

Operating System Interfaces: Bridging the Gap Between CPU and FPGA Accelerators

John Kelm*, Isaac Gelado†, Kuangwei Hwang*, Dan Burke*, Sain-Zee Ueng*,
Nacho Navarro†, Steve Lumetta*, Wen-mei Hwu*

*Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{jkelm2, khwang3, drburke, ueng, steve, hwu}@crhc.uiuc.edu

†Computer Architecture Department
Universitat Politecnica de Catalunya (UPC)
{igelado, nacho}@ac.upc.edu

ABSTRACT

We have developed operating system interfaces for CPU/FPGA hybrid systems running standard applications. This hybrid model has not enjoyed widespread success, partly due to the lack of software support on the processor. This paper presents a fully functional system with the GNU/Linux operating system running on a PowerPC processor embedded in a commercial FPGA. Further, this system is used to prototype a hybrid processor/accelerator execution model, implementing the accelerators in the FPGA fabric. This model of computation combines the flexibility of general purpose processors with the performance and energy efficiency of tailored accelerator hardware. The system supports two alternative models for integrating hardware accelerators with the application running on the embedded processor. The direct access model provides simple, low operating system overhead communication between the CPU and the accelerators, whereas the indirect access model increases reliability and protection at the cost of additional run-time operating system overhead. Three standard desktop applications are ported to the system. All showed positive speedup when executing with the accelerators. Detailed real hardware measurements demonstrate the considerations that must be taken into account in order to achieve speedup when porting applications into our system.

Categories and Subject Descriptors: C.1.3 [Computer Systems Organization]: Processor Architectures—*Heterogeneous (hybrid) systems*; C.5.m [Computer Systems Organization]: Computer System Implementation—*Miscellaneous*

General Terms: Design, Experimentation, Performance

Keywords: FPGA, Interconnects, Operating System Interfaces

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

1. INTRODUCTION

Trends in hardware suggest that transistor density will continue to increase, but better usage of the available transistors and better performance per watt is needed. General purpose processors provide good performance across a large variety of programs, but are often too power hungry or too slow for many embedded applications. Specialized circuitry can provide the desired performance and power efficiency for computation kernels, but is not suitable for supporting a dynamic mix of common applications. A possible hybrid model is to supplement a general purpose processor with accelerators. In such a model, the processor executes sophisticated applications as well as operating system services while off-loading compute intensive parts of the application to accelerators for enhanced performance and/or power efficiency. FPGA devices that incorporate general-purpose processor cores, such as the Xilinx Virtex-II Pro series [19], provide a promising medium for implementing this hybrid model.

However, the work to construct and deploy accelerators in these hybrid FPGA devices for sophisticated applications that execute under standard operating systems has been slow to come. The need has become even more urgent as many video, audio, image and game applications that require standard operating system services such as file systems and networking, are now being ported from desktop Windows and Linux environments to cell phones and mobile media devices. It is extremely desirable that future embedded systems can run these applications while still achieving performance and power efficiency goals without significant changes to the source code.

We present two interface designs in this paper. The first design provides a direct form of access to the application on the general purpose processor while the second enables better virtualization and protection by further leveraging the operating system. Furthermore, we have developed a switchable interconnection network that simplifies the interactions between reconfigurable accelerators and the operating system and serves as a vehicle for future development. We explore our two interface designs using three application examples: H.263 video encoding, MP3 audio processing and JPEG image compression. We obtained our applications from open source Linux software repositories. All of these applications depend upon Linux operating system support, such as file system

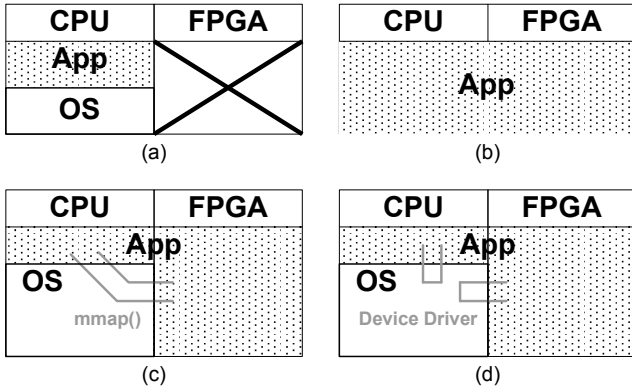


Figure 1. Computation Models

and audio and graphics device drivers. If such support is absent, a substantial amount of application and system modifications are necessary just to get the application to function.

This paper is organized as follows: Section 2 describes in detail our system setup; Section 3 explores a direct method for accessing accelerators; Section 4 examines a second interface method and a switchable interconnect framework incorporated into this model; Section 5 covers related work; Section 6 describes future work; and Section 7 concludes the paper.

2. THE SYSTEM

2.1 Application Design Models

Four different application development models are considered in the context of the prototype platform: two models familiar to the reconfigurable computing community and two that we propose for moving toward the integration of reconfigurable computing with general purpose applications. Figure 1 shows the abstraction layers for each of the defined models.

The way of approaching software design on contemporary general purpose computing platforms is shown in Figure 1(a). In such a model, the entire application runs on a general purpose processor without acceleration but with operating system support. Figure 1(b) shows one method for developing applications that more fully utilizes the resources of an FPGA with an embedded processor. This hybrid CPU with accelerator model does not allow for standard user and commercial applications to run on the processor. Furthermore, the embedded CPU is frequently relegated to merely data transfer and minimal control tasks, which under-utilizes the power of the general purpose processor.

