

# HybridOS: Runtime Support for Reconfigurable Accelerators

John H. Kelm  
Dept. of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801  
jkelm2@crhc.uiuc.edu

Steven S. Lumetta  
Dept. of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801  
lumetta@uiuc.edu

## ABSTRACT

We present HybridOS, a set of operating system extensions for supporting fine-grained reconfigurable accelerators integrated with general-purpose computing platforms. HybridOS specifically targets the application integration, data movement and communication overheads for a CPU/accelerator model when running a commodity operating system. HybridOS provides a simple API for applications and a well-defined hardware interface for reconfigurable accelerators. The goal is to reduce the difficulty in mapping applications into a CPU/accelerator model compared to an unrestrained FPGA platform while achieving whole-application speedups.

HybridOS is integrated into a full Linux distribution running on the embedded processor of an FPGA. Application-specific accelerators are implemented in the reconfigurable fabric of the FPGA that are allocated to user applications running on Linux. We have developed and evaluated four methods for accessing the data buffers required by hardware-accelerated applications using our prototype. The results of our work show the feasibility of our system for a case study, JPEG encoding with two accelerators, and an evaluation of HybridOS for varying data movement requirements that can be used as a guide for future applications developers.

## Categories and Subject Descriptors

D.4.7 [Software]: Operating Systems—*Organization and Design*

## General Terms

Design

## Keywords

CPU/Accelerator Architecture, Operating System

## 1. INTRODUCTION

We present HybridOS, a set of CPU/accelerator operating system (OS) extensions that address the need for high-performance computing platforms which can better exploit reconfigurable accelerators by providing consistent interfaces to applications developers.

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.

FPGA'08, February 24–26, 2008, Monterey, California, USA.  
Copyright 2008 ACM 978-1-59593-934/08/02 ...\$5.00.

Furthermore, we develop and characterize data access methods, the core of HybridOS, for an architecture model integrating reconfigurable accelerators into common applications. We separate the data access mechanism from the calling mechanism to provide consistency while also allowing for flexibility of implementation. Our platform supports a commodity desktop- and server-class OS. HybridOS enables common software methodologies to be employed, easing applications developers' transition to CPU/accelerator platforms. The end goal is to provide a framework for applications developers to target multi-user reconfigurable platforms and have appropriate data transfer mechanism assigned dynamically.

The CPU/accelerator model need not replace existing parallel programming models, but instead is a complementary technique for exploiting data parallelism present in applications. Our model is complementary to threaded models which aim to separate code into explicitly parallel tasks, exploiting control independence among flows of execution. The granularity of parallelism utilized in threaded models is not the granularity of computation that will necessarily be mapped into accelerators. Instead, we envision multiple data-parallel kernels mapped into accelerators to be accessed by multiple threads of execution in an interleaved and space-multiplexed fashion. However, before hybrid CPU/accelerator models can gain wide acceptance, protocols and interfaces must be developed to ease the integration of accelerators with current programming models. Furthermore, the mechanisms must provide protection between applications and accelerators, while also protecting the system from defective or malicious accelerators. HybridOS achieves these goals by enabling common tool flows and programming models to be maintained, while extending a widely-used operating system and computing environment.

We develop a CPU/FPGA prototype that runs the Linux OS to study data access methods in our CPU/accelerator model. We build the necessary support into the OS to allow accelerators to be accessed by user space applications by way of a library call-like interface. The application developer uses an application programming interface (API) that abstracts away the details of the hardware implementation. Different modes of communication are implemented to transfer data from the application running on the general-purpose processor to the accelerators. We refer to the different models for data transfer as *accelerator access methods*. HybridOS creates the abstraction between application and accelerator such that many different modes of operation are possible, while maintaining the same application interface used to access the accelerator data. We refer to the encapsulation of the access method and of the computation method used by the OS as *accelerator virtualization*. The virtualization of the accelerator resources allows applications to access the accelerators without the need to understand the structure of the underlying resources.

We evaluate HybridOS using an accelerator framework that we have developed. The framework includes embedded memories, bus interfaces, control logic, and a switchable interconnect for accelerator developers to target. The embedded logic of our framework would be more efficient than a general reconfigurable platform in terms of area, cost, speed, and power. Our framework reduces the mapping complexity experienced by accelerator developers compared to an unrestricted platform at the expense of flexibility. Targeting our accelerator framework allows accelerator developers to take advantage of the software interfaces we develop as part of the HybridOS platform. As a result, the accelerator framework enhances efficiency and performance, while reducing the complexity experienced by designers.

As a case study in accelerator development, and to demonstrate our CPU/FPGA platform, we develop a set of accelerators for JPEG image encoding. We have characterized our different access methods to provide the needed insight for data transfer mechanisms to adapt to changing conditions within the system. For each access method, our results show the cost of each component of the overhead associated with accessing accelerators. The results demonstrate the feasibility of our approach, the need for adaptive data transfer mappings between applications and associated accelerators, and directions that future research can pursue to advance the state of CPU/accelerator platforms.

The contribution of this paper is the demonstration and evaluation of HybridOS, protected access methods constituting a set of OS extensions, targeting a CPU/accelerator model. We extend previous work in operating system support for reconfigurable systems by adding data mapping techniques that may be incorporated into the various reconfigurable resource allocation techniques previously studied. The entire HybridOS platform provides software and hardware interfaces, decoupling two dissimilar design processes. The model allows for better use of the reconfigurable resources by allowing the OS to map data access methods to applications dynamically in response to varying workloads. The mapping is made possible by the virtualization provided by the methods we develop and characterize. We have developed an accelerator framework that exploits the commonalities of many accelerators to provide consistent, easy-to-use interfaces, while taking advantage of the benefits of embedded logic.

