Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link

Emerging hybrid chips containing CPU and FPGA components are an exciting new development promising commercial off-the-shelf economies of scale, while also supporting hardware customization.

David Andrews
Douglas Niehaus
Razali Jidin
Michael Finley
Wesley Peck
Michael Frisbie
Jorge Ortiz
Ed Komp
University of Kansas

Peter Ashenden
Ashenden Designs

Embedded- and real-time system designers are continually challenged to provide increased computational capabilities to meet tighter system requirements at ever-improving price/performance ratios. Best practices have long promoted the use of commercial off-the-shelf (COTS) components to reduce design costs and time to market. Creating COTS components that are reusable in a wide range of real-time and embedded applications is still a difficult challenge, partly because it requires the simultaneous satisfaction of apparently contradictory design forces: generalization and specialization. System designers are all too familiar with the tension caused by these opposing forces when trying to balance cost versus performance. Adopting COTS components reduces costs and time to market, but can prove inadequate for meeting challenging performance requirements. Custom components can achieve significantly higher performance when compared to COTS components, but require significantly higher development costs and longer times to market.

Recently emerging hybrid chips containing both CPU and field-programmable gate array (FPGA) components are an exciting new development that promises COTS economies of scale, while also supporting significant hardware customization. For example, Xilinx offers the Virtex II Pro, which combines up to four PowerPC 405 cores with up to approximately 4 million free gates, while Altera offers the Excalibur, which combines an ARM 922 core with approximately the same number of free gates. Designers can program the free FPGA gates with a widening range of standard FPGA intellectual property (IP) system library components, including serial and parallel I/O interfaces, bus arbiters, priority interrupt controllers, and DRAM controllers. They now have the freedom to select a set of FPGA IP to create a specialized system-on-chip (SoC) solution. This capability allows designers a COTS-like economy of scale while *specializing* the design for particular application requirements. One of the most interesting aspects of specialization is that the free FPGA gates can support customized application-specific components for performance-critical functions. Although an FPGA-based implementation's performance is still lower than that of an equivalent application-specific IC, the FPGA-based solution often provides acceptable performance and a significantly better price/performance ratio.

Tapping the full potential of these hybrids presents an interesting challenge for system developers. Although the ability to program them with a standard set of FPGA IP to replace current COTS designs with a single chip already exists, specifying custom components within the FPGA is tedious and device specific—contrary to the desired goals of modularity, portability, and reuse. Knowledge of hardware design methods and tools is also required, which places the full potential of these hybrids just out of reach of the majority of system programmers. New capabilities are being developed that allow the synthesis of hardware designs from a more familiar high-level language syntax: a fundamental step toward allowing programmers to form custom components within the FPGA. However, in addition to the high level syntax provided by a programming language, a complete programming model is necessary to completely abstract away the hardware-specific details, making it unnecessary for the designer to distinguish the use of the FPGA and CPU hardware components to produce a transparent system implementation platform. This will enable programmers to access the full potential of the hybrid system and apply familiar componentization and reuse best practices that reduce cost and time to market. To achieve this potential requires the definition of a new hybrid computational model and hardware-software codesign of runtime services.

Programming languages for reconfigurable architectures

Researchers are seeking solutions to system-level design for embedded systems by investigating new design languages, hardware-software specification environments, and tools. Projects such as Ptolemy,[1] Rosetta,[2] and System-C seek system-level specification capabilities that can drive software compilation and hardware synthesis. Other projects such as SPARK,[3] Streams-C,[4] and Handel C (www.celoxica.com) focus on raising the level of abstraction for FPGA programming, from gate-level parallelism to that of a modified and augmented C syntax. System Verilog (www.eda.org/sv-cc/) and a newly evolving VHDL standard (www.eda.org/vhdl-200x/) are also now under development, with the goal of abstracting away the distinction between the traditional low-level hardware-software interface and moving toward a system-level perspective. Although these approaches differ in the scope of their objectives, they all share the common goal of raising the level of abstraction required to design and integrate hardware and software components.

In light of all this work, a good question is then, if high-level programming-language capabilities for FPGAs continue maturing at current rates, will this be sufficient to allow software engineers to apply their skills across the FPGA-CPU boundaries? Unfortunately, current hybrid computational models are still immature, generally treating FPGAs as computational accelerators that are invoked passively as subroutines, or for essentially independent portions of a data flow computation, requiring only input and output queues.

Effectively programming across the FPGA-CPU boundary will require a high-level programming model that abstracts the FPGA and CPU components, bus structure, memory, and low-level peripheral protocols into a transparent computational platform. Programming models form the definition of software components as well as the interactions among them.[5] Message passing and shared-memory protocols are two familiar forms of component interaction mechanisms in use today. Both have been successfully used in the embedded world and practitioners enjoy debating the relative merits of their personal choice. We have chosen to discuss the multithreaded shared-memory model with the basic principles described here being equally appropriate for the message-passing model.

