

Using the Multi-Threaded Computation Model as a Unifying Framework for Hardware-Software Co-Design and Implementation*

Douglas Niehaus and David Andrews
EECS Department
Information and Telecommunication Technology Center
University of Kansas
{niehaus,dandrews}@eecs.ku.edu

Abstract

The range of distributed real-time and embedded (DRE) system applications has continued to expand at a vigorous rate. Designers of DRE systems are constantly challenged to provide new capabilities to meet expanding requirements and increased computational needs, while under pressure to provide constantly improving price/performance ratios and shorter times to market. Recently emerging hybrid chips containing both CPUs and FPGA components have the potential to enjoy significant economies of scale, while enabling system designers to include a significant amount of specialization within the FPGA component. However, realizing the promise of these new hybrid CPU/FPGA chips will require programming models supporting a far more integrated view of CPU and FPGA based application components than that provided by current methods. This paper describes methods we are developing for supporting a multi-threaded programming model providing strongly integrated interface to CPU and FPGA based component threads.

1. Introduction

The set of applications using distributed real-time and embedded (DRE) system components has continued to expand at a vigorous rate. System designers are constantly challenged to provide new capabilities to meet the expanding requirements and increased computational needs of new application types, while under pressure to provide constantly improving price/performance ratios and shorter times to market. One way to reduce cost and development time is to emphasize reuse of commercial off the shelf

(COTS) components, both hardware and software, in preference to custom designs. Creating COTS components that can be reused in a wide range of real-time and embedded applications is a difficult challenge, in part, because it requires the simultaneous satisfaction of apparently contradictory design forces: generalization and specialization. This paper describes our current work using the multi-threaded computation model as a unifying framework for hardware-software co-design of DRE applications. We believe this approach provides an effective resolution of the tension between generalization and specialization by making multi-threading the first-level design view of a computation, making the CPU or FPGA implementation for a given thread a more secondary and well encapsulated consideration.

1.1. Specialization-Generalization Tension

COTS components are generalized to make them useful for as wide an audience as possible in order to maximize their economies of scale, but this limits the specialized support that they can offer for any given computation. DRE system designers are attracted to the lower cost and shorter times to market of COTS components. However, DRE system designers also often use specialized components to provide provide unusual capabilities required by a given application or to meet stringent performance constraints. This usually forces them to accept higher component costs and longer times to market as a result. This creates a tension between the forces favoring use of generalized components and those favoring specialized components.

We believe two recent trends, one at the hardware and one at the software level, provide an opportunity to improve the ability of DRE system developers to address this tension between generalization and specialization. At the hardware level, recently emerging hybrid chips containing both CPUs and FPGA components have the potential to enjoy

* This work partially supported by NSF EHA Contract CCR-0311599

significant COTS economies of scale, while enabling system designers to include a significant amount of hardware specialization within the FPGA component. At the software level, the increasing popularity of open source component oriented system software offers a similar combination of COTS economies of scale and considerable scope for component selection and specialization. We believe the convergence of these trends offers an exciting opportunity to advance the state of the art in DRE system implementation.

The Xilinx Virtex II Pro and Altera Excalibur chips are two examples of the CPU/FPGA hybrids in which we see such promise. The Xilinx [21] Virtex II Pro combines up to four Power PC 405 cores with up to several million free gates [20, 19], while Altera [2] offers the Excalibur, which combines an ARM 922 core with approximately the same number of free gates [3, 4]. At this level specialization takes two forms: FPGA Intellectual Property (IP) component selection and specialization. The free FPGA gates can be programmed with a wide range of standard system components, including: serial and parallel I/O interfaces, bus arbiters, priority interrupt controllers, and DRAM controllers. Designers select the set of FPGA IP, configuring and perhaps modifying them to create what is, in effect, a specialized System-on-a-Chip (SoC) processor, while enjoying the economies of scale of a COTS device. Further, the free FPGA gates may also be used to support customized application specific components for performance critical functions. While the performance of FPGA based implementations is non-trivially lower than that of an equivalent ASIC, the FPGA based solution often provides perfectly acceptable performance with a better price/performance ratio.