The goal of HybridOS is to understand the overheads associated with data access in a protected, OS-supported CPU/accelerator model allowing for better performing accelerated applications, thereby gaining wider acceptance for reconfigurable accelerators on general-purpose platforms.

## 2. SYSTEM MODEL

In this section, we describe the interfaces that both applications and accelerator developers use to target applications at our platform. The interfaces decouple and simplify the process of integrating reconfigurable accelerators into common applications. We describe the library-like interface that encapsulates the applications' interactions with HybridOS. We introduce the reconfigurable accelerator framework to simplify accelerator/processor integration. We conclude the section by discussing the implementation of the interfaces and framework on our CPU/FPGA prototype platform.

### 2.1 Application Interfaces

Applications access accelerators in our model through a well-defined API managed by the OS layer. The API is a set of calls to a shared library subsystem that uses both user-space and kernel-space components to transfer data and communicate control information to accelerators resident in the accelerator framework. The

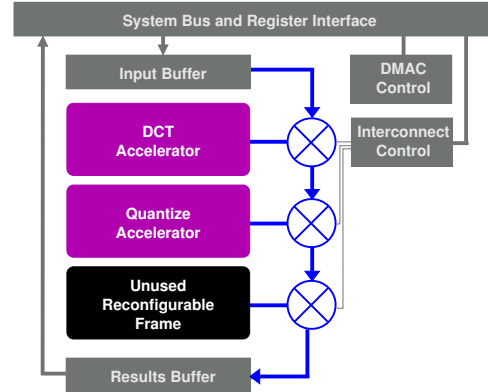


Figure 1: Block diagram of the accelerator framework.

HybridOS platform leverages software developers' familiarity with library and system call use and the application support of a widely-used client- and server-class OS.

The application interface of HybridOS enables the OS layer to perform dynamic resource allocation and virtualize accelerator resources transparently to the user application. When accelerators are directly accessed by an application the ability to reallocate accelerator resources and well-defined portable interfaces are lost, making accelerators' use in a preemptive multi-tasking OS environment difficult. Dispensing with an OS to avoid the overhead of accelerator access has the drawbacks of system services being lost, possible loss of protection boundaries, and lost abstraction. The loss of system services results in increased difficulty for application development and debugging. HybridOS is an attempt to avoid these difficulties.

Applications built for HybridOS can continue to operate in a multiprocessing environment, while exploiting the data-parallel processing capabilities of hardware accelerators. The scheduling and resource sharing capabilities provided by the OS, critical for multiprocessing environments, will be lost if no OS is used. Making the operating system aware of the accelerators allows the accelerator resources to be allocated dynamically to applications, while still allowing for the system services and multiprocessing capabilities necessary for a CPU/accelerator model to be adopted for general-purpose computing platforms. HybridOS remains agnostic to scheduling policy and defining such a policy is not the purpose of this work.

Virtualization of accelerator resources enables compatibility and resource sharing. The virtualization of the reconfigurable resources is similar to the virtualization of physical memory whereby every process believes that it has full access to the physical resources, but in reality the processes operate using an abstraction of the physical resources. For HybridOS, we use in-memory buffers to virtualize accelerator memory and do copying as necessary, transparently to the application and accelerator. In this environment memory resources are allocated by virtual memory mechanisms supported by the OS and processor architecture. As physical can be virtualized, so too can hardware accelerator resources. Adding support to the architecture and to the OS allows accelerators to be virtualized, providing low-overhead access and a simplified programming model for developers.

### 2.2 Accelerator Interfaces

We demonstrate a reconfigurable hardware accelerator frame-

work for accelerator developers to target. A block diagram of the framework is depicted in Figure 1. The accelerator framework simplifies the process of accelerator development by reducing the number of signals a designer must model on our platform by nearly 80%. The reduction is achieved by defining a less general interface than the unrestricted bus interface attached to the embedded processor core of our design. The defined interface allows for accelerators to be chained together and accessed through a defined interface from the general-purpose processor. The functions provided by the framework would be implemented as embedded, hardwired logic in future CPU/accelerator systems, taking advantage of the density, power efficiency, and speed of fixed logic. Furthermore, the framework interface provides a consistent platform that enables standard accelerator libraries to be built. As a part of achieving the goal of reduced complexity when developing CPU/accelerator systems, we use the framework as a target for the HybridOS platform.

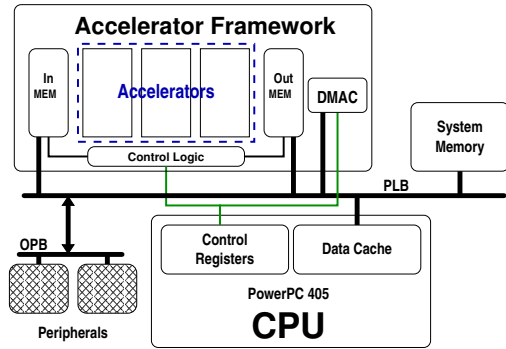
The system interface of the accelerator framework is a set of control registers and data buffers. Control registers for the DMA controller and registers to start and stop the accelerators are accessible to the OS layer. A single control register is used for setting up the interconnect network that chains together the separate accelerators and the processor-accessible buffers. Control registers can also be mapped into individual accelerators to define various modes of operation. The data buffers are addressable by the processor and do not overlap with the address space of system memory. The control registers and buffers provide a set of primitives, supported by HybridOS, for accelerator developers to build upon.

