RECONOS: AN RTOS SUPPORTING HARD- AND SOFTWARE THREADS

Enno Lübbers and Marco Platzner

University of Paderborn
email: {enno.luebbers, platzner}@upb.de

ABSTRACT

Modern platform FPGAs integrate fine-grained reconfigurable logic with processor cores and allow the creation of complete configurable systems-on-chip. However, design methodologies have not kept up with the rise in complexity of the target hardware. In particular, there is little overlap between the programming model for embedded software running on a real-time operating system and the programming model for digital logic.

In this paper, we present the operating system ReconOS which supports both software and hardware threads with a single unified programming model. ReconOS is based on eCos, a widely-used real-time operating system (RTOS). We investigate the incurred time and area overheads, especially for inter-thread communication across the hardware/-software boundary, and present a case study demonstrating the feasibility of the RTOS-centric design approach.

1. INTRODUCTION

The increase in density and complexity of reconfigurable logic, as well as the inclusion of microprocessor cores on FPGAs, has moved reconfigurable devices from their classic roles as glue logic modules, prototyping platforms and ASIC replacement to a viable platform for integrating complete embedded systems-on-chip. Modern platform FPGAs allow both complex control-dominated tasks and data-centric parallel processing tasks to be efficiently implemented on the same device. However, design methodologies for such configurable systems on chip have not kept up with the rise in complexity of the target hardware. In particular, there is little overlap between the programming model for embedded software running on a real-time operating system (RTOS) and the programming model for digital logic.

Most of today's RTOS, e.g., VxWorks [1], RTXC [2], eCos [3], provide the system designer with a set of clearly defined objects and associated services, which are encapsulated in language-specific application programmer interfaces, e.g. POSIX [4]. A few basic primitives, among them *threads*, *semaphores* and *shared memory*, make up the programming model for embedded software development – it may be considered a crude model, none the less is it a well-established one. To support real-time systems, RTOS typically offer dynamic priority-based preemptive scheduling for threads, minimized interrupt latencies, bounded execution times for system calls, and are highly configurable to satisfy small memory footprint requirements.

It seems natural to extend the RTOS approach to include customized hardware cores. Analogous to software threads, a hardware core performing a specific task can be thought of as a *hardware thread*. Covering hardware and software threads with the same programming model requires us to give hardware cores access to RTOS services such as semaphore management or shared memory. We need to investigate to what extent the known RTOS primitives can be applied to (re)configurable hardware, and where they need to be modified or replaced.

To facilitate the design of embedded systems that integrate microprocessors with fine-granular configurable logic, we have been developing ReconOS, an operating system build on top of eCos [3], a widely-used open source RTOS. In ReconOS, software and hardware threads integrate and communicate seamlessly and transparently with the operating system using the same set of operating system services. We believe that such an RTOS-centric approach to integrating hardware cores into a processor-based system eases application development and thus increases productivity and portability.

The main contribution of this paper is the presentation of the novel hardware/software operating system ReconOS, including the programming model for hardware threads and a runtime system for Xilinx Virtex-II-Pro/Virtex-4FX FPGAs.

The paper is organized as follows: In Section 2, we discuss related approaches that provide operating system support for hardware cores. ReconOS is presented in Section 3, including the programming model and the runtime system. In Section 4, we report on operating system overheads and present a case study. Finally, Section 5 concludes the paper and lists future work.

2. RELATED WORK

During the last years, much research in operating systems for reconfigurable hardware has been presented, including

conceptual work, e.g., in [5], algorithms for task and resource management, e.g., in [6, 7], and even sophisticated techniques such as relocation and preemption of hardware tasks, e.g., in [8]. In contrast to this research, we start out with an available RTOS and seek to integrate hardware cores into the system. In this paper, we are not concerned with dynamic and partial reconfiguration. However, we will take up on the previously developed techniques in our future work.

Peck et al. [9] promote thread-level parallelism by generating sequential hardware threads directly from multithreaded C source code. They support a significant subset of the language, along with several POSIX-like operating system (OS) calls related to thread creation and communication. Their *hthreads* project also maps some OS functionality to hardware, resulting in shorter interrupt latencies, a reduced number of CPU context switches, and less jitter [10]. With ReconOS, we follow a slightly different approach. Our hardware threads are not generated from a sequential language, but written in VHDL. In this way, we exploit both thread-level parallelism by allowing multiple hardware threads to run concurrently, as well as low-level parallelism which is key to constructing high-performance hardware cores. Only the part of a hardware thread that controls the OS interaction needs to be specified sequentially.

