ReconOS: Multithreaded Programming for Reconfigurable Computers

ENNO LÜBBERS and MARCO PLATZNER
University of Paderborn

Rising logic densities together with the inclusion of dedicated processor cores push reconfigurable devices from being applied for glue logic and prototyping towards implementing complete reconfigurable systems-on-chip. The mix of fast CPU cores and fine-grained reconfigurable logic allows to map both sequential, control-dominated code and highly parallel data-centric computations onto one platform. However, traditional design techniques that view specialized hardware circuits as passive coprocessors are ill-suited for programming these reconfigurable computers. In particular, the programming models for software—running on an embedded operating system—and digital hardware—synthesized to an FPGA—lack commonalities, which hinders design space exploration and severely impairs the potential for code reuse.

In this article, we present ReconOS, an execution environment based on existing embedded operating systems that extends the multithreaded programming model established in the software domain to reconfigurable hardware. Using threads and common synchronization and communication services as an abstraction layer, ReconOS allows for the creation of portable and flexible multithreaded applications targeting CPU/FPGA systems. This article discusses the ReconOS programming model and its execution environment, presents implementations based on modern platform FPGAs and the operating systems eCos and Linux, evaluates time and area overheads of the proposed mechanisms and, finally, demonstrates the feasibility of the multithreading design approach on several case studies.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Design

Additional Key Words and Phrases: Reconfigurable computing, operating systems, multithreading

ACM Reference Format:

Lübbers, E. and Platzner, M. 2009. ReconOS: Multithreaded programming for reconfigurable computers. ACM Trans. Embedd. Comput. Syst. 9, 1, Article 8 (October 2009), 33 pages.
DOI = 10.1145/1596532.1596540 http://doi.acm.org/10.1145/1596532.1596540

1. INTRODUCTION

Reconfigurable hardware devices have evolved from small logic-centric chips to powerful platforms combining microprocessor cores with dense logic fabrics.

Author's email address: enno.lucbbers@uni-paderborn.de

DOI 10.1145/1596532.1596540 http://doi.acm.org/10.1145/1596532.1596540

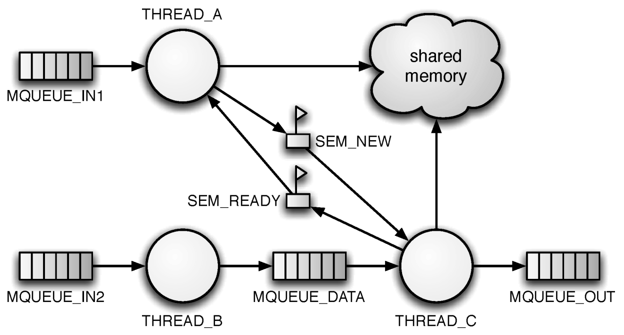


Fig. 1. Example application in the multithreaded programming model.

Accordingly, the application domains for such devices have been extended from the original glue logic over prototyping and ASIC replacement to modern reconfigurable computers that allow for the mapping of both complex control-dominated tasks and data-centric parallel processing tasks to the same device. However, design methodologies for such configurable systems-on-chip have not kept up with the rise in complexity of reconfigurable hardware. In particular, there is little overlap between programming models and practices for embedded software and digital logic.

In this context, we are especially interested in the operating system layer. In the embedded systems domain, real-time operating systems such as VxWorks [Wind River, 2007], RTXC [Quadros Systems, Inc. 2007], eCos [eCosCentric 2008], and many proprietary systems provide the designer with a set of clearly defined objects and associated services, which are encapsulated in application programmer interfaces (e.g., the POSIX API [IEEE and The Open Group 2004]). Among these basic objects, we typically find threads and processes as units of execution, semaphores and related services for synchronization, and mailboxes and their derivatives for communication. Threads are mostly characterized as light-weight processes featuring a fast context switch. While threads within a process share one address space, different processes are isolated from each other. Real-time operating systems 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.