Multithreaded programming model

The multithreaded programming model is convenient for describing embedded applications composed of concurrently executing components that synchronize and exchange data. The popularity of the multithreaded programming model is apparent from the widespread use of the Posix threads standard.[6] One example of this is the significantly improved Pthreads multithreading library included with new releases of Linux. Because of its acceptance within the software engineering community and expressiveness for embedded applications, the multithreaded model is a good candidate for a programming model that can provide a high-level abstraction; it provides a largely uniform view of computations whose set of components can cross the FPGA-CPU boundary.

Under such a multithreaded programming model, system developers can specify applications as sets of threads distributed flexibly across the system's CPU and FPGA assets. This approach promises several advantages.

First, the use of the familiar multithreaded programming model will greatly reduce design and development costs since the computational structure of hybrid applications remains familiar at the highest level of abstraction.

Second, whether the threads implementing a computation are CPU- or FPGA-based will become just one more of the available design and implementation parameters with resource use and application performance implications. How to perform this partitioning to best support the needs of an application or system is yet another challenging problem currently under investigation.

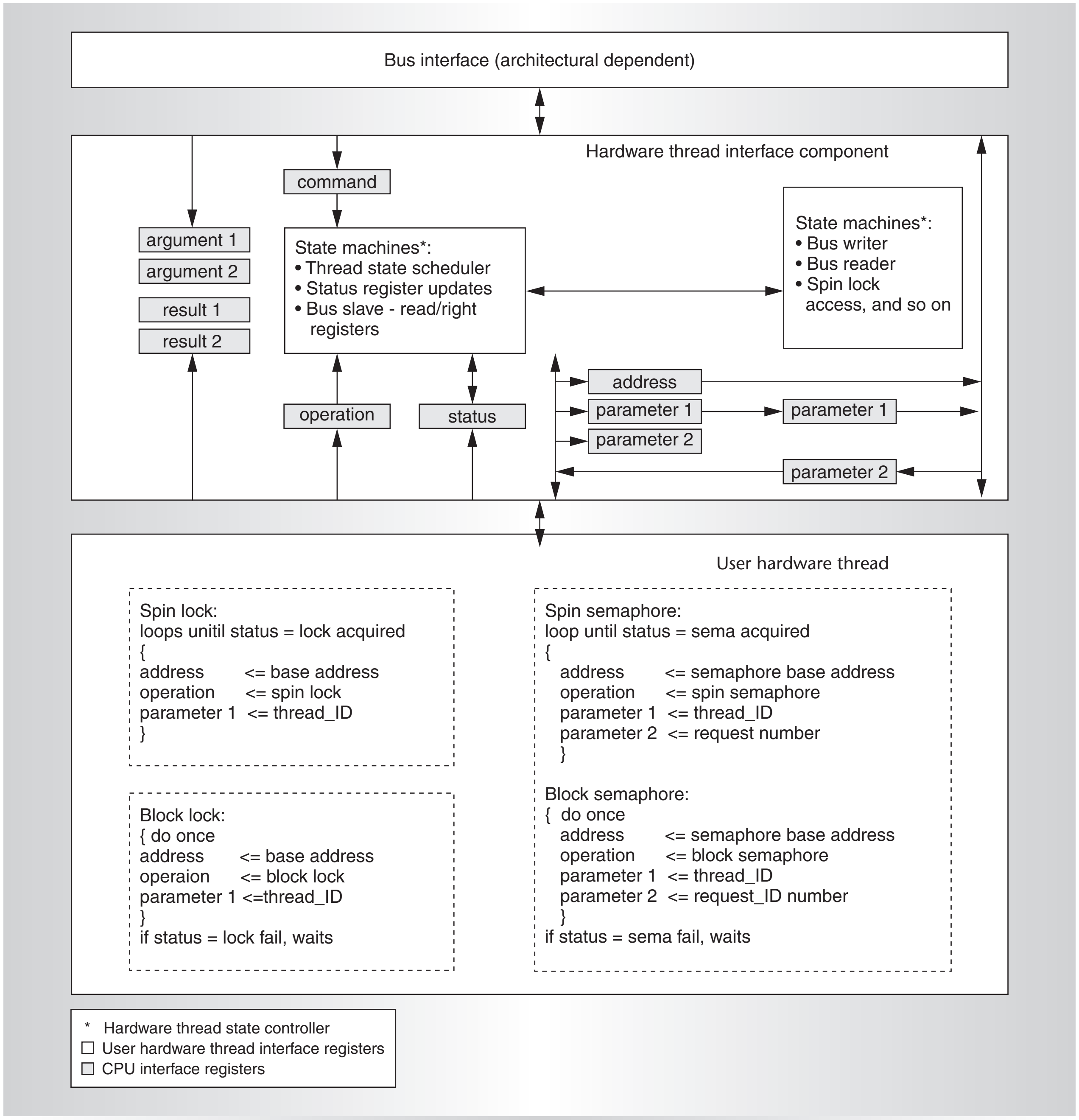
Third, it's possible to iteratively develop applications, beginning with an exclusively CPU-based multithreaded implementation and gradually transferring specific threads to FPGA support. Such an approach can accelerate time to market, as well as make it possible to focus FPGA support on the portions of the application that will benefit the most as established by performance measurements.