Modern embedded applications often require standardized interfaces for existing or legacy software and support for commodity hard- and software, such as networking protocols. Bergmann et. al [11] implemented a Linux-based OS for reconfigurable hardware, running on a MicroBlaze soft core CPU. They wrap hardware circuits in software wrappers, called *ghost processes*, so that an unmodified Linux Kernel can interact with them. As a result, hardware circuits are modeled as processes, with their own address space. The preferred method of communication between the CPU and a hardware process relies on the MicroBlaze's Fast Simplex Links and maps them to Linux FIFOs on the file system level [12]. Another Linux-based run-time environment for reconfigurable hardware, called BORPH, has been developed by So et al. [13][14]. BORPH modifies and extends a standard Linux kernel for the PowerPC architecture with a hardware interface, providing conventional UNIX IPC mechanisms to the hardware using a message passing network. Their current research focuses on file system-based data communication between software and hardware processes. In contrast to the Linux-based approaches, ReconOS builds on top of and extends eCos, a widely-used *real-time* OS. In ReconOS, all threads share the same physical memory space. Therefore, hardware threads have direct access to any location in the system's memory, or memory mapped peripherals, if desired. As our research into the semantics of programming model primitives progresses, we will also modify the internals of the eCos kernel. This is greatly facilitated by the modular configurability of eCos.

3. RECONOS

In this section we present the concepts of our programming model for hardware/software systems and the implementation of a runtime system on Xilinx Virtex-II-Pro and Virtex-4 FPGAs. We pursued the following three design goals: First, hardware threads should have access to all relevant operating system services offered by eCos in a transparent way. For example, communicating threads need not know whether their communication partners are executed as hardware or software threads. Second, the model should support true parallel execution, i.e., the concurrent execution of one software thread on the CPU and several hardware threads mapped to reconfigurable logic. Third, we wanted to offer a way to integrate existing hardware IP cores as hardware threads into our system. Thus, it should be possible not only to re-use existing software by leveraging standard APIs supported by eCos, but also existing hardware IP by supplementing it with a proper operating system interface.

3.1. Programming Hardware Threads

Software threads have sequential execution semantics. To use an operating system service, a software thread simply calls the corresponding function in the operating system library. Hardware tasks, on the other hand, are inherently parallel. Mostly, there is no single control flow and, thus, no apparent notion of calling an operating system function. In particular, typical hardware description languages, such as VHDL, offer no built-in mechanism to implement *blocking calls*.

To present as unified a programming model as possible to the user, we rely on the following approach: We structure a hardware thread such that all interactions with the operating system are managed by a single sequential state machine. To this end, we have developed an operating system function library for VHDL. This library contains code implementing the system call signaling wrapped into VHDL procedures, e.g., *reconos_sem_wait()*. Together with the operating system interface (OSIF), a separate synchronizing logic module serving as the connection between the hardware thread and the OS, these procedures are able to establish the semantics of blocking calls in VHDL. A hardware thread thus consists of at least two VHDL processes: the synchronization state machine and the actual user logic. The state transitions in the synchronization state machine are always dependent on control signals from the OSIF; only after a previous operating system call "returns", the next state can be reached. Thus, the communication with the operating system is purely sequential, while the processing of the hardware thread itself can be highly parallel. It is up to the programmer to decompose a hardware thread into a collection of user logic modules and one synchronization state machine. Besides the increased complexity due to the par-

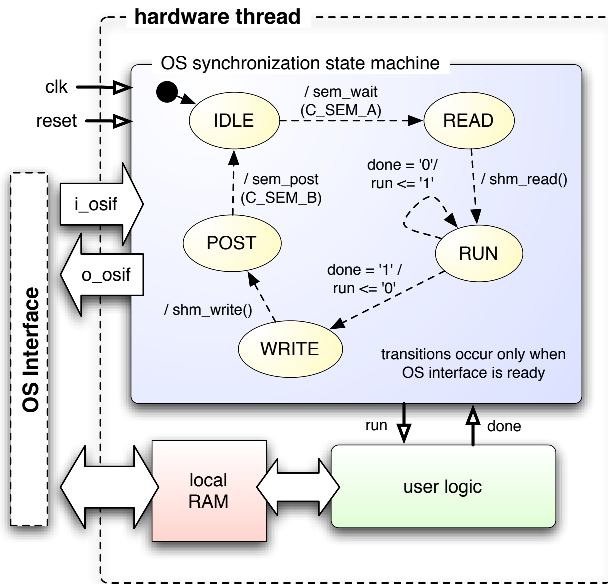

Fig. 1. Example of an OS synchronization state machine

allel nature of hardware, this process is no different from programming a software thread.