The set of objects offered by an operating system together with the used scheduling policy can be considered a programming model. While this model is not comparable to formal models of computation, it does provide a designer with an established way of structuring an application. Figure 1 sketches an example for an application composed of several threads, semaphores, message queues, and a shared memory region. In this example, `THREAD_A` reads data out of a message queue, processes it, and writes the result to a shared memory region, synchronizing concurrent access to the shared memory with `THREAD_C` via two semaphores. When going from a CPU-based system to a CPU/FPGA platform, it seems natural to simply extend the services offered by the operating system to customized hardware cores. Analogous to a software thread, a hardware core performing a specific task can be thought of as a hardware thread.

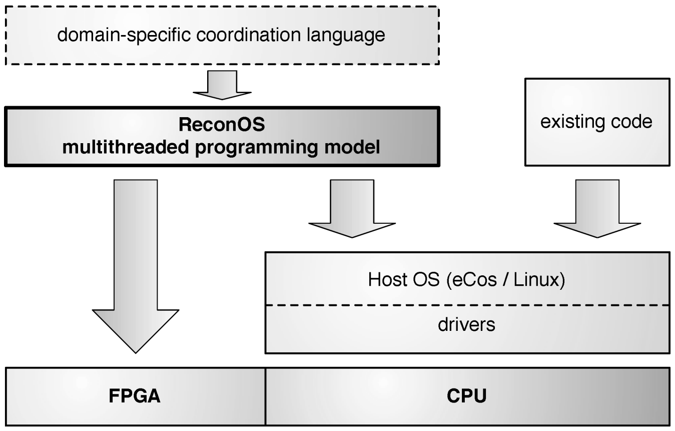

domain-specific coordination language
ReconOS
multithreaded programming model
existing code
Host OS (eCos / Linux)
drivers
FPGA
CPU

Fig. 2. ReconOS abstraction layer for CPU/FPGA systems.

In this article we present ReconOS, an operating system for configurable systems-on-chip that extends the multithreaded programming model from software to reconfigurable hardware. The multithreaded programming model represents an established abstraction layer used throughout the software development field, from embedded devices using small real-time operating system kernels to high-performance computing systems relying on full-fledged multiuser operating systems. Moreover, multithreading is also supported by several programming languages. 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 argue that such an approach to integrating hardware cores into a processor-based system greatly eases application development and increases productivity and portability.

Figure 2 shows the abstraction layer provided by ReconOS. ReconOS leverages standard operating system kernels, which allow for running any existing code and facilitate the access to a variety of I/O devices. The ReconOS application programmer interface (API) essentially provides POSIX functions as one single development model for both software and hardware execution contexts. As we show in this article, a designer is now able to map an application to a portable model that can be directly executed on a variety of CPU/FPGA execution platforms.

Although well-established, multithreading as a model is often criticized for its implicit nondeterminism, for example, by Lee [2006]. Partly, this can be addressed by designing applications in a more formal model of computation or coordination language, which are often domain-specific, and an automatic mapping from such a formal model down to a multithreaded representation. In essence, the ReconOS multithreaded programming model then forms an abstraction layer in-between a domain-specific formalism and the underlying heterogeneous execution resources. Although such formal models and their mapping to ReconOS objects are not subject of this article, Figure 2 outlines this option.

The novel contribution of this work is the extension of the well-known multithreaded programming model across the software/hardware boundary. ReconOS leverages existing operating system kernels and allows threads to be executed either on the CPU or in reconfigurable hardware, achieving an unparalleled flexibility and portability for the emerging class of CPU/FPGA systems. Compared to previously published conference contributions [Lübbers and Platzner 2007, 2008b, 2008a], which focused on single aspects of ReconOS, this article provides a comprehensive overview over ReconOS as well as a detailed discussion of its programming and execution models, several implementations, and elaborate experiments.

