Programming Models for Hybrid CPU/FPGA Chips

David Andrews and Douglas Niehaus, University of Kansas
Peter Ashenden, Ashenden Designs

Designers of embedded and real-time systems are continually challenged to meet tighter system requirements at better price-performance ratios. Best-practice methods have long promoted the use of commercial-off-the-shelf components to reduce design costs and time to market, but creating COTS components that are reusable in a wide range of applications remains difficult.

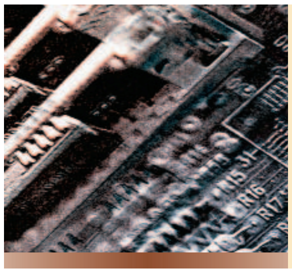
In part, the challenge lies in satisfying the contradictory design forces of generalization and specialization. Systems designers are all too familiar with the tension these opposing forces cause in trying to balance cost versus performance. Adopting COTS components reduces costs and time to market but often fails to meet the most demanding performance requirements; custom-designed components can achieve significantly higher performance but at greater development costs and longer times to market.

HYBRID CPU/FPGA CHIPS

Emerging hybrid chips containing both CPU and field-programmable gate array (FPGA) components are an exciting new development. They promise COTS economies of scale while also supporting significant hardware customization. For example, the Virtex II Pro from Xilinx (www.xilinx.com) combines up to four Power PC 405 cores with up to approximately 4 million free gates, while the Excalibur from Altera (www.altera. com) combines an ARM 922 core with about the same number of free gates.

Designers can configure the free FPGA gates with a widening range of standard FPGA system library components. This intellectual property includes serial and parallel I/O interfaces, bus arbiters, priority interrupt controllers, and DRAM controllers. The IP components let designers select an FPGA IP set to create a specialized system-on-chip. SoC solutions can achieve COTS economies of scale with IP selected to meet specific system requirements.

The free FPGA gates can also support customized application-specific components for performance-critical functions. While FPGA-based implementations do not perform as well as equivalent ASICs, they can often provide acceptable performance with a significantly better price-performance ratio.



Components that combine a CPU and reconfigurable logic gates need a programming model that abstracts the computational hardware.

FPGA COMPONENT SPECIFICATION

Tapping the full potential of these hybrid chips presents an interesting challenge. System developers can replace current COTS board designs with a single hybrid chip, but specifying custom components within the FPGA requires knowledge of hardware design methods and tools that most system programmers do not have.

Researchers are seeking solutions to this problem by investigating new design languages, hardware/software specification environments, and development tools. Projects such as Ptolemy (ptolemy.eecs.berkeley.edu/), Rosetta (sldl.org), and System-C (systemc.org) are exploring system-level specification capabilities that can drive software compilation and hardware synthesis.

Other projects such as Streams-C (rcc.lanl.gov/Tools/Streams-C/index.php) and Handel C (www.celoxica.com) focus on raising the abstraction level for programming FPGAs from gate-level parallelism to modified and augmented C syntax. System Verilog (eda.org/sv-cc/) and a newly evolving VHDL standard (eda.org/vhdl-200x/) attempt to abstract away the distinction between the two sides of the traditional low-level hardware/software interface into a system-level perspective.

Although these approaches differ in the scope of their objectives, they all share the goal of raising the abstraction level required to design and integrate hardware and software components.

Crossing the boundary

A question remains as to whether high-level FPGA programming languages will mature to a point that lets software engineers apply their skills across the CPU/FPGA boundaries.

Unfortunately, current hybrid programming models are still immature. They generally treat FPGAs as independent accelerators with computations outside the scope of the programs running on the CPU. They generally use simple I/O queues for communications between the FPGA-based and CPU-based computations, and the hybrid model is not mature enough for synchronizing the execution of component computations—a critical capability in distributed and parallel computation.

A mature high-level programming model would abstract the CPU/FPGA components, bus structure, memory, and low-level peripheral protocols into a transparent system platform. In general, programming models provide the definition of software components as well as the interactions between these components, as Edward Lee described in a discussion of embedded systems' software frameworks ("What's Ahead for Embedded Software?," *Computer*, Sept. 2000, pp. 18-20).

Message passing and shared-memory protocols are two familiar component interaction mechanisms. Practitioners have successfully used both in embedded systems and enjoy debating the relative merits of their personal choice. We describe the multithreaded shared-memory model here, though many aspects of the discussion are equally appropriate for the message-passing model.