Similar component oriented flexibility is a continuing trend at the software level as well. Under such systems, the specific set of components required by the application can be selected, just as with FPGAs, and in the case of open source software the implementation of components can also be modified as required. For example, the ACE middleware is a collection of components that has had many years of steadily increasing popularity because it is continuously evolving set of components of use to designers of concurrent, distributed, and real-time systems [12, 13, 14].

At the computation level, DRE developers have always had to choose which portions of the application computation should be supported directly by hardware and which by software executing on a CPU. The new hybrid components have greatly increased the size of this design space by increasing the size of the set of application components that can be supported either by the FPGA hardware or by the CPU.

The Open Source trend is significant in this context because it helps increase the DRE system designers' set of choices concerning how to resolve the tension between specialization and generalization. The Open Source trend is at-

tractive to DRE designers because, in principle, it permits them to focus specialization effort on precisely the system components requiring it. Many factors affect the practicality of specialization of a given component within a given situation, of course. The cost of creating and then maintaining a specialized version of an Open Source component can be non-trivial, but so can the cost of a supplier removing a proprietary product and its support from the marketplace.

It is worth noting that the Open Source trend is also present at the FPGA level. For example, Opencores.org [11] provides a wide range of freely available FPGA IP. As with all Open Source projects, the quality and applicability of the available components to a given project varies considerably. It is important to note that commercially produced and licensed FPGA IP can also provide a similar degree of flexibility, by permitting the modification of the licensed IP to address specialization required by a specific project.

1.2. HW/SW Co-Design

The trend toward highly configurable component oriented offerings at both the hardware and system software levels has thus significantly increased the range of choices available to DRE system designers. While providing many advantages, this also significantly complicates DRE system design and implementation because an application's implementation may involve the co-design and integration of a wide range of components in both hardware and software. The situation is further complicated by the fact that during DRE system design and implementation the boundary between components supported by the CPU and those supported by FPGA IP must remain fluid to produce the best resolution of the tension between generalization and specialization. While no single technique or technology is capable of addressing such a fundamental design force, this paper describes an approach we are using to implement a new system software capability that we believe will significantly improve the flexibility and efficiency with which DRE system developers can co-design the HW and software for DRE systems.

The key element of our approach is to provide an environment emphasizing a uniform view of hybrid computations; those involving the use of both CPU based and FPGA components. Our approach uses the familiar, widely used, and widely ported multi-threaded programming model to generalize the description of a computation as a set of concurrently executing threads of execution. The key point is that we are minimizing the distinction that developers must make between threads of execution that are supported by the CPU, by the FPGA, or by both.

We are thus designing and implementing a *hybrid* multi-threaded programming model by standardizing support for CPU and FPGA based threads of execution. In most situa-

tions, developers need only view a computation component as an independent thread of execution interacting with other concurrently executing threads through standard semaphore based synchronization methods.

Given this model, developers requiring specialized support from the FPGA for some portion of an application can begin by separating that portion from the rest of the application as an independent thread. Use of semaphores in a variety of ways can implement the desired relationship between the new thread and those already existing. Then, when this implementation is working satisfactorily in the software domain, the segregated thread can be switched to an FPGA based implementation providing the required unique capabilities or improved performance.

The unified concurrency control of CPU and FPGA based threads in any combination is the most important aspect of our approach. It substantially reduces the set of situations in which a developer must consider the degree of CPU and FPGA support for the thread, and thus increases the flexibility with which application components can migrate across the CPU/FPGA boundary, either during development, or as a system configuration option. Our approach unifying CPU and FPGA implementations under the multi-threaded programming model helps resolve the generalization-specialization tension by making it easy to use generalized components enjoying economies of scale, while permitting focused specialization through FPGA support of specific threads.