The remainder of this article is structured as follows: Section 2 reviews related work in operating system approaches for reconfigurable computers. The programming model used and implemented by ReconOS is explained in Section 3. Section 4 details the execution model for hardware threads, with Section 5 describing ReconOS implementations on two different existing software operating system kernels and the underlying hardware architecture. In Section 6, the performance of these implementations is evaluated and two case studies are presented. Finally, Section 7 concludes the article and gives an overview of ongoing and future work.

2. RELATED WORK

In the last decade, operating systems for reconfigurable computers have been researched from a number of different angles. In the following, we first review early work on concepts and functions for such operating systems, followed by more recent efforts toward integrating reconfigurable hardware circuits as tasks into mainstream operating systems and, finally, approaches to extend the multithreaded programming model from software to hardware.

Brebner [1996, 1997] was one of the first to discuss hardware multitasking. He proposed so-called swappable logic units that can be swapped in and out of a partially reconfigurable device, driven by an operating system. Other authors discussed further operating system functionalities. For example, Burns et al. [1997] described operating system functions that perform translation and rotation operations on hardware circuits to better fit them to the device. Merino et al. [1998] split the reconfigurable surface into so-called slots and used the operating system to schedule hardware tasks to these slots. Shirazi et al. [1998] structured the run-time system into a monitor, a loader, and a configuration store and investigated trade-offs between reconfiguration time and circuit quality. Wigley and Kearney [2001] made the case for including partitioning and allocation of reconfigurable resources as well as routing of hardware tasks as operating system functions.

In the following years, a large body of work has been focusing on single functions of future hardware operating systems. A prominent example is placement and scheduling of hardware tasks, which has been studied under a variety of task and resource models as well as optimization objectives. Examples can be found in Jean et al. [1999], Bazargan et al. [2000], Teich et al. [2000], Steiger et al. [2004], Danne and Platzner [2006], Danne et al. [2007], and Pellizzoni and

Caccamo [2007]. Many efficient scheduling techniques, especially for real-time systems, rely on task preemption. The preemption of hardware is a challenging problem and has been studied and prototyped by, for example, Simmler et al. [2000] or Kalte and Porrmann [2005]. Another issue related to placement and scheduling is the fragmentation of the reconfigurable logic area. While many of the presented placement and scheduling approaches try to avoid too much fragmentation, some authors proposed to compact the reconfigurable area from time to time. Examples can be found in Diessel et al. [2000] and Compton et al. [2002]. Most of these works were either theoretical or, if experimental, evaluated their algorithms by simulation studies on synthetic workloads, given the absence of available hardware operating system implementations and accepted benchmarks.

A number of prototypes have been created to demonstrate the feasibility of reconfigurable hardware operating systems. For example, a networked reconfigurable platform for multimedia appliances that enables multitasking in hardware and software was shown by Mignolet et al. [2002] and Nollet et al. [2003]. Prototype creation has always been hindered by limitations of available technology and design tools. More importantly, all the presented approaches viewed hardware tasks as coprocessors rather than independent execution units. A first step toward an approach that integrates hardware tasks as independent units was shown by Walder and Platzner [2003] and Steiger et al. [2004], respectively. In their prototype, hardware tasks have a higher degree of autonomy and can access operating system objects, such as FIFOs, memory blocks, and I/O drivers, and signal events to the operating system in order to drive the scheduler.