An example demonstrating this mechanism is illustrated in Figure 1. In this example, the hardware thread waits on a semaphore (*C_SEM_A*), reads a block of data from shared memory into a local RAM, processes it, writes the result back to shared memory, and then posts another semaphore (*C_SEM_B*). The OS synchronization state machine and the user logic communicate via the two handshake signals *run* and *done*. Listing 1 shows the corresponding VHDL implementation of the synchronization state machine, using ReconOS system calls.

To further exemplify the underlying mechanism, consider the following sequence of events. Upon reaching the state *IDLE*, the VHDL procedure *reconos_sem_wait()* asserts the appropriate handshake signals in the OSIF to signal a ReconOS "semaphore wait" call. The *state* signal is set simultaneously to the next state, *READ*. However, the OSIF immediately asserts a *blocking* signal, indicating that the request is being processed. On the next rising clock edge, the blocking signal, evaluated in *reconos_ready()*, prevents the synchronization state machine from entering the *READ* state. Only after the operating system call returns, the OSIF will deassert the blocking signal which allows the synchronization state machine to complete the state transition.

Listing 1. Code for the example of Figure 1

```
1  osif_fsm: process(clk, reset)
2  begin
3    if (reset = '1') then
4      state <= IDLE;
5      run <= '0';
6      reconos_reset(o_osif, i_osif);
7    elsif rising_edge(clk) then
8      reconos_begin(o_osif, i_osif);
9      if reconos_ready(i_osif) then
10       case state is
11         when IDLE =>
12           reconos_sem_wait(o_osif, i_osif, C_SEM_A);
13           state <= READ;
14
15         when READ =>
16           reconos_shm_read_burst(o_osif, i_osif,
17                                  local_address,
18                                  global_address);
19           state <= RUN;
20
21          when RUN =>
22           run <= '1';
23           if done = '1' then
24             run <= '0';
25             state <= WRITE;
26           end if;
27
28         when WRITE =>
29           reconos_shm_write_burst(o_osif, i_osif,
30                                   local_address,
31                                   global_address);
32           state <= POST;
33
34         when POST =>
35           reconos_sem_post(o_osif, i_osif, C_SEM_B);
36           state <= IDLE;
37
38         when others => null;
39       end case;
40     end if;
41   end if;
42 end process;
```

It should be noted that the local RAM is optional; single-word bus access is possible through the OS interface.

3.2. System Architecture and Runtime System

We have implemented a runtime system that can execute both software and hardware threads concurrently on Virtex-II Pro or Virtex-4FX FPGAs. Figure 2 shows the overall system architecture. Hardware threads are placed in the reconfigurable fabric; software threads together with the RTOS kernel are executed on the devices' embedded PowerPC 405 CPU. The system is based on an IBM CoreConnect bus topology, which connects the CPU to system peripherals, such as memory controllers, timers and I/O, and the OS interfaces associated with the hardware threads. Design, synthesis and simulation of the hardware is done with the Xilinx Embedded Development Kit and supplementary custom-built tools for thread instantiation and integration, as well as a hardware thread simulation environment.

eCos, which is used as the underlying software RTOS, is configurable on an extremely fine-grained level. The designer selects the modules required for a particular application (e.g., a bitmap scheduler, POSIX API support, device drivers, a TCP/IP stack, and the ReconOS hardware interface support), configures them for the particular target platform, and builds a customized static library. Software threads are then written in C or C++, using the eCos or another supported standard API for operating system calls. To-

