an operating system, requires complex simulation models for the bus, processor, and other system components. As we will show, relying on our accelerator framework replaces the otherwise onerous task of full hardware platform debugging with a reduced-complexity unit test, that has a common interface for all accelerators.

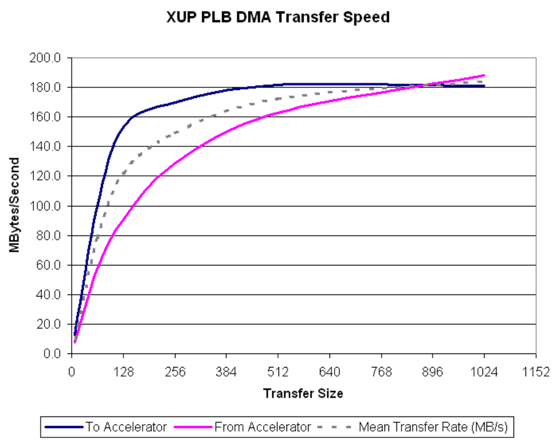

Figure 4.3 DMA transfer rates from system memory across the 100-MHz PLB to the accelerator buffer with transfer sizes in bytes.

4.3.1 Accelerator design

The DCT core allows for a fully pipelined design, that takes a 12-bit sample, sign-extended by the software encoder to a 16-bit value, every 10-ns clock period. The latency of the device depends on internal datapath widths, coefficient value sizes, and input/output precision. All combinations used in this study result in a latency of 100 ± 10 cycles. It is possible to clock the DCT at over 140 MHz on a Virtex FPGA, a lower-speed FPGA relative to our platform. However, a 100-MHz clock is chosen so that there is no need to cross clock domains between accelerators and the bus. Furthermore, decreasing the clock frequency reduces pressure on place and route, yielding faster design build times.

Data transfers from main memory to the accelerator are accomplished via DMA. Figure 4.3 shows the throughput of the DMA engine used in our platform. Each transfer must be set up as a sequence of four stores to the PLB memory-mapped control registers for the DMA controller. The DMA engine local to the accelerator must then arbitrate for the bus. The DMA controller will proceed to move an entire 64-pixel macroblock in a single burst transfer. The DMA controller has an internal buffer for rate decoupling between the bus and the framework. The dual-ported internal memory we integrate into the accelerator has a 64-bit port attached to the bus, allowing for a read or write every bus cycle, providing for maximum bus utilization.

The stand-alone version of the DCT accelerator implements a single buffer instantiated as a BRAM. The accelerator is attached to the PLB and data transfers are accomplished using an integrated DMA engine. Due to the design of the DMA engine and parameterized interface to the PLB provided by Xilinx, an entire macroblock (16-bits per sample * 64 samples per block = 128 bytes) is the maximum that can be transfered in a single burst transfer. The size of the data array could be increased to allow for multiple macroblocks, but due to the limited buffering of the DMA controller, the macroblock array would need to be partitioned into 128-byte transfers with the DMA engine re-arbitrating for the bus between burst transfers.

Cache coherence, between the data to be transfered via DMA from system memory and the data that is present in the data cache of the PPC, is an issue that must be addressed. Both the DMA controller and the processor are accessing memory that may be cached. The entirety of the JPEG encoder is not implemented in hardware and as such, the host application must first process parts of the data stream before handing the data off to the DCT accelerator for processing. It is possible—and depending on implementation, likely—that the next macroblock to be processed by the accelerator is resident in the processor's cache and differs from the copy of that data in main memory. To maintain coherence between

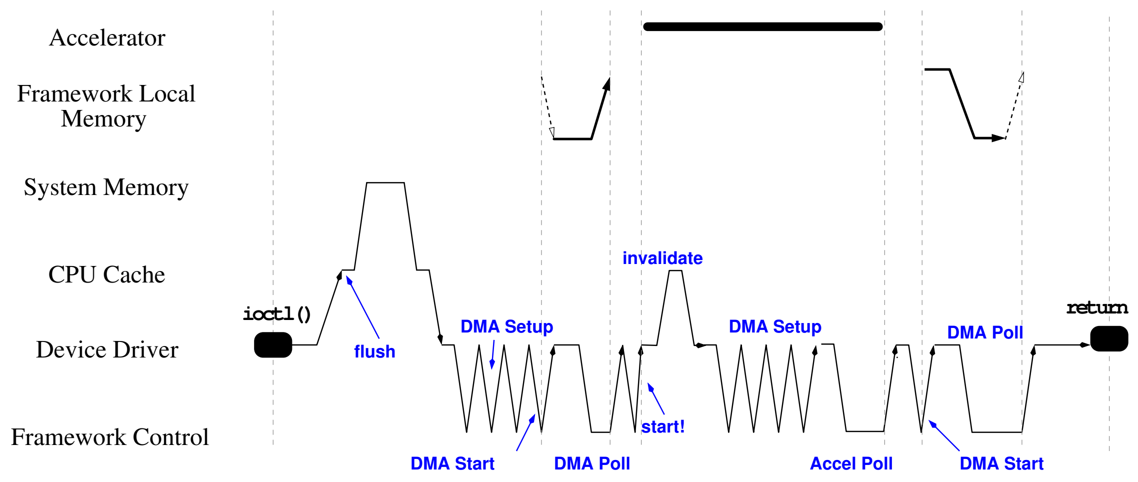

Figure 4.4 Flow chart of accelerator execution using the `ioctl()` system call.

the version of the macroblock to be processed resident in memory and in the cache of the processor, an explicit flush must be performed prior to any DMA transfers taking place.

In the stand-alone accelerator, a set of control registers is memory-mapped on the PLB for accessing the accelerator. There is a register that is set by the host application to being executing the accelerator once all the data is available in the accelerator local memory. Another register is used for polling by the host application and is set by the accelerator when all the output is available in the local output buffer of the accelerator.

4.3.2 Software integration

Integrating the accelerator with the driving application (`cjpeg`) was accomplished by developing a character device driver for the stand-alone accelerator. System calls used for accessing character devices under Linux are already implemented and wrapped in user-accessible library calls. An `ioctl()` system call provides the necessary interface for the accelerator, but with unnecessary checks and data structure maintenance that have the effect of reducing performance. A new system call is developed to provide the same service with reduced overhead in Section 5. However, having such an interface that can be taken "off the shelf," expedites development efforts, requires fewer changes to the operating system, and enhances consistency across accelerators.

Figure 4.4 shows the interactions between the driver and the accelerator framework. To access the accelerator, the user application passes the `ioctl()` call associated with the device driver for the accelerator a pointer in its virtual address space. The pointer is to a 128-byte region containing a single 8x8 pixel macroblock. Each macroblock is stored as 16-bit, signed integers with big endian byte ordering. The device driver performs protection checks and then copies the data from user space to a statically allocated kernel buffer used for the DMA transfer. If invalid input is provided by the application, an error code is returned by `ioctl()`. The DMA buffer is marked cacheable by default in our implementation to reduce the cost of single word writes to an uncachable buffer in system memory. The region of the data cache the kernel buffer maps into is invalidated and flushed to system memory. The DMA transfer is initiated using a series of word writes to memory-mapped control registers. Once the DMA transaction begins, the driver polls the DMA controller status register until completion.

Once all of the data has been moved from system memory to the accelerator local memory, the PLB-mapped control register for the DCT accelerator is written to begin DCT accelerator execution. The driver then polls on the accelerator completion status register until it is cleared, signaling completed execution. Concurrent with accelerator execution, the driver invalidates the cache region of the DMA input buffer to ensure that reads from the accelerator will reflect the output of the accelerator and not be referencing stale, cached data. The accelerator framework could have been implemented to handle DMA transfers without intervention on the part of the processor. However, due to the platform we were using, the desire to keep as much flexibility as possible in how the software accesses the framework, and the transfer sizes of our test application, we did not add the functionality into the framework that would have allowed the processor to do additional independent work while DMA transfers occur. When computation is complete, another DMA transfer is initiated, moving data from the accelerator local memory back to system memory. The driver copies the data from the kernel buffer back to the user space buffer. The driver returns control to the calling application. A more in-depth look at device driver access to the accelerators using the framework is presented in the next chapter.

The stand-alone design, using the device driver interface, is far from optimal and could be improved by finding ways to move larger amounts with each transaction to mitigate the effects of protection boundary crossings and to allow for more computation and communication concurrency. With respect to data moving from the application to the accelerator, communication entails the cost of memory transfers from the system memory to the local store of the accelerator, and the computation cost is the time spent executing the accelerator. With only one macroblock being sent via the `ioctl()` call per transfer, the cost of the system call and kernel/user space memory protection boundary crossings must be amortized over a small amount of execution time. A far more efficient method would allow for a large number of macroblocks to be transfered to the driver in a single transaction. Doing so allows the system call overhead to be amortized by the longer runtime of the accelerator. Furthermore, larger block transfers to the accelerator allow for overlapping communication and computation, mitigating the performance impact of the DMA setup time, PLB arbitration, and coherence actions. Even though streams of blocks must be broken into 128-byte chunks, the control required for doing so is handled by the DMA controller[1] allowing the CPU to make forward progress during data transfers. Another approach is to have more macroblocks available concurrently, allowing for multiple DCT and quantization pipelines. Doing so increases performance while also reducing the system interface/memory transfer to reconfigurable accelerators ratio.

An issue with the DCT core used was encountered in the design of the stand-alone DCT accelerator. We were unable to get a bit-accurate DCT at any level of precision (matching neither Matlab nor libjpeg output). The errors present in the final design are apparent when viewing the compressed images, but with little quality degradation. The loss of quality could be avoided by making use of an accurate 2D DCT core, but developing such a core is beyond the scope of this work. The errors introduced by the DCT core have no impact on timing or interface method, which are the measures of interest to this study.

4.4 Switchable Interconnect Implementation

The switchable interconnect accelerator framework is implemented in VHDL and synthesized using the Xilinx toolflow [34]. The targeted hardware platform is the Xilinx University Program (XUP) board. The prototype framework system has two reconfigurable regions, or *reconfigurable frames*, that can be interconnected and/or interfaced with input and output buffers. A reconfigurable frame is similar to a page frame in that it is only the location where the accelerator will be mapped, but not the actual accelerator. We refer to actual accelerators as accelerator blocks. Figure 4.6 shows the interconnections in the framework with three reconfigurable frames and input/output buffers. The interconnection network is connected to the PLB for data transfers and DCRs for all control functionality except for DMA setup. The DMA control registers are memory-mapped registers on the PLB. The input/output buffers and DMA control registers are accessible to the processor via physical addresses. The physical address range

[1]As stated earlier, DMA bursts can only be 128 bytes, however the size of the entire transfer is limited only by the size of address buses and counters internal to the controller. Our current implementation can transfer up to 2 048 bytes (16, 128-byte bursts) in a single DMA transaction.

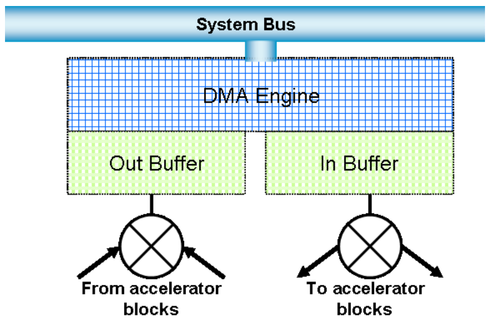

Figure 4.5 System interface to accelerators via DMA data transfers.

can be mapped into the virtual address space of the kernel or user space applications for debugging purposes.