Under our unified model, designers also have the advantage of an iterative development approach that can begin by implementing an application as a purely CPU based multi-threaded application, perhaps interacting with a simulation of target hardware that is not yet available. Then, migration of the multi-threaded application to the target platform, and of specific threads to either partial or complete FPGA support, can be done iteratively. Stepwise refinement of a working application is a familiar way to increase overall efficiency by simplifying the resolution of problems that arise at any given stage.

It is important to note that our approach is not a panacea; implementation of applications using a set of concurrently executing threads will remain a difficult challenge because concurrent programming is hard. Further, iterative refinement of an application will not always be as easy as migrating a single thread from CPU support to full or partial FPGA support. However, this is also true of purely CPU based multi-threaded application development, and thus adds no complexity to the overall development problem, while providing a well defined migration path from CPU to FPGA support.

While relevant for design of any computer based system, we are particularly interested in the application of our hybrid model to real-time embedded systems, and are using

KURT-Linux [15] as our development platform. It is important to note that we are investigating the advantages of co-design at the system software level as well as the DRE application level.

1.3. Summary

The attractive combination of economies of scale with significant specialization potential makes these hybrid CPU/FPGA chips likely to play an important role in the creation of COTS hardware platforms for future DRE systems. A major challenge that remains is the creation of a system design and implementation environment within which whether a component is supported by the CPU or the FPGA is just one of many implementation details.

The rest of this paper discusses the hybrid multi-threaded programming model with which we will support co-design in Section 2, and then discusses various ways in which we plan to use co-design to improve the behavior of Kurt-Linux in Section 3 in the course of which we will both test the components of the hybrid model and gain experience with their use. Section 4 then considers the application level examples we will use to exercise the approach at the user level. Section 5 then discusses some related work, and Section 6 briefly presents our conclusions.

2. Hybrid CPU/FPGA Programming Model

The great promise of reconfigurable computing has made it an attractive topic for a number of years, but the opinions of researchers differ on how that promise might best be realized. Some believe that a wide range of applications can be *completely* migrated into FPGA support. Under this view, the CPU will disappear and developers will write computations that will be combined with others on the same system in a process of synthesis producing a monolithic configuration for a large pool of FPGA computational assets. This view arose fairly early in application areas with a strong DSP component for at least three contributing reasons: (1) FPGA components were particularly well suited to these kinds of computations, (2) the CPU components being replaced were often DSP chips, and (3) input and output data streams for these applications could often be handled directly in hardware, rather than through a supporting operating system.

Our view emphasizes *hybrid* computations, those with both CPU and FPGA based components because we believe that many DRE application domains will continue to include elements that are best supported by CPU based components for at least the next several years. Indeed we see the creation of the Xilinx Virtex II Pro [20] and Altera Excalibur [3] chips as a reflection of this position. We be-

lieve that such hybrid architectures are also important because they maximize the flexibility of developers in choosing whether the components of their application will be CPU or FPGA based. Indeed, it is possible to imagine that members of a family of implementations for a given application might have many components in common, but distribute them across the CPU/FPGA boundary differently. This would be one way to provide customers with several different price-performance points while maximizing component reuse and minimizing development time.

We thus believe that designing a programming model capable of describing computations with both CPU and FPGA based components would be a significant contribution to current practice. The rest of this section first describes the elements of the multi-threaded model that are required to achieve this, and then describes how we are planning to implement a hybrid concurrency control model, support for FPGA based threads, and how threads that use both CPU and FPGA support will work.

2.1. Multi-Threaded Computations

The attraction of the multi-threaded computation model is its combination of simplicity and power. A thread is an independently executing entity tracing a path through an address space it shares with other threads. If these threads do not interact in any way, they can execute without restriction. Interactions among threads may exist for several reasons, some fundamental to an application's semantics and some incidental to a particular implementation. These interactions address a number of different issues, including: measures for protecting the validity of shared data, control of access to specific resources, and control of thread pools.

However, despite the wide range and large number of issues addressed, only a small number of mechanisms are used to constrain thread execution. These execution constraints can all be implemented using semaphores. Creation of a hybrid CPU/FPGA multi-threaded programming model thus crucially depends on the creation of a method for permitting hybrid semaphores to be used by both CPU and FPGA based threads.