Accelerators can be made resident inside an area of reconfigurable fabric which we refer to as a *reconfigurable frame*. The reconfigurable frame has the input and output stubs necessary for attaching an accelerator to the accelerator framework. The inter-accelerator model that the accelerator framework employs is stream-like and uses a simple accelerator-to-accelerator protocol, with precise semantics, that allows for streams to pass through a set of accelerators. Details of the inter-accelerator protocol and access model are available in [13]. A key feature of the model is that the accelerator developers need not be concerned with what system interfaces are available and the protocols that govern their usage. Accelerator developers who use our model need only target a simple stream-like accelerator interface and applications developers can take advantage of our platform, OS support, embedded framework resources, and a simplified programming model.

HybridOS arbitrates access to and partitions the resources of the framework by setting up address mappings and manipulating control registers transparently to the user application. The consistent but general interfaces provided by the accelerator framework allows HybridOS to target a broad range of accelerators. Moreover, HybridOS is able to interface applications run on a general-purpose CPU with a set of accelerators resident in the framework in a transparent, efficient, and well-supported manner.

### 2.3 Implementation Details

We have implemented our CPU/accelerator model on a CPU/FPGA prototype platform to show its feasibility and to explore the design space of our model. Figure 2 depicts our system implemented in a Xilinx Virtex-II Pro FPGA [25]. The design uses a single 32-bit PowerPC405 [24] processor, running at 300 MHz, supporting the Linux 2.6.18 Kernel. The system memory of the platform is external DRAM attached to a synthesized memory controller. The accelerator framework is developed in VHDL and is attached to the 64-bit processor local bus (PLB), both of which are clocked at 100 MHz. Control information is transmitted to the accelerator framework using a set of 32-bit device control registers



**Figure 2: Block diagram of the CPU/accelerator platform. Note that the system memory is located off chip.**

(DCR), separate from the PLB to allow for concurrent data and control communication. Peripherals are supported by an on-chip peripheral bus (OPB) that is bridged to the PLB.

The synthesized VHDL of the accelerator framework maps into 1638 slices of the Virtex-II FPGA. The framework occupies 1735 flip-flops, 3000 4-LUTs, and nine 2 kB block RAMs. The accelerator framework is not highly optimized and is mainly used in this work as a prototype.

The platform is capable of running standard Linux applications that are compiled for the PowerPC. The Xilinx CPU/FPGA system running Linux serves as the basis for the models and platform developed in this work.

## 3. HYBRIDOS ACCESS METHODS

In this section we describe four access methods for applications to transfer data to accelerators that we have developed. Each method provides the same interface to the application, allowing for transparent switching between access methods by the runtime. We describe two methods based on direct memory access (DMA): *user-space buffers* with low startup costs and *user mapped DMA* with added startup cost, but reduced per-access cost. We will conclude by examining our *cacheable direct mapping* and *uncacheable direct mapping* access methods that can provide efficient bus usage and reduced copying by mapping accelerator local memories directly into user-space applications and libraries.

### 3.1 Example: Character Device Driver Access

As a baseline for comparing the four access methods and to illustrate how it is possible to integrate reconfigurable accelerators using existing software support, we have developed a simple Linux device driver. The accelerator-as-a-device model has been used by commercial products [16]. However, as we will show, the model has deficiencies when applied to closely-coupled, fine-grained reconfigurable accelerators intended for use with a multitasking operating system. All methods we present follow the same pattern of executing a setup routine, writing data to a (possibly virtualized) buffer and then calling the HybridOS runtime, and in doing so, transfer control of the buffer to the runtime system allowing HybridOS to initiate the necessary actions to run the accelerator. In a blocking model, as we will assume here for simplicity, the call returns when the accelerator result is ready. The application can read and write the buffer as needed except when blocked in a call.

The device driver we have developed is accessed via the `ioctl()` system call. Once the application has set up the device interface by calling the `open()` system call, it can access the accelerator by



**Table 1: Mapping overhead for user access methods.**

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

calls from a user application running on our test platform. In the character device driver example, both `open()` and `close()` system calls need to be issued, requiring as many as 44,000 cycles. If setup were required by all of our access methods, the high startup costs would reduce the applicability of our interfaces to only those applications that access the accelerators numerous times or use large data sets. Instead, we provide a variety of methods that take advantage of special mappings using setup routines when setup costs can be tolerated, but also user-space buffers that allow for accelerator access without the high cost of setup to enable closely-coupled fine-grained accelerators.

### 3.2.2 User Mapped DMA Buffers

User mapped DMA buffers avoid the additional copies associated with the user space buffer method by mapping a DMA-capable buffer directly into the application accessing the accelerators. DMA transfers off-load the task of data movement to the DMA controller, providing an opportunity for increased concurrency between accelerators and the general-purpose processor. The main tradeoffs for reduced overhead are the cost of setting up the mapping and the need to pin DMA-capable buffers into system memory for each user mapping. For each new mapping, HybridOS must allocate a new set of memory pages in pinned physical memory. When the pages are first accessed, the application will fault-in the page and suffer the cost of having the OS insert the page table entry into the translation look-aside buffer (TLB). Our results show that above a certain number of transactions with the accelerators, the cost of setting up the initial mappings can be amortized. User mapped DMA buffers are a viable option when the startup costs can be made negligible, providing low-overhead data transfers and opening up greater possibilities for concurrency between the processor and the accelerators.

## 3.3 Accelerator Direct Mapping

Accelerator direct mapping methods attempt to avoid the cost associated with setting up and executing DMA transfers for small data sets. These mechanisms map the accelerator local memories directly into the virtual address space of the calling application. Both direct mapping methods require a setup routine to create the mapping, but offer the lowest overhead access method on a per-call basis. Uncacheable direct mapping writes directly to the accelerator memory, bypassing the cache, thereby avoiding cache pollution. Cacheable direct mapping writes to the accelerator local memory using the processor’s data cache as a buffer. The caching provides more efficient use of the bus by leveraging cache line transfers and exploits other benefits of caching data such as lower latency when locality is present.