At this point, it is appropriate to draw a distinction between policy and mechanism. The *policy* of the multithreaded model is fairly simple: to allow the specification of concurrent threads of execution and protocols for accessing common data, and synchronizing the execution of independent threads. On a general-purpose processor, the mechanisms used to achieve this policy include the definition of data structures that store state information about thread execution, and the semantics of how thread synchronization primitives interact with the operating system's thread scheduler. For example, both the synchronization (semaphore) control and the thread-scheduling portions of the system software access common data structures for thread context. Additionally, synchronization primitives make use of architecture specific assembly language instructions such as load linked and store conditional.

The FPGA computational model employs a different-enough level of abstraction from that of the CPU that there is no immediately obvious equivalent to the CPU structures used to hold a thread's context such as the register set, program counter, and stack pointer. Additionally, with current FPGA technology, system developers must synthesize and map the data paths and operations that represent the computations of the thread into the FPGA before runtime. These differences require new mechanisms for achieving the basic policies of the multi-thread model for threads running within the FPGA, and to support interactions among threads across the FPGA-CPU boundary. Although, at first glance, the lack of a computational model seems to be a liability, it is instead an asset, because it presents an opportunity to create efficient mechanisms for implementing FPGA threads and for supporting thread synchronization within the FPGA and across the FPGA-CPU boundary. The implementation of threads in FPGAs must maximize the advantages of using the FPGA platform, while preserving the common multithreaded programming model. Blindly adhering to historical thread syntax and semantics that were developed for general-purpose CPUs is a trap. It can easily lead to an implementation that falls far short of the reconfigurable logic's performance potential. At the same time, the FPGA thread implementation must not use methods that degrade the efficiency of synchronization across the FPGA-CPU boundary. Neither should the FPGA methods create any form of unfairness between the FPGA and CPU threads.

Hybrid thread abstraction layer

As part of our ongoing research to extend the thread programming model across hybrid FPGA-CPU components,[7,8,9] we have defined the Hardware Thread Interface (HTI) library component shown in Figure 1. We implement the Hardware Thread Interface component as three subcomponents: a CPU interface, a hardware-thread state controller, and a user hardware-computation interface. The user



Bus interface (architectural dependent)
Hardware thread interface component
command
argument 1
argument 2
result 1
result 2
State machines*:
• Thread state scheduler
• Status register updates
• Bus slave - read/right registers
State machines*:
• Bus writer
• Bus reader
• Spin lock access, and so on
operation
status
address
parameter 1
parameter 2
parameter 1
parameter 2
User hardware thread
Spin lock:
loops unitil status = lock acquired
{
address <= base address
operation <= spin lock
parameter 1 <= thread_ID
}
Block lock:
{ do once
address <= base address
operaion <= block lock
parameter 1 <=thread_ID
}
if status = lock fail, waits
Spin semaphore:
loop until status = sema acquired
{
address <= semaphore base address
operation <= spin semaphore
parameter 1 <= thread_ID
parameter 2 <= request number
}
Block semaphore:
{ do once
address <= semaphore base address
operation <= block semaphore
parameter 1 <= thread_ID
parameter 2 <= request_ID number
}
if status = sema fail, waits
* Hardware thread state controller
☐ User hardware thread interface registers
☐ CPU interface registers

Figure 1. Hardware Thread Interface component.

interface binds our device independent API's accessible from within the application program to platform-specific implementation methods. Thus the HTI component provides a general-purpose register set that supports platform independent user level semantics to promote thread migration across the system.

We provide the Hardware Thread Interface component as a library for inclusion with each user-defined hardware thread. The set of Hardware Thread Interface components and a software interface component form a system-level hybrid thread abstraction layer as shown in Figure 2. The hybrid thread abstraction layer implements all interactions between user threads and other system components through the command and status registers. This capability is particularly useful for debugging and is

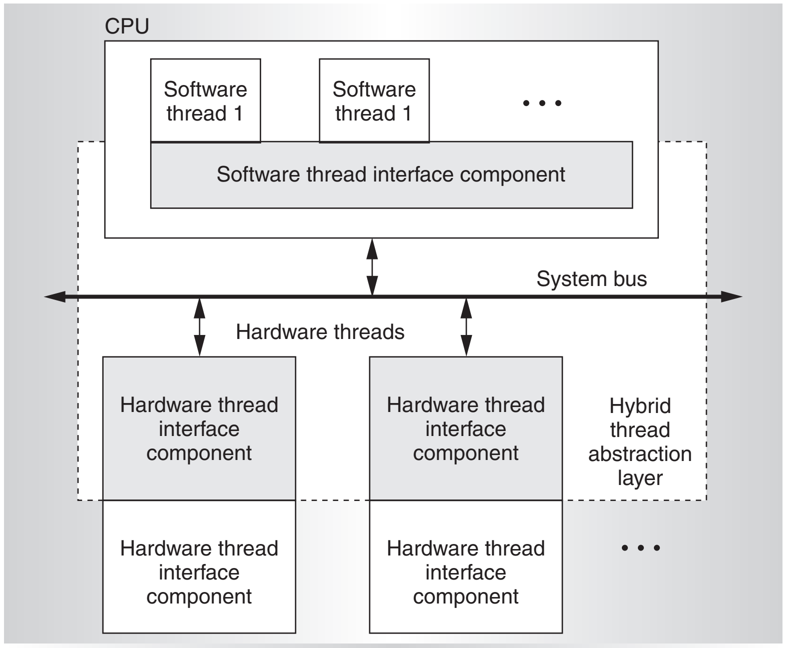

CPU
Software thread 1
Software thread 1
• • •
Software thread interface component
System bus
Hardware threads
Hardware thread interface component
Hardware thread interface component
Hybrid thread abstraction layer
Hardware thread interface component
Hardware thread interface component
• • •

Figure 2. Hybrid thread abstraction layer.

