A PORTABLE ABSTRACTION LAYER FOR HARDWARE THREADS

Enno Lübbers and Marco Platzner

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

ABSTRACT

The multithreaded programming model has been shown to provide a suitable abstraction for reconfigurable computers. Previous implementations of corresponding runtime systems have been limited to a single host operating system, hardware platform, or application domain.

This paper presents the implementation of ReconOS, our hardware/software multithreaded programming model, on both eCos and Linux-based host systems as well as on PowerPC and MicroBlaze CPUs. This demonstrates that ReconOS provides a truly portable abstraction layer for programming reconfigurable computers. Further, we quantify the performance of operating system calls and measure the resulting application level performance.

1. INTRODUCTION

Due to the lack of pervasive high-level programming abstractions most reconfigurable hardware accelerators are implemented as passive coprocessor-like extensions invoked via procedure calls, which does not fit well with the increasing densities and capabilities of reconfigurable fabrics. A promising approach toward a higher-level programming abstraction is to express applications as a collection of threads and to manage the reconfigurable resources from an operating system layer. While earlier work focused on concepts and single runtime management functions such as placement and scheduling, the focus now lies on a true embedding of hardware circuits as independent execution units into the programming abstraction offered by a host operating system (OS). Lately, extensions of Linux-based OS have emerged that promote integration of software threads and hardware processes. Kosciuszkiewicz et al. [1] build on top of an existing Linux kernel and try to model hardware tasks as a drop-in replacement for software tasks. Processes executing in the FPGA's fabric appear as regular threads to the operating system kernel. However, the existing implementation is limited to thread interactions via FIFOs only, and does not exploit the fine-grained parallelism of FPGAs, but maps threads to be executed in hardware to synthesized processors. Xie et. al. [2] support a similar heterogeneous multiprocessor system consisting of soft processor cores, synthesized to an FPGA, with Linux OS services. Bergmann et al. [3] wrap arbitrary hardware circuits in software wrappers, so called ghost processes, that provide a transparent interface for interactions from the kernel and other processes. Their approach considers sharing the same address space between hard- and software processes as unsuitable and hence uses processes instead of threads to encapsulate hardware modules. So et al. [4][5] modify and extend a standard Linux kernel with a hardware interface, providing conventional UNIX IPC mechanisms to the hardware using a message passing network. On the operating system level, they also map inter-thread communication to FIFOs.

These approaches try to connect processes implemented in reconfigurable hardware to single existing operating system objects to ease communication. While simplifying the design, this also presents restrictions to the thread designer and hinders portability. We believe that supporting a unified programming model consisting of various objects for communication, synchronization, and thread control is essential for exploiting the full potential of reconfigurable systems while maintaining portability across different software OS bases and hardware platforms.

In our previous work, we have developed ReconOS [6], which demonstrated hardware/software multithreading based on eCos [7] as a host OS running on the PowerPC core of modern platform FPGAs. With ReconOS, we have created an abstraction layer common to both software and hardware threads, allowing the latter to autonomously initiate OS calls. A related project is hthreads [8], which implements the OS components managing synchronization and scheduling as hardware IP cores, sacrificing the flexibility of a software OS kernel for exceptionally low response time and jitter [9].

A cardinal argument for using an operating system is *portability*, a feature important for both classic and reconfigurable computers. We are interested in investigating the portability of our hardware/software multithreading abstraction layer across different processors, different host operating systems, different reconfigurable fabrics, and different application domains. This paper proves the first two points and lays the groundwork for expanding the applicability of ReconOS from embedded systems to the domain of reconfigurable high-performance computing.

In both domains, we observe an increased use of reconfigurable accelerators and the acceptance of multithreading as a programming abstraction. However, while embedded applications are typically dependent on deterministic OS calls, small memory and area footprints, low interrupt latencies, and minimal power consumption, high-performance systems, on the other hand, need complete multiuser OS functionality including memory protection, as well as scientific support libraries, network protocol stacks, and specialized communication abstractions. To be able to support our programming model in both areas, we need to show that ReconOS can be adapted to and run on existing operating systems that already cater to the needs of their respective application domains.

The novel contributions of this paper are to present the implementation of the ReconOS hardware/software multithreaded programming abstraction on Linux, to contrast it with the existing eCos-based version, and to quantify and compare the performance of both systems.