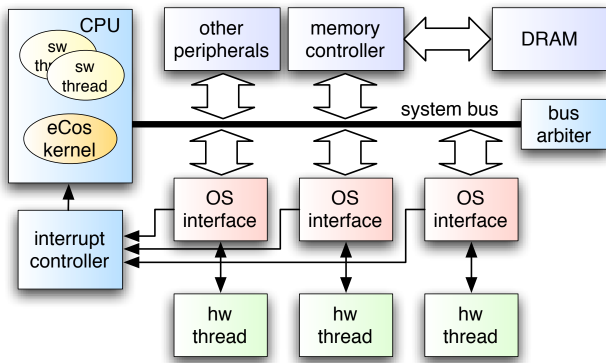

Fig. 2. ReconOS system architecture

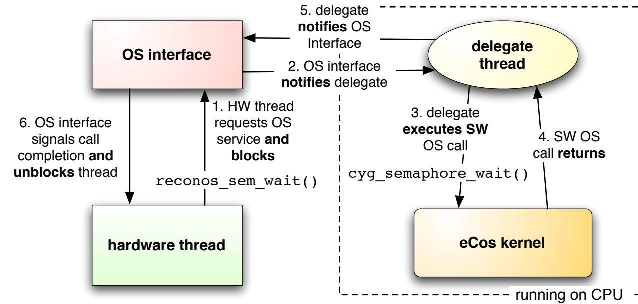

Fig. 3. OS call sequence

gether with initialization code for system startup and thread creation, the threads are compiled and linked against the customized eCos library, resulting in a single executable image to be loaded into system memory.

Hardware threads are written in VHDL, using the OS call mechanisms described in Section 3.1, and connect to the OS interface as shown in Figure 1. This OS interface contains the busy/blocking handshaking logic and implements both master and slave interfaces to the system bus, providing a hardware thread with access to any memory region or memory mapped peripheral in the system. Thus, hardware threads that access shared memory do not generate any CPU load. Calling an operating system function, however, requires the execution of the eCos kernel. To this end, each OS interface module can raise a hardware interrupt. The OS interfaces and the interrupt controller module are instantiated automatically by our build system.

At run-time, hardware threads are created similar to regular software threads. For each hardware thread, a new dedicated eCos thread, the *delegate*, is created and connected to the corresponding OS interface. Whenever an OSIF raises an interrupt, the corresponding delegate is scheduled for execution. The delegate thread receives the interaction requests from the OS interface, translates them into their eCos equivalents, and executes them on behalf of the hardware thread. In our current ReconOS implementation, all hardware threads are statically configured into the FPGA and can run in parallel. While there is no need to schedule hardware threads, the priority of a hardware thread is set by assigning a priority to the associated delegate thread and its OSIF interrupt line.

An example sequence for a semaphore wait OS call is shown in Figure 3. The hardware task requests the service by executing the corresponding VHDL procedure, i.e., *reconos_sem_wait()*, which blocks the hardware thread's OS synchronization state machine. The OS interface then raises an interrupt, which is forwarded to the CPU by the system's interrupt controller. There, a corresponding interrupt handler wakes up the associated software delegate thread, which queries the OS interface across the system bus as to which OS call was requested. If the call in question is non-blocking, the OS interface deasserts the blocking line, allowing the hardware thread to continue. Otherwise, blocking continues until the request is served. Again, blocking affects only the OS synchronization state machine – in principle, parts of the user logic in the hardware thread may run continuously. The OS call is then executed by the delegate thread using the standard eCos API, e.g., *cyg_semaphore_wait()*. After that call returns, the OS interface is notified, deasserts the blocking signal and thus allows the hardware thread to continue execution. This execution model provides a great deal of flexibility and basically enables a hardware task to use any operating system primitive accessible by software, albeit at the cost of the OS' interrupt handling latency and an additional context switch to the delegate thread.

4. CASE STUDY AND EXPERIMENTAL RESULTS

The efficiency of an application running on an embedded real-time operating system hinges on the efficiency of the OS services' implementations, especially when running applications with significant inter-thread communication and synchronization. We have performed intensive tests with synthetic hardware and software threads performing system calls. In particular, we have analyzed the average latencies of synchronization operations between several combinations of software and hardware threads using semaphores. The tests have been conducted on a Xilinx XUPV2P board equipped with an XC2VP30-7 FPGA. The embedded CPU has been run at 300 MHz, while the system bus, memories, peripherals and hardware threads have been run at 100 MHz. The area overhead introduced by the operating system is mainly given by the OS interfaces attached to the system bus. In our implementation, an OS interface including the system bus interfaces amounts to 1051 slices, which constitutes about 7 % of the used FPGA.