The accelerator framework has a memory management unit (MMU) with an integrated translation lookaside buffer (TLB). The accelerator MMU mechanism provides accelerator-initiated, protected access to data across the PLB bus. A DMA controller is provided to transfer data into input and out of output buffers integrated into the framework from the system memory as shown in Figure 4.5. The basic mode for accessing the framework is via a character device driver using `ioctl()` system calls. More methods are discussed in the next chapter. All accelerators are synchronized to the same clock. The current implementation runs at 100-MHz, but could be scaled depending on the maximum speed of the slowest accelerator instantiated in framework.

4.4.1 Interconnection network

The accelerator framework and its embedded interconnection network provide for synchronous dataflow between resident accelerators, electronic isolation of accelerators for runtime reconfiguration, and export a simple accelerator-to-system interface. The accelerator framework is controlled by a set of registers connected to the PPC DCR bus. The interconnect network can effectively be switched in a single accelerator clock cycle, but is asynchronous with respect to the processor. Protected access via the MMU is provided to each accelerator block.

Once a transaction with the framework is initiated and data is available in the input buffer, the data is allowed to stream through the sequence of accelerator blocks in the framework. Each accelerator frame provides an input and an output interface for interacting with the other accelerator blocks of the switchable interconnect network. A feature of the accelerator frame interface is a means to assert `backpressure` between two accelerator blocks. Backpressure, in our model, is the ability for an accelerator block in the role of sink to signal its source that it is unable to take any more data as input. The framework requires that any source node hold its current output state starting the cycle its sink asserts backpressure. If no internal buffering is implemented in the source, the source must propagate the backpressure signal to its source.

Our backpressure mechanism differs from other stream-based schemes, such as [10], where queues are used for rate matching. Instead, our model delegates the responsibility for internal buffer management to each accelerator. Therefore, no state is maintained in the channels. Placing no restrictions on interaccelerator buffering mechanisms allows developers to choose the level of buffering complexity commensurate with need. For accelerators that are rate-matched and do not use the variable-latency side-path for data, not supporting build-in buffer management is of little consequence and simplifies the framework architecture. Backpressure allows for flexible buffering for those accelerators that require it, while simplifying the design of accelerators that are fully stream-oriented and rate-matched.

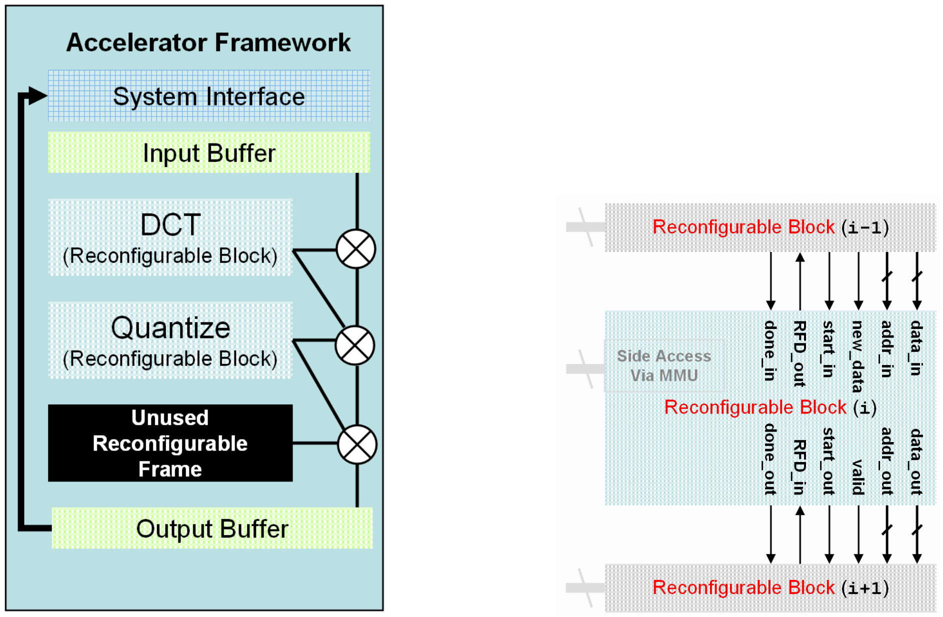

Figure 4.6 Accelerator framework block diagram.

The interfaces developed in this work serve as a starting point. As software to hardware accelerator mapping technologies evolve and the target applications shift toward more variable grained accelerators, the accelerator-to-framework connections can be changed to provide a more suitable interface. In our prototype, the bus widths are chosen for their applicability to JPEG encoding and to fall on byte boundaries, but could be scaled in future implementations. However, further upward scaling in an FPGA implementation places more strain on routing. The strain leads to higher synthesis times and a reduction in the number of accelerator frames that can be generated in an FPGA implementation. The intended target for implementing the framework includes interconnects that would be hardwired logic. These could then be switched (i.e., redirect data flow from one frame to another), reducing complexity and overhead, and obviating the need to route the signals as they would already be present in the fabric at fixed locations. Accelerators would plug into the embedded interconnect network, achieving dense, high-speed channels while using the interfaces we develop here to interact with software applications.

Figure 4.6 shows the input and output signals that are available for reconfigurable blocks to use. The top of the reconfigurable block shown goes to the preceding block while the bottom is attached to the succeeding block. On the input side of a reconfigurable frame are the `data_in` and `data_in_addr` buses for data flowing into the accelerator frame. The initial implementation uses a 16-bit data bus and a nine bit address bus. The handshaking protocol used to communicate between accelerator frames uses the signals `new_data`, `start_in`, `RFD_out` (Ready For Data), and `done_in`. The output side of a reconfigurable frame includes `data_out` and `data_out_addr` buses with widths that match the data_in widths. The signals associated with the data_out side of the frame include the signals `valid`, `start_out`, `RFD_in`, and `done_out`. Each reconfigurable frame also has a 16-bit control signal mapped to a DCR as an input that is left unused for the example accelerators in this work. A total of 58 separate signals are required without considering the signals necessary for the control which may be unused by the accelerator or side-path access using the accelerator MMU described in a later section.

The interfaces provided by the reconfigurable frames in the framework allow for both "push" and "pull" semantics. It is envisioned that data will be pushed through the framework in a streaming fashion. However, data enters the pipeline by being pulled in by the head accelerator. The flow of data is started when the reconfigurable accelerator block that is connected to the input buffer, the head accelerator, pulls data from the buffer using its `data_in` and `data_addr_in` ports. Our implementation includes a small state machine that feeds data automatically to the first accelerator in row-major or column-major ordering based on an 8x8 matrix of values. There is no need for pull semantics if the configurable state machine provided offers the stride needed by the head accelerator. Future designs could incorporate more complex logic to feed data to accelerators, placing the burden of pulling data from the buffers onto the state machine and not the accelerator designer.

When a new burst of data (e.g., a new macroblock in JPEG or a new loop iteration in more fine-grained approaches) is available inside a reconfigurable block, the control for that block asserts the `start_out` signal along with the `valid` signal for a single cycle. New data blocks enter the pipeline of accelerator blocks at the head block, which is attached to the framework local memory, and either receives the data as a push from the state machine or pulls the data. The actions of the state machine are based on control register values. The `start_out` signal communicates to the subsequent accelerator that a new block of data is starting and `valid` signifies a new data value is ready to be consumed. The `start_out` signal could also be used as a reset, cutting off the execution of a prior task on the subsequent frame.

Every cycle data is available to be passed to the next accelerator frame, `data_out`, `data_out_addr`, and `valid` are asserted. Upon completion of the current block, concurrent with the last input value, `done_out` is asserted. If `valid` goes low between the assertion of a block starting and it completing, the subsequent reconfigurable block stalls until the next `valid` signal qualifies the `data_out` and `data_out_addr` buses.

Next we give an example from the point of view of an accelerator in the role of source. Figure 4.7 shows the timing diagram for a sample data block, seven words in size, being transferred between two accelerators. All transitions are on the rising edge of the clock. At time zero it is assumed that there are no pending transactions and therefore the address and data buses are not valid and the `valid` signal is not asserted. In cycle 1 the `start_out` signal communicates to the sink that a new block is starting. The `valid` signal qualifies the data on `data_out` and `addr_out` in each cycle it is brought high. Three data values are sent at which point the source deasserts the `valid` signal to stall the sink. In cycle 5 the source reasserts `valid` to stop stalling and one more data item is read by the sink. In cycle 6 the sink lowers its `RFD` signal. The source holds the values of `data_out` and `addr_out` until the sink can accept data again as signaled by the sink raising its `RFD` signal. In cycle 8 the sink raises `RFD` and reads the value produced by the source in cycle 6. Two more data items are read out by the sink before the block is finished at cycle 10. The `done_out` signal is asserted by the source at cycle 10 to communicate to the sink that the last value in the current data block is available. There is no data block immediately following the current one and therefore, in cycle 11, `valid` is brought low by the source.

There are tradeoffs that must be made when reducing the number of signals that are used by the accelerator framework. The developer loses the ability to freely access memory-mapped regions and system memory. The internal accelerator data and address buses have also been reduced in size to match our target applications. The PLB allows for byte-addressable random accesses, but our prototype framework only allows 16-bit accesses when streaming. However, the accelerator developer does not have to be concerned with arbitration and can access the local memories provided by the framework deterministically. The accelerators built using our framework cannot arbitrarily set up control registers or physically addressable buffers. The payoff for the reduced flexibility is simplified generation of accelerators that fit into our computation model and the ability to take advantage of the software interfaces we develop for our framework.

The accelerator framework is switchable in the respect that the channels between each of the blocks can be redirected from one accelerator to another. There are an equal number of lines on the output bus as there are on the input bus. The signals have been selected such that a direct connection from input of one accelerator to the interface of the next is possible. The processor redirects the channels in the framework by setting a DCR. A DCR command can be executed in approximately ten processor cycles, or three accelerator cycles, adding a small latency to switching of the interconnects. It is assumed switching will take place at transaction boundaries as there is no way to synchronize the accelerators with the switching, other than at points in the framework's operation where all accelerators are idle.

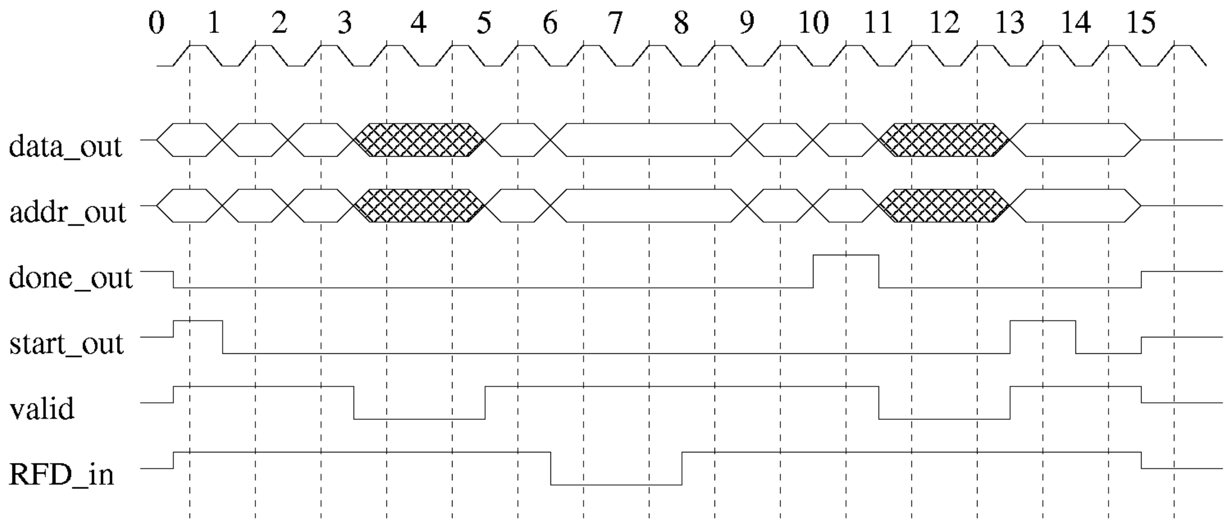

Figure 4.7 A timing diagram from the source point of view writing six words. Stalling and backpressure situations are shown.