The remainder of this paper is organized as follows: A short overview of the ReconOS multithreaded programming model is given in Section 2, with the role of *delegate threads* explained in Section 3. Section 4 presents the implementations of the ReconOS software runtime system on eCos and Linux using differing hardware platforms; their performance is evaluated in Section 5. Section 6 summarizes the results and discusses future work.

2. PROGRAMMING MODEL

The ReconOS programming model builds on top of established programming models for real-time operating systems, and extends them to the hardware domain. The basic unit of execution is a thread, which can be implemented as a "regular" software thread of the respective operating system, scheduled and executed on the system's main CPU, or as a hardware thread, implemented as a hardware circuit on an FPGA. Essentially, hardware threads and software threads can access the same operating system *services*, such as communication and synchronization, using the same operating system *objects*, such as semaphores, shared memory, or mutexes. In ReconOS, these objects are managed by the software operating system kernel; software threads access them using the existing and established software API provided by the respective operating system.

To allow hardware threads to access these OS objects, the corresponding software API calls are exposed to the hardware through a VHDL library. The procedures provided by this library, listed in Table 1, can be used in a state machine description to synchronize the hardware thread's operation with the operating system and other threads. Since VHDL does not provide as transparent a memory access model as C, reads and writes to system memory also require their own

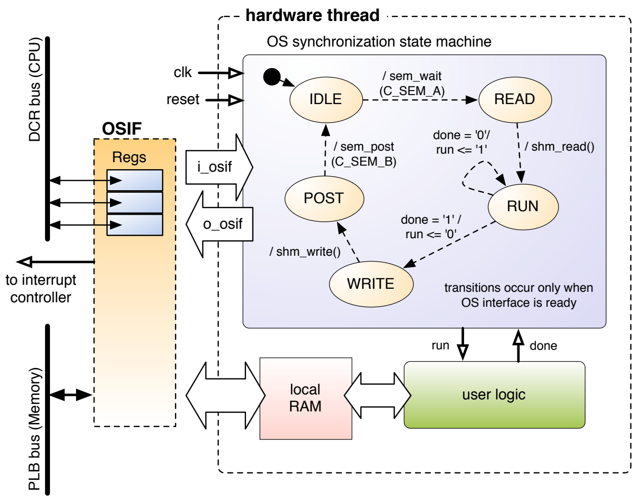

Fig. 1. Communication between HW thread and OSIF

ReconOS API calls. By applying this "hardware API", the hardware thread manipulates defined OS interface signals, which in turn evoke the appropriate control functions in a separate operating system interface module (OSIF), which is shown in Figure 1. The OSIF then either executes the requested function, if it can be handled in hardware, or relays the OS call to the CPU by means of an interrupt line and software-accessible registers. To ensure low and deterministic latencies for operating system calls, the communication between the OSIF and the CPU is routed through a device control register (DCR) bus that is separate from the memory bus (PLB).

Apart from the mentioned OS synchronization state machine, a hardware thread can contain an arbitrary number of concurrent "user logic" VHDL processes which carry out the thread's intended functionality. Usually, these processes are synchronized with the state machine through handshake signals; for simple tasks, it is also possible to integrate the processing directly into the state machine. More detail and an example for hardware thread programming can be found in [6].

3. DELEGATE THREADS

A fundamental assumption of the ReconOS programming model concerns the transparency of thread-to-thread communication and synchronization, regardless of the execution context (hardware or software) of the respective communication partners. This enables the designer to easily replace, for example, a software thread with a functionally equivalent hardware thread, allowing for rapid design space exploration with respect to the hardware/software partitioning.

In ReconOS, every hardware thread is associated with exactly one software thread, its *delegate*, to achieve this trans-

Table 1. ReconOS hardware API

Class	VHDL procedure
Initialization	reconos_get_init_data()
Thread termination	reconos_thread_exit()
Memory access	reconos_write()
	reconos_read()
	reconos_write_burst()
	reconos_read_burst()
Semaphores	reconos_sem_post()
	reconos_sem_wait()
Mutexes	reconos_mutex_lock()
	reconos_mutex_trylock()
	reconos_mutex_unlock()
	reconos_mutex_release()
Condition variables	reconos_cond_wait()
	reconos_cond_signal()
	reconos_cond_broadcast()
Message boxes	reconos_mbox_get()
	reconos_mbox_tryget()
	reconos_mbox_put()
	reconos_mbox_tryput()

parency. The delegate is responsible for executing operating system calls on behalf of the corresponding hardware thread, making it appear as a software thread to the operating system kernel. Delegate threads are created as standard OS threads and passed additional parameters necessary to access the OSIF hardware. After creation, the delegate resets, initializes and starts (i.e. unblocks) the hardware thread. It then waits for an incoming OS request from the hardware to execute.