We explore two alternative methods to fully exploit the capabilities of a hybrid CPU and FPGA system. The first method we call *direct access* and is illustrated in Figure 1(c). Part of the application has been synthesized as accelerators in the reconfigurable fabric. The accelerators are mapped directly to the application, indicated by the tunnel between the part of the application on the CPU and the part on the FPGA. This achieves the low overhead of the model shown in Figure 1(b) while keeping the application in the context of a widely available, conventional operating system. Figure 1(d) illustrates further encapsulation of the reconfigurable resources in our *indirect access* model. In this model, parts of the application are isolated and mapped through well-defined interfaces into the reconfigurable substrate as accelerators that can be accessed in a fashion similar to library calls. The operating system abstracts and protects the accelerator resources by providing indirect access via stubs extending from the different parts of the application into the

Bus	First Access		Pipelined		Arbitration
	Read	Write	Read	Write	
PLB	21	20	3	3	15
OCM	4	3	2	2	–
DCR	3	3	3	3	–

Table 1. Bus Parameters (Processor Cycles)

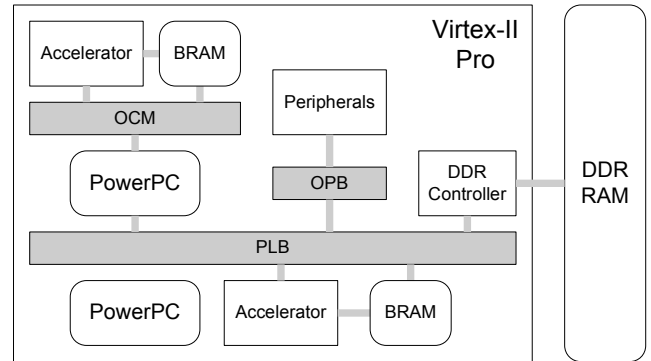


Figure 2. System Figure. Embedded entities are represented with rounded corners and those implemented in the FPGA with square corners.

operating system. Although the direct access method provides less virtualization support than the indirect access method, we were able to perform low level timing optimizations under that method, as is discussed in Section 3.4.1.

The direct and indirect models enable the application to make use of all of the services the operating system offers and to take advantage of the wide array of development, hardware, and software library support available under Linux. This work explores the direct method in Section 3 and the indirect method in Section 4.

2.2 Hardware

We use the *Xilinx University Program* (XUP) development board that uses the VP30 Virtex-II Pro [19] FPGA. As shown in Figure 2, two PowerPC405 CPUs are embedded in the FPGA fabric with 30,816 logic cells. Only one of the processors, running at 300 MHz, is used for this work. There are 136 hardwired 2 KByte blocks of dual-ported memory embedded in the reconfigurable fabric. They are referred to as *Block RAM* (BRAM). The BRAMs are separate from the main memory and are capable of running at processor clock speeds. The platform supports the 64-bit Processor Local Bus (PLB) at 100 MHz, the 32-bit Data Side On-Chip Memory bus (OCM), the On-chip Peripheral Bus (OPB), and 32-bit Device Control Registers (DCR). Not all of the buses are instantiated for all of the test applications. It is important to note that all buses and peripherals are implemented in the FPGA fabric.

Trade-offs abound within the context of the XUP infrastructure (See Table 1). The OCM enables deterministic, low latency access to small amounts of memory realized as BRAMs and is accessed by the processor using 32-bit loads and stores. The OCM requires four cycles for the first load, three cycles for the first store and can complete a load or store every two cycles after the first. Furthermore, the OCM requires that physical and virtual addresses match, placing an additional constraint on the bus. The PLB can access a 32-bit address space allowing for large memories. Furthermore, the PLB provides high throughput transfers. The PLB can make use of *direct memory access* (DMA) with the processor, system memory and accelerator local buffers. The DCRs are a separate I/O facility providing 1024 possible registers synthesized in the FPGA.

The registers are accessed using privileged instructions and take three processor cycles to complete. Their accesses are not routed through the *memory management unit* (MMU) of the PPC405, removing contention issues with other buses. See [12] for an in-depth study of the trade-offs inherent in these buses. The OPB connects peripherals to the system. The OPB is inferior to the PLB in every respect and was not used to interface accelerators.

2.3 Software

The hardware platform runs the GNU/Linux operating system running on one of the PowerPC405 processors. It is based on the official PowerPC port of the Linux 2.4 kernel.

All timing measurements are made using the 64-bit time base facility of the PowerPC [18] to give cycle accurate measurements. Each call to the time base is encapsulated by `sync` instructions to ensure that all pending—and possibly costly—bus transactions have completed prior to taking a time measurement. Inserting synchronization instructions also makes certain that no unrelated instructions are allowed to contribute to the measurement. Our measurements show that accessing the time base and synchronization adds fewer than ten processor cycles (34 ns) to our measurements.

3. DIRECT ACCESS MODEL

In this model of applications running on a conventional operating system accessing reconfigurable accelerators, the accelerator resources are made directly accessible to the applications. The accelerators are accessed with standard load and store instructions. In the direct access model, the operating system sets up and removes the necessary mappings, but is otherwise transparent to the application. The direct access model is based on memory mapping facilities provided by the Linux kernel and are accessible via standard libraries.

3.1 Operating System Interface

The operating system interface in the direct access model consists of a single `open()` system call to access the system's physical address space followed by an `mmap()` system call to map the accelerator's address space into the virtual address space accessible by the application, as illustrated in Figure 1(c). The mapping enables proper virtual-to-physical address translation to access BRAMs, memory-mapped registers or internal memory of the accelerators attached to either the OCM or PLB.

The memory mapping approach allows the application to interact directly with the accelerator using standard load and store instructions to virtual addresses, reducing software complexity. It also reduces the overhead of setting up and accessing the accelerator on a per-call basis.

The region mapped into the application can be accessed by multiple applications if the `mmap()` call is made and the region is not subsequently locked. However, preemption by the operating system places the burden of synchronization on the application. When the mapping is made exclusive by locking, only one application can access the accelerator at a time. If the accelerator is locked, each application holds the accelerator until it releases access to it with the `ummap()` system call or the application exits. If any other application attempts to map the accelerator while another application has it mapped, the operating system will cause the `mmap()` call to fail. Disallowing sharing in this model renders critical sections unnecessary and avoids unnecessary overhead if there is a one-to-one mapping between applications and accelerators.