Once the DCR write instruction commits on the processor, the accelerator framework is set to the new configuration.

A further explanation of backpressure is required. Backpressure it is a necessary component of the protocol developed for the framework due to the variable latencies associated with the side-path accesses. Furthermore, by providing backpressure, a slower accelerator can signal to a faster accelerator without any mind paid by the developer. Rate decoupling allows for heterogeneous, asynchronous execution units that ease development by not forcing all accelerators to operate in lock-step or the developer to reason about timing, while supporting more accelerator execution models. In the current implementation, however, all accelerators and interfaces are synchronized to a common clock. The two `RFD` signals are used for providing backpressure. When asserted by the sink, `RFD` signals to its source that the flow of data must be stalled beginning that cycle. The data and address values at the start of that cycle that are valid at the output of the source are not consumed by the sink. The source must hold the values present on `data_out`, `addr_out`, and `valid` until the `RFD` signal from the sink is brought high. Since no buffering is available in the channel, if no buffering is implemented internally by the sink, the whole pipeline must stall, resulting in potential slowdowns. However, without the need to understand the complexity of asynchronous events, correct execution is maintained due to the handshaking protocol employed.

The backpressure asserting, handshaking protocol demonstrated here allows for rate decoupling between connected accelerators, variable latency loads/stores via the protected side access path, random access to input and output memories, and most importantly, a simplified model for mapping reconfigurable accelerators into a conventional computing environment. Our model is not necessarily the best model for interactions between reconfigurable blocks, but serves as a starting point and a basic model for mapping hardware accelerators into applications.

4.4.2 Accelerator MMU

The Accelerator Memory Management Unit (aMMU), as shown in Figure 4.2, provides accelerators with a method to access data outside the stream provided by the input buffers and other accelerator blocks. The accelerator MMU is depicted in Figure 4.8. Accesses to system memory are addressed using

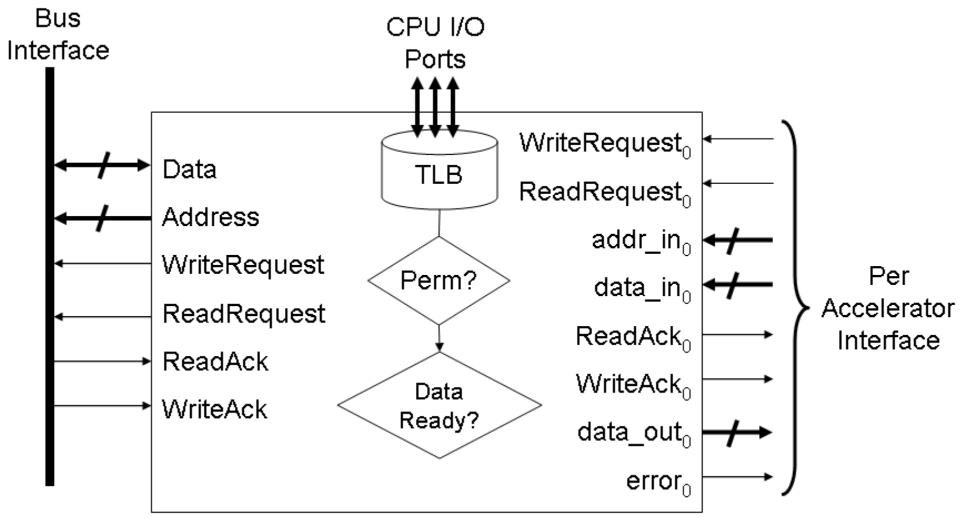

Figure 4.8 Block diagram of the accelerator MMU.

the virtual address space of the application associated with the accelerator. The aMMU has a translation lookaside buffer (TLB) to provide the translation and protection. The translation mechanism maps fixed-sized regions of virtual memory and is filled by the host system. Each entry maps a page-aligned, 4-kB region of virtual memory into the systems physical address space. The current implementation uses a four entry, direct mapped TLB that associates a 3-bit accelerator identification number (AID) with each entry, allowing each accelerator to have independent entries in the TLB. The software managed TLB provides independence among accelerators and thus different applications can share a single accelerator framework.

One of the goals of the framework is to provide a platform for developing reconfigurable accelerators that can be implemented with varying degrees of closeness in their coupling (i.e., the framework will still work when implemented as a coprocessor sharing the cache with another core, or when it is a stand-alone unit on a peripheral bus as demonstrated in this work). To that end, the framework uses an asynchronous protocol to handle the variable latencies possible when accessing memory via the aMMU.

The accelerator frames access memory via the aMMU using a simple request-acknowledge protocol. A block diagram for the aMMU is shown in Figure 4.8. When an accelerator requires access to data via the side-path, it sets either its `ReadRequest` signal or `WriteRequest` signal. Concurrently, the accelerator places data on the data out bus (for a write) and the valid virtual address of the data to access. The accelerator must then wait for an acknowledgment signal from the aMMU. If the accelerator exceeds its ability to sink data while waiting on the request, it asserts backpressure to stop the flow of data in from preceding accelerators.

When a request is received by the aMMU from an accelerator, the TLB is accessed using the two least significant bits of the stored page address (bits 18 and 19 for the 32-bit addresses used in the framework). The entry is checked for validity, permission, and a matching accelerator ID number. If the checks all succeed and access is granted, the associated physical page address is concatenated with the least significant 12 bits of the virtual address from the accelerator request. A request is then made to system memory in our prototype or, in future implementations, the cache or local accelerator store. When the data is available, in the case of a read, or has been written, as is the case of a write, the acknowledgment signal is asserted for the particular accelerator block.

Currently, TLB fills are handled by the device driver associated with the framework using device control registers (DCR) to communicate data between the host system and the framework. The host system is able to create a protection boundary between the accelerators mapped into the framework and the system memory. The framework creates a protection boundary by only allowing specific regions of physical memory set by the operating system to be mapped by the TLB of the aMMU. Any access not mapped by the TLB will result in an error signal delivered to the violating accelerator and, in future implementations, can interrupt the processor to fault in missing mappings. The TLB maintains a single protection bit signaling read-write access or read-only access. Each entry also has a valid bit that can be cleared by the host system to disallow future access to a previously installed mapping. The TLB fill is handled by the operating system setting two DCRs, the second write signaling the fill is complete to the accelerator.

The aMMU has the ability to access the PLB directly as a bus master. As such, the framework can access any of the system resources on behalf of a requesting accelerator. For example, an audio dithering accelerator could place its result data directly into the sound card buffer. Accessing device buffers directly removes the need to copy data twice more across the bus between the accelerator, CPU, and audio device. Future implementations could take advantage of different access methods, but the translation and accelerator interface would remain unchanged, furthering the goal of coupling agnosticism.

Many research efforts in developing reconfigurable systems have concentrated on stream-based computation models; the limitations of such a model were discussed in Chapter 2. In the design of a shared resource, such as the aMMU and its associated TLB, it is necessary to understand how arbitration, prioritization, and resource sharing will affect the aMMU design and the implications for framework mapped applications. We implement a basic configuration for side-path access necessary for our stream/random access model, while leaving room for future implementations to tailor the interface to their needs.

4.4.3 Framework DCT accelerator

A stated goal of the framework is to decrease the effort required to develop and debug accelerators in the context of our hybrid CPU/accelerator model. The entire system is comprised of the processor running applications and a full operating system, the bus models, hardware accelerators, and external interfaces such as input peripheral devices. It is possible to model the entire system using a register transfer level (RTL) simulator such as ModelSim [37]. Using a simulator to debug the entire system is too slow to use effectively and/or too difficult to set up in practice. By constricting the interface to fewer, predefined signals, the switchable interconnect accelerator framework enables developers to design accelerators without considering the entire system, expediting the design and debug processes.

By avoiding modeling the entire system and avoiding modeling the accelerators' interactions with the host, the framework makes debugging more manageable. The framework brings the number of signals necessary for debug below 100, provides a way of interconnecting multiple accelerators without accessing the system bus directly, and provides a target for our application-level and operating system-level interfaces. In the stand-alone design, there are over 400 signals that connect the DCT and quantizer accelerators with the buses. The high number of signals, coupled with the difficulty in simulating the PLB and operating system interaction accurately, makes independent accelerator development overly burdensome. The interfaces to DCT and quantization are redesigned to work in the accelerator framework as a demonstration of the framework's capabilities, but also to demonstrate the framework as a reduced complexity interface that provides all the resources necessary for accelerator development.

Another difference of note is the separation of control and data transfers, with polling taking place using a separate resource from the data bus, in the framework version of the accelerator. In the stand-alone DCT accelerator, memory-mapped control registers accessed across the PLB are used to communicate out of band information to the accelerator. However, memory-mapped control registers reduce concurrency as a DMA transfer or cache line fill cannot occur while the accelerators are being controlled or polled. The framework enabled version implementation uses the device control register (DCR) bus that has 10 processor cycle (3.33 accelerator clock cycle) access. Accessing the DCR is independent of the PLB, which reduces contention and increases performance. Due to the limitations of the provided DMA controller, DMA control registers remain mapped to the PLB, but with some effort could be implemented as DCRs as well, fully separating data and control channels.

Debugging the stand-alone DCT design is a long and laborious process, even with high-quality tools to assist the developer. The debugging process consists of both simulation using ModelSim and emulation on the board (assisted by software debug utilities, such as Xilinx's ChipScope [38]). While still using the same tools, porting the DCT accelerator to the framework was accomplished in an order of magnitude less time than was required to developed the initial system interface. Furthermore, the constrained interface allowed for complete, fast simulation and verification in software prior to the design ever being run on the FPGA. The changes to the design were minimal when porting, as the framework was developed using JPEG as a motivating example. However, some redesign is necessary to provide mechanisms for communicating solely through the framework's handshaking mechanism and removing assumptions about how one accelerator feeds data into the next. Porting of the quantizer to the accelerator framework followed a similar design path.

4.5 Summary

We have developed a framework for building accelerators that provides a target for our software interfaces and simplifies the design and debugging processes. We reduce the complexity of debugging by bringing the core number of signals that must be modeled by the developer down from over 400 to 58. Our protocol for interaccelerator communication allows for accelerators to be developed as logically separate components and then integrated using our framework. We have built an accelerator memory management unit that allows protected access to system memory using processes' virtual address space. We have constructed the necessary buffering, data flow protocols, control logic and registers, and bus interfaces to expedite accelerator development and to provide a possible target for future automated tools.