There are no DMA buffers that must be pinned into memory for this model and no backing store for the memory hosted by the accelerator local memories. As a consequence of writing directly to the local memory of the accelerator, no copies must be made between user space and kernel space nor from a DMA buffer to

the accelerator local memory. Furthermore, removing the need for pinned buffers is of greatest benefit to accelerators with potentially large buffer requirements, e.g., GPUs with 100’s of MB of memory.

Both models have the drawback that only one mapping between the application and the local buffer can exist at any time. Future systems could leverage the abstraction we have created for the user-space application to change the mappings at runtime, allowing multiple applications to share direct mapped access.

### 3.3.1 Uncacheable Direct Mapping

Uncacheable direct mapping transfers data directly from the software application to the local memories of the accelerator using standard load and store instructions. Once the mapping is set up by the HybridOS initialization routine, the application writes data into the virtually addressed input buffers and then starts execution of the accelerator by executing an accelerator system call. While servicing the system call, the OS determines that the mapping is uncacheable direct, meaning that all of the input data is already available in the buffer, and immediately starts the computation. When the accelerator is finished, the call returns, and the application can access data directly from the accelerator local memory.

Uncacheable direct mappings avoid extra copies, avoid cache pollution, remove the cost associated with DMA setup, and remove the need for explicit cache coherence mechanisms. However, uncacheable direct mappings do not efficiently use the bus and may force computations to stall waiting for writes to the accelerator local memory to complete. The cost of not caching becomes performance-limiting for transfers greater than a few words due to inefficient bus utilization. For instance, uncached writes limit each transfer to be at most 32 bits while the system bus that attaches the processor to the accelerator is a 64-bit bus, leaving the system bus only half-utilized when uncacheable mappings are employed. Furthermore, each write will require arbitration for the bus, reducing performance if multiple bus masters exist. Write combining could alleviate the low utilization of the bus, but still does not provide the caching benefits of our cached direct mapping method.

The benefit of not caching the accelerator input buffers is that doing so avoids cache pollution associated with other methods. The data-parallel kernels that are amenable to implementation as reconfigurable accelerators are often streaming in nature. Streaming applications [12] produce data sets that are read, manipulated by a set of functional units, and then produce a result. Much of the locality a CPU data cache is meant to exploit is lost in such a model since once an input to the accelerator is produced, the processor will not access it again. Not caching inputs to streaming accelerators allows the cache to be put to more effective use to achieve better performance.

### 3.3.2 Cacheable Direct Mapping

Cacheable direct mappings use the CPU data cache to leverage existing prefetchers when streaming and locality when there exists persistent data shared between accelerator and CPU. Blocks of data to be written to the accelerator local memory are allowed to reside in the cache. When the application issues the start command to the accelerator, the only additional step is to flush the cache lines that may not have written back to the accelerator local memory. The benefits of the cacheable direct mapping are more efficient use of physical memory, increased bus utilization over uncached direct mapping, and no copy overhead associated with DMA-based methods.

Better bus utilization in the cacheable direct mapped method comes from two factors: overlapped communication and computation and burst transfers for cache lines. Writing to the cache takes

many fewer cycles than writing directly to memory due to the in-order pipeline and uncacheable store policy of our processor. The reduced write time allows for fewer cycles wasted waiting for writes to complete. With the data in the cache, the cache controller is able to combine the writes and flush the data back to the accelerator local memory in parallel with computation, increasing concurrency. In the best case, all the data generated during the computation phase will have been written back prior to starting the accelerator, rendering the explicit flush a null operation. In a fashion similar to write-combining on contemporary microprocessors, the bus will be able to make use of full-width bursts between the cache and the accelerator local memory that are not possible when issuing single, uncacheable loads and stores. For cache-coherent multicore processors we avoid unnecessary cache probing, which could otherwise reduce performance, and we avoid the complexity of participating in the cache coherence protocol. Both probing and protocol participation would be required if the accelerator local memory were kept coherent.

The major drawbacks to using direct access are coherence concerns, cache pollution, and the need for one-to-one mappings between user-space buffers and the local memories of the accelerator. Our current prototype platform does not provide support for cache coherence using the system bus. The accelerator library must therefore ensure that the data in the CPU cache is coherent with the data that the accelerator will access from the accelerator local memory. In our implementation, we explicitly flush the cache lines corresponding to the accelerator local memory and add a synchronization barrier prior to allowing the accelerator to begin. Cache pollution is a valid concern for streaming applications that may trigger many writebacks to cache-allocated accelerator data; that is, data that may never be read again. We avoid total cache pollution by mapping a fixed-sized region of accelerator memory into the cache to create a cacheable accelerator memory window. The case where multiple buffers need to map into the same local memory is beyond the scope of this work, but the virtualization provided by our CPU/accelerator interface allows for the mapping method to be altered. The virtualization allows a method that does not require one-to-one mappings to be used and the data for the accelerator buffered. Although concerns exist with using cacheable direct mappings, our results demonstrate that for certain workloads, they are the most appropriate mechanism.

We use the same mechanism already presented for the accelerator input buffer to handle the accelerator result buffer. Thus far we have focused on writing data to the accelerator local memory through the cache; however, once the accelerator is finished, a mechanism must exist such that results can be read by the processor. To do so, the application accesses the result buffer using normal load instructions, but instead of going out to the accelerator local memory on each access, the first access triggers the corresponding cache line to be transferred into the processor data cache. The cache line transfer provides faster access to subsequent data while at the same time not affecting the latency of the critical word. Moreover, the data can now be cached, allowing for faster accesses when result data is repeatedly accessed after the accelerated computation completes.