Implementing hybrid semaphores that provide correct support for concurrency control to both CPU and FPGA based threads is fairly straightforward. Creating implementations that support efficient execution of both types of threads across the full range of desirable computation architectures will certainly be more challenging, and will probably require a set of implementations from which system designers may choose according to characteristics of their systems. This is not surprising given the range of possible application areas for which hybrid concurrency control will be attractive, and is another situation in which a range of

configurable components is desirable for the design and implementation of DRE systems.

2.2. Hybrid Concurrency Control

Concurrency control among CPU and FPGA based threads depends on a set of semaphores that can be used by both. These hybrid semaphores are mapped into the system's physical address space at a specific location, and can be implemented using either memory (DRAM, SRAM, BRAM) or as memory mapped registers within the FPGA. The key aspect of thread operations on the semaphores is that they must be *atomic*. Many architectures, for example, provide atomic test-and-set operations to support efficient semaphore implementation.

In a system with no configuration constraints, atomic test-and-set to a semaphore in any memory location would be possible from both CPU and FPGA threads. In practice, current platforms, while significantly configurable, are not yet *completely* configurable. For example, on our target platform, the Xilinx Virtex II Pro (V2P), the CPU core is a PPC 405. Careful analysis and several experiments have demonstrated that the CPU signals necessary to implement atomic test-and-set on semaphores shared between CPU and FPGA based threads are not available on the busses available to the FPGA.

This is reasonable since the designers of the V2P were under the familiar pressure to reuse an existing component, the PPC 405 CPU core. Further, it is likely that making these signals available would have been a difficult and expensive task. Also, there are other ways to implement hybrid semaphores. Our first approach will use an FPGA based implementation. Each semaphore will have a set of registers to support the request, release, and ownership inquiry operations. Every thread using a hybrid mutex semaphore, whether FPGA or CPU based, will have a unique ID. Threads wishing to obtain possession of a semaphore in the hybrid set will write their ID into the semaphore's request register, and then wait for their ID to appear in the ownership register. The release register can be used by the thread holding the semaphore to release it. Such an implementation will be appropriate for the basic spin-locks that will then enable implementation of semaphores with more subtle semantics. A major advantage of this approach is that it depends only on the atomicity of reading and writing memory locations, which makes it portable among an extremely wide range of CPU architectures.

Support for blocking semaphores is slightly more complex, since a waiting thread must block and then be unblocked when granted the semaphore. On the CPU side, it is easy to block the requesting thread and run the scheduler to select another. When the semaphore is granted to the

blocked thread, the state of the thread must be changed to indicate it is able to run, and the scheduler invoked again. The simplest approach is to have the FPGA semaphore generate a CPU interrupt, and the CPU interrupt handler changes the thread state and arranges for the scheduler to run. Other designs are possible which make different choices about how much information is kept and used by the FPGA based semaphores and under what conditions the CPU is notified of a semaphore state change using an interrupt. The capabilities offered by the reconfigurable FPGA hardware may make it possible to create forms of semaphore support that increase the efficiency of thread execution, especially on the CPU side, by making it possible to reduce interrupt overhead related to semaphore use.

In the case of an FPGA based thread it might always spin on a given semaphore, as there is nothing else for that FPGA component to do, but they need not. Instead, the FSM controlling the FPGA thread might wait in a blocked state until told by the semaphore support hardware that it had been granted possession of the semaphore.

It should now be reasonably clear that an implementation of common semaphore types should be fairly simple on a wide range of hybrid hardware. The designs we are currently implementing depend only on the atomicity of read and write operations on memory addresses. It is interesting to note that the difference in time scales between the CPU and FPGA based threads will probably favor a queuing implementation, as will the need for the FPGA semaphore controller to keep track of all elements waiting on a given semaphore. Clearly, as with most systems design problems, there are many issues of scalability and efficiency that will motivate the creation of many alternate designs after the first correct implementation.