CHAPTER 5

OPERATING SYSTEM INTERFACE

In this chapter the model for accessing accelerators using the device driver interface developed is explored. We demonstrate the device driver as a basic model, then present a set of four interface methods that offer a variety of tradeoffs for accelerator developers. The interfaces demonstrated in this chapter only vary in how the operating system handles the accelerator requests from the user application, while most of the application interfaces, and all of the hardware accelerators/accelerator framework, remain unchanged. This chapter demonstrates a set of OS arbitrated access methods and system architectures that can be matched with the data transfer requirements of varied workloads based on results that will be discussed in Chapter 6.

5.1 Device Driver Access

The hardware accelerator-to-framework interface provides an abstraction that frees the accelerator developer from the onerous task of system integration via buses and control registers. Likewise, the interface provided to the application can be made consistent by wrapping the system specific details in a set of system calls. A simple interface method could have the application write data out to the accelerator and then read them back in at a later time as shown in Figure 5.1 Abstracting away the underlying details of implementation allows for a layer in which applications developers are provided with a simple interface and the underlying system can make best use of its resources. The simple device driver interface, first shown in Chapter 4, serves as a starting point. Using a device driver provides the applications developer with a set of system calls, wrapped in library routines using the well-known character device model, allowing the accelerators to be integrated with the operating system in a modular and well-understood fashion.

The software-only version of our test application `cjpeg` was augmented as little as possible to enable hardware accelerator integration. The simple character device driver is accessed by only three system calls: `open()`, `close()` and `ioctl()`. The `ioctl()` character device driver system call was chosen because it had the lowest overhead for a call that allowed the application to pass two parameters to the driver. A new system call could have been inserted for the accelerator and tailored to the application. However, the simple device driver method fits into the existing interfaces provided to character device drivers under Linux. Doing so allows the accelerators to look exactly like any other device in the system to the application.

Access to the hardware accelerator traces a simple flow starting with an `open()` system call to reset and initialize the accelerators, followed by multiple `ioctl()` calls to transfer macroblocks to the accelerator, and finally a `close()` system call to free access to the accelerator.

Initializing the accelerator, using the `open()` system call, resets the accelerator framework and DMA engine to ensure that everything is in a known state prior to accessing the accelerators. The shared implementation (i.e., one where multiple applications share the same accelerator framework) does not reset all the accelerators and the DMA controller on `open()`. In such multiaccess situations, the framework is reset only between transactions, or at accelerator context switch points, when a single

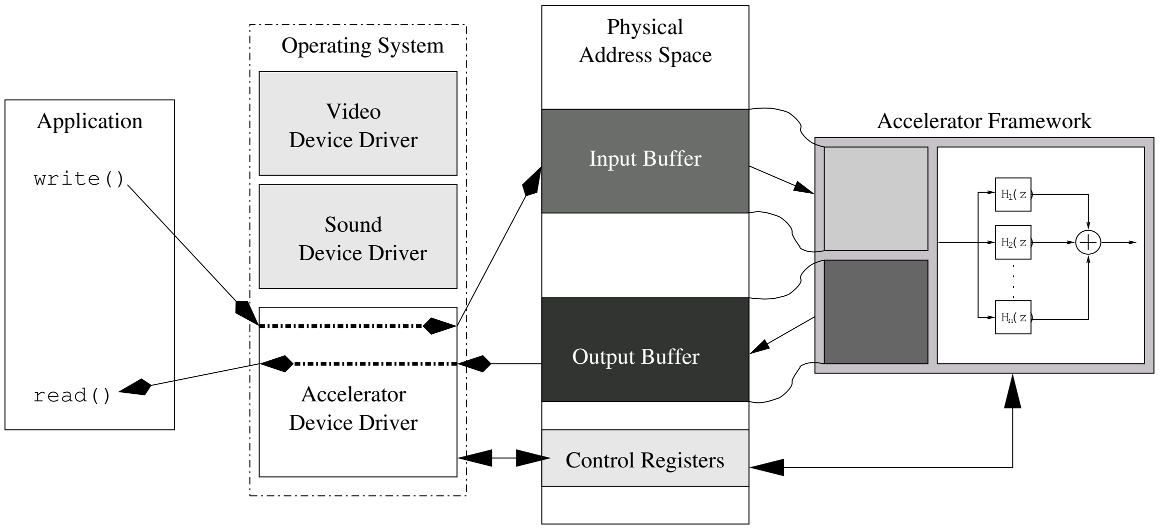

Figure 5.1 Example operating system interface using `read()` and `write()` system calls.

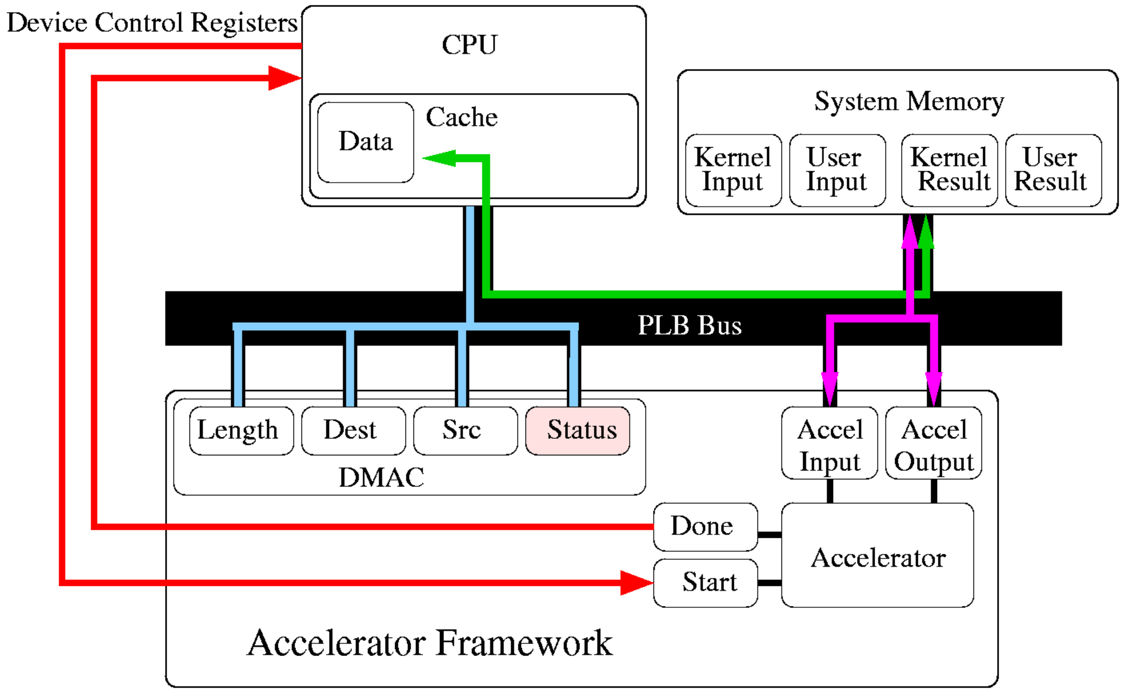

Figure 5.2 System block diagram with control and data transfers.

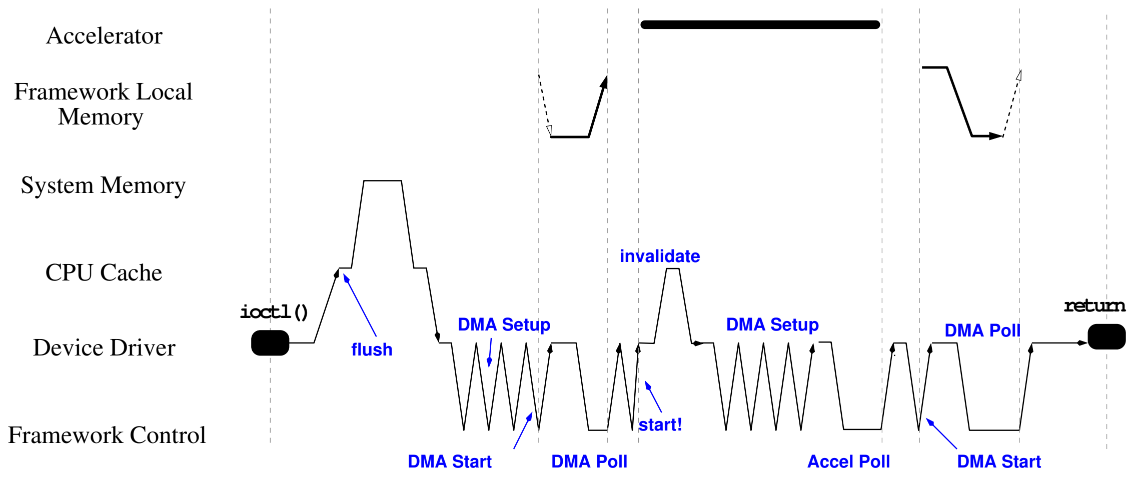

Figure 5.3 Flow chart of accelerator execution using the `ioctl()` system call.

applications issues accelerator requests and is eventually swapped out, leaving the accelerator unlocked, and another process is then swapped in.

In a strict model of access, where sharing is only allowed between uses, all `open()` calls block until the resource is free, at which point the operating system chooses the next process to give access to the framework. A blocking `open()` model limits concurrency severely. One model discussed later leverages virtualization of hardware resources to work around the case where the hardware must be locked between uses and two applications desire access to the accelerator framework. For the hardware infrastructure developed here, it is only necessary to maintain serial execution on a transaction-by-transaction, or `ioctl()`, basis, so `open()` locking is unnecessary. Instead, no state is maintained between transactions with the accelerator, allowing independent threads of execution to share the accelerator on a block–by–block basis.

Figure 5.2 depicts the flow of control and data for a call to the accelerator. Red lines denote control register transfers between the CPU and accelerator, green lines are used to represent data accesses by the processor and flushes, magenta lines represent the DMA transfers between the system memory and the accelerator local memory, and light blue lines show the PLB-mapped control registers for the DMA controller. Figure 4.4 has been reproduced as Figure 5.3 to assist in tracing the flow of control and data in the example below. In the reference to both of the figures, the following actions take place:

1. The application generates some data that is to be operated on by the accelerator. The data is placed in the user input buffer. The user input buffer exists as a contiguous region of the application's virtual address space, that is dynamically or statically allocated.

2. The application makes a call to the accelerator by issuing an `ioctl()` system call any time after initializing the framework with an `open()` system call, passing the call both a pointer to the user input buffer and the number of items in the buffer.

3. The data in the user input buffer represents a single macroblock in the simplest case for JPEG. The second parameter can be ignored. The kernel input buffer is a statically allocated array that is physically contiguous and marked as reserved so that it is capable of being accessed via DMA. The data in the user input buffer is copied to the kernel input buffer.

4. The cache lines occupied by the kernel input buffer are flushed from the cache to ensure coherence with system memory.

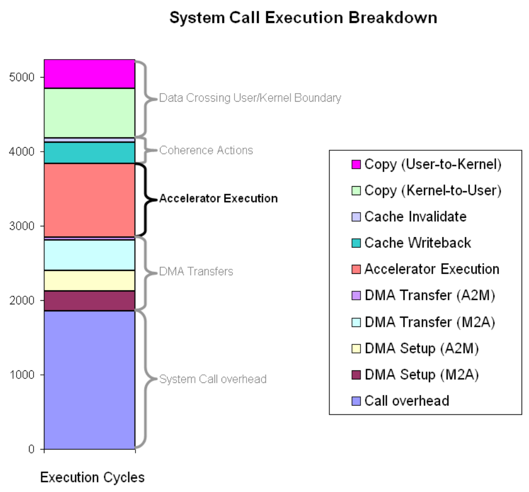

Figure 5.4 Percentage of time spent in each stage of the accelerator system call.