To be able to map the OS objects referenced by the hardware thread to actual instances in the operating system kernel, the delegate thread maintains a table of object instances that are used by the hardware thread. Individual resources are represented towards the hardware thread as an index into this table. Hence, a single hardware thread description (i.e. VHDL source code, netlist or possibly a relocatable bitstream) can be used for multiple instances in the system; giving different instances access to different resources is simply a matter of changing the delegate's OS object table. This mechanism is also a prerequisite for partial reconfiguration of hardware threads, which is planned as a future extension.

4. IMPLEMENTATIONS

The programming model described in Section 2 has been implemented on top of two wide-spread operating system kernels: eCos and Linux. While eCos is designed primarily for embedded systems with limited resources, Linux is targeted at a wide range of application areas and corresponding target platforms. As a consequence, both systems provide a somewhat disparate feature set. Many differences, however, can be hidden from the application programmer behind the POSIX API, which is supported by both systems. Since hardware threads are written with a seperate API that is similar to POSIX and supported by the ReconOS abstraction layer, both systems are capable of running the same software and hardware threads with little to no changes to the source code.

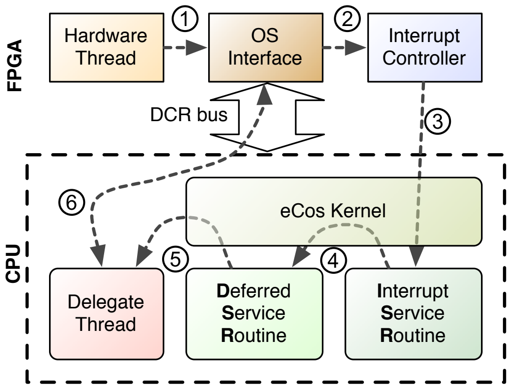

Fig. 2. Communication between HW and Delegate in eCos

4.1. eCos

The eCos [7] real-time operating system provides a modular and configurable framework of operating system services. Application designers can select the necessary packages from the eCos repository and compile them into a library, which the final application is linked against. eCos is also configurable on a source code level; using preprocessor macros, unneeded code is removed at compile time, resulting in small code sizes, which suits the targeted embedded segment. eCos is written in C/C++ and supports a range of target processor architectures, including the PowerPC 405, but not the MicroBlaze soft core, which limits applying eCos to Xilinx FPGAs of the Virtex-II Pro and Virtex4FX families, for now.

To transparently include ReconOS delegate threads in the eCos programming model, we have extended the eCos thread class to include additional information relevant to hardware threads, such as OSIF addresses, interrupt numbers, and OS object tables. Together with C wrappers for thread creation that are very similar to the eCos and POSIX API, reusing the existing thread code allows ReconOS delegate threads (and, by extension, the associated hardware threads) to take advantage of all OS services provided by the eCos kernel.

Because eCos does not distinguish between user and kernel space but runs entirely in the processor's real mode, hardware access from user threads is greatly simplified. Although the delegate thread is logically part of the user application rather than the kernel, it can directly access the DCR bus to communicate with its corresponding OSIF. eCos also lets hardware and software threads share the same address space, since it disables the MMU, sacrificing memory protection and privilege management for a greatly simplified memory access model and higher performance. While unreasonable for larger-scale multiuser systems, this is entirely appropriate for small-footprint self-contained embedded systems, as targeted by eCos.

The sequence of events that is performed to relay an OS call from hardware to the eCos kernel is shown in Figure 2. When a hardware thread uses a VHDL API call to request an operating system service, the respective VHDL procedure asserts certain handshake lines between the thread and its OSIF (1). Pending OS call requested by the OSIF are signalled to the CPU's interrupt controller via a dedicated interrupt line (2). In eCos, interrupt processing is split in two steps to ensure rapid response to incoming interrupt requests. First, a very small *interrupt service routine* is invoked (3), which executes in its own context, performs the necessary operations to enable reception of the next interrupt as quickly as possible, and marks the *deferred service routine* (4) for execution. The latter is scheduled by the regular eCos scheduler so as not to interfere with the low-level interrupt processing, which keeps interrupt response times low. As the last step before the actual delegate thread is invoked, the DSR posts a semaphore (5) which the delegate is waiting on, essentially signalling an incoming request. The delegate thread then directly accesses the OSIF's registers via dedicated DCR access instructions to retrieve call parameters (6) and executes the requested eCos kernel function.