2.3. FPGA Based Threads

An FPGA based computation executing concurrently with the CPU is already, in essence, an independent FPGA based thread of execution. It is important to note that FPGA based threads will be preallocated, in the sense that their portion of the FPGA will be established during system configuration. This need not be true as dynamically loadable FPGAs become more common and loading times decrease, but is how our first implementation will work. At the API level, FPGA based threads will still be “created”, but at the FPGA level they will be provided with values for any runtime parameters and given permission to run.

What we require to make an FPGA based thread part of the hybrid threading model is a control structure that will permit the FPGA based thread to interact with the hybrid concurrency control provided to the entire thread set by the FPGA based semaphore set. We are using a basic finite state machine (FSM) approach to controlling computations real-

ized in the FPGA. The most basic thread is one that interacts with no others, and the FSM for such a thread has two states: *running* and *waiting*. Transition between these states depends on when the thread is “created” by another during application execution. As mentioned, creation in this context means being given input values and permission to run. An FPGA thread “exits” by entering the waiting state.

FPGA based threads that interact with semaphores during their execution require a slightly more complex FSM with a *blocked* state where they can await their ownership of the relevant semaphore. Clearly, the control component of a semaphore *S* will have to be able to interact with the FSM controlling each FPGA based thread using *S*. Fortunately the V2P FPGA assets include a number of components that will make thread interactions relatively simple to implement.

The previous discussion considered how the computation implemented by the FPGA thread will be *controlled* but not how it would be *created*. Specification of the FPGA based computations will initially be done in VHDL, but we hope to be able to use the results of one or more recent advances in languages and compilers that are enabling the use of higher-level programming language constructs to specify gate-level behavior within FPGAs [6, 5]. These efforts will substantially expand the set of developers who can use FPGAs to support computations beyond those comfortable with the vocabulary of hardware design. In combination with a hybrid programming model, such as that described here, such higher-level approaches to FPGA use would substantially lower the entry cost for designing and building many DRE systems.

2.4. Hybrid CPU/FPGA Threads

So far, we have considered a programming model that was of hybrid character because individual threads could be either CPU or FPGA based. However, it is also possible for an individual *thread* to be hybrid as well. The most obvious example is when an FPGA component is used to provide a specific high performance function that is invoked as a subroutine. In this case, the FPGA component is “passive” in the sense that it awaits activation by a CPU based thread in the desired context.

At the programming model level, the CPU based thread of control activates the FPGA component and waits for the result. From one point of view, a single thread of control exists, and crosses the CPU/FPGA boundary. From another point of view, there are two threads of control that operate as co-routines. Both are reasonable ways to describe the implementation.

One interesting point is that if more than one thread can use such passive FPGA assets, then their allocation must be made safe with respect to concurrent access. More complex

considerations arise when the execution time of the FPGA based portion of the computation is significant. This favors use of a blocking semantics, so the CPU can do some useful work while the FPGA component is executing. All of these variations can be accomplished through fairly simple uses of the basic hybrid concurrency control capabilities discussed earlier.

3. KURT-Linux Co-Design

The purpose of developing the hybrid multi-threaded programming model is to improve support for the development and execution of DRE systems. The hybrid thread programming model requires a new co-design of many parts of existing system software to take advantage of the capabilities of the reconfigurable hardware. Further, some aspects of co-design at the system level are opportunities for early use of some elements of the hybrid model, particularly concurrency control. In this section we discuss some of the system level co-design projects that we are working on and how they relate to the overall goal of improving DRE system behavior.