5. The source address, destination address, and setup word are sent to the memory mapped registers of the DMA engine instantiated in the accelerator framework. The length of the transfer (0x80 in the case of the 8 x 8 16-bit macroblock used for JPEG) is written to the memory mapped DMA register, initiating the transfer from system memory into the local memory of the framework. On either side of the store instruction to the DMA length register is a synchronization instruction (`eieio`) that ensures all the information required for the DMA transfer has been written to the DMA controller and so any future access to the control registers of the DMA controller reflect that a transfer is taking place.

6. While the data is being transfered into the BRAM constituting the local memory of the framework, the system call polls on the DMA memory mapped status register. When the register is cleared by the DMA controller, the accelerator driver proceeds.

7. The system call sets up the interconnect network as needed by setting a DCR mapped into the framework design. With the data in the input buffer of the framework, the system call starts the execution of the accelerators.

8. The design polls on a DCR mapped into the framework that is cleared when accelerator execution completes.

9. The kernel result buffer is a second kernel space buffer that is used by DMA transfers from the accelerator. The cache lines associated with the kernel result buffer are invalidated to ensure coherence.

10. Another DMA transfer back into the kernel result buffer resident in system memory is initiated by the accelerator driver.

11. The kernel copies the data from the kernel space DMA buffer back to the same user space buffer that provided the input data, and returns.

The time spent in each stage of the system call described above is shown in Figure 5.4. The simple device driver interface has many opportunities for pipelining, concurrent execution, optimization, and alternative access methods. However, to its credit, the character device driver access method provides a well-understood interface for hardware resources in the system. Using the simple character device driver model as a basis, we present a set of access methods that are employed with a near-identical, application-level interface, but handle buffer management and data transfers in alternative ways.

5.2 Protected Mode Data Transfer Methods

The operating system provides an abstraction, separating the user-level application from the underlying hardware. Even when only considering a library call-like interface, and without changing the way in which the application accesses the underlying hardware accelerators, there are many possible avenues to be taken. Decoupling the logical flow of data from the application to operating system and from the operating system to the accelerator local memory allows for swapping between access methods transparently by the operating system. This section explores four different access modes for moving data between the application running on the host processor and the reconfigurable accelerators built into the switchable interconnect network.

The alternative access methods developed in this section differ from the simple device driver model in the way in which data is transfered to the accelerator local memory. However, the application interface is kept as similar to the simple device driver model as possible to allow for future work to transparently swap between access methods, without requiring modifications to the application code. Further changes include the `ioctl()` call being replaced by a system call specific to the driver to reduce call overhead and provide added flexibility. Furthermore, for three of the models a special implementation of `mmap()` has been created to allow direct access to accelerator local memories and kernel DMA buffers from within user applications.

5.2.1 User space buffers

Due to the existent library support and its widespread use, a simple method for accessing accelerator resources from an application is to pass the operating system a pointer to a user space buffer via a character device driver. The operating system can check the pointer to ensure it is a valid location and then access the hardware resources as needed. The standard model for device driver interactions allows for the user to provide the operating system kernel with a pointer to the application's virtual address space via calls such as `ioctl()`, `read()`, and `write()` system calls. A device driver model provides the most abstract interface to the user space application. The user space buffer access method we develop is very similar to the character device driver method; however, we implement our own system call, to reduce the overhead associated with standard device driver system calls, and use it as a basis for our various access methods. The flow of events in the user space buffer method is similar to the simple device driver method and will not be repeated. While keeping most of the semantics of the simple device driver model, we present how the user space buffer access method can be used and how it is implemented.

The user space buffer method allows the application to access the accelerator resources without first setting up any sort of mapping or initializing any data structures internal to the operating system. The application can generate a block of data in user space and hand it off to the operating system for processing by the accelerator. The operating system is able to recognize that the application is using this access method due to the system call accessed and the fact that there is no previously set up mapping between the application and the accelerator resources, as is the case for the other models we will describe.

To access the accelerators in our simple model, the user generates a linear array of data and passes a pointer to the data to the operating system which then takes the appropriate actions to move data into the accelerator local memory. Figure 5.5 shows the user space buffer named `buf` and its location in virtual memory. The pointer references any part of the application's memory space that is allocated to it either dynamically (e.g., using `malloc()`) or statically (e.g., a global array allocated on the program stack). The physical location of the data is shown as "Buffer for Accelerator Input" in the figure. A sequence of copies must be initiated by the operating system to move data from the user space buffer into a DMA-enabled region of physical memory. Once in a DMA-enabled region, the data is transfered to the accelerator local memory via DMA. The local memory of the accelerator is mapped into the physical address space of the processor, but is not system memory, such as the DDR in our prototype

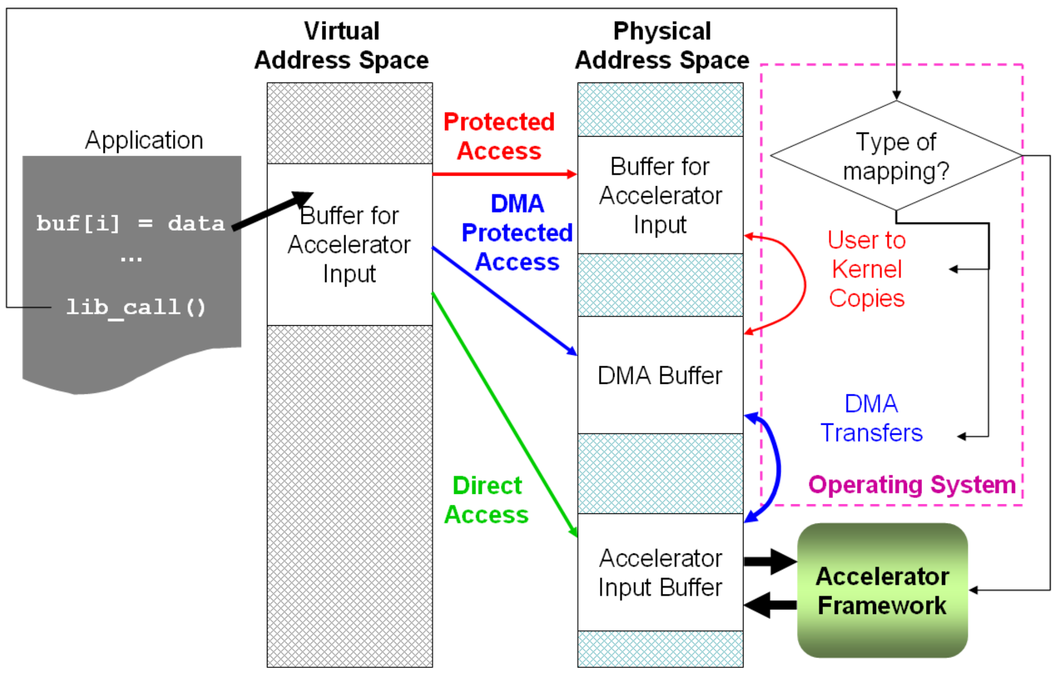

Figure 5.5 Protected access mode data copy methods.

platform. Instead, the local memory of the accelerators can be any type of memory located anywhere in the system, as long as it is physically addressable by the processor. We implement BRAM buffers, internal to the framework, that are physically addressable by the processor and DMA controller.

The call semantics describe what the programmer needs to know about the operating system interface that are necessary for correct program execution. Both blocking and nonblocking call semantics are allowable using the direct access method. A *blocking* system call begins when the user calls into the operating system at which point the kernel performs the request action immediately, or defers the execution of the call while the calling application sleeps. A *nonblocking* system call returns immediately informing the user of whether the access was performed without waiting; or it returns indicating that the access would require the application to sleep; or it returns an error message if the call is invalid. We extend the nonblocking model to defer execution of the accelerator by having the system call perform the necessary protection checks, scheduling the application's request, and returning. The aim of nonblocking calls is to allow the operating system to handle the user request without the application waiting on the result. For nonblocking calls, access to shared data must be protected by some form of synchronization. The programmer must ensure that it does not manipulate any data used by the kernel call that is not copied during the call by the operating system. Furthermore, the application must be careful to not access any results of the nonblocking call until the deferred work inside the kernel has finished. Our implementation uses only blocking system calls, for simplified semantics, such that when a call into the kernel returns, all accelerator interactions have completed. However, future work could explore nonblocking calls to increase concurrent execution on accelerators and the general-purpose processor.

User space buffers allow the operating system to check the permissions of the accessing application by providing an explicit transfer of control between the application and the operating system, while also isolating the accelerator resources from the user application. Furthermore, the operating system can reallocate resources on a call-by-call basis due to the explicit call into the operating system on every accelerator access. However, the user space buffer method also incurs added cost due to memory copying, coherence actions, system call overhead, and sanity checks that must be conducted before the kernel can dereference a user space pointer. However, a further advantage of passing a user space pointer is obviating the the need to set up a possibly costly mapping. An application need not perform any initialization prior to accessing the accelerator if user space buffers are used. The user space buffer model will now serve as the basis for more other access methods that provide better efficiency under many circumstances, but maintain the same calling semantics as the user space buffer access method.

5.2.2 User mapped DMA

The purpose of user mapped DMA buffers is to avoid the extra copy shown in Figure 5.5 as a red line between the user buffer and the DMA enabled buffer. User space buffers provide a simple abstraction that is familiar to UNIX systems programmers. However, the extra copies required when moving data from user space to a DMA enabled buffer incur added cost. The extra copies can be rendered unnecessary by having the operating system map a DMA enabled buffer directly into the user's virtual address space. Previous work has shown that such direct mappings for fast message passing in clustered environments [39] can improve performance dramatically by reducing the number of unnecessary copies. The user mapped DMA mode of accelerator access still encapsulates the hardware resources while providing faster access, at the cost of increased initialization time.

The user mapped DMA access method differs from the user space buffer method described in the last section in that it must first set up a mapping, through a driver controlling the accelerator, prior to use. The application sets up the necessary mapping by making a `mmap()` system call, created specifically for this device driver. During the mapping call, the operating system allocates a DMA-enabled buffer that will be mapped into the user space application. The buffer is pinned into memory, so that it cannot be swapped out, and made present in physical memory. The mapping is inserted into the page tables of the user application, enabling the proper address translation to occur between the virtually addressed buffer, `buf[]` in the figure, and the DMA enabled buffer. The operating system sets up, and henceforth maintains, a mapping between each application and the DMA-enabled buffer. When the user makes a call to the accelerator, the coherence actions and DMA transfers can take place immediately, without a need to copy from a user space buffer. The operating system is able to recognize what actions must occur based on the previously recorded mapping. The decision box in the figure shows the operating system performing the necessary actions to move data into the accelerator local memory. The accelerator can then proceed and place the output in its local memory. The system call now reverses its previous actions by performing a DMA transfer out of the accelerator local memory and back into the user mapped DMA buffer, and then returns. Two copies are rendered unnecessary by using user mapped DMA buffers: both when entering and exiting the call, no copy between a user space buffer and the DMA enabled buffer is performed.