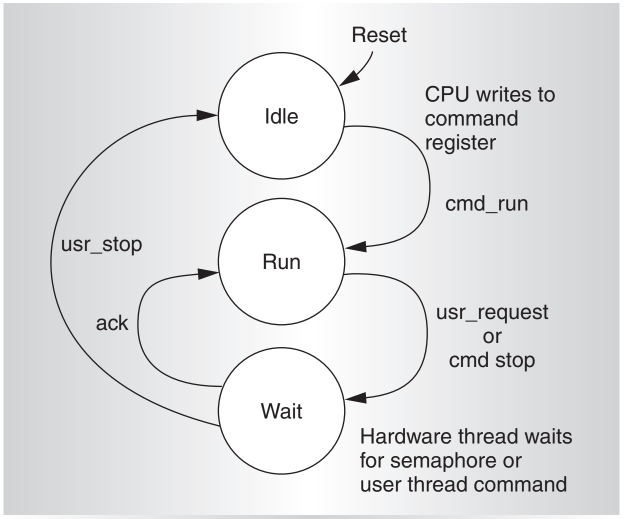

Reset
Idle
CPU writes to command register
cmd_run
usr_stop
Run
ack
usr_request or cmd stop
Wait
Hardware thread waits for semaphore or user thread command

Figure 3. Hardware thread controller state machine.

used during runtime to interact with the other system components, such as our semaphore IP for thread blocking and wake-up. During debug, the status register is accessible, so that the developer can determine the execution state of a hardware thread. We have also included control signals within the hardware thread interface component to perform a system soft stop and soft reset. The soft stop is similar in capability to setting a break point within the software and causes all state machines within the hardware thread interface to temporarily halt. The soft reset is useful when a particular IP needs to be reset without requiring a complete system reboot. Compared to existing approaches that implement simple slave coprocessors within an FPGA, our approach has one new requirement for supporting hardware threads: A thread must also write to—in addition to read from—memory-mapped locations across the bus. The bus interface supports both bus slave and master modes; as bus master, a thread must also be able to request bus transfers, such as memory accesses, on behalf of the user thread.

The state machine in Figure 3 controls the execution of a user hardware thread, which will be in one of three states: idle, running, or waiting. Threads that have not yet started or have terminated are in the idle state. Threads that are currently in the run state transition into the wait state when a thread requests a semaphore, or continues to be blocked on a semaphore. Each hardware thread's state is maintained in the status register. Dedicated hardware threads require no context switching when transitioning the thread into the wait state. Instead, the hardware thread simply idles. This allows the adoption of the same approach for both spinning and blocking semaphores. The hardware thread interface component, however, does perform different processing for spinning and blocking semaphores. For the spinning semaphore, the hardware thread interface transitions the thread state into the wait state, issues a single request for the semaphore, and returns the thread to the running state when the request's status is returned in the status register. The thread then checks to see if it owns the semaphore or not. In contrast, for a blocking semaphore, the hardware thread interface transitions the thread to the wait state, issues the request for the blocking semaphore, and leaves the thread in the wait state while the semaphore is in use. Upon a grant or release, the state machine will then transition the thread back into the run state. The semaphore IPs we discuss later exemplify the differences in the semaphore logic for the two types of semaphores.

Operating-system codesign

The operating system is the most fundamental system software layer, abstracting a