More recently, extensions of Linux have emerged that promote an OS-controlled integration of software and hardware processes. Kosciuszkiewic et al. [2007] built on top of an existing Linux operating system kernel and viewed so-called hardware tasks as a drop-in replacement for software tasks. These hardware tasks were executed on synthesized PicoBlaze softcore processors and did not exploit the fine-grained parallelism provided by FPGAs. In the described implementation, the interaction between software threads and hardware tasks was limited to FIFO communication. Xie et al. [2007] presented a similar heterogeneous multiprocessor system consisting of soft processor cores synthesized to an FPGA. Again, the Linux integration was limited to FIFO communication. Bergmann et al. [2006] encapsulated access to hardware modules into software wrappers, the so-called ghost processes, which provide a transparent interface for interactions from the kernel and other processes. The authors considered sharing the same address space between hardware and software execution units as unsuitable. Technically, they used processes instead of threads to encapsulate hardware modules. For communication between software and hardware, FIFOs mapped to the Linux file system as well as dual-ported memory accessible from both software processes and a hardware process were used, as shown by Williams et al. [2005]. So et al. [2006] also modified and extended a standard Linux kernel with a hardware interface, providing conventional UNIX IPC mechanisms to the hardware using a message passing network. Again, communication between hardware and software processes was implemented by FIFOs and mapped to file system-based operating system objects.

All these approaches tried to connect circuits implemented in reconfigurable hardware to existing operating system objects to ease communication between software and hardware. While simplifying the design of hardware/software systems to a certain degree, such an approach poses severe restrictions to the thread designer as only one specific communication service is available. In contrast, we believe that supporting a unified programming model for both software and hardware threads alike, supported by a rich set of operating system functions, is essential for exploiting the full potential of hybrid reconfigurable hardware/software systems while maintaining portability across different operating systems and hardware platforms.

ReconOS extends the multithreaded programming model from software to hardware. The multithreaded model was also taken up by Duchenne and Hanna [2005] who presented a high-level synthesis approach starting from Java. Instead of trying to extract parallelism from sequential code, they synthesized explicitly concurrent (i.e., multithreaded) Java programs to a hardware architecture. A more closely related effort to ReconOS is the hthreads project [Peck et al. 2006]. In hthreads, hardware threads are managed by the operating system and are able to access various OS functions through a dedicated hardware thread interface while sharing memory through a sophisticated interthread memory model [Anderson et al. 2007]. hthreads is based on the POSIX pthreads programming model for both hardware and software threads and implements the OS components managing synchronization and task scheduling as hardware IP cores. In comparison to ReconOS, hthreads sacrifices the flexibility of a software operating system kernel for exceptionally low response time and jitter [Agron et al. 2006], which caters to the needs of the targeted embedded systems domain.

3. THE RECONOS PROGRAMMING MODEL

An important design goal of the ReconOS programming model is its portability, and—somewhat closely related—its flexibility. ReconOS tries to (re)use much of the interface and functionality already present in established APIs, such as POSIX or the eCos kernel API. Consequently, most ReconOS programming model primitives or operating system objects are implemented by an operating system kernel running on the system CPU. ReconOS applications are typically crafted from the following operating system objects:

—*Threads* are the basic units of execution that make up an application. It is up to the developer to partition an application into threads, which then communicate and synchronize using other operating system objects.

—*Semaphores* and *Mutexes* provide means for high-level synchronization; they can be used to sequentialize execution of threads, to protect critical code regions, or to manage exclusive access to shared resources.

—*Shared memory*, *message queues* and *mailboxes* are used for interthread communication. Generally, access to shared memory must be protected by synchronization primitives, as is necessary for any shared resource. Message queues and mailboxes occupy a special niche among the operating system

Table I. Overview of ReconOS API Functions

OS Object	POSIX API (software)	ReconOS API (hardware)
semaphores	sem_post()	reconos_sem_post()
	sem_wait()	reconos_sem_wait()
mutexes	pthread_mutex_lock()	reconos_mutex_lock()
	pthread_mutex_unlock()	reconos_mutex_unlock()
condition variables	pthread_cond_wait()	reconos_cond_wait()
	pthread_cond_signal()	reconos_cond_signal()
	pthread_cond_broadcast()	reconos_cond_broadcast()
message queues /		reconos_mq_send()
mail boxes	mq_send()	reconos_mbox_put()
		reconos_mbox_tryput()
		reconos_mq_receive()
	mq_receive()	reconos_mbox_get()
		reconos_mbox_tryget()
shared memory	**ptr = value*	reconos_write()
		reconos_write_burst()
	*value = *ptr*	reconos_read()
		reconos_read_burst()
threads	pthread_exit()	reconos_thread_exit()
	pthread_create()	—

objects—they provide both communication and synchronization at the same time.