MULTITHREADING MODEL

The multithreaded programming model is convenient for describing embedded applications composed of concurrently executing components that synchronize and exchange data. Its popularity is apparent in the widespread use of the Posix threads standard.

High-level abstraction

The multithreaded programming model specifies applications as sets of threads distributed flexibly across the system CPU and FPGA assets. At the highest abstraction level, the computational structure of hybrid applications remains familiar. Whether the threads implementing a computation are CPU- or FPGA-based becomes just another design and implementation parameter with resource use and application performance implications.

The programming model suports iterative application development.

How to perform this partitioning to best support application or system requirements is yet another challenging problem. However, the model supports iterative application development that begins with an exclusively CPU-based multithreaded implementation and gradually transferring specific threads to FPGA support.

All of these attributes speed the time to market. They also let designers focus FPGA support on those portions of the application that performance measurements indicate will benefit most from it.

Policy and mechanisms

We can draw a useful distinction between policy and mechanism. The multithreaded model *policy* is fairly simple: to allow the specification of concurrent execution threads and protocols for accessing common data and synchronizing the execution of independent threads. The *mechanisms* that a general-purpose processor uses to achieve this policy include the definition of data structures that store thread execution state information and the semantics of thread synchronization interactions with the operating system thread scheduler.

Both the synchronization and the thread-scheduling portions of the system software access data structures for semaphore control and thread context. In addition, most microprocessors provide the minimum hardware support required to implement atomic semaphore operations—for example, test-and-set or compare-and-swap support.

The FPGA computational model is expressed at an abstraction level different enough from that of the CPU to leave no immediately obvious equivalent to the CPU thread context of register set, program counter, and stack pointer. Additionally, current FPGA technology synthesizes the data paths and operations that represent the thread computations and maps them into the FPGA before runtime.

These differences require new mechanisms for achieving the basic multithreaded model policy relative to threads running within the FPGA and threads interacting across the CPU/FPGA boundary. Although the lack of an existing computational model seems to be a liability at first glance, it actually presents an opportunity to create efficient mechanisms for implementing FPGA threads and for supporting thread synchronization.

OPERATING SYSTEM CODESIGN

A key challenge in giving programmers access to these new hybrid computations is extending the operating system across the CPU/FPGA boundary in a form that abstracts the differences between the computational models used within the CPU and FPGA components.

Operating systems provide the underlying synchronization and control mechanisms for higher-level programming models as well as a generic set of interfaces through which application programs can access system functions. These OS functions relieve application programmers from needing to know the low-level protocols and device-specific requirements. For hybrid systems, this functionality will require hardware/software codesign across the CPU/FGPA boundary.

System developers have always regarded OS codesign as a means to increase system performance through parallelism and to improve the predictability of system behavior. Thus, the codesign work required to enable a hybrid programming model can also enhance general OS performance.

Function migration to FPGA domain

A wide range of OS functions will certainly benefit from either a partial or a complete migration into the FPGA domain. Examples include receiving and evaluating interrupts, time keeping, event queue management, task scheduling, clock synchronization, support for distributed computation, and concurrency control. FPGA support can increase accuracy, performance, and predictability.

System benefits

In this context, FPGAs provide a means to refine system properties such as the resolution and precision of the system time standard. This translates to finer event scheduling, a fundamental aspect of any real-time system.

FPGA codesign can also reduce scheduling jitter, or variable delays of the scheduling decision. Jitter can originate from several sources, including variable execution time of system software, existing mechanisms used to implement interrupt-handling methods, task scheduling, and concurrency control. FPGA-based implementation of some of these OS functions will significantly reduce system overhead by transferring computational loads from the CPU into FPGA-based concurrent state machine components.

These and many other possibilities make codesign of OS functions an exciting area of current research.

System-level multithreaded programming requires new hardware/software codesign approaches that support operating system and application functions. Current efforts to develop this capability will make hybrid CPU/FPGA computational components accessible to a much broader community of system programmers, increasing OS performance and reducing design times and development costs. ■

Acknowledgment

The work is partially sponsored by NSF EHS contract # CCR-0311599.

David Andrews *is an associate professor of electrical engineering and computer science at the University of Kansas. Contact him at dandrews@ittc.ukans.edu.*

Douglas Niehaus *is an associate professor of electrical engineering and computer science at the University of Kansas. Contact him at niehaus@ittc.ukans.edu.*

Peter Ashenden *is director of Ashenden Designs. Contact him at peter@ashenden.com.au.*

Editor: Wayne Wolf, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, 08544-5263; wolf@princeton.edu