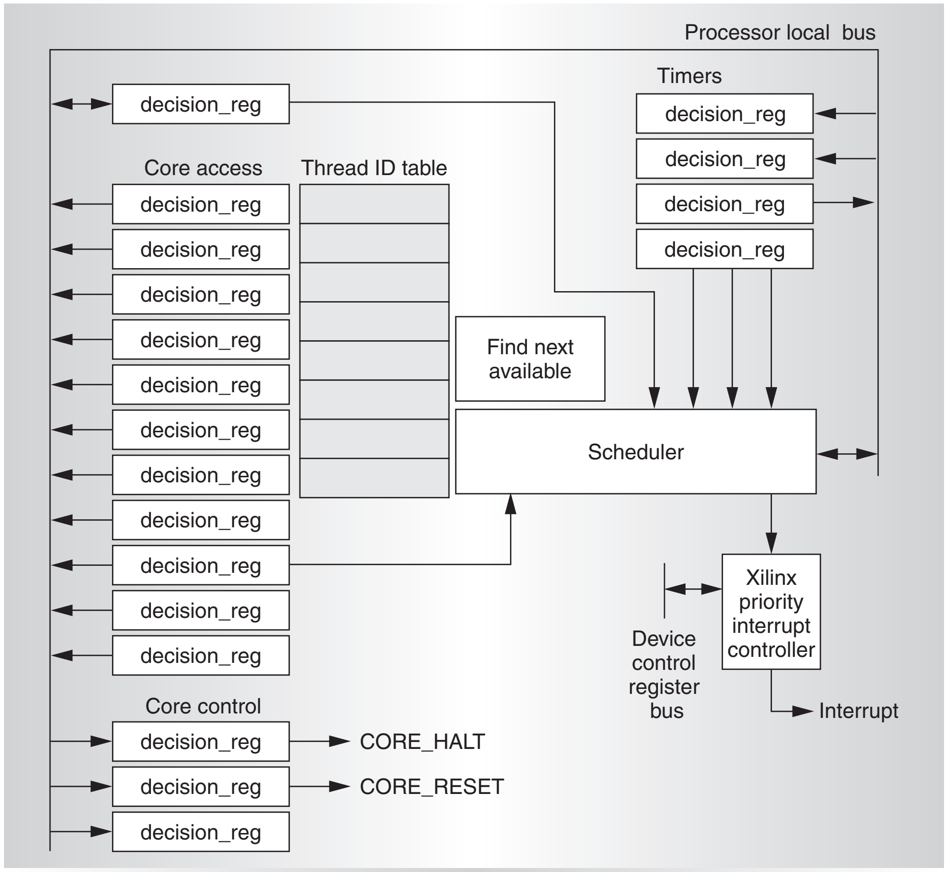


Figure 4. Software thread manager subsystem.

wide range of low-level system services and device-specific requirements into a generic set of interfaces through which both application and system programs access specific operating-system capabilities. Creation of a hybrid thread programming model, which eliminates the distinction between a CPU- and FPGA-based thread from the developer's point of view, requires a hardware-software codesign of portions of the operating system to extend system services across the FGPA-CPU boundary.[10] One of the most attractive goals of hardware-software codesign of operating system services is to reduce the overhead of servicing asynchronous interrupts while reducing their effect on overall system behavior. Hardware-software co-design of operating system services such as timekeeping, event queue management, interrupt handling, and task scheduling hold the promise of creating a real-time system with extremely low scheduling jitter and enhanced performance. An extremely important goal for our hardware-software codesign of operating-system components is to create a system that will only interrupt the CPU when a change in the system state requires the CPU to switch to another activity. Such changes in system state include timers expiring, devices completing an assigned activity and generating an interrupt, or external inputs arriving in the form of network packets. Current systems fall far short of the ideal because the CPU must be interrupted whenever a change in system state might require a switch of CPU activity.

Task management

Figure 4 shows the hardware component block diagram of our hardware-software codesigned thread manager for software threads, the software thread manager (SWTM). The SWTM hardware component contains a thread state table, scheduler for software threads, system timers, and interrupt processing. The SWTM provides an interface for software thread state change requests from all

```
htread create()
{
   if (attr>detached)
       threadStatus = create_thread_detached;     ⇒ if( thread unavailable ) return 0 + ERR_BIT
                                                    thd's status=used,~exited,~queued,~joined,detached
                                                    thd's pid = 0
   else                                             return thread's Id
       threadStatus = create_thread_joined;       ⇒ if( thread unavailable ) return 0 + ERR_BIT
                                                    thd's status=used,~exited,~queued,~joined,detached
                                                    thd's pid = current_thread
   if( !hasError (threadStatus))                    return thread's Id
       threadID = extractID(threadStatus);
       update software data structures
       addStatus - add_thread(threadID);          ⇒ if( threadID->status=used,~exited,~queued ) {
                                                        threadID->status = queued
                                                        add threadID to RZR_QUEUED
                                                        update RZR_QUEUE
       if( hasError(addStatus) )                        return 0;
       {                                            } else return threadID->pid,status,ERR_BIT
           clrStatus - clear_thread(threadID);    ⇒ if( threadID->pid == current_thread ) {
           update software data structures              threadID->status = ~used, ~exited, ~queued,
           return RUN_QUEUE_FULL;                                          ~joined, ~detached
        } else                                          threadID->pid - 0;
           thread->id = threadID                        return 0;
     } else                                         } else return threadID->pid,status,ERR_BIT
        return NO_THREADS_AVAILABLE;

      return SUCCESS;
}
```

Figure 5. Thread-create pseudo-code and processing.

sources, including software threads, hardware threads, and system devices. However, the SWTM does not perform the scheduling of hardware threads; instead, hardware threads are controlled in a distributed fashion through their hardware thread interface components. We have migrated the basic control mechanisms for software threads into the SWTM to reduce overhead, minimize code requirements, and support our goal of migrating the scheduler for CPU-based threads into the hardware. Thread management functions are invoked by reading or writing to the register set. A write to, or a read from, a register invokes a state machine that performs the associated processing. Figure 5 shows an example of our hardware-software codesign thread-create API. The pseudocode that the CPU executes appears on the left of Figure 5; the register read or write invokes the subsequent processing that appears on the right. Similarly, Figure 6 shows the pseudo code and processing for joining a thread.