Memory mapping from an application running on top of a fully-fledged operating system enables consistent interfaces with low overhead to be developed. Application developers can encapsulate a portion of their application as a function and map it to a reconfigurable accelerator. A simple software stub will then be generated to encapsulate the access to the accelerator. The developer replaces the function call in the software application with a call to the stub instead. We present two examples of such access semantics in Section 3.3 and Section 3.4.

3.2 Lessons Learned

The data communication overhead between the CPU and the accelerator is critical to performance when using fine-grained accelerators, ones that require frequent communication with the rest of the application. Polling or an interrupt synchronization mechanism is needed for accelerators with larger granularity, ones that operate on bigger segments of data. Polling using the PLB will lead to a bigger overhead, since each read to the PLB costs up to 21 CPU cycles when there is no contention on the bus and far more should arbitration be necessary. The impact of arbitration can easily erase any gains of fine-grained accelerators in our prototype as each arbitration costs fifteen cycles.

The cost of data transfer and synchronization reads would increase on a system where data is streamed from one accelerator to another using the main system bus. In such an environment, synchronization reads coming from the processor and data transfers from one accelerator to the other would be in contention for the bus. As a result, half of the accesses would incur added latency and could in effect slow down the computation, further underscoring the importance of prudent accelerator-to-interface mapping.

From a practical point of view, developing hardware accelerators requires understanding a set of bus protocols. Our experience shows that much of the development time is devoted to dealing with the interface between the accelerator and the bus. Accelerator development is protracted by the use of multiple buses, optimizing for acceptable performance, and the need to move between differing platforms. If a restricted view of the system—optimized for accelerator interactions—is presented to the accelerator designer, development effort can be reduced greatly.

3.3 Example: Video Encoder

H.263 video encoding [8] achieves high compression ratios by taking advantage of both spatial and temporal redundancy in the signal. Spatial compression is achieved through *discrete cosine transform* (DCT) and quantization, similar to JPEG encoding. Temporal compression is achieved through *motion estimation* (ME). The current frame of the video is divided into *macroblocks*, each of which is a 16 by 16 pixel block. The macroblocks are then compared against the previous frame to find as close a match as possible by calculating the *sum of the absolute differences* (SAD). The differences between the sample with the smallest SAD in the search area and the current macroblock is encoded as a *motion vector* (MV). Since only the changes from frame to frame are encoded, the encoded file size is greatly reduced.

3.3.1 Design

Full-search ME makes up 73% of the total execution time in a software implementation of H.263. Consequently, cheaper search algorithms such as diamond-search are normally employed in software. There are various ME algorithms allowed under the H.263 standard. The primary difference between each algorithm is the number

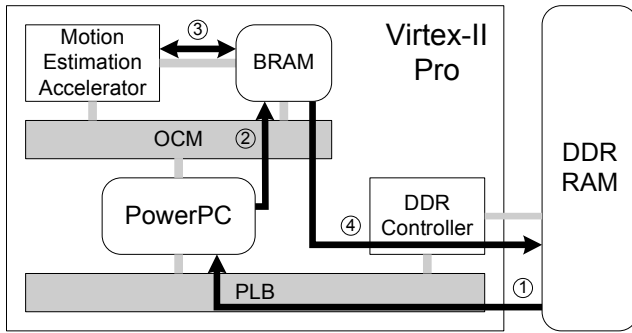


Figure 3. H263 Video Encoder Data Flow

of comparisons against the previous frame performed. Full-search ME is very expensive in software due to the large number of memory accesses, branches and subtractions required for computing and comparing the SAD values. Diamond search is a heuristic that reduces the computational cost of motion estimation by examining only a small subset of the macroblocks in the previous frame. However, this results in inferior encoding quality as the search could be trapped in a local minimum. Our experiments demonstrate a typical increase of 20% in file size for diamond search over full-search ME.

A version of full-search ME is implemented as a hardware accelerator to demonstrate the potential for coarse-grained acceleration. Every macroblock in the current frame is compared against a search area, a 31 by 31 pixel area, in the previous frame. The center of the search area is aligned with the center of the current macroblock. The current macroblock is compared against all possible combinations within the search area, shifting pixel by pixel to search for the best match. This results in 256 comparisons, or SAD computations, per current macroblock/search area pair. Since the SAD computations can be performed in parallel, the accelerator was made with 256 SAD units to realize full parallelization. The software-only version performs all 256 computations sequentially.

The original `MotionEstimation()` function is replaced with a new function to perform data marshaling, data transfers, and setting the ready bit on the accelerator to initiate computation—all of the core computations of ME are now performed on the hardware accelerator. The software polls a pre-defined address and waits until the accelerator is finished. A single 2 KByte block of BRAM configured to be 32 bits wide with 512 entries is instantiated to act as the memory interface between the CPU and accelerator. Architecturally, the BRAM attached to the OCM is mapped into a range within the physical address space of the processor at synthesis. The application then uses the method described in Section 3.1 to obtain direct access to the accelerator memory.

3.3.2 Data Flow

Figure 3 shows the data flow for the H.263 video encoder when using the hardware accelerator:

1. The application on the CPU reads the data needed by the accelerator from main memory. This consists of the macroblock from the current frame (256 bytes) and the search area from the previous frame (961 bytes).
2. The application on the CPU writes the data for the accelerator into the BRAM that has been memory mapped into the application's virtual address space. The application tells the accelerator to begin.