Table 1. Average latencies of semaphore operations

test	data cache enabled	data cache disabled
semaphores (turnaround)		
SW-to-SW	7.69 μs	78.92 μs
SW-to-HW	13.84 μs	93.91 μs
HW-to-SW	27.13 μs	168.16 μs
HW-to-HW	34.19 μs	211.96 μs
non-blocking OS call (semaphore post)		
SW	1.59 μs	14.51 μs
HW	16.51 μs	161.09 μs

The results are listed in Table 1. The first set of measurements shows the semaphore turnaround time, the time taken between one thread posting a semaphore and another thread returning from a wait on that semaphore. Since all operating system calls are handled by software, it is expected that calls from hardware threads exhibit latencies inferior to the same calls issued from software. A software-to-software semaphore synchronization involves only one CPU context switch from the posting to the waiting thread. If the waiting thread is running in hardware, it has to be notified across the system bus, which takes additional time. As described in Section 3.2, any OS call from hardware involves calling an interrupt handler and an additional CPU context switch, as well as extra bus access to the OS interface's registers. This is confirmed by the second set of measurements shown in Table 1, listing the latencies of non-blocking OS calls such as a semaphore post. It is further interesting to note that disabling the CPUs data cache incurs a severe penalty on both software and hardware threads.

While ReconOS offers a simple and flexible way of providing operating system services to hardware threads, the experiments demonstrate clearly that there is a penalty involved. Nevertheless, designers are more likely to use the precious logic resources for heavy data-parallel processing rather than implementing synchronization-intensive control dominated code. Under this premise the synchronization latencies are within reasonable bounds. To minimize synchronization latencies for hardware threads, one could either implement the delegate's functionality directly into the eCos kernel and save one context switch, or map the semaphore management together with the scheduler into hardware. The first option requires a modification of the eCos kernel; the latter leads to a massive reorganization and hardware/software partitioning of the kernel which is pursued in [10].

Next, we present a more elaborate case study to show the feasibility of system design based on an operating system for hardware/software threads. Grayscale image data is acquired from a webcam and streamed into the embedded target system through Ethernet, using a TCP/IP stack running on eCos. The image data is then run through a convolution filter (in this case, a $w \times w$ Laplacian edge detection kernel), and subsequently copied to a framebuffer for display on an external monitor. The application consists of three threads, a capture, a filter and a display thread, as depicted in Figure 4. Data is passed between the threads through shared memory, while semaphores synchronize access to the memory and enforce a sequential scheduling of the threads.

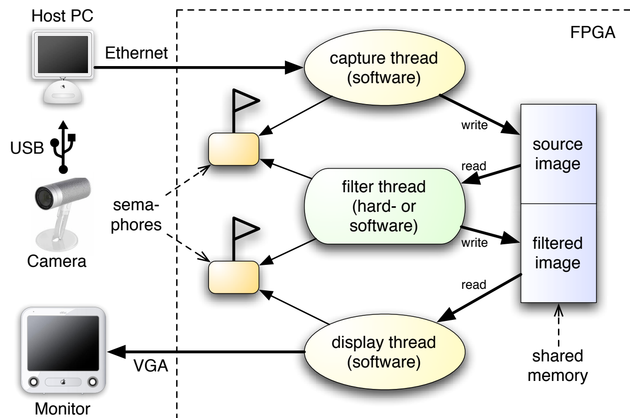

Fig. 4. Image processing case study

Table 2. Raw execution times and theoretical speedup

w	filter	capt.	filter	disp.	total	S
		raw execution time [ms/frame]				
3	SW	16.0	23.9	22.5	62.3	1.4
	HW		6.0		44.4	
5	SW	16.0	86.8	22.5	125.3	2.7
	HW		7.6		46.1	

The application was first implemented and tested purely in software. Then, we have coded the Laplacian in VHDL and turned it into a hardware thread. Convolution filters are amenable to parallelization which promises a considerable performance boost if the filter thread is moved to hardware. Again, the CPU has been operated at 300 MHz and the rest of the system at 100 MHz.

Table 2 lists the execution times in ms per frame for the different threads and Laplacian kernel sizes, excluding any overhead due to OS calls. It can be seen that the hardware filter thread outperforms its software counterpart by a factor of 3.98 and 11.42 for a 3×3 and 5×5 kernel, respectively. The last column of Table 2 presents the theoretically attainable speedup S. To process a frame, all three threads need to execute in sequence. Hence, the theoretical speedup is fundamentally limited by Amdahl's law. In practice, the theoretical speedup will not be reached due to the overhead of the OS.