The user mapped DMA method saves an unnecessary copy from user to kernel space at the cost of a `mmap()` system call; however, there are other implications for providing user applications with direct mappings to DMA-enabled buffers. The operating system must now manage the allocation of a possibly fixed amount of DMA capable memory among many applications. Flexibility may be gained by allowing the DMA mapping to be removed by the operating system by marking the page not present and flushing the TLB entries associated with the mapped page. A change must also be made to the page fault handler to fault in a mapping when requested by a user, and possibly removing another pinned mapping from another application. Our design does not implement shared DMA mappings, but for architectures with limited DMA enabled regions, or fixed amounts of physical memory, such additions can be considered. Furthermore, pinning a large amount of memory, especially for accelerators that require large data transfers, is not an effective use of system resources and can reduce the operating system's ability to provide applications with the memory they need to execute. Performance would be degraded, in a system with reduced available physical memory, if pages must begin being swapped in and out of the few remaining free physical page frames. For future systems, where many applications are accessing an assortment of accelerators, it will be necessary to develop more cooperative models for DMA mapping access to the accelerators, but such models are not explored in this work.

On the PPC platform, all physical memory is DMA-capable. The only restriction for allocating buffers in our present model is the need to find a free area of physical memory that is large enough for the DMA buffer and is contiguous. Another method for enabling noncontiguous DMA buffers would require making use of the scatter-gather facilities of the DMA controller we employ. The DMA buffer would be presented to the application as a contiguous region of virtual addresses; however, it would be a scattered collection of physical pages mapped into a contiguous region. As none of the applications make use of buffers larger than a page in size and our DMA controller implementation cannot support regions larger than a page, scatter-gather DMA was not examined, but is available for future development.

Mapping DMA-enabled buffers directly into the virtual address space of applications allows those applications to access the accelerators more efficiently. The system bus is more fully utilized by enabling eight byte transfers every 10-ns bus cycle during a burst. However, efficient DMA transfers come at the cost of overhead associated with setting up the DMA transfers. The setup costs may negate the performance gains earned by using DMA as opposed to alternative access methods. For many applications, however, the benefits of DMA-enabled access may outweigh any of the access method's shortcomings. With the mapped DMA buffer method we are able to take advantage of the benefits of DMA, but with reduced cost compared with the user space buffer method, while only incurring the one-time setup cost of the buffer mapping.

5.2.3 Direct access

Direct access refers to the operating system, at the request of an application, mapping the local memories of the accelerators directly into the virtual address space of the application. Both cacheable and noncacheable mappings are possible. The two direct access methods can provide a high level of protection and can allow resources to be virtualized, while also removing the need to perform possibly costly DMA setup routines. A setup cost to initialize the mapping must be paid, however, making direct access less attractive for certain applications. The direct access method for accessing accelerators, and its implementation on our prototype, is now discussed.

Direct access is instrumented by providing the driver interface with an `mmap()` system call that maps the physical address range of the accelerator memory into the virtual address space of the application. The `mmap()` call uses `ioremap()` to setup the necessary page tables and then inserts those page tables into the virtual memory area (VMA) of the calling application. The buffer used by the application is still the same as the user space buffer, named `buf[]`, shown in Figure 5.5. However, now the buffer maps directly to the local memories of the accelerator, bypassing the two other buffers in physical memory shown in the figure. Instead of writing to system memory, data generated by the application is transfered directly to the local memory of the accelerator. When the operating system is called, using the same model as with the other access methods, the operating system takes the necessary actions, starts the accelerators, and returns to the application when the output is available. Since the accelerator buffer is mapped directly into the user application, no explicit transfer of data must take place before returning to the application.

To implement direct mappings, two methods are possible: the page table entries can either be marked cacheable or noncacheable. If the entries are marked as noncacheable, then every time the application accesses the accelerator with a load or store instruction, the processor must write out to the memory. By forcing a write on every access, the processor write buffer fills quickly as it cannot flush data to the cache, but must make high latency transfers all the way out to memory for each access. Eventually the processor will stall, reducing performance. However, marking the mapping as noncacheable removes the need to perform an explicit cache line flush for the regions in the mapping. If a cacheable mapping is allowed, the inefficiency of noncacheable access is mitigated.

The mapping from virtual addresses used by the application to the physical addresses used by an I/O device are normally marked noncacheable. By making the mapping cacheable, the accelerator can take advantage of cache line transfers that allow for better bus utilization, fewer stalls due to write buffer blocking, and possibly overlapping cache line flushes with computation. However, marking the mappings as cacheable requires explicit cache flushes that may take many cycles. Depending on the application's buffer sizes and data production rate, one method may be favored over the other. The discussion of which method is appropriate for a given data set is left to the next chapter.

Direct mapping makes the accelerator resources more visible to the user application at the cost of reduced abstraction. The interface provided by all the methods so far remains unchanged in the direct mapped method, as the application must still call into the kernel to enable the transaction with the accelerators to proceed. If the direct access method were taken to an extreme, by making all accelerator resources available, nearly all abstraction would be lost. However, the operating system could implement a model similar to Exokernel [40]. In doing so, the whole accelerator platform would be virtualized, giving each application the illusion of having direct access to the device. The loss of abstraction at the system call level breaks the transparent model and therefore is not explored in this work, but remains as an avenue for future work.

Direct mapping provides an alternative to the user space buffer and user mapped DMA buffer methods. Mapping the accelerator resources directly into the application's virtual address space allows all extraneous copies to be removed. The uncacheable version of the method removes the need to perform coherence actions as well and results in the lowest possible overhead per memory access. However, word-by-word transfers may cause unnecessary stalling of the processor. Cacheable direct mapping provides an alternative that allows for faster, cacheable transfers at the cost of explicit cache management instructions being needed when performing the system call to initiate an accelerator transaction. One restriction placed on direct mapping, in either cacheable or uncachable form, is the requirement that there be a one-to-one mapping between accelerator buffers and applications. If preemption is allowed, there must be a means to save the state of the buffers on an accelerator context switch and restore that state when the preempted application becomes runnable once again. Our work does not implement mechanisms that allow for multiple accessors, using the direct access model, and is left for future work.

5.3 Summary

The hybrid CPU/accelerator platform described in this chapter provides a rich set of choices for system-to-accelerator interfaces. The character device driver model represents a well-understood method for accessing hardware resources. The model was used as the basis for the application-to-accelerator framework interfaces. However, the character device driver method for accelerator access performs unneeded protection checks, requires added maintenance of kernel data structures, and requires more calls than are necessary for accessing hardware accelerators. A set of methods that deviate from the pure character device driver method have been described and compared. Each method represents a node in the design space exploring the tradeoffs between setup overhead, accelerator access, level of virtualization, and data transfer overhead. All methods provide a means to access the hardware accelerators, mapped into the accelerator framework via a common interface, while maintaining protection boundaries.

CHAPTER 6

EXPERIMENTS

Experiments were conducted to determine the performance of the different interface models presented in the previous chapter. The cacheable direct mapped method is found to be optimal for JPEG encoding, but not necessarily for all applications. We show that for a low number of accesses, the user space buffer method proves to be the optimal choice by avoiding the costly overhead of initialization. The other three access methods prove optimal over a range of data set sizes, where the initialization cost can be amortized over the lifetime of the application.

We present the different methods we employ for quantitatively measuring the performance of the operating system, example application, and reconfigurable accelerators using our CPU/FPGA prototype platform. The issues associated with these methods are discussed and their use in this work is noted. Performance results are presented for platform-specific operations, required operating system calls, and all phases of the system calls and data transfer time necessary for accessing the accelerator framework. The space requirements of the interconnect relative to accelerators in our model platform are given. A brief summary of the relative merits of each access model concludes the chapter.

6.1 Methodology

All hardware accelerator experiments were conducted using the XUP Board platform, with Linux 2.4 running on one of the PowerPC cores embedded in the FPGA. The performance of the experiments was measured by one of three methods: the Linux `time` application, the *gettimeofday()* system call, or by reading the timebase for the PowerPC processor.

The `time` utility calls the Linux system call *times*, which returns a kernel generated structure listing the application's time usage in terms of system ticks. A tick on the PowerPC405 occurs every 10 ms, causing it to have the poorest resolution of the three methods used. The *gettimeofday()* system call on the PowerPC405 running Linux 2.4 includes the overhead of the system call, which is on the order of a microsecond. For full application measurements, *gettimeofday()* was used instead of the `time` application.

All fine-grained measurements, and all measurements made inside the kernel, were performed using the 64-bit PowerPC405 timebase. The timebase is split into upper and lower 32-bit registers. The timebase lower register increments every cycle while the timebase upper register increments every time the lower register overflows. At 300 MHz, the timebase lower register overflows approximately every 14 s. The execution of a `mftbl` instruction takes approximately three processor cycles to execute, causing each read to incur approximately 10 ns of overhead. To ensure completion of any outstanding instructions prior to making a measurement, synchronization instructions are inserted prior to all timebase measurements.

Application profiling was used to determine the relative portion of time spent in each phase of the computation. The GNU profiler was used for all profiler analysis in this work. Profile data returned by `gprof` is stochastic and the sample rate is limited by the operating system and the system architecture. As such, profiling results are not used for fine-grained measurements. However, profiling is an invaluable tool when determining the appropriateness of a portion of software for implementation as a hardware

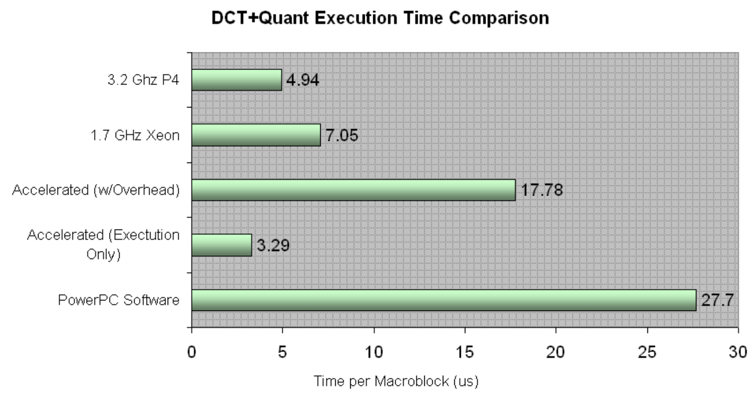

Figure 6.1 Comparison of average DCT and Quant execution time for cjpeg compressing a 1.4-MB bitmap using standard JPEG.

accelerator. Coupled with the results presented for our interface methods in this chapter, a system designer can make use of the techniques listed above to partition his design to achieve desired results.

6.2 Results

Various data transfer methods are available for application designers to choose from. The choice of transfer method depends on the required call semantics, transfer latency, transfer size, and number of transactions per use. Furthermore, speed, overhead, and FPGA area implications of each transfer method must be understood to develop accelerators that provide gains in performance, cost, and energy efficiency. Our results provide insight into what tradeoffs must be made when developing accelerators for applications running in a hybrid CPU/FPGA environment.

6.2.1 Performance

We will look at application performance in terms of time to execute a single call to the accelerator framework. For our test application JPEG, one access to the framework constitutes a single, 128-byte macroblock being processed. Figure 6.1 shows the results of profiling done on the JPEG encoder application for two different x86 machines and our test platform using the simple device driver interface method. We provide these results to show the large speedup possible when using hardware accelerators that can easily be hindered by the large overheads associated with accessing these accelerators. The speedup of the quantization and DCT accelerator combination, running at 100 MHz, is faster than both x86 machines running at an order of magnitude higher clock frequencies and is 8.4x compared to the PowerPC software-only implementation. However, once the call and data transfer overhead is accounted for, the CPU/accelerator model is slower than the x86 machines and only 1.5x faster than the PowerPC software-only version. We will now deconstruct the accelerator access time and show the speedups provided by our alternative access methods.