The fact that all interthread activity is carried out using only these objects provides complete transparency within these interactions; a thread does not need to know whether its communication or synchronization partners are located in hardware or software—which, in turn greatly facilitates design space exploration with respect to the hardware/software partitioning. Also, as long as the interfaces to the respective operating system objects are supported, the interoperability and portability of threads can be easily maintained when moving to a different target platform.

It should be noted that it is the set of programming model primitives, not the individual API calls, that provides these benefits of the multithreaded programming model as implemented by ReconOS. For example, a software thread using the POSIX API can communicate seamlessly with a hardware thread using the ReconOS API (outlined in Section 3.2), as long as they use the same programming model abstraction (e.g., a mailbox/message queue).

An overview of the operating system objects and their related ReconOS and POSIX API calls, as used by hardware and software threads, respectively, is shown in Table I. Most hardware functions are direct counterparts to the POSIX software API. Notable exceptions include the mailboxes, which provide separate sets of calls for blocking and nonblocking put and get operations, and memory accesses, which can explicitly request single-word or burst transfers. Currently, the ReconOS hardware API supports the most important subset of the calls available in POSIX; the incorporation of additional functions, such as calls for thread creation or scheduler control, requires only minimal extensions to the execution environment and is planned as future work.

```
mqd_t mqueue_in1;
sem_t sem_new, sem_ready;
void *shared_mem;

void *thread_a_entry( void *data ) {
   uint8 buf[ MSG_SIZE ];

   while ( true ) {
       mq_receive( mqueue_in1, buf, MSG_SIZE, 0 );
       do_something( buf );
       sem_wait( sem_ready );
       memcpy( shared_mem, buf, MSG_SIZE );
       sem_post( sem_new );
   }
}
```

Listing 1. Code implementing software thread THREAD A.

In the following, we first present the programming of software threads and, more importantly, hardware threads under ReconOS. Then, we discuss the creation and termination of threads as well as access to shared resources.

3.1 Software Threads

ReconOS software threads are identical to regular threads of the host operating system both in concept and implementation. Since software threads are handled by the standard OS scheduler, they are independent from the ReconOS extensions.

Currently, software threads can be implemented using either the eCos kernel API or the POSIX pthreads API—the ReconOS operating system objects can be seamlessly mapped to either API. It is recommended to use POSIX where possible, as it is the more portable API of the two and is also supported via a compatibility layer in eCos. Listing 1 shows an example of a software thread implementing `THREAD_A` from the application shown in Figure 1, using the POSIX API. Here, data is received from a message queue (MQUEUE_IN1), processed, and then copied to shared memory, which is synchronized using semaphores (SEM_READY and SEM_NEW).

3.2 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

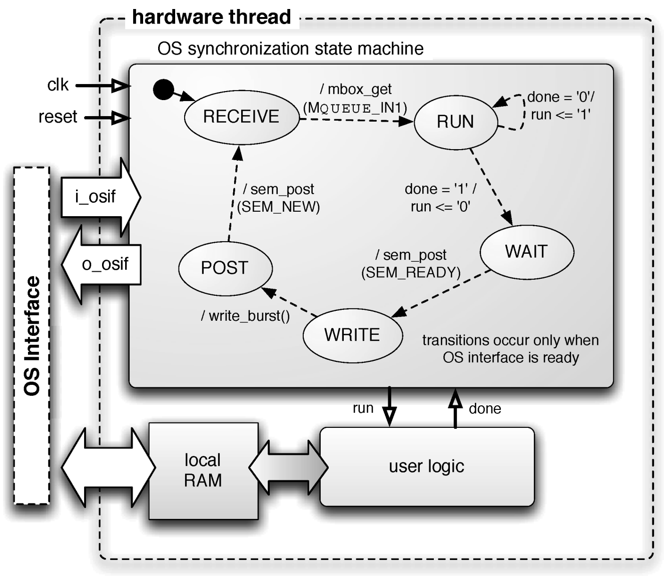


Fig. 3. Example of an OS synchronization state machine.