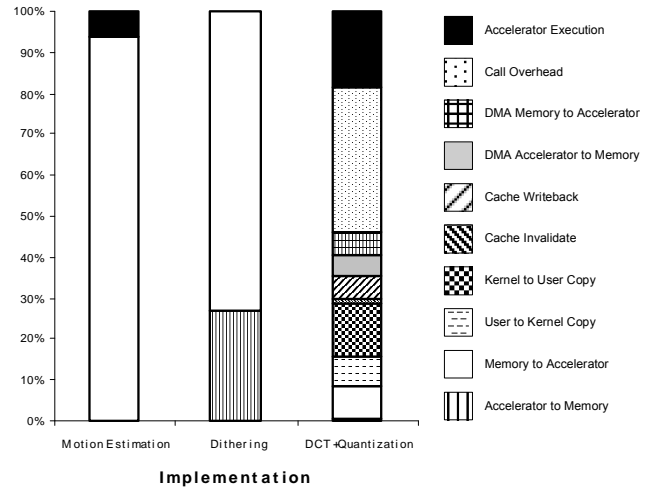


Figure 4. Breakdown of Accelerator Execution and Data Transfer Times

Accelerator Action	Overhead	
	cycles	μ seconds
<code>open</code> System Call	34,000	113.3
<code>mmap</code> System Call	24,000	80.0
Data Marshaling+Write	90,600	302
Initiation	180	0.600
Accelerator Computation	5,400	18
Read From Accelerator	300	1
<hr/>		
Total time per call		
Full Search (HW)	96,480	322
Diamond Search (SW)	174,911	583
Full Search (SW)	639,925	2,133

Table 2. `MotionEstimation()` Transaction Times

3. The accelerator reads and writes to the BRAM while performing its computations.
4. The accelerator signals to the application on the CPU that it is finished with its computations. The data is read from the BRAM and put back into main memory.

3.3.3 Results

We use the time base facility for all measurements, as detailed in Section 2.3. Table 2 shows the breakdown of the time spent using the ME accelerator and how much time ME takes in software. The accelerator implementation of full-search ME speeds up execution by 6.6x. It is even 1.8x faster than diamond search in software. `open()` and `mmap()` are called once at the beginning of the application, thus they are overheads that are easily amortized. A lot of time is spent marshaling and transferring the data from the CPU to the accelerator due to the large amount of data involved. This cost is incurred on every invocation of the accelerator. Figure 4 shows the breakdown in terms of percentages. 93.73% of the time is spent marshaling and transferring the data from main memory to BRAM on the accelerator, 6.06% is spent waiting for the accelerator to complete computation, and 0.21% is spent reading the motion vector and SAD values back to the CPU. However, the ratio of data transferred to and computation performed on the accelerator can be improved. One possibility is to leverage the overlap of the search areas for different macroblocks. Currently the entire search area is sent anew with each macroblock.

3.4 Example: Audio Processing

Madplay [14] is an MP3 player application available on the Linux platform. It decodes an input file and sends the output to the audio hardware, which expects audio samples of 16 or 24 bits. However, the representation of audio samples frequently grows beyond these bit widths while undergoing processing. Consequently, the samples need to be truncated. Unfortunately, truncation introduces error into the signal. *Dithering* before truncation reduces the error by adding random noise to each sample.

3.4.1 Design

The main portion of *madplay* consists of two nested loops. The outer loop reads a block of data from disk and decodes it to produce 1192 samples per channel. There are two channels, left and right. The inner loop processes all of the samples, from both channels, by performing dithering and then truncation.

Dithering takes one input sample and produces an output sample while maintaining previous samples necessary for computation. Dithering can perform better on hardware than software due to the inherent parallelism of its algorithm, where several intermediate data can be produced in parallel. The dithering accelerator consists of two dithering blocks that operate at 25 MHz, each with a 32-bit input register and a 16-bit output register. The two dithering blocks represent the left and right channels in audio. The accelerator interfaces with the processor by means of three memory-mapped registers. The registers consist of two 32-bit input registers and a single 32-bit output register that merges the output from the two internal dithering blocks. The CPU writes to the input registers one at a time and then performs a single read to retrieve the output.

Madplay maps the dithering accelerator into its virtual address space by using the method described in Section 3.1. Each call to the software dithering has been replaced with two standard stores to the input registers and a single read from the output register.

Even though we leveraged the interface support in mapping the dithering accelerator into the virtual address of the application, we were also able to take advantage of certain timing characteristics in the hardware. By taking into account the intricacies of communicating over the PLB bus, we were able to completely overlap the communication between the CPU and the accelerator with the execution of the accelerator itself, provided the execution time is short enough. This also allowed us to avoid introducing synchronizations. Our experiments using the time base show that in the best case scenario, i.e., no arbitration, reads and writes to the accelerator take 21 and 20 CPU cycles, respectively. However, these are the delays as seen by the CPU. The accelerator actually sees the write sooner because there is an acknowledgment from the PLB bus back to the CPU. The accelerator also sees the read some cycles after it is issued by the CPU before the result is taken back. Therefore, there is at least an 18 cycle window between when the accelerator receives the write and the read request even if the two instructions are issued back-to-back. Since the dithering accelerator takes 12 CPU cycles to complete, its execution time is always completely masked by the communication time, even if arbitration lengthens the communication time. As long as the dithering accelerator is not shared, its design ensures that the output remains valid until read by the CPU.

3.4.2 Data Flow

The call to the software dithering routine has been substituted by two stores to the accelerator input registers and one read from the output register. This value is then written into the application output

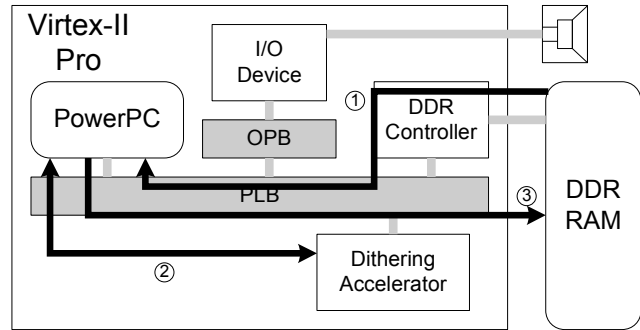


Figure 5. Dithering Data Flow

buffer by the processor. Figure 5 shows the data flow for *madplay* when the hardware accelerator is used:

1. One sample per channel is read from the main memory by the processor.
2. The application writes both samples into the accelerator and gets the result back.
3. The processor does some post-processing and stores the result into the main memory.