There are many components that contribute to the time required to do an FPGA accelerated computation. There are startup costs associated with access methods. There are overheads from going into the kernel and performing the added resource allocation. Protection checks add to the time a call takes to complete. There is time spent moving data to and from the accelerator, from system memory, or the processor cache. This section illustrates system-level effects of using system call library-like interfaces to reconfigurable accelerators. Future work can develop hardware accelerated applications and the as-

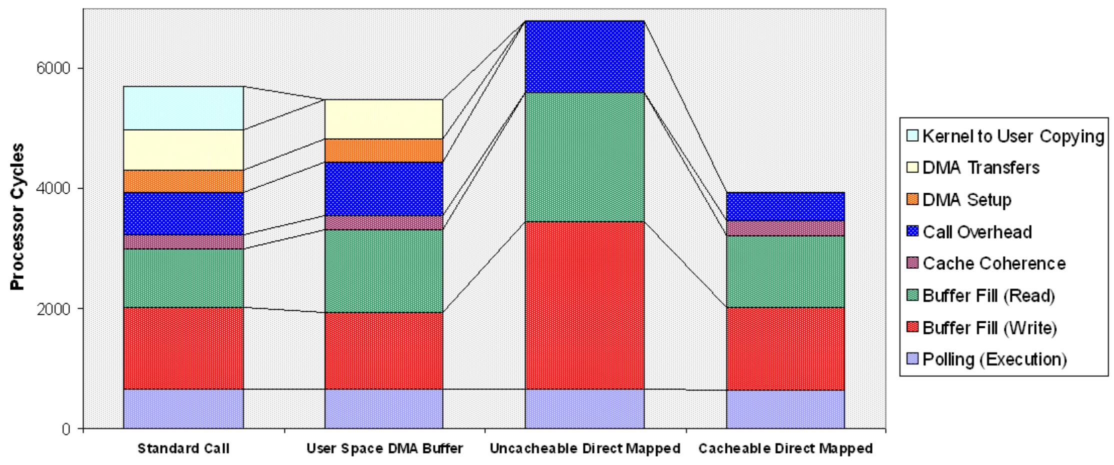

Figure 6.2 Accelerator access breakdown across four access modes.

sociated infrastructure, using our framework and results, that will break away from the present state of accelerated applications that require embarrassing levels of parallelism and provide little protection.

6.2.2 Accelerator access cost

Four access methods were described in the last chapter: user space buffers (standard method), user mapped DMA buffers, cacheable direct mapping, and uncacheable direct mapping. To test these methods, the JPEG encoder was implemented to use the accelerator framework with DCT and quantization. Each access to DCT and quantization in software was replaced by a system call to access the hardware accelerators, and the performance measured. We find that the direct cacheable method provides the best speedup for the hardware-accelerated version of JPEG encoding. The access cost of each method for a set of data set sizes is also provided to enable developers to choose the appropriate access method based on their application's needs.

During JPEG encoding, each macroblock must be transformed and quantized. In our experiments, a 1.4-MB test bitmap image is used and requires 11 900 calls to the accelerator. For each method used, the time spent filling buffers and executing each call was independently measured using the timebase while encoding our test image. Due to the large number of accelerator accesses for JPEG, startup costs are easily amortized, and are therefore not considered in our per-call experiments. However, startup costs are described in further detail in the the next section. Figure 6.2 compares the time spent in each part of the accelerator call for the four methods.

Figure 6.2 shows the cost of system call overhead is not the main factor in determining the performance of the JPEG accelerator calls. A great deal of time is spent transferring data and performing the actions necessary to maintain coherence and set up the transfers. From the figure it is clear that finding low-latency, high-throughput methods for getting data into accelerators is key to extracting the full potential of hardware accelerators. Another point to be made about the figure is the optimal case being the cacheable direct mapping. Even though cacheable direct mapping requires more time to transfer data to the accelerator local memory than both of the DMA methods, it provides the greatest speed due to not requiring costly DMA setup time and the additional cost of a DMA transfer on top of a buffer being filled.

If an application developer were to develop an application with similar data transfer characteristics to JPEG, the application accelerated in our experiments, the cacheable direct mapping method would be

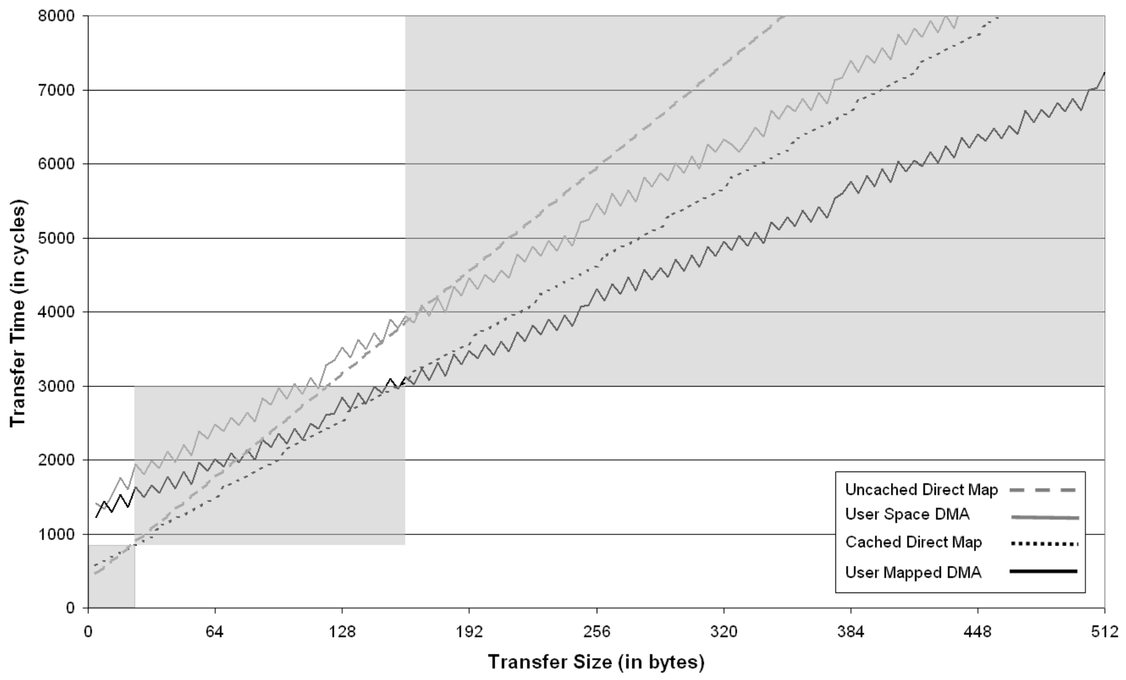

Figure 6.3 Cycles required for the four access methods with varying transfer sizes without the cost of the initial mappings factored in.

the best choice. However, applications have vastly different requirements in terms of number of accesses, sizes of data transfers, and hardware accelerator execution time. To provide more general results, we have timed each of our methods for different data transfer sizes. Figure 6.3 depicts the call time, in cycles, for data transfer sizes ranging from 4 bytes to 512 bytes, in 4-byte increments. To include all the overhead associated with each call, we time the following: data copies to temporary buffers, system call overhead, DMA setup and transfers (if required), cache coherence actions (if required), and data copies from temporary buffers. No accelerator execution is included in the timing measurements.

There are three distinct phases in our data transfer characterization, highlighted in Figure 6.3. Uncacheable direct mapping is the fastest access method, from 0 bytes to 16 bytes, due to its non-existent startup cost and lack of explicit cache flushes. However, the cached direct mapped method is the optimal method for transfer sizes of 20 bytes to 160 bytes. The low utilization of the bus causes the cached direct mapping to overtake the uncached version. The 32-byte cache line size has some effect on the smoothness of the curve, with the cached direct mapping showing inflection points at each 32-byte interval.

For applications with large transfer sizes, that can bear the cost of setting up a DMA-capable buffer, user mapped DMA method is the best choice. Above transfer sizes of 160 bytes, the overhead of setting up DMA transfers is overtaken by the higher bus utilization of DMA-based transfers. The user space buffer method, which also employs DMA but requires the cost of an extra copy from user space buffers to DMA-capable regions inside the accelerator system call, is never optimal when comparing just the transfer cost. However, as discussed in the next section, startup costs render the other three methods less effective for low numbers of accelerator calls, due to the high overhead required for setting up mappings. Lastly, the jaggedness of the curves for two DMA methods is due to the reduced efficiency of bus transfers that are less than the bus width of 64 bits. Due to the DMA controller we employ, any transfer that is

Table 6.1 Mapping overhead for user DMA and both cacheable and noncacheable direct mapping.

	Call Overhead			
	Average Case		First Access	
Call	Cycles	Time (μs)	Cycles	Time (μs)
`open()`	14 381	47.94	34 472	114.8
`mmap()`	8 422	28.01	24 487	81.62
`munmap()`	7 357	24.52	14 467	48.22
`close()`	4 509	15.03	9 581	31.94
Total	34 699	115.5	83 007	276.7

not a multiple of 8 bytes will require a separate transfer, resulting in reduced performance for such data transfer sizes.

Using the results presented in this section, an access method can be chosen to match an application based on the data set size. The values provided in this section are only for accelerators that can easily amortize the startup cost over a large number of accesses. For those accelerators that cannot, the next section will provide the needed insight that, when coupled with the data presented in this section, will allow for the proper choice of access method for a given application.

6.2.3 Accelerator startup cost

The times spent executing system calls necessary to initialize the mappings and set up DMA buffers in our model are presented in Table 6.1. The applications we study that are amenable to hardware acceleration either access accelerators repeatedly (e.g., JPEG), or are long running (e.g., filtered back-projection). The startup costs can be nontrivial, as is shown in the table by the first access times. The JPEG accelerators are able to amortize the cost of a possibly long initialization process. In cases where the time spent setting up the accelerator is trivial compared to the time spent accessing the accelerators, only the access cost need be examined. For more fined-grained, or less often accessed accelerators, the negative effects of startup and access cost must be mitigated to achieve an optimal system design. Experimental results show that the cost of setting up the mappings required for user-level DMA buffers and direct access are listed in Table 6.1. All of the methods presented require these calls to be made once during the application's execution, except for the user space buffer method that requires no setup prior to use. Both average (best) case and first access results are given. The slowdown for first access to the system calls necessary to set up a mapping can be attributed to the cost of cache misses and TLB misses. When an application first accesses the accelerator by calling `open()`, or uses `mmap()` to set up mappings between operating system controlled resources and the application, the areas of kernel memory where those instructions and the associated data reside are likely not in the CPU caches and may not have a mapping in the TLB. The side effect of not having the instructions in the cache is added latency for the call. A TLB miss on the PowerPC requires operating system intervention, and can waste many hundreds of cycles. Ways to avoid the high cost of startup may be developed to enable better use of startup-sensitive accelerators.