4.2. Linux

The Linux operating system is employed on a wider range of target architectures and therefore enjoys a wider adoption than eCos. The list of architectures includes, as the most interesting to us, the PowerPC 405 and the Xilinx MicroBlaze soft core. The latter widens the range of target FPGAs, including those without an embedded CPU core. The MicroBlaze can be synthesized in variants with or without an MMU. For our MicroBlaze prototype, we have opted for the omission of an MMU, which simplifies memory transfers between software and hardware threads.

While offering a wide set of configurable options, it is not possible to reduce the memory footprint of a Linux kernel as much as is possible with eCos. Absolute values on the size of the respective kernel images are difficult to obtain, as the code size greatly depends on the selected features, the target architecture, and the employed compiler. Also, an eCos kernel image already includes all necessary API implementations, the libc, and possibly a network stack. It can be expected, though, that a Linux kernel's size will exceed an equivalent eCos kernel by about an order of magnitude.

To communicate with its OSIF, a delegate thread needs access to the DCR bus. On a PowerPC system, this is accomplished through the `mtdcr` and `mfdcr` instructions, both of which are *privileged*. In Linux, user-space code, such as a delegate thread, typically cannot execute privi-

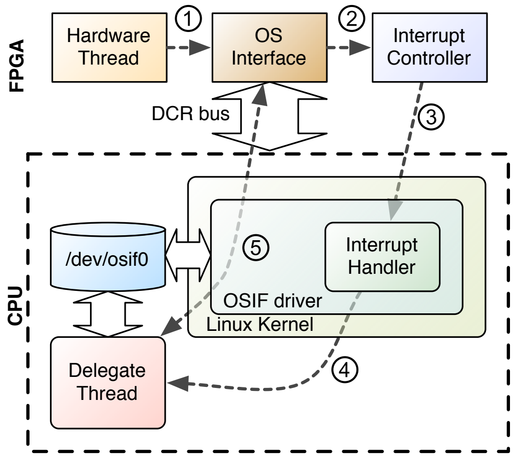

Fig. 3. Communication HW and Delegate in Linux

leged instructions. To make the OSIF registers accessible to the delegate, we have implemented the low-level hardware access to the OSIF registers in a kernel driver, which publishes the registers through a device node, as depicted in Figure 3. The hardware-independent code, such as the API wrappers and the delegate thread code, is implemented through a library that is linked with the user application.

Due to the separation of hardware-dependent and independent code, the sequence of events to relay an OS call from hardware to the Linux kernel differs from the one described in Section 4.1. The signal assertions between hardware thread and OSIF (1) and the interrupt request to the system's interrupt controller (2) are identical. When a delegate thread needs to access its OSIF, it does so through filesystem accesses to the kernel driver's device node. In eCos, synchronization between the delegate thread and the OSIF was achieved through a dedicated semaphore; in Linux, this synchronization is implemented through read accesses blocking until an interrupt from the OSIF is registered (3). Only then is the blocking delegate thread resumed (4) and the read access translated into DCR operations (5). Write operations to an OSIF do not block.

Data transfers between software and hardware threads are complicated by the fact that Linux usually employs virtual memory. This means that memory buffers set aside by an application as shared memory to transfer data to or from hardware threads are not necessarily contiguous; furthermore, the hardware thread operates on physical addresses, while software threads use virtual addresses which are translated by the MMU. Also, it is not possible for user applications to flush or invalidate the processor's caches in order to maintain cache coherency.

Therefore, shared memory for thread communications across the HW/SW boundary has to be implemented in one of three ways: a) by using a separate, uncached memory buffer which is advertised to the kernel as a memory mapped device, b) by allocating contiguous buffer of kernel memory via `kmalloc()`, mapping it into user space, and providing hardware threads with its physical address, or c) by providing every OSIF with a small-scale MMU which keeps a mirror of the processors TLB and other virtual memory information, similar to [10]. The first two ways, which involve a separate device driver, are currently being investigated. The third approach bears the biggest complexity, but also provides a transparent memory access model, and will be the target of our future research.