Our goal in producing hybrid designs for several parts of the OS, using the KURT-Linux system as a working example, is to improve four real-time system performance metrics: (1) Event Response Latency (ERL), (2) Worst Case Execution Time (WCET), (3) ERL variance, and (4) execution time variance. Each of these factors has an important influence on decisions made by designers of DRE systems, and are influenced by a number of factors in the system software architecture and implementation. The ERL of a DRE system is a significant metric of its ability to respond to external events, including timer events used by time triggered systems. WCET of various components is a fundamental parameter of many approaches to DRE system design and scheduling. Changes in system implementation which lower the magnitude of these performance metrics are generally considered good. However, variation in behavior can also increase design complexity. For this reason both ERL and execution time variance are also significant influences on the design of many DRE systems. Many designers are, for example, willing to accept slightly larger values for ERL or WCET if in return the variance of ERL and execution is significantly lowered.

There are several aspects of the system for which we believe co-design implementations will improve system behavior as measured by these metrics, which will interact helpfully with the hybrid multi-threaded programming model, and which will serve as a reasonable set of driving examples for a the concurrency control portion of the hybrid model's implementation. The majority of these form an ordered set addressing time keeping and computation scheduling within the system. Specifically, (1) time standard imple-

mentation, (2) timer interrupt generation, (3) event queue management, (4) task scheduling, and (5) interrupt handler scheduling.

We have already implemented on another platform, and are porting to the V2P, a set of related time standard and timer interrupt designs which covers Steps 1 and 2. In general these provide a time standard of configurable resolution, up to that provided by the underlying FPGA component which provides a 10ns clock. The basic design supports the KURT-Linux model of a jiffy and sub-jiffy timer. Note that a "jiffy" is the basic time unit tracked by the generic Linux kernel. KURT-Linux increases the resolution of time keeping and of event scheduling by using the sub-jiffy, preserving the jiffy, to limit the set of changes to generic Linux required. The timer and event interrupt generator improve time keeping and scheduling accuracy, but do not by themselves improve ERL, WCET, or reduce the variance of either ERL or execution time.

We have recently completed the first version of Step 3, the co-designed event queue. This migrates the task of tracking the set of scheduled events into the FPGA, and uses the time standard and event interrupt components we implemented in Steps 1 and 2. This design established methods by which we could have the FPGA read and write memory, interacting with the KURT-Linux system software. Putting the event queue into the FPGA modestly reduces the system overhead, thus modestly reducing ERL and its variance since the programming of the timer for the next event is now solely in the FPGA.

The first benefit of more than modest magnitude for system overhead and ERL variance comes with Step 4, which will move the selection of the next real-time thread to run into the FPGA. This will be done in two stages. First, a simple scheduling decision function will be used, such as EDF or fixed priority. In the second stage, a group oriented hierarchic scheduling model will be used which groups threads and associates a unique scheduling decision function with each group. the advantage at Step 4 will be in reducing ERL by reducing the WCET of the scheduling decision, as well as the variance of the scheduling decision execution time.

However, Step 5 will provide the greatest benefit to DRE system behavior, because it addresses the single largest influence on ERL and its variance: interrupt processing. We have implemented a modified semantics for interrupt processing in KURT-Linux purely in software, and realized a dramatic decrease in both maximum ERL and its variance. Performing co-design for interrupt handling will have a dramatic effect on DRE behavior because the system will interrupt the CPU only when the current thread *should* be substituted with another according to the scheduling policy, not when this only *might* be true. Other co-design projects will address other issues affecting DRE system behavior are planned, including FPGA support for system performance

data gathering, device driver support, clock synchronization in distributed system, and distributed concurrency control.

4. Applications

We are still considering the specific applications to select for driving examples of application computations. However, one of our students spent 8 weeks at Berkeley recently working on KURT-Linux support for the Embedded Machine (E-Machine) which is a virtual machine abstraction for a portable programming model of real-time and embedded system developed by Henzinger and others [8]. The core idea is that concurrency control and scheduling aspects of a computation can be separated from others and expressed as E-Machine code which arranges for elements of the computations known as *tasks* to be called at the proper times and in the correct contexts. Software development environments operating at a higher level of abstraction, such as Ptolemy [10] and Giotto [7] can use the E-Machine as a compilation target. We also experimented with using KURT-Linux as a direct compilation target using a Ptolemy based real-time audio application.