Future architectures could make use of special calls to reduce the overhead associated with setting up the access methods we describe. One example is pinned mappings that would enable average case system call overhead to revert toward the best case by removing TLB miss overhead. However, performance elsewhere may suffer with fewer TLB entries available. Furthermore, doing so many not be possible in other architectures that, unlike the PPC405, do not provide software management of the TLB. The opportunity to improve startup is immense. As our results show, setting up such mappings may add as much as 276 μs to the accelerator total access cost just for setting up and tearing down the mappings.

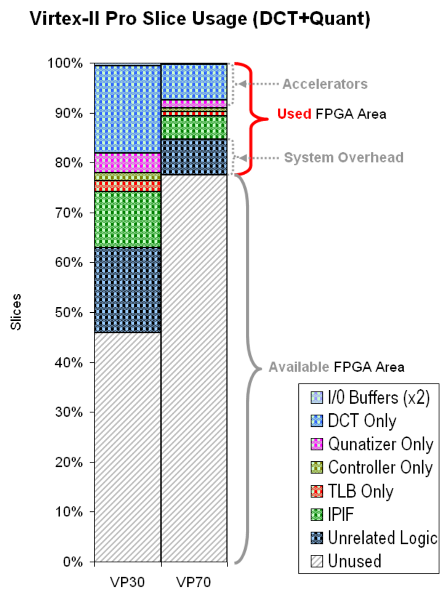

Figure 6.4 FPGA area utilization on the XUP Board's VP30 and the VP70 for comparison.

6.2.4 Area utilization

Figure 6.4 shows the free resources left on the FPGA after instantiating the processor support logic and the accelerator framework. The VP30 is one of the smaller FPGAs in the Virtex-II Pro product line, but there is space remaining for multiple accelerator blocks for processing independent macroblocks. Some additions to the control logic could allow for parallel access to the local store and the ability to feed multiple pipelines. However, as shown earlier, the throughput of the system bus is limited. If each accelerator were allowed to process data at full speed, 2 bytes per accelerator would be consumed every bus cycle. Optimistically, the accelerators would become I/O limited at fewer than two pipelines using a 100-MHz PLB. Therefore, area is not a major concern for our platform, but the available throughput of the interconnects and the processor's ability to provide input data should be considered more critical when developing similar, future systems.

6.3 Summary

In this chapter, we have presented results for our framework, based on our JPEG encoder example. Furthermore, we have characterized the setup, access, and area costs of our interfaces and prototype framework. User space buffers that are passed to the kernel have the largest per call cost, but require no setup. For a short running accelerator, the user space buffer approach is desirable. The price that is paid for a low per-call cost is the need to set up a mapping. If the accelerator is accessed enough times to amortize the cost of the initialization across the time saved by setting up such a mapping, then DMA or direct mappings are ideal. For our test application, JPEG encoding, we find that cacheable direct mappings are the optimal choice for our platform.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

A hybrid CPU/accelerator computer system model is presented that is capable of running common applications on a real hardware platform. The goal set for the project was to develop and characterize a system in which common applications could be accelerated using reconfigurable hardware accelerators. The hybrid model allows for more efficient, faster computation while maintaining the protection boundaries within the system. Two major contributions are described here. First, we present a framework that allows hardware developers to map their accelerators into our hybrid CPU/accelerator system model. It allows the hardware developer to do so without worrying about how an applications developer will interface with the accelerator. Second, we present a set of interfaces that can be used by applications developers to access hardware accelerators without concern for how they are implemented, or if the hardware resources are even available. The computation model we discuss provides the logical abstractions necessary for integrating our interfaces and accelerator framework into a contemporary, general-purpose computing environment.

The interfaces developed abstract away the details of the framework, and decouple the accelerator development from the application development. Skilled applications developers can continue to develop high-quality applications while utilizing hardware accelerators, much as they are accustomed to utilizing shared libraries today. Skilled hardware developers can continue produce high quality accelerators that can now be seamlessly integrated with a general-purpose operating system. Our contributions provide a basis for future development of heterogeneous systems that wed FPGA-based accelerators and general-purpose microprocessors by building a set of interfaces for both accelerators on one end, and applications on the other, that can communicate effectively using our computation model as a basis for the connections between the two.

7.2 Future Work

We present a platform for future hybrid CPU/accelerator systems. Moreover, the platform developed here demonstrates the viability of our heterogeneous computing model by providing a real working prototype. However, our model and framework are only the beginning; many tools still must be developed and parts of our model refined.

Independent transactions are unnecessarily serialized and could take advantage of multiple parallel pipelines. Even though the processing of a single block is fully pipelined between the accelerator local store and the DCT/quantizer, the parallelism inherent in the code is not fully exploited.

The ultimate goal of this project is to allow a developer to take an application and map it into a hybrid CPU/FPGA platform seamlessly. To do so requires compiler tools, with possible annotations in the code, that allow for the heavily used, data-parallel kernels of an arbitrary application to be extracted from the source. Once extracted, data-parallel kernels of code must be mapped into reconfigurable logic. The reconfigurable accelerators can be integrated with the system using our accelerator framework. The application can access the accelerator through a library-like interface that is managed by the operating system. The user need not worry about the size or availability of reconfigurable resources. The operating system loads the accelerator into hardware or into memory as a normal shared library depending on the system resources available. In the case of software execution, resources are virtualized by the operating system, leaving the calling semantics intact.

Efforts exist to provide mappings from high-level languages, such as C, into hardware description languages, such as VHDL or Verilog. Porting these tools into our framework is one step necessary for enabling widespread use of the CPU/accelerator model developed here. Tools to address the hard problem of taking general program code written in a high-level language and finding appropriate sections to map into our platform must be developed for seamless integration to be possible.

Our framework could be further developed to provide more autonomy to the accelerator framework. A small degree of unnecessary latency is experienced when performing operating system to accelerator transactions. The latency is the product of the CPU accessing the DMA controller exclusively. If the accelerator framework were allowed to access the DMA controller directly, instead of the CPU, the accelerator could be provided with all the necessary information to complete a single transaction at the transaction's start, and then allowed to run to completion, obviating the need for CPU intervention. For larger data transfers, and future implementations where concurrent software and hardware threads are executing, the longer runtime of accelerators, without CPU intervention, could be exploited.

REFERENCES

[1] J. H. Kelm, I. Gelado, K. Hwang, D. Burke, S.-Z. Ueng, N. Navarro, S. Lumetta, and W. mei Hwu, "Operating system interfaces: Bridging the gap between cpu and fpga accelerators," Tech. Rep. UILU-ENG-06-2219, University of Illinois at Urbana-Champaign, October 2006.

[2] T. G. Lane, *Using the IJG JPEG Library.* Independent JPEG Group, 6th ed., March 1998.

[3] Xilinx Staff, *PowerPC 405 Processor Block Reference Guide.* Xilinx, 2100 Logic Drive, San Jose, CA 95124, 2nd ed., July 2005.

[4] Altera Staff, *Excalibur Devices Hardware Reference Manual.* Altera, 101 Innovation Drive, San Jose, CA 95134, 3rd ed., November 2002.

[5] Nallatech Staff, *BenONE-PCI Reference Guide.* Nallatech, One Napier Park, Cumbernauld, Glasgow G68 0BH, United Kingdom, 7th ed., April 2005.

[6] W. Thies and M. Karczmarek and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *International Conference on Compiler Construction*, 2002.

[7] J. R. Hauser and J. Wawryznek, "Garp: A mips processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[8] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1997, pp. 87–96.

[9] E. M. Panainte, S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, and S. M.-G. Kuzmanov, "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[10] A. DeHon and et. al., "Stream computations organized for reconfigurable execution," *Journal of Microprocessors and Microsystems*, vol. 30, pp. 334–354, September 2006.

[11] Xilinx Staff, *MicroBlaze Processor Reference Guide.* Xilinx, 2100 Logic Drive, San Jose, CA 95124, 6th ed., June 2006.

[12] R. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, IEEE Computer Society Press, 1996, pp. 126–135.

[13] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, New York, NY, USA, ACM Press, 1999, pp. 145–154.

[14] M. Vuletic, L. Pozz, and P. Ienne, "Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors," in *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, 2004.

[15] H. Schmit, "Incremental reconfiguration for pipelined applications," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, IEEE Computer Society Press, 1997, pp. 47–55.

[16] M. Welsh, A. Basu, and T. von Eicken, "Incorporating memory management into user-level network interfaces," Tech. Rep. TR97-1620, Cornell, 1997.

[17] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina, "Virtual-memory-mapped network interfaces," *IEEE Micro*, vol. 15, no. 1, pp. 21–28, 1995.

[18] InfiniBand Trade Association, *InfiniBand Architecture Specification*, vol 1. 1st ed., 2004.

[19] H. Walder and M. Platzer, "A Runtime Environment for Reconfigurable Hardware Operating Systems," in *Proc. of Field Programmable Logic*, Leuven, Belgium, Springer, Aug. 2004, pp. 831–835.

[20] B. C. Watson, "Filtered backprojection implementation for a field-programmable gate array," Master's thesis, University of Illinois at Urbana-Champaign, 2006.

[21] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE)," in *Field-Programmable Logic and Applications*, 2000, pp. 605–614.

[22] C. Giraud-Carrier, "A reconfigurable dataflow machine for implementing functional programming languages," *SIGPLAN Not.*, vol. 29, no. 9, pp. 22–28, 1994.

[23] B. Lee and A. Hurson, "Dataflow architectures and multithreading," *Computer*, vol. 27, pp. 27–39, 1994.

[24] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, 2004.

[25] L. A. Corts, P. Eles, and Z. Peng, "A survey on hardware/software codesign representation models," save project report, Linkping University, Linkping, Sweden, 1999.

[26] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *International Journal in Computer Simulation*, vol. 4, no. 2, 1994.

[27] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[28] M. M. Chu, "Dynamic runtime scheduler support for score," Tech. Rep. UCB/CSD-01-1134, EECS Department, University of California, Berkeley, 2001.

[29] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, and A. DeHon, "Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine," in *International Symposium on Field-Programmable Gate Arrays*, 2002.

[30] W. Fu and K. Compton, "An execution environment for reconfigurable computing," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, vol. 13, pp. 149–158, 2005.

[31] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee, and W. J. Dally, "Concurrent event handling through multithreading," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 903–916, 1999.

[32] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, "An empirical study of programming language trends," *IEEE Software*, vol. 22, no. 3, pp. 72–78, 2005.

[33] J. E. Smith and A. R. Pleszkun, “Implementatin of precise interrupts in pipelined processors,” in *12th International Symposium on Computer Architecture*, 1985.

[34] Xilinx Staff, *XST User Guide.* Xilinx, 2100 Logic Drive, San Jose, CA 95124, 7th ed., 2005.

[35] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgeford, “Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas,” in *16th International Conference on Field Programmable Logic and Applications*, 2006.

[36] Xilinx Staff, *2-D Discrete Cosine Transform (DCT) Product Specification.* Xilinx, 2100 Logic Drive, San Jose, CA 95124, 2th ed., March 2002.

[37] ModelSim, *ModelSim SE User’s Manual.* Mentor Graphics, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777, 6.0e ed., June 2005.

[38] Xilinx Staff, *ChipScope Pro Software and Cores User Guide.* Xilinx, 8th ed., October 2005.

[39] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: A mechanism for integrated communication and computation,” in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, pp. 256–266.

[40] D. R. Engler, “The exokernel operating system architecture,” Ph.D. dissertation, MIT, 1998. Supervisor-M. Frans Kaashoek.