5. EXPERIMENTAL MEASUREMENTS

The somewhat different nature of the two operating systems ReconOS has been ported to presents the interesting question of how the complete HW/SW systems compare perfomance-wise. To enable quantitative measurements on the system performance, we have run a set of benchmarks on three prototype implementations: ReconOS/eCos and ReconOS/Linux on a XC2VP30 FPGA, and ReconOS/Linux on a XC4V35SX FPGA. The software parts of the first two prototypes executes on the FPGA's embedded PowerPC 405 running at 300 MHz, while the third prototype runs a noMMU variant of Linux on a MicroBlaze v4.0 soft core processor clocked at 100 MHz. All implementation feature a similar hardware layout, each with PLB, OPB and DCR buses running at 100 MHz, as well as external DDR memory. The Linux systems run a 2.6 kernel.

The first set of experiments employs a set of synthetic threads analyzing the performance of timing critical OS calls. The mutex and semaphore primitives from Table 1 serve as representative examples, as most other supported API calls are either based on them or are not of interest for timing. The threads measure the raw execution time of single OS API calls to lock/unlock a mutex or post/wait for a semaphore, respectively, as well as a measure we call the *turnaround time*. This is the time it takes for one thread to release a mutex / post a semaphore, and the next thread waiting on that mutex / semaphore to acquire a lock / receive the semaphore and continue. The experiments have been run with different combinations of software and hardware threads; results are shown in Table 2 and Figures 4 (a) and (b).

Synchronization operations on the eCos kernel behave as expected: calls from hardware are more expensive than their software counterparts due to the additional interrupt processing and hardware accesses. The Linux implementations show a similar behaviour but differ in certain details. Overall, OS calls are significantly more expensive in a Linux kernel than in eCos; a fact which can be attributed to context switches to and from kernel mode when executing OS functions. On a PowerPC CPU running at the same speed, the Linux calls take about an order of magnitude longer than the

Table 2. Synchronization benchmark results

	eCos/PPC	Linux/PPC	Linux/MicroBlaze
Mutex (raw OS calls)			
SW lock	83	821	9178
SW unlock	171	551	9179
HW lock	959	7769	35855
HW unlock	679	2636	22360
Mutex (turnaround)			
SW → SW	453	8821	83657
SW → HW	629	9824	90515
HW → SW	1449	14371	121673
HW → HW	1460	14102	126668
Semaphore (raw OS calls)			
SW post	73	598	13180
HW post	695	1972	22116
Semaphore (turnaround)			
SW → SW	305	9094	203221
SW → HW	528	9575	207824
HW → SW	908	12291	145924
HW → HW	1114	12196	154013

All values given in bus cycles (1 cycle = 10 ns)

corresponding eCos calls. Using a considerably less powerful MicroBlaze soft core processor clocked at a third of the clock frequency, the execution times rise by another order of magnitude, except for one anomaly: software-initiated semaphore operations exhibit about twice the latencies that we expected. This inconsistency will be further investigated.

The second set of experiments focuses on the real-world implications of the OS call overheads by evaluating system performance in a complete application. A list of 2^{18} unsorted 32 bit integers is sorted, using a combination of bubble-sort and merge sort. First, the list is divided into 128 chunks, which are sorted individually using bubble sort. The resulting lists are then merged. To map this application onto our system, we divided it into two threads, one for the bubble sort routine, which has a software and a hardware implementation, and one for the merge operation, which is always performed in software. The threads communicate using shared memory and use message boxes for synchronization. The benchmark has been run on two of our three prototype platforms, one running ReconOS/eCos on a PowerPC, the other running ReconOS/Linux on a MicroBlaze. Three tests were performed: the first running the sort thread in software (SW); the second running the sort thread in hardware (HW); and the third running two sort threads concurrently, one in software, the other in hardware (SW+HW). The results of the measurements are shown in Figure 4 (c). In the sum above each bar, the first (bold) value denotes the time spent sorting, while the second is the merge time.