3.4.3 Results

On the reference platform, the software dithering takes 50 CPU cycles per channel, whereas the hardware implementation takes 12 CPU cycles, resulting in an achieved speed-up of about 4.1x for dithering execution. However, there is a lot of data movement involved in using the accelerator, as indicated by steps (1) through (3) in Section 3.4.2. If the data movement is taken into consideration, the achieved speed-up becomes 1.42x for dithering.

The number of CPU cycles spent on each of the previously described stages has been measured using the time base register of the PowerPC as described in Section 2.3. Figure 4 shows these results as a percentage of the total time for a single call to the dithering accelerator. The computation time in the accelerator does not appear in the figure since it happens in parallel with the data transfer. Each input is processed while the acknowledgment for the store is sent back to the processor. The communication time from the CPU to the accelerator takes 82% of the total time, whereas getting the input from the accelerator is 18%.

4. INDIRECT ACCESS MODEL

We have implemented a method for interfacing reconfigurable accelerators with embedded processors in a consistent and easily debuggable fashion using the indirect access model as depicted in Figure 1(d). Currently, every time an accelerator is implemented on a new platform, the system designer must learn the intricacies of a bus. Moreover, the designer must be concerned with application integration and debugging in a heterogeneous system. Doing so requires both extensive software development and hardware design skills. Accelerator integration is further exacerbated by unavailable, or inaccurate models of the system. Accelerators interacting with the operating system and user applications raises the complexity that must be navigated by the system designer. In an effort to reduce the burden on application developers and accelerator designers we have implemented a switchable interconnect interfaced with user applications using the indirect access model developed. Figure 6 depicts a high-level view of our system. We give JPEG compression running on our FPGA-based platform as an example.

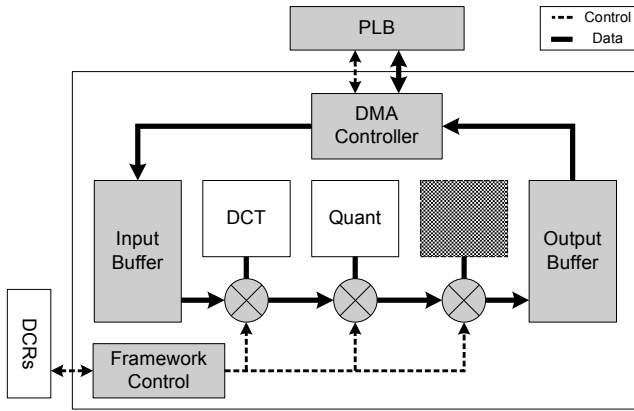


Figure 6. Switchable Interconnect Accelerator Framework

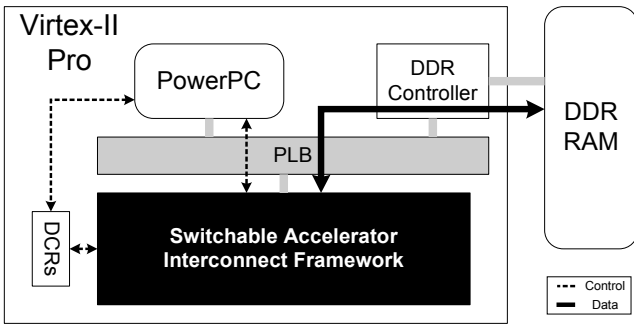


Figure 7. JPEG Encoder

4.1 Operating System Interface

Accessing the hardware directly enables the development of an efficient access model without the recurring overhead incurred by adding an abstraction layer. However, there are many caveats to such a design. Direct mapping of device resources into an application can limit concurrency. In such a model only a single application can access the device, blocking access to it for an indeterminate amount of time. Moreover, the nature of direct mapping does not provide a mechanism to protect user applications from each other nor the system from the applications. The indirect access model forces each call to the accelerator to go through the operating system providing a layer of protection. Furthermore, such an access model allows for more fine-grained resource management.

The accelerators are interfaced with the application through a Linux character device driver. A device driver interface is necessary for DMA and access to protected resources such as the DCRs on our platform. The application obtains access and resets the accelerator by first issuing an `open()` system call. The call to the hardware accelerated function in the software-only version is replaced by an `ioctl()` system call. When the application no longer requires the accelerator, the application calls the driver implementation of `close()` to free the accelerator. Concurrent access to the device is controlled at the granularity of an `ioctl()` call, i.e., multiple applications can have the accelerator open concurrently and will block when attempting to access the busy accelerator.

When the application makes an `ioctl()` call, it includes a user space pointer to an array of independent, related data values that are processed by the accelerator, or *blocks*, and the number of blocks in the array. The device driver then processes the blocks, executing

them on the accelerator. The call returns control to the calling application upon completion of the last block. Processing blocks inside the system call begins by copying the data of a single block from the user array into a statically allocated DMA-capable buffer inside the driver. The buffer is then explicitly flushed from the cache to ensure coherence with the system memory via the PLB. DMA control registers are set up for the transfer and the data movement is completed. Once all of the data is moved from the buffer in system memory via the DMA controller into the local input buffer of the accelerator, the driver starts the execution of the accelerator. DCRs are used by the accelerators to communicate control and status information. The driver polls on a DCR, waiting for the accelerator to complete. When the accelerator is finished, the data is placed into an output buffer and the driver is allowed to proceed. The driver invalidates entries in the cache to ensure coherence before setting up a DMA transfer from the accelerator input buffer to the system memory. The result data is then copied from the DMA buffer to the user space buffer that originally contained the block. The accelerator continues to process blocks of data in this manner and when all have completed, the system call returns to the application.