### 3.4 Summary

Four access methods for transferring data between software applications and accelerators are presented. The methods implemented are encapsulated in a library call API that accesses the reconfigurable accelerator HybridOS platform. The device driver model for accessing hardware devices is described to demonstrate the lack of accelerator-specific OS support available to accelerator developers

and serves as a baseline for performance comparisons. The user-space buffer access method provides a custom-tailored accelerator system call along with runtime support that requires no set up prior to use. The user-space mapped DMA buffer method provides the same benefits of the user-space buffer method, but also reduces unneeded copies at the expense of requiring initialization prior to use. Uncacheable direct mappings provide a very low-overhead mechanism for accessing accelerators, but provide low bus efficiency. For slightly more overhead, bus efficiency is greatly improved by our cacheable direct mapping model.

## 4. EXPERIMENTAL RESULTS

In this section the accelerator access methods are evaluated. We first explore the overhead for our various access methods using JPEG image compression as an example. Next we provide general results to show that for different data sets, different access methods are optimal. A discussion of the results and the utility of those results for future hybrid system builders concludes the section.

### 4.1 JPEG Case Study

We have integrated hardware accelerators with a commonly used integer-based JPEG encoding library [15]. Our motivating application is a JPEG encoder test application built on top of the hardware-accelerated library. We have built a quantizer accelerator and a 2D discrete cosine transform (DCT) accelerator into our framework. The interconnect is set up such that the results of the DCT accelerator flow directly into the input of the quantizer via the in-framework interconnect. In the best case, on a per-call basis, the hardware-accelerated version of DCT and quantization provides a speedup of 1.85x over software execution *when taking into account data movement and call overhead*, while maintaining protection boundaries set by the OS.

Minimal changes must be made to the source to enable accelerator access in our model. Library call wrappers are used to integrate the hardware accelerators, resident in the accelerator framework, into the JPEG encoder application. Two source code changes must be made to the JPEG encoder to access the accelerators. First, the call to DCT and quantization in the original code is replaced with a call to our accelerator library, resulting in no additional lines of C source code. Second, a call to initialize the desired mapping is added to the initialization code of the JPEG encoder. The initialization routine adds a single line of C code to the application. The encoder is now able to use the accelerators for DCT and quantization with minimal changes having been made to the source code of the original application.

The augmented version of the JPEG encoder application, which integrates two hardware accelerators, is used to study the overhead associated with accelerator accesses. The accelerated call takes as input a single 8x8 pixel region called a *macroblock*. Each pixel is represented by a 16-bit signed integer. The pixels are packed into a contiguous 128-byte region of memory for all access methods. The accelerator produces a quantized 8x8 matrix of 16-bit signed integer coefficients as a result of its computation.

Through major refactoring of the source, the original application could be accelerated by reconfigurable accelerators to a greater degree. However, we choose not to modify the application thereby demonstrating the ability of HybridOS to provide speedup for fine-grained accelerators while not increasing the burden on software developers. The HybridOS interface aims to avoid having developers change the architecture of their software applications to accommodate reconfigurable accelerators. The streaming nature of the application is suppressed to study our model's ability to accelerate applications of finer granularity.

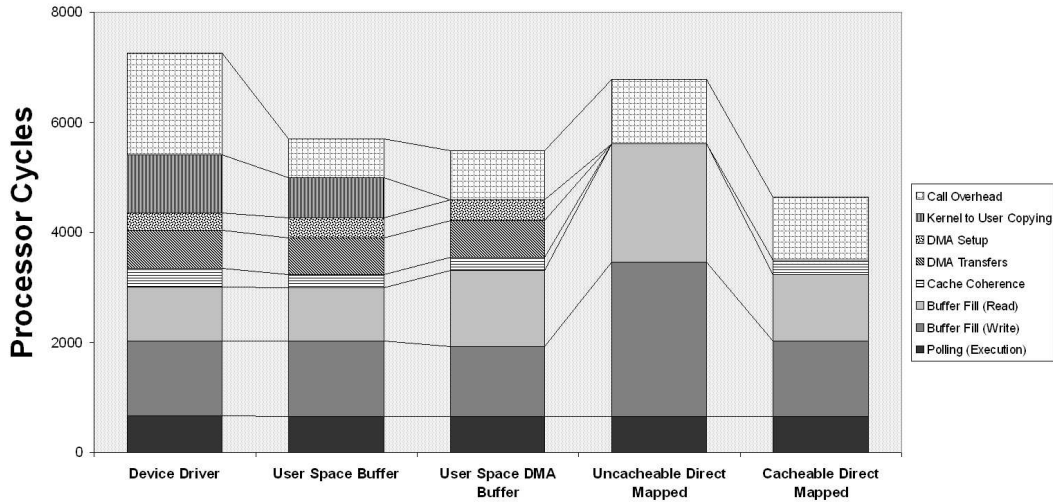


Figure 4: A single call to DCT and Quantizer using each access method.

There are eight possible operations that constitute a single transaction with the accelerator framework in our implementation of HybridOS, but not all are necessary for each method, e.g., DMA setup is unnecessary for direct mappings. *Polling* is the time required for the accelerators to process a single macroblock when not considering any data movement overhead. The *buffer fill* is the time spent writing data from its source in the original code to the accelerator input buffer in the virtual address space of the application. The *cache coherence* component is the time spent flushing the cache lines of the macroblock explicitly from the CPU data cache using the PowerPC *dcbf* instruction. *DMA transfers* is the time from when the DMA transaction is initiated until the DMA status register, which is polled, signals the completion of the DMA transaction. *DMA setup* is the time required to set the four DMAC control registers. *Kernel-to-user copying* is the time required to perform the Linux copy calls to move data from a user-space buffer to a statically-allocated one inside the kernel. *Call overhead* is the time spent going into the kernel and any other time that is not accounted for in the above sections.