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

An example demonstrating this mechanism is illustrated in Figure 3, which again represents THREAD_A from the previously example described. In this example, the hardware thread receives a message into the local RAM, processes it, waits on a semaphore (SEM_READY), writes the result to shared memory, and then posts another semaphore (SEM_NEW). The OS synchronization state machine and the user logic communicate via the two handshake signals *run* and *done*. Listing 2 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 *WAIT*, 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

```
osif_fsm: process(clk, reset)
  variable completed, success : boolean;
begin
  if (reset = '1') then
    state <= RECEIVE;
    ram_addr <= 0;
    run <= '0';
    reconos_reset(o_osif, i_osif);
  elsif rising_edge(clk) then
    reconos_begin(o_osif, i_osif);
    if reconos_ready(i_osif) then
      case state is

        when RECEIVE =>
          reconos_mq_receive(completed, success, o_osif, i_osif,
                             MQUEUE_IN1, 0, MSG_SIZE);
          if completed then
              state <= RUN;
          end if;

        when RUN =>
          run <= '1';
          if done = '1' then
            run <= '0';
            state <= WAIT;
          end if;

        when WAIT =>
          reconos_sem_wait(o_osif, i_osif, SEM_READY);
          state <= WRITE;

        when WRITE =>
          reconos_write_burst(o_osif, i_osif, 0, SHARED_MEM);
          state <= POST;

        when POST =>
          reconos_sem_post(o_osif, i_osif, SEM_NEW);
          state <= RECEIVE;

        when others => null;
      end case;
    end if;
  end if;
end process;
```

Listing 2. Code for the example of Figure 3.

to the next state, *WRITE*. 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 *WRITE* 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.

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

```
// shared OS objects
mqd_t my_mqueue;
sem_t my_sem;

// resource array for hardware thread
reconos_res_t hwthread_resources[2] = {
 { &my_mqueue, POSIX_MQD_T },
 { &my_sem,    POSIX_SEM_T }
};

// software thread object and attributes
pthread_t      swthread;
pthread_attr_t swthread_attr;

// hardware thread object and attributes
rthread        hwthread;
pthread_attr_t hwthread_swattr;
rthread_attr_t hwthread_hwattr;

// initialization of software thread attributes
pthread_attr_init(&swthread_attr);

// initialization of hardware thread attributes
pthread_attr_init(&hwthread_swattr);
rthread_attr_init(&hwthread_hwattr);
rthread_attr_setresources(&hwthread_hwattr,
                          hwthread_resources, 2);

// software thread creation
pthread_create(
   &swthread,                // thread object
   &swthread_attr,           // attributes
   swthread_entry,           // entry point
   ( void * ) data           // entry data
);

// hardware thread creation
rthread_create(
   &hwthread,                // thread object
   &hwthread_swattr,         // software attributes
   &hwthread_hwattr,         // hardware attributes
   ( void * ) data           // entry data
);
```

Listing 3. Creation of software and hardware threads compared.

3.3 Thread Creation and Termination

The creation of threads within the ReconOS programming model is almost identical for software threads (using `pthread_create()`) and hardware threads (using `rthread_create()`). The POSIX-like creation of both variants is shown in Listing 3. A hardware thread takes the same scheduling and stack size parameters as a software thread, encapsulated in a `pthread_attr` structure. These are used for the hardware thread's associated delegate thread (see Section 4.2) and influence the hardware thread's priority when contending for access to operating system objects. The main difference between the creation of software and hardware threads exists in the passing of information about the shared resources to the latter, which is done via a `rthread_attr` structure. Currently, creating a hardware thread using the `rthread_create()` assumes that the hardware thread is already present in the reconfigurable fabric. Usually, hardware threads are configured to the FPGA together with the static hardware architecture (e.g., buses and peripherals) during system bootup. With partial reconfiguration, as outlined in Section 7, the loading of hardware threads could be implicitly done by the scheduler on thread creation.