The first and last test, which perform (at least part of) the sorting routine in software, reveal, unsurprisingly, that the MicroBlaze processor performs the sort operation vastly slower than the PowerPC. However, when executing the sorting thread solely in hardware, both systems are almost on par. In this situation, the hardware threads interact with

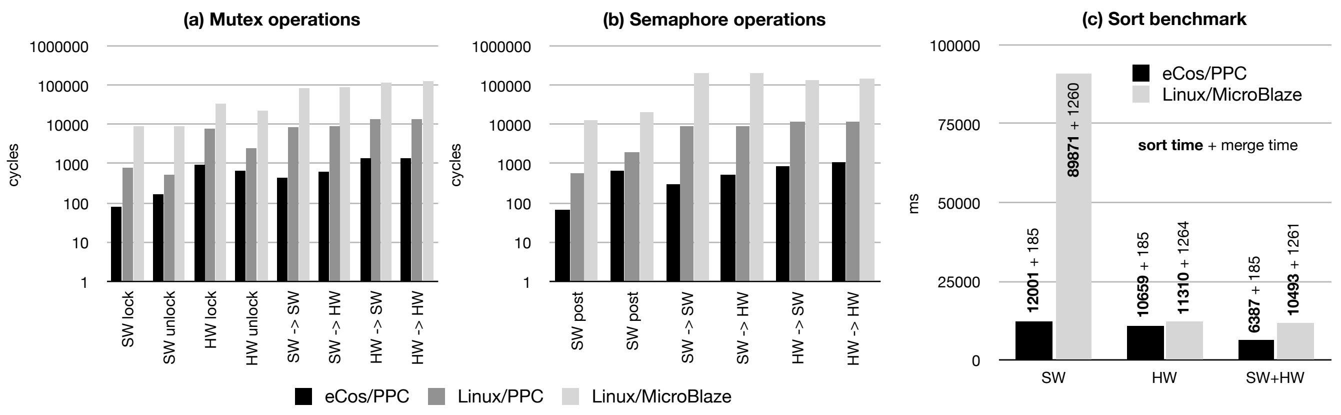

Fig. 4. Experimental results

the OS synchronization primitives infrequently enough so that the performance penalty due to additional software processing remains within acceptable limits. This is a common scenario: an application designer will be likely to use the precious hardware resources for data-centric computations with relatively enfrequent OS synchronization operations, and perform most control-dominated tasks inside software threads. Therefore, while the penalty incurred by the low-level synchronization and communication between delegate thread and OS interface is substantial for OS calls alone, the effect on overall application performance is not.

6. CONCLUSION

In this paper, we have presented implementations of the ReconOS hardware/software programming model on both eCos and Linux-based systems running on an embedded PowerPC or the soft core MicroBlaze CPU. This work demonstrates the portability of the ReconOS abstraction layer, significantly extending the range of reconfigurable target platforms and application domains that can take advantage of the multithreaded programming model. We have also analyzed and quantified the performance of both single operating system calls and a complete application. The results show that single OS services involving hardware threads are executed about one order of magnitude slower than the same services for software threads. On the application level, however, this performance difference is not observed since typical hardware threads make rather infrequent use of OS services.

Future research will focus on several areas: First, we will investigate the integration of virtual memory models as used by MMU-based Linux implementations into ReconOS (see Section 4.2). Second, we plan to integrate partial reconfiguration techniques into our implementation while preserving portability. Third, we will port and apply ReconOS to high-performance reconfigurable computing platforms.

7. REFERENCES

[1] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Run-Time Management of Reconfigurable Hardware Tasks Using Embedded Linux," in *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2007, pp. 209–215.

[2] X. Xie, J. Williams, and N. Bergmann, "Asymmetric multi-processor architecture for reconfigurable system-on-chip and operating system abstractions," *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pp. 41–48, 2007.

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

[4] H. K.-H. So and R. W. Brodersen, "Improving Usability of FPGA-based Reconfigurable Computers through Operating System Support," in *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 349–354.

[5] H. K.-H. So, A. Tkachenko, and R. Brodersen, "A Unified Hardware/software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH," in *Proceedings of the 4th International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*. ACM Press, 2006, pp. 259–264.

[6] E. Lübbers and M. Platzner, "ReconOS: An RTOS Supporting Hard- and Software Threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2007, pp. 441–446.

[7] "eCos," Website, 2008, http://ecos.sourceware.org/.

[8] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "hthreads: A Computational Model for Reconfigurable Devices," in *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 1. IEEE, August 2006, pp. 885–888.

[9] 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 *Proceedings of the 27th International Real-Time Systems Symposium (RTSS)*. IEEE, 2006, pp. 3–12.

[10] M. Vuletić, L. Pozzi, and P. Ienne, "Seamless Hardware-software Integration in Reconfigurable Computing Systems," *Design and Test of Computers, IEEE*, vol. 22, no. 2, pp. 102–113, March-April 2005.