The SWTM scheduler in Figure 4 runs in parallel with the execution of the software thread on the CPU. To reduce overhead and CPU thread scheduling jitter, all requests to interrupt the CPU, including external-device interrupts, expiring timers, terminating, blocking threads, and unblocking threads, go to the SWTM scheduler. This approach is distinctly different from existing approaches that allow external interrupts to cause state changes outside of the CPU thread scheduler's control. When a state change requires a context switch away from the software thread on the CPU, as determined by the SWTM scheduler, the SWTM generates an interrupt. The SWTM interrupt service routine on the CPU is very lightweight, only needing to read the thread_id of the next thread to run from the next_thread register and performing the specified context switch.

The first version of the SWTM implemented simple first-in first-out and round robin schedulers to simplify development and debugging. The next version will implement our recently developed, component-oriented and flexibly configurable scheduling framework.[11] We call our approach *group scheduling* because it explicitly addresses an important trend in system design: The implementation of computations as groups of computational components, most often but not exclusively

```
htread_join()
{
   joinStatus = join_thread(thread-?id)          ⇒

   if( !hasError (joinStatus) )    {
       if( joinStatus != ALREADY_TERMINATED )   {
           run thread scheduler
       } else
           return joinStatus
   }

   clearStatus = clear_thread(thread->id)        ⇒

    return clearStatus
 }
```

```
if( (threadID->pid == current_thread ) &&
    (threadID->status==~used,~joined,~detached, {
    if( threadID->status != exited
        threadID->status = joined
        return 0;
    }   else return 0+ALREADY_TERMINATED
} else return threadID->pid,status,ERR_BIT
```

```
if( threadID->pid == current_thread ) {
    threadID->status = ~used, ~exited, ~queued,
                       ~joined, ~detached
    threadID->pid - 0;
    return 0;
} else return threadID->pid,status,ERR_BIT
```

Figure 6. Thread join pseudo-code and processing.

threads, and the most-appropriate scheduling semantics for the group of components depends on the semantics of the computation they implement. Group scheduling addresses these issues by permitting the grouping of threads and other operating-system computation components, such as interrupt handlers, according to the computations they support. In addition, this approach associates a scheduling decision function (SDF) with the appropriate semantics within each group. We then form the SDF for the system as a whole by hierarchically composing group SDFs into a decision tree that controls the execution of all threads and other computation components on the system. We have implemented the group-scheduling framework in software under the most recent version of Kurt-Linux.[12]

Semaphores

Semaphore implementations on modern general-purpose CPUs are based on pairings of atomic read and (conditional) write operations such as the load-linked and store-conditional instructions. These existing mechanisms can be integrated in with memory coherence protocols in shared memory multiprocessors (SMPs) to provide synchronization between applications running on multiple CPUs. This approach introduces a level of complexity that is not easily extendable to the independent hardware threads running within the FPGA. Instead of reproducing these mechanisms, we use the FPGA to implement more efficient mechanisms that are CPU family independent and require no additional control logic to interface into the system memory coherence protocol. Figure 7 shows such a semaphore implementation, our binary spin-lock mechanism. The basic mechanism uses a standard write of a thread_id into a memory-mapped request register. We define a simple control structure within the semaphore IP that conditionally accepts or denies the request. The thread requesting the write then performs a read operation of the owner register to see if the thread_id has been accepted as the new owner of the lock.

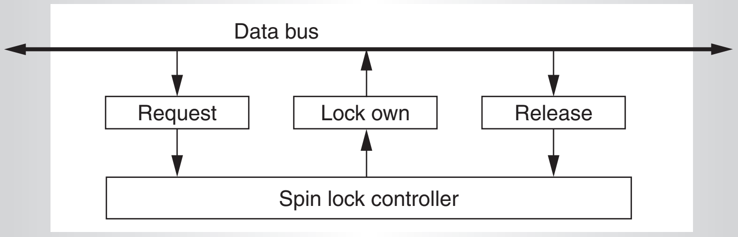


Figure 7. Spin-lock semaphore implementation.

Blocking semaphores allows the queuing and suspension of threads that cannot gain access to the semaphore, thus providing more efficient use of shared computing resources, and decreasing congestion on the system bus. Figure 8 shows our basic mechanism, which includes queue structures associated with each blocking semaphore to hold the thread_ids of blocked threads. Our semaphore API writes the thread_id of the requester into the request register and then checks the owner register, as users of the binary spin-lock do. If the thread

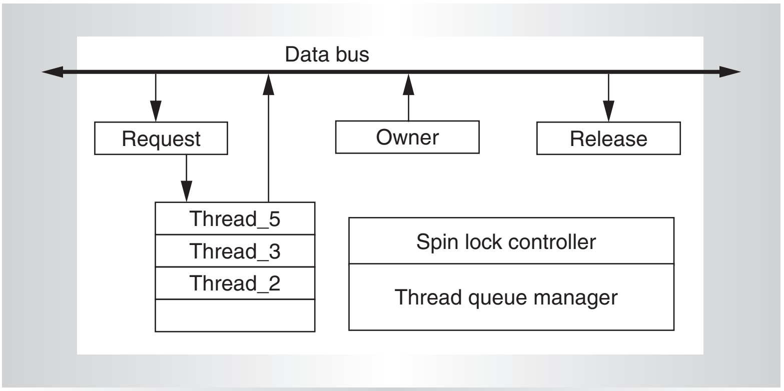


Figure 8. Blocking-semaphore implementation.

did not receive the lock, then the semaphore API saves the context of the (software) thread and calls the scheduler. . The semaphore API called from a hardware based thread puts the thread into the wait state until the semaphore is granted. In both cases, he semaphore IP control logic enqueues the thread_ids of blocked threads.