4.2 Accelerator Interconnect Framework

To provide interfaces for accelerator developers, we present a switchable accelerator interconnect network, a framework providing the components necessary for mapping the data flow of accelerators into a simple interface. When combining our framework with the indirect method of accelerator access, our switchable interconnect network provides interface consistency and simplifies development. In contrast, by directly mapping hardware resources, every new application must develop a hardware interface, breaking compatibility and exacerbating the already difficult task of debugging in a heterogeneous environment. Moreover, redeveloping the interconnections in the reconfigurable logic for each accelerator fails to take advantage of the commonality. Parts of the design that do not change between accelerators (e.g., the switchable interconnect presented here) can be implemented in faster, denser embedded logic in the FPGA as multipliers and BRAMs are today.

The development platform as described in Section 2.2 allows for a wide range of choices regarding how accelerators are interfaced with the system. Due to the medium-sized data transfer bursts possible with our examples the PLB is chosen as the system interface for the accelerators as shown in Figure 7. By placing the accelerators on the PLB, burst-oriented DMA transfers are enabled between system memory and the accelerators' input and output buffers as depicted in Figure 6. In future systems, the abstraction provided by the framework will allow designers to explore different buses without constantly rebuilding their accelerators. Control signals are exchanged with the accelerator framework using Device Control Registers which are independent of the PLB bus. Choosing DCRs avoids contention for the bus between the processor and the accelerator control registers as would be the case if memory-mapped registers were used.

The accelerator input and output buffers are instantiated as two fixed-logic BRAMs constituting independent banks for input and output. Data movement of blocks between system memory and accelerator input and output buffers is handled via DMA transfers. Providing larger internal buffers, added accelerator control, and exposing more parallelism in the application would allow for multiple blocks to be sent per transfer, amortizing the startup cost (Table 3 lines 1, 2, and 5) of a DMA transaction across more execution on the accelerator framework.

All components of the accelerator interconnect framework have been implemented as library components available to accelerator

Action	Overhead	
	cycles	μ seconds
System Call Overhead	1,853	6.18
DMA Setup	549	1.83
DMA Transfers	448	1.49
Accelerator Execution	987	3.29
Cache Coherence	348	1.16
Data Copies	1060	3.53
Total Time	5,244	17.5

Table 3. JPEG System Call Breakdown for a Single Macroblock

design. Each accelerator is connected to the switchable network via a set of well-defined interfaces for moving data into and out of the individual accelerators. The goal is to provide a reduced complexity interface that removes the burden of bus interfacing, DMA transfers, and flow control between multiple accelerators working in concert. Furthermore, providing a constrained view of the system enables future technologies (i.e., compilers) to more easily map portions of a software application into reconfigurable accelerators running on the switchable interconnect network.

We define a *reconfigurable frame* as the predefined area of reconfigurable fabric that can be configured as a single accelerator entity. Each frame consists of a set of input and output signals that include: a data bus in each direction, an address bus identifying the current value entering or exiting the accelerator, handshaking to convey to the subsequent accelerator that data is available, and back-pressure assertion when the current accelerator must stall. The use of handshaking, an asynchronous interface, allows for different stages to take variable amounts of time and allows for the removal of centralized control logic from the accelerator framework. All data in the network flows in one direction from input to output. Besides providing a consistent interface so that accelerators can be plugged into and out of the switchable network easily, the onerous task of debugging is simplified. The accelerator developer need only be cognizant of the reduced complexity interface to the switchable network and not with the timing of memory movement and application interfacing.

A long term goal of this project is to enable dynamic reconfiguration of accelerators to match the changing needs of the applications running on the system. An opportunity to adapt the hardware accelerators on-the-fly is partial runtime reconfiguration [4]. However, partial runtime reconfiguration is a slow process compared to accelerator execution and operating system context switch times. Having a switchable network allows for fast switching between resident accelerators. An example is one user encoding a JPEG image while another is compressing an audio file. As the system shares the processor resources between the two applications, it may also be advantageous to share reconfigurable resources. By having both accelerators for JPEG and for dithering resident in the switchable interconnect framework, the operating system can time multiplex the accelerator resources with the granularity of a few cycles. If runtime reconfiguration were used as the only means of sharing the reconfigurable resources, many microseconds would be wasted during accelerator swapping. A further benefit of a switchable network for interconnecting reconfigurable frames is the electrical isolation needed to enable runtime reconfiguration.

4.3 Example: JPEG Encoder

JPEG image compression has been accelerated by instrumenting discrete cosine transform (DCT) and quantization in the FPGA fabric using the indirect mapping operating system interface and

the switchable interconnect framework. JPEG image compression provides numerous opportunities for acceleration in FPGA-based logic due to its inherent medium-grained and coarse-grained parallelism. The conversion of an uncompressed image to a JPEG compressed image is performed on 8x8 pixel blocks called *macroblocks*. The original image is partitioned into macroblocks that can be processed in parallel. Each macroblock is converted from RGB color to YUV color, representing luminance and two chrominance channels. A 2-dimensional DCT is then performed on each of the macroblocks for each of the channels. Quantization is then applied to the transformed macroblock. All the macroblocks are then sequentially compressed in two steps: run-length encoding (RLE) and Huffman coding yielding the final, compressed image. As motivating examples, we have chosen DCT and quantization as components of JPEG to accelerate using the platform described in Section 2.2.

4.3.1 Design

We have chosen to accelerate the Independent JPEG Group’s *libjpeg* implementation of a JPEG compression utility—*cjpeg* [7]. In the context of *cjpeg*, DCT and quantization serve as examples of medium-grained to coarse-grained accelerators. The minimum block size of a transaction with either accelerator is the 64 elements constituting a single macroblock. Each element is composed of two bytes resulting in a minimum transfer to or from the accelerator of 128 bytes of data. An additional motive for choosing DCT and quantization to implement as accelerators is their data flow relationship—each macroblock must first be transformed and then quantized. In *cjpeg* on a 1.4 megabyte test image using integer-only arithmetic and default quality setting, 27 percent of the processing time was found to be spent in quantization and DCT. If both DCT and quantize are implemented in hardware and connected, the results of DCT can be pipelined directly into the quantizer. Connecting two accelerators end-to-end saves a costly bus transfer further increasing performance.