This preliminary work showed that KURT-Linux could be used as an effective target, and as this type of support progresses, we believe it will be possible to use these applications, particularly the Ptolemy audio application, as driving examples for application level use of the hybrid threading model. The focus in the context of that problem would be to embody portions of the audio processing computations in the FPGA and experiment with various kinds of multi-threaded software architectures.

5. Related Work

Many researchers are working on various aspects of improving the methods used to design and implement DRE systems. Many are investigating new design languages, hardware/software specification environments, and tools. Projects such as Ptolemy [10], Rosetta [1], and System-C [16] are seeking system level specification capabilities that can drive software compilation and hardware synthesis. Other projects such as Streams-C [6] and Handel C [5], are focused on raising the level of abstraction at which FPGAs are programmed from one of gate-level parallelism to that of modified and augmented C syntax. System Verilog [17] and a newly evolving VHDL standard [18] are also now being designed to abstract away the distinction between the traditional low level hardware/software interface to produce 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.

Others recognize the need for a programming model that more completely abstracts the use of underlying FPGA/CPU components, bus structure, memory, and low level peripheral protocols into a more unified system platform. Programming models support the definition of software components and of the interactions between them, see Lee [9] for a discussion of software frameworks for embedded systems.

6. Conclusions

This paper has discussed how recently emerging hybrid chips containing both CPUs and FPGA components have the potential to significantly improve many aspects of distributed real-time and embedded system design and implementation by providing ways to resolve much of the tension between generalization and specialization. The result is likely to be systems that enjoy significant COTS economies of scale, while enabling system designers to include a significant amount of specialization within the FPGA component. This paper also discussed why we believe that to realize this promise of improved productivity, significant improvements in the level of abstraction and integration in the programming environments will be required. Specifically, we discussed the unifying multi-threaded programming environment we are developing that will provide a largely uniform programming interface to both CPU and FPGA based threads, as well as hybrid threads using both types of support. We also discussed how co-design projects at the system software and application levels will serve as driving examples for the techniques required to support the unified multi-threaded programming model.

References

- [1] P. Alexander and C. Kong. Rosetta: Semantic support for model centered systems level design. *IEEE Computer*, 34(11), November 2001.
- [2] Altera. Altera home page. www.altera.com.
- [3] Altera. *Excalibur Device Overview Data Sheet*. http://www.altera.com/literature/ds/ds_arm.pdf.
- [4] Altera. *Excalibur Devices Hardware Reference Manual*. http://www.altera.com/literature/manual/mnl_arm_hardware_ref.pdf.
- [5] Celoxica. Celoxica home page. www.celoxica.com.
- [6] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. *Proceedings of the Eight Annual IEEE Symposium on Filed-Programmable Custom Computing Machines (FCCM)*, pages 49–56, April 2000.
- [7] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

- [8] T. Henzinger and C. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.
- [9] E. Lee. Whats ahead for embedded software ? *IEEE Computer*, pages 18–26, September 2000.
- [10] E. Lee. Overview of the ptolemy project. *Technical Memorandum*, UCB/ERL M01/11 Universit of California(6), March 2001.
- [11] OpenCores. Opencores.org home page. www.opencores.org.
- [12] D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.
- [13] D. C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/~schmidt/ACE.html, 1997.
- [14] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*. Addison-Wesley, Boston, 2002.
- [15] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the shelf hardware and free software. In *Proceedings of the Real-Time Technology and Applications Symposium*, Denver, June 1998.
- [16] SystemC. Systemc home page. www.systemc.org.
- [17] www.eda.org/svcc/.
- [18] www.eda.org/vhdl200x/.
- [19] Xilinx. *Virtex II Pro Platform FPGAs: Functional Description*. <http://direct.xilinx.com/bvdocs/publications/ds083-2.pdf>.
- [20] Xilinx. *Virtex II Pro Platform FPGAs: Introduction and Overview*. <http://direct.xilinx.com/bvdocs/publications/ds083-1.pdf>.
- [21] Xilinx. Xilinx home page. www.xilinx.com.