The lock is released when the current owner writes its thread_id into the release register. When a thread releases a blocking semaphore, the semaphore IP either enqueues the unblocked thread_id into the ready to run queue by writing into the add_thread register of the SWTM, or in the case of a hardware thread writes into the appropriate hardware thread interface component. This approach, having the semaphore IP interface with the SWTM, removes jitter associated with interrupting the CPU to perform updates of structures and calling the scheduler. Instead, all interactions are within the hardware components. The CPU will be interrupted only if unblocking the thread should generate an immediate context switch to a new thread as determined by the scheduler.

Timekeeping and timer services

Hardware-software codesign of timekeeping and timer services provides an opportunity to improve system performance in several ways that are particularly important for real-time systems. First, providing support for timekeeping services in the FPGA enables developers to choose among a wide range of timekeeping and timer resolutions while keeping the overhead of time management at a constant and low level. A second, more subtle advantage arises from the ability to modify the relationship between the timer interrupt and the scheduler. Modern systems use the timer interrupt to invoke the scheduler. In our system with the hardware based scheduler running concurrently with the application program, the semantics of the timer interrupt changes to that of a context switch interrupt that directs the CPU to switch from the currently running thread to the next thread selected by the scheduler.

The FPGA support for timers manages elements of the system timer queue in local memory on the FPGA, and detects the occurrence of the next timer event using a combination of match registers and periodic interval registers. As an example, we have used the FPGA to implement a pair of linked registers, jiffy and sub-jiffy, to support the existing Linux jiffy timekeeping variable under KURT-Linux. This approach transfers timer programming and timer queue management from the CPU to the FPGA, and optimizes performance in several ways. Most obviously, it provides the desired timer resolution up to the FPGA performance limits and modestly reduces overhead because the system no longer has to program the timer hardware for every timer event. More subtly, it maximizes timer accuracy at the desired resolution. This is because the use of the match register ensures that the calculation required to set the hardware timer does not affect when the event interrupt occurs, which is true of the standard timer hardware in the x86 architecture, for example.

Interrupt processing

FPGA-based timekeeping and timer services are an important element of support for more general improvements in system performance. They do not, however, significantly improve one of the most important aspects of real-time system performance: scheduling jitter, or "random" variation when scheduled system events actually occur primarily because of how systems handle asynchronous interrupts and control the concurrency they represent. Systems generally control concurrency arising from interrupts by blocking them. This increases scheduling jitter because it can delay delivery of a timer interrupt. Some approaches to improving real-time performance under Linux (http://www.fsmlabs.com/products/rtlinuxpro) use a dual-executive approach, which disable interrupts for the shortest possible periods, but run Linux as a non-real-time, best-effort computation (http://www.aero.polimi.it/~rtai/).

Our approach under Kurt-Linux also minimizes the periods during which interrupts are disabled at the hardware level and integrates interrupt processing under group scheduling.[11] This approach notes when interrupts occur, but defers the interrupt-handler execution until the group-scheduling framework chooses to execute the deferred interrupt handler according to the policies chosen by the developer. The current software version of this approach significantly reduces scheduling jitter and provides a dramatically improved worst-case scheduling latency. However, it still briefly considers each interrupt whenever it occurs, which is a form of randomized interference with CPU use by the computation components that the group-scheduling framework selects.

FPGA support will improve this in two ways. First, it will provide direct support for the group-scheduling framework. Second, the FPGA will directly support the tracking of interrupt occurrences required to integrate interrupt processing under the group-scheduling framework. The combination of FPGA support for interrupt tracking and group scheduling will all but eliminate scheduling jitter from our existing Kurt-Linux kernel, because it will ensure that the computational component using the CPU is never interrupted except when it should be stopped in favor of another. The only remaining sources of scheduling jitter will be the few, remaining brief periods when interrupts are disabled at the hardware level. These include small sections of code where KURT-Linux reconfigures memory management hardware registers and reconfigures the interrupt handling hardware. In addition, the CPU itself disables interrupts in response to interrupts and exceptions, but KURT-Linux quickly re-enables them in its interrupt and exception handling code.

Significant advances in fabrication technology are providing new COTS components that combine a general-purpose CPU and FPGAs. These new devices are a significant step toward realizing a single component that can support both the generalization of COTS components and the specialization required for individual embedded applications. Work is currently underway in developing unified programming models that allow computations within and across hybrid components expressed using the familiar multithreading programming paradigm. Creating a system-level multithreaded programming capability requires new hardware-software codesign approaches to supporting operating system and application functions. When complete, this capability will enable these new devices to be accessible by the broad community of system programmers and provide increases in operating-system efficiency. Enabling the multithreaded model across the hybrid components will ultimately provide shorter design times and lower development costs. MICRO

Acknowledgment

The work in this article is partially sponsored by National Science Foundation EHS contract CCR-0311599.