To illustrate the switchable interconnect network, an example of JPEG using separate DCT and quantizer accelerators is implemented and integrated with the framework as shown in Figure 6. A macroblock is computed within the JPEG encoder software and is available in a 64-entry array of 16-bit values. The application delivers this block to the accelerator through the operating system interface as a pointer to the user level buffer. The call to the kernel replaces the original DCT function call. A system call is then made, moving the data from the system memory to the accelerator input buffer via a DMA transaction. The only changes made to the original software version of *cjpeg* are that the loop performing quantization is removed and the call to a software DCT routine is replaced by an `ioctl()` system call to our driver interface.

4.3.2 Data Flow

The data flow through the accelerators begins when the driver:

1. sets a control register with the configuration of the interconnects for the given block,
2. copies the input user array into a system DMA buffer and invokes DMA to transfer the input data to the accelerator input buffer and
3. sets another register signaling that the accelerators a macroblock is available in the input buffer.

A simple state machine that is internal to the active accelerator starts the flow of data by communicating with the local storage (i.e., the input buffer) and obeying the interface protocol. The accelerators were developed using the Xilinx DCT and divider cores that are fully pipelined accepting one sample (of the 64 total) each cycle. The accelerators run at the PLB clock frequency of 100 MHz to avoid crossing clock domains and added place and route effort but are capable of much higher speeds. The DCT and quantizer are fully pipelined but have many cycles of latency between final input and initial output. Handshaking occurs between the quantizer and the DCT to synchronize the hand off of coefficients to be quantized when new data is available from DCT. The quantizer writes the results in row-major order into the output buffer. When all 64 results are complete, the accelerator toggles a bit in the control register that the driver is polling. The driver then sets up the DMA transfer back to system memory, completing the transaction with the accelerator. The driver returns to the application. The application now has the transformed and quantized data available in the same buffer passed to the system. Application execution continues unaltered.

4.3.3 Results

In the original software version of *cjpeg*, DCT and quantization take 8,310 processor cycles per macroblock. Table 3 shows the time to complete DCT and quantization using the hardware accelerator. In hardware, DCT and quantization take 987 cycles, yielding a speed-up of 8.42x. When the overheads of the indirect access method and switchable interconnect framework (Figure 4) are taken into account, the speed up is 1.58x for a single DCT/ quantization call.

When the indirect access method is used with the switchable interconnect network, there is a fixed cost for making the system call that limits potential speed-up. A lighter-weight system call could be implemented to reduce the overhead from 35% of the call but would still be of the same order of magnitude and thus limiting. However, the remaining contributors to call overhead can be reduced in an effort to achieve better performance. Furthermore, implementing multiple pipelines in hardware and exposing more parallelism to the accelerator further mitigates the cost of the call overhead.

The cost of setting up DMA transfers constitutes 10% of the call time. The per block cost could be reduced by increasing the size of the data blocks sent to the accelerator. In the case of JPEG this would mean sending an array of macroblocks as opposed to processing a single macroblock with each call. Further alterations to *cjpeg* would be necessary to expose the parallelism required to send a large number of macroblocks to the accelerator during each system call.

Transferring a macroblock from the system memory to the input buffer and back to the system memory from the output buffer of the accelerator takes 9% of the call time. The data transfer time can be overlapped with processor execution further increasing concurrency, as DMA does not require CPU supervision once initiated. The data copying needed to move data into DMA enabled regions and to guaranty protection can be avoided by using user space DMA buffers. The application data arrays that hold the accelerator input can be marked uncachable. If the processor is not going to make use of the input data again soon (or in the case of JPEG, never again), marking the region uncachable will remove the 20% of the overhead attributable to coherence and redistribute it in the form of word transactions across the PLB instead of possibly more efficient cache line flushes.

5. RELATED WORK

Previous hybrid platforms have tightly incorporated reconfigurable logic with a microprocessor [2, 6, 15]. The tight integration comes from augmenting the instruction set architecture (ISA) and microarchitecture of the processor to access the accelerators via special purpose instructions inserted into the application. The model presented here takes a less tightly-coupled approach and does not seek to alter existing ISAs. By adopting the DLL model and not altering the ISA, we retain the portability of the applications across the general purpose processor family.

Data movement is the critical bottleneck of many accelerators and we have incorporated the following ideas into our platform. In [9], the authors investigate tight integration of accelerators while emphasizing the need for fast, coherent memory interfaces. Recent work using the same hardware platform has investigated various bus interfaces available on the Virtex-II Pro using a single motivating example [12]. In [12], hardware interfaces in the context of free-standing applications and the trade-offs involved in attaching hardware accelerators to the system buses are investigated. The system does not have operating system support nor can it run standard applications.

A more radical approach to system integration and computability is the streaming computation model [3]. By providing a consistent interface, the developers no longer need to worry about being tied to a particular platform. Larger degrees of abstraction are provided to the hardware and software developers, expediting development in a heterogeneous system. Operating system level interfaces and integration into software applications such that they remain compatible is not a priority of the streaming computation work to date.

Studies integrating hardware accelerators into the operating system and standard applications are less mature. Past efforts have investigated scheduling and resource sharing for reconfigurable platforms by application [5]. However, the interfaces utilized by user applications are not discussed. There are several proposals to extend thread abstraction to expose hardware accelerators to user applications [1, 11, 16, 17]. These systems impose an underlying programming model where one or more software threads are implemented into the logic (hardware thread). In order for the software and the logic to communicate with each other, synchronization mechanisms such as semaphores have to be implemented into the reconfigurable logic [10]. The interface between the accelerators and the operating system is a piece of logic commonly called the *hardware thread interface* that is implemented by each accelerator. Supporting additional features, such as message passing, into hardware threads adds more complexity to both hardware and software threads [13]. These works may become necessary as we explore interfacing accelerators into multi-threaded applications.