Table 3. System performance in [frames/s]

w	filter	block size 4	8	20	40	80
3	SW	14.4	15.5	16.1	16.2	16.3
	HW	15.5	17.6	18.6	19.0	18.9
Speedup		1.08	1.14	1.16	1.17	1.16
5	SW	8.1	8.3	8.5	8.5	8.5
	HW	15.3	17.0	18.4	18.6	18.6
Speedup		1.89	2.05	2.16	2.19	2.19

The application was run with differently sized blocks of data. Larger block sizes reduce the system call overhead for semaphore synchronization, but require more shared memory. Table 3 lists the resulting performance figures in frames per second for different Laplacian kernel sizes and for software and hardware filter threads. The data shows that increasing the block size to more than 20 image lines does not result in a significantly higher performance. Further, we observe that by switching from a 3×3 to a 5×5 Laplacian kernel the software filter's performance drops dramatically while the hardware filter can exploit more parallelism and delivers an almost constant performance. Finally, we see that the resulting overall speedups are quite close to the theoretically achievable speedups of Table 2. This points to an acceptable overhead of the ReconOS system calls.

Naturally, the performance of the application could be further improved by allowing concurrent execution of multiple threads, using a double-buffering of data blocks, or by implementing another thread in hardware. However, the case study served to demonstrate that by moving data-intensive threads to hardware *while maintaining the underlying programming model* – and thus making changes to the remaining parts of the system unnecessary –, appealing performance increases can be achieved.

5. CONCLUSION AND FUTURE WORK

We have presented the novel operating system ReconOS, which supports the execution and transparent interaction of hardware and software threads on a configurable system-on-chip. Time and area overheads have been quantified and the practicability of the proposed RTOS-centric design approach has been demonstrated by a case study.

Future work will be conducted along two lines: First, we will investigate partial reconfiguration to load and execute hardware threads dynamically. So far, all applications implemented on ReconOS have used static hardware threads placed during system synthesis. Second, we intend to optimize the communication between hardware threads, and between hardware threads and the operating system objects running on the CPU. For example, we could provide separate communication channels for physically adjacent hardware threads. This promises higher throughput and shorter latency together with a reduction of the load on the system bus.

6. ACKNOWLEDGEMENT

This work was supported by the German Research Foundation under project number PL 471/2-1.

7. REFERENCES

[1] "VxWorks 6.x," Website, 2007, http://www.windriver.com/products/run-time_technologies/Real-Time_Operating_Systems/VxWorks_6x/.

[2] "RTXC 3.2 real-time kernel," Website, 2007, http://www.quadros.com/products/operating-systems/rtxc-32/.

[3] "eCos," Website, 2007, http://ecos.sourceware.org/.

[4] IEEE and The Open Group, "The Open Group Base Specifications Issue 6, IEEE Std. 1003.1, 2004 Edition."

[5] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," in *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, June 2003, pp. 284–287.

[6] K. Danne, R. Muehlenbernd, and M. Platzner, "Executing hardware tasks on dynamically reconfigurable devices under real-time conditions," in *Proc. IEEE 16th Int. Conf. Field Programmable Logic and Applications*, 2006, pp. 541–546.

[7] C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1392–1407, November 2004.

[8] H. Hinkelmann, A. Gunberg, P. Zipf, L. S. Indrusiak, and M. Glesner, "Multitasking Support for Dynamically Reconfigurable Systems," in *Proc. IEEE 16th Int. Conf. Field Programmable Logic and Applications*, August 2006, pp. 219–224.

[9] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "hthreads: A Computational Model for Reconfigurable Devices," in *Proc. IEEE 16th Int. Conf. Field Programmable Logic and Applications*, vol. 1, August 2006, pp. 885–888.

[10] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, "Run-Time Services for Hybrid CPU/FPGA Systems on Chip," in *Proc. IEEE 27th Int. Real-Time Systems Symposium*, 2006, pp. 3–12.

[11] N. W. Bergmann, J. A. Williams, J. Han, and Y. Chen, "A Process Model for Hardware Modules in Reconfigurable System-on-Chip," in *Workshop Proc. 19th Int. Conf. Architecture of Computing Systems*, vol. 81, March 2006, pp. 205–214.

[12] J. A. Williams, N. W. Bergmann, and X. Xie, "FIFO Communication Models in Operating Systems for Reconfigurable Computing," in *Proc IEEE 13th Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, 2005, pp. 277–278.

[13] H. K.-H. So and R. W. Brodersen, "Improving Usability of FPGA-based Reconfigurable Computers through Operating System Support," in *Proc. IEEE 16th Int. Conf. Field Programmable Logic and Applications*, 2006, pp. 349–354.

[14] H. K.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/-software runtime environment for FPGA-based reconfigurable computers using BORPH," in *CODES+ISSS '06: Proc. 4th Int. Conf. Hardware/software Codesign and System Synthesis*. ACM Press, 2006, pp. 259–264.