The results represent the arithmetic mean of 11,900 accelerator accesses when encoding a 1.4 MB reference image. Measurements were taken on our test platform to determine the overhead of each component of the JPEG quantizer and DCT call using each of our four access methods. Figure 4 depicts the results of these measurements along with the time for the device driver implementation included for comparison. All timing measurements are taken using the 64-bit timebase of the PowerPC with synchronization instructions inserted to ensure accurate timing results.

We now highlight some of the key insights gained from the results presented in Figure 4. The device driver method is the highest-overhead access method demonstrating that our low-overhead system call interface provides net benefit over using the character device driver library code when only considering performance. Much of the overhead associated with all five methods is due to context switch overhead and the associated cache and TLB misses shown by the top section of each bar. For the access methods that require them, polling while waiting on accelerator completion, cache coherence actions, DMA setup, and DMA transfers are the same. Kernel-to-user copying overhead incurred by the user-space buffer method represents the non-trivial direct cost for ad hoc protection checking. The protection checking is the overhead that is avoided

through the use of pre-mapped DMA buffers or direct mapping.

The time required to write data into the accelerator input buffer and subsequently read data out of the accelerator result buffer from user space is the source of the greatest variability across access methods. The variability of the buffer reads and fills is due to the use of varying levels of indirection. The application is always filling and reading the same virtually-addressed buffer for all four methods; however, those buffers reside in different places in physical memory. For the device driver and user space driver methods, the input and results buffers are both in the application's heap area. The benefit of this type of allocation is that the corresponding pages of the buffer share data with other often accessed variables, potentially avoiding TLB misses<sup>1</sup>.

The results show that the lowest fill and read times are for the device driver and user-space buffer access methods, but these access methods require additional copies that negate any benefit. The user mapped DMA buffer method avoids the additional copies, but at the cost of having potentially higher TLB miss rates as the corresponding page may only be accessed when the buffer is accessed. The uncacheable direct mapping has the worst performance of all four methods we develop and by far the worst read and write times. The poor performance of uncacheable direct mappings is to be expected due to the poor bus utilization and long-latency write operations. Uncacheable accesses would have even worse performance if each of the data elements were word-aligned such that each 32-bit read and write only contained a single pixel instead of two packed into a single word. The cacheable direct mapping yields the best performance for JPEG encoding due to low buffer fill and read times, avoiding extra copies, and obviating the need for DMA setup. One additional benefit of caching the results comes from reads to the result buffer pulling the entire eight-word cache line into the processor cache on the first access. A cache line transfer results in low-latency cache hits for the remaining words on the cache line.

The JPEG encoding case study provides a concrete example of our access models at work, the virtualization they provide, and the viability of such a model for incorporating hardware accelerators

<sup>1</sup>The TLB replacement strategy does not favor more recently accessed page table entries and therefore does not provide as much gain on our platform as is possible, but the gain could be substantial on other platforms.

into applications run on general-purpose processors. We examine the overheads associated with each method for accelerators that process a small, fixed amount of data per transaction. We will return to the broader implications of these results after investigating accelerator access methods when considering a variety of transfer sizes.

## 4.2 Data Transfer Overhead

Each of the access methods we describe provides benefit to a class of workloads. The JPEG example provides a single point in this design space using our platform; however, the JPEG example does not provide the information necessary to make general statements about the appropriateness of the access methods for different applications, or even JPEG using different transfer sizes per accelerator transaction. We measure the time required to transfer data sets of varying size between the application running on the general-purpose processor and the accelerator framework in our model to provide the information necessary to best match an application with the corresponding optimal access method.

Figure 5 shows the number of processor cycles per transaction with the accelerator framework for each of our four methods. The results capture the cost associated with call overhead and data transfers by setting the accelerator execution time to zero. Three distinct regions are highlighted in the figure showing where uncacheable direct mapping, cacheable direct mapping, and user mapped DMA are the optimal transfer method from left to right, respectively. A small price is paid for unaligned DMA transactions using our setup resulting in the oscillation seen in the figure for the DMA access methods. For all transactions with the accelerator local memory exceeding 160 bytes in length, user mapped DMA is the lowest-overhead access method on a *per-call* basis; however, possible startup costs must be considered when assessing the best solution for a particular application. We discuss the trade offs below.

Uncacheable and cacheable direct mapping methods are the preferred methods for small data transfers. The uncacheable method allows for data to be written directly to the local memory of the accelerator framework, thus avoiding cache coherence actions, DMA setup, and DMA transfer overhead. However, such a method is inefficient for larger transfers and, even for a small number of transfers, has a high per-byte cost due to the system call overhead. Cacheable direct mappings provide better per-byte transfer costs above 16 bytes by having better bus utilization. Our current platform requires explicit cache management instructions to be issued due to lack of cache coherence support in the architecture. Therefore, the advantage of better bus utilization is not reaped until the flush cost can be amortized. Up to a certain transfer size, direct mapping is the optimal choice for accelerator access; however, for larger data transfers, other methods must be explored.

DMA approaches are shown to be better for larger data transfers, overlapping computation and communication. For transactions with data transfers greater than 160 bytes, user mapped DMA is the optimal choice of data access method. The cost associated with DMA setup, cache coherence actions, and the additional copy from a DMA-capable buffer to the local storage of the accelerator framework is overcome with larger transfer sizes. An added benefit of a DMA-based approach is the opportunity to overlap more computation with the transfer of data from the DMA buffer to the accelerator local memory. User mapped DMA provides the lowest overhead for applications with a given transfer profile, overhead that continues to become a smaller fraction of overall access cost as the transfer per transaction size is increased.