6. FUTURE WORK

This work represents the beginning of a two-pronged approach to developing future heterogeneous reconfigurable platforms with efforts focused on operating system interfaces and novel reconfigurable platforms. The first major thrust of future work is operating system interfaces advances for reconfigurable accelerators. Our second line of future research is to develop models that reflect advances in system interfaces and computer microarchitectures for use in our framework and in our interface models.

Protection boundaries must be maintained. Users need protection from each other and the system from malicious or faulty accelerators. Operating system interfaces and the development of our framework are key to investigating the protection model necessary

for integrating reconfigurable accelerators into contemporary systems.

We are developing operating system interfaces that allow for *virtualization* of the reconfigurable resources by providing both hardware and software implementations of the accelerator. Virtualization allows the operating system, guided by system policies, to share reconfigurable resources efficiently between processes while providing a consistent interface to the user.

Going forward we must investigate the trade-offs possible when integrating our accelerator framework with contemporary processors. Synthesized FPGA logic provides a rich prototyping platform, but routing the signal necessary for a reconfigurable interconnect network represents a challenge. We envision an accelerator network, bus interfaces, and protection mechanisms built into the processor/reconfigurable logic hybrid CPU. We connect a small number of accelerators, but for future systems where many accelerators are available, a more rich set of interconnections will be needed. To that end, appropriate topologies for future reconfigurable accelerator networks is a focus of our future research.

Finally, the new model of accessing reconfigurable hardware accelerators through a library call interface requires a method for software developer to target reconfigurable platforms. The ultimate goal is to automatically identify sections of code suitable for acceleration at compile time either through profile data, code analysis, or programmer notation.

7. CONCLUSION

In this work two operating system interface models are developed for reconfigurable accelerators. We use these interfaces for applications running on the embedded PowerPC processor of a Virtex-II Pro FPGA. The direct access model provides only the support needed to expose the accelerator directly to the application. The indirect access model further encapsulates the accelerators, providing greater operating system control at the expense of added overhead. A reconfigurable interconnect network has also been presented with the goals of easing development and debugging of applications accelerated with reconfigurable hardware. The framework provides a consistent interface, protection mechanisms, and reduced debugging complexity for both applications developers and hardware designers. We have provided three applications capable of being run on conventional computing systems and instrumented them within the context of our prototype platform.

As more die area becomes available to processor designers and the performance limits of power hungry, out of order superscalar microarchitectures are reached, parallelism must be exploited to achieve future performance gains. Simply adding more cores to a chip has been one venue explored in commercial designs, but such a philosophy does not immediately yield increased performance. Integrating hardware accelerators for heavily used sections of code can provide speed-up without the added complexity, cost and increased power budget of added cores. This work is an effort to motivate and enable fine-grained reconfigurable hardware accelerators to be integrated into a broader range of applications by utilizing a reconfigurable interconnect network and well-defined hardware/software interfaces.

The OS interface implementation is available from the UIUC GSRC Soft System Research web site at:
<http://www.crhc.uiuc.edu/IMPACT/gsrc>

8. ACKNOWLEDGMENTS

This work was supported in part by the MARCO/DARPA Gigascale Systems Research Center (<http://www.gigascale.org>). Their

support is gratefully acknowledged. We would also like to thank Xilinx and Intel for their donation of the hardware and their support.

The UPC authors have also been partially supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIC 2001-0995-C02-01 and the HiPEAC European Network of Excellence.

References

- [1] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. *IEEE Micro*, 24(4):42–53, July 2004.
- [2] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, 2000.
- [3] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *FPL*, pages 605–614, 2000.
- [4] A. Donlin. New tools for FPGA Dynamic Reconfiguration. The Future of Configurable Hardware Symposium, Dec. 2005.
- [5] W. Fu and K. Compton. An execution environment for reconfigurable computing. *FCCM*, 00:149–158, 2005.
- [6] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, 1997.
- [7] Independent JPEG Group. coderules.doc. Text file in zipped archive, 1998. <ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz>.
- [8] International Telecommunication Union. H.263. Technical report, Underbit Technologies, <http://trace.eas.asu.edu/tools/H.263.pdf>, 2005.
- [9] 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*, pages 145–154, New York, NY, USA, 1999. ACM Press.
- [10] R. Jidin, D. Andrews, and D. Niehaus. Implementing Multi Threaded System Support for Hybrid FPGA/CPU Computational Components. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, June 2004. CSREA Press.
- [11] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, Apr. 2003. IEEE Computer Society.
- [12] J. Noseworthy and M. Leiser. Efficient use of communications between an FPGA's embedded processor and its reconfigurable logic. In *FPGA*, page 233, 2006.
- [13] C. Steiger, H. Walder, and M. Platzer. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov. 2004.
- [14] Underbit Technologies. MAD: Mpeg audio decoder. Technical report, Underbit Technologies, <http://www.underbit.com/products/mad>, 2006.

- [15] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.
- [16] H. Walder and M. Platzer. A Runtime Environment for Reconfigurable Hardware Operating Systems. In *Proc. of Field Programmable Logic*, pages 831–835, Leuven, Belgium, Aug. 2004. Springer.
- [17] G. B. Wigley and D. A. Kearney. The First Real Operating System for Reconfigurable Computing. In *Proc. of the 6th Australian Computer Science Week (ACSAC)*, Gold Coast, Australia, Jan. 2001. IEEE Press.
- [18] Xilinx. *PowerPC 405 Processor Block Reference Guide*. Xilinx Inc., San Jose, USA, 2005.
- [19] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet, October 2005.