References

1. E. Lee, "Overview of the Ptolemy Project," tech. memo., Mar. 2001, http://ptolemy.eecs.berkeley.edu.
2. P. Alexander and C. Kong, "Rosetta: Semantic Support for Model Centered Systems Level Design," *Computer,* vol. 34, no. 11, Nov. 2001, pp. 64-70.
3. S. Gupta et al., "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," *Proc. 16th Int'l Conf. VLSI Design,* IEEE Press, 2003, pp. 461-466.
4. M.B. Gokhale et al., "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *Proc. 8th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 02), 2000, pp. 49-56.
5. E. Lee, "What's Ahead for Embedded Software?" *Computer,* Sept. 2000, pp. 18-26.
6. D. Butenhof, *Programming with POSIX Threads,* Addison-Wesley, 1997.
7. D.L. Andrews, D. Niehaus, and P. Ashenden, "Programming Models for Hybrid CPU/FPGA Chips," *Computer,* vol. 37, no. 1, Jan. 2004, pp. 118-120.
8. D.L. Andrews, D. Niehaus, and R. Jidin, "Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational

Components," *Proc. 1st Workshop on Embedded Processor Architectures, Proc. 10th Int'l Symp. High Performance Computer Architecture (HPCA 10),* Feb. 2004.

9. R. Jidin, D. Andrews, and D. Niehaus, "Implementing Multithreaded System Support for Hybrid FPGA/CPU Computational Components," *Pro. Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms,CSREA Press,* June, 2004. pp. 116-122.
10. D. Niehaus and D. Andrews, "Using the Multithreaded Computation Model as a Unifying Framework for Hardware-software Codesign and Implementation," *Proc. 9th Int'l Workshop on Object-Oriented Real-Time Dependable Systems* (WORDS 03), IEEE Press, 2003, pp. 317-324.
11. M. Frisbieet al., "Group Scheduling in Systems Software," *Proc. 18th Int'l Parallel and Distributed Processing Symp.* (IPDPS 18), IEEE CS Press, 2004, pp. 120-127.
12. B. Srinivasan et al., "A Firm Real-Time System Implementation Using Commercial Off-the-Shelf Hardware and Free Software," *Proc. 4th Real-Time Technology and Applications Symp.,* IEEE CS Press, 1998, pp. 112-120.

David Andrews is an associate professor in the Electrical Engineering and Computer Science Department at the University of Kansas. His research interests include embedded systems architectures, and parallel and reconfigurable computing. Andrews has a BS and an MS, both in electrical engineering, from the University of Missouri, Columbia, and a PhD in computer science from Syracuse University. He is a senior member of the IEEE.

Douglas Niehaus is an associate professor in the Electrical Engineering and Computer Science Department at the University of Kansas. His research interests include the design and implementation of real-time and distributed systems, operating systems, system and network performance evaluation, and tools for programming environments. Niehaus has a PhD in computer science from the University of Massachusetts, Amherst.

Razali Jidin is a research assistant at the Information and Telecommunication Technology Center and a PhD candidate in electrical engineering at the University of Kansas. His research interests include embedded system, hardware-software codesign, and computer architecture. Jidin has an MSEE from the University of Bridgeport, Connecticut, and an MSEE from the University of Leeds, England. He is a member of IEEE.

Michael Finley is pursuing an MS in computer science at the University of Kansas. His research interests include the use of FPGAs to simplify system software by moving traditional software functionality to within FPGAs. Finley has a BS in electrical engineering from the University of Kansas.

Wesley Peck is a research assistant at the Information and Telecommunication Technology Center and a PhD candidate in computer science at the University of Kansas. His research interests include algorithm design, hardware-software codesign and real-time operating systems. Peck has a BS in computer science from the University of Kansas.

Michael Frisbie is a staff engineer at Garmin International Inc. His research interests include embedded systems software and operating systems design. Frisbee has an MS and a BS in computer science from the University of Kansas.

Jorge Ortiz is a PhD candidate and is a research assistant in electrical engineering at the University of Kansas. His research interests include hardware-software codesign and reconfigurable hardware for real-time and embedded systems. Ortiz has a BS and an MS in computer engineering from the University of Kansas.

Ed Komp is a research engineer at the Information and Telecommunication Technology Center at the University of Kansas. His research interests include specialized computer language design for application-specific domains, functional programming, software development environments, and networking. Komp has a BA in mathematics and an MS in computer science from the University of Kansas.

Peter Ashenden is director of Ashenden Designs Pty Ltd. His research interests include

electronic design automation and computer architecture. Ashenden has a BS and a PhD in electrical engineering from Adelaide University, South Australia. He is a senior member of the IEEE and a member of the ACM.

Direct questions and comments about this article to David Andrews, Information and Telecommunication Technology Center, University of Kansas, 2335 Irving Hill Rd., Lawrence, KS 66045-7612; dandrews@ittc.ku.edu. Further information on the Hybrid Threads project can be found at www.ittc.ku.edu/hybridthreads.