For no transaction size in the figure does the user space DMA access method have the lowest overhead. The user space DMA ac-

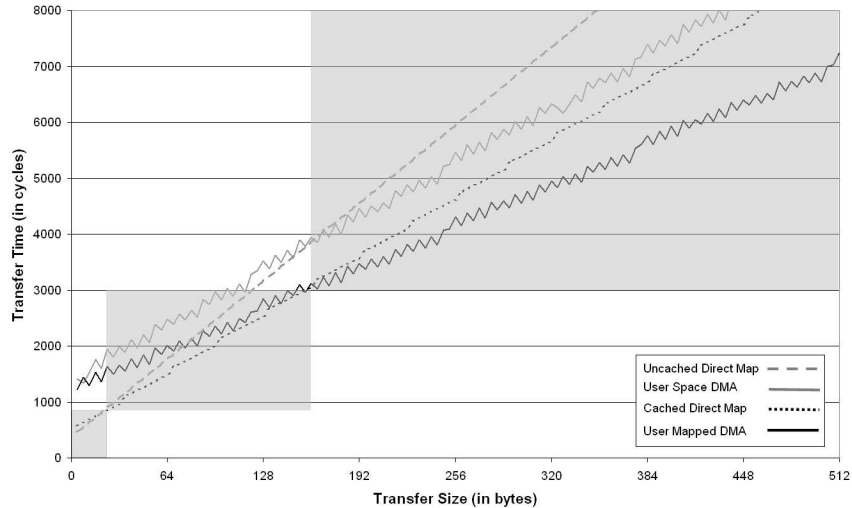
cess method is still an optimal method for certain workloads, however, since it has no initial startup cost associated with it. While the other three access methods shown in the figure have lower *per-call* overhead, the figure does not reflect the high cost associated with setting up a mapping. As Table 1 shows, the memory mapping necessary inside the library API initialization call of our model has an overhead on the order of tens of thousands of processor cycles. For accelerator accesses that will only be used a few number of times and cannot amortize the cost of initialization and unmapping, user space DMA offers a viable solution as it requires low initial setup cost.

## 4.3 Discussion

A key concern for software developers is finding an efficient way to represent internal data structures that can be easily transferred to the accelerators for processing. The process of aggregating the data needed for a transaction prior to relinquishing control to the accelerator is referred to as *data marshaling*. In the JPEG case study we present, data is always accessed in linear blocks. For cases where the input to the accelerator must be extracted from larger data structures or accessed via pointer-based data structures, extra data copies and pointer dereferencing would be required. As another example, we pack two 16-bit values per transfer for the uncacheable direct mapping to provide best-case speedup. However, if the marshaling is not performed, writes could be as small as one byte per transfer resulting in at best only one-eighth of the bus bandwidth being utilized. Answering the questions of how data should be laid out and what architecture support should be available to achieve optimal performance for applications run on our hybrid system is left for future work.

Direct mapping can be extended to allow applications to have more control over accelerator actions and reduce overhead. The system call interface developed in this work results in an overhead that can be removed by mapping control registers into the application's virtual address space. The tradeoffs for suffering a small performance degradation by using a system call interface instead of directly mapping all accelerator resources into the application are the opportunity for virtualization by handing off control decisions to HybridOS, well-defined security boundaries to protect the system and accelerators from malicious or defective entities, and a simplified access model. It is possible to allow more control to be put into the user space library code and still maintain protection boundaries by adding greater complexity to the design, but is left for future work. Nevertheless, the encapsulation and abstraction provided by the library call interface enables our programming model to remain consistent even as the spectrum of abstraction levels is explored. The access methods explored here also remain intact as the responsibility for accelerator interactions shifts between user space and kernel space.

As hybrid system workloads vary at runtime, so too must the way the hybrid applications access the accelerator resources. We have shown that for different transfer sizes and potential number of accesses per setup, different access methods must be utilized for the best performance to be achieved. Each access method has an array of tradeoffs that further complicates the assignment of applications to access methods. For general-purpose applications to be built upon hybrid systems, there must be a means to allow for dynamic adaptation that takes into account the access method tradeoffs. Without the possibility of dynamically mapping between applications and accelerator access methods, multiprogramming workloads will be unable to efficiently and effectively utilize hybrid systems. We present a set of application and accelerator interfaces that are used by HybridOS that we have developed along with a charac-



**Figure 5: Per-byte transfer cost for the access methods.**

terization of the methods that can be used as the basis for realizing a high-performance adaptive hybrid system.

## 5. RELATED WORK

HybridOS presents a set of techniques that can be used on a wide array of hardware platforms that incorporate general-purpose CPUs with coprocessors. Previous work has investigated an array of hybrid CPU/accelerator systems [5–7, 9, 10, 18, 19, 23]. Furthermore, many devices that integrate a CPU with the reconfigurable fabric of an FPGA are available commercially [1, 25].

Methods for providing access to data have been investigated for CPU/accelerator architectures. SCORE [7] uses the concept of a configurable memory block to link applications and accelerators physically and stream buffers as a mechanism to convey data logically. OneChip-98 [11] provides a mechanism for ensuring consistency of data shared by the accelerator and the general-purpose processor. Although both techniques could be used to implement the low-level memory architecture of a system running HybridOS, HybridOS is meant to provide a greater degree of abstraction to such memory access techniques to allow for remappings to be done transparently to the accelerated application. HybridOS also provides a different data access model than what stream buffers provide, allowing for irregular access and persistent data that may not be consumed in FIFO order as with streams.

Previous work has investigated the allocation of reconfigurable resources to hardware threads [8]. HybridOS is not in itself a reconfigurable resource allocation mechanism, but a set of techniques that allow for transparent remapping of the data resources used across software and hardware threads of execution. Providing a consistent interface to applications by leveraging virtual memory for reconfigurable devices [20] is adapted for use in HybridOS. Furthermore, a consistent view of system resources that does not deviate greatly from current practices has the benefit of support for existing tools and development techniques. An example of a similar model used to map applications to a CPU/acceleratorsystem are presented in [14].

Threaded models for reconfigurable systems [2,3] provide a way to encapsulate the state of reconfigurable accelerators and their execution over time. Previous work has focused on reconfigurable resource allocation and scheduling for multi-user machines [22]. HybridOS extends resource allocation to the memory that hosts

data accessed by the reconfigurable accelerator. HybridOS is meant to provide a consistent view of the data accessed by the software running on the general-purpose CPU and does not impose requirements on how the hardware task is executed. We leave investigation of how HybridOS could be adapted for use in a thread model to future work.

Using a platform similar to the one HybridOS is based upon, Noseworthy [17] investigates a variety of configurations demonstrating the need to map applications to appropriate interconnects and memories on reconfigurable platforms. In the high-performance computing arena there are examples of user-level access mechanisms to provide low-latency access to shared resources. Blumrich et al. have investigated virtualization of device memories and protected user-level interfaces in the context of high-speed network devices [4]. Welsh et al. look at hosting of device memory in [21]. Memory access considerations specific to hybrid system design are investigated in [11]. HybridOS builds on these previous works by providing the means to do resource remapping transparently to the application while still maintaining the protection guarantees of an OS managed design.

## 6. CONCLUSION

We present HybridOS and its data transfer mechanisms in the context of a protected interface model for reconfigurable accelerators attached to general-purpose CPUs. We utilize an accelerator framework that integrates reconfigurable hardware accelerators into common applications with OS support. The complete HybridOS platform decouples hardware and software design to reduce complexity and to enable the adoption of hybrid CPU/accelerator architectures and programming models. We provide both interfaces to the HybridOS runtime for applications, using a library call approach, and interfaces for the accelerator to plug into our hybrid CPU/accelerator model, using our accelerator framework. Our accelerator model allows for reduced complexity by providing an access model for accelerators with which software developers are comfortable.

We explore different accelerator access methods for transferring data between applications running on the general-purpose processor and the reconfigurable accelerators. We have found that the most effective access method depends on data transfer size and the number of times the application will use an accelerator. It is our

goal to take the insights gained from this study of the access methods on our platform to enable transparent, dynamic allocation of accelerator resources to applications, thereby allowing HybridOS to allocate the most effective access method at runtime.

The characterization of data access in our hybrid CPU/accelerator HybridOS model will allow for the wider acceptance of reconfigurable accelerator architectures by enabling a consistent library API that adapts to dynamic workloads. Having a set of application interfaces and an accelerator framework that decouple the process of application development from accelerator development is key to the future success of CPU/accelerator systems. Our study of the overhead associated with a variety of data access methods provides the insight necessary for developing effective CPU/accelerator systems. Furthermore, the framework, our HybridOS interfaces, and most importantly, our experimental results provide the basis for systems that incorporate reconfigurable accelerators with a general-purpose processor.

## ACKNOWLEDGEMENTS

The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. We would also like to thank Intel and Xilinx for their generous donations of equipment.

## 7. REFERENCES

- [1] Altera Staff. *Excalibur Devices Hardware Reference Manual*. Altera Inc., 3.1 edition, November 2002.
- [2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a uniform programming model across the software/hardware boundary. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 89–98, 2006.
- [3] 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, 2004.
- [4] M. Blumrich, C. Dubnicki, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *High-Performance Computer Architecture*, pages 154–165, February 1996.
- [5] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 141–150, New York, NY, USA, 2001. ACM Press.
- [6] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzyniek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *FPL*, pages 605–614, 2000.
- [7] A. DeHon et al. Stream computations organized for reconfigurable execution. *Journal of Microprocessors and Microsystems*, 30:334–354, September 2006.
- [8] W. Fu and K. Compton. An execution environment for reconfigurable computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, 00:149–158, 2005.
- [9] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, 1997.
- [10] J. R. Hauser and J. Wawrzyniek. Garp: A mips processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
- [11] 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.
- [12] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.
- [13] J. H. Kelm. Operating system interfaces to reconfigurable systems. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, December 2006.
- [14] J. H. Kelm, I. Gelado, M. Murphy, N. Navarro, S. Lumetta, and W. W. Hwu. CIGAR: Application partitioning for a CPU/coprocessor architecture. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2007.
- [15] T. G. Lane. *Using the IJG JPEG Library*. Independent JPEG Group, 6b edition, March 1998.
- [16] E. Lindholm and S. Oberman. NVIDIA GeForce8800 GPU. In *HOT CHIPS 19*, 2007.
- [17] J. Noseworthy and M. Leeser. Efficient use of communications between an fpga's embedded processor and its reconfigurable logic. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, page 233, 2006.
- [18] E. M. Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.
- [19] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In *13th Reconfigurable Architectures Workshop*, 2006.
- [20] M. Vuletic, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Design Automation Conference*, volume 41, pages 948–953, 2004.
- [21] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical Report TR97-1620, Cornell University, 13, 1997.
- [22] 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.
- [23] R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.
- [24] Xilinx Staff. *PowerPC 405 Processor Block Reference Guide*. Xilinx Inc., v2.1 edition, July 2005.
- [25] Xilinx Staff. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, October 2005.