Thread termination can either be initiated by the respective threads themselves—using `pthread_exit()` (within software threads) or `reconos_thread_exit()` (within hardware threads)—or threads can be explicitly aborted using `pthread_kill()`.

Apart from POSIX, hardware and software threads can also be created and terminated using the eCos API, if supported by the host OS. For hardware threads, the same `rthread_attr` structure can be used.

3.4 Shared Resources

Operating system objects will almost always be shared among different threads. For software threads, this is usually achieved by representing the respective resources as global variables accessible by all threads. This approach creates significant problems when dealing with hardware threads. Since the synthesis tool for a hardware thread written in VHDL cannot easily access the symbol tables of associated software threads, we cannot use global variables defined in software to share an operating system object. Passing a pointer to the data structure representing the OS object to a hardware thread is a possible option; however, it does not replicate the simplicity of the global variable approach.

To provide hardware thread designers with a comparably simple mechanism, ReconOS associates an array of resources with every hardware thread. The thread designer can then define integer constants in VHDL that act as indices into the resource array, and use the symbolic constants as arguments to the respective ReconOS API calls. The definition of the resource array—and thus the mapping between symbolic VHDL constants and actual objects of the operating system kernel—is established at design time; an example is shown in Listing 3. This mechanism also transparently separates the hardware thread API from the API used to define the OS objects (e.g., the eCos kernel API or the POSIX API), and it provides a concise overview of the resources used by the individual threads.

Conceptually, ReconOS hardware threads can directly access the same memory regions as software threads, which allows for efficient sharing of data among threads. As far as the programming model is concerned, all threads share the same address space. To manage concurrent accesses to these memory areas, the threads will usually use synchronization mechanisms from ReconOS' programming model, such as semaphores or mutexes.

In many execution environments, two mechanisms complicate the implementation of shared memory: *caching* and *virtual memory*. Many embedded processors, such as the PowerPC 405 core included in some Xilinx FPGAs, feature a cache unit in the data path between CPU and the memory bus. Sharing an address space between software threads running on this CPU and hardware threads directly connected to the memory bus requires the application designer to explicitly manage cache coherency issues, for example, by manually flushing or invalidating cache lines before or after a synchronized data transfer between the threads. This, however, somewhat dilutes the transparency of the programming model. The concept of virtual memory, which is also present in the ReconOS/Linux execution environment (see Section 5.3), further complicates sharing memory. Although software threads usually share the same page tables and virtual memory mappings, this information is not necessarily available for hardware threads. We currently circumvent this problem by allocating a separate, physically contiguous block of memory that is marked as uncachable and advertised to software threads through a memory-mappable filesystem node. This area is then explicitly used for data exchange between hardware and software threads. Section 7 discusses a more transparent solution.

4. THE RECONOS EXECUTION MODEL

Hardware circuits modeled as threads and synthesized to an FPGA require a run-time environment to connect them to an existing operating system kernel. On the hardware side, a well-specified interface is required to manage the requests and responses of a hardware thread. On the software side, many of these requests need to be forwarded to the host operating system kernel, and their responses need to be relayed back to the hardware threads. This section details the mechanisms employed by ReconOS to manage these tasks: the operating system interface (OSIF) and the delegate threads.

4.1 The Operating System Interface

To be able to model hardware circuits executing on reconfigurable logic as threads, it is necessary to carefully define mechanisms for low-level synchronization and communication between the hardware circuitry and the operating system. In ReconOS, this is the task of the OSIF. Figure 4 gives an overview of the OSIF's structure and its interfaces. On one side, the OSIF connects to the hardware thread's OS synchronization state machine and local RAM. On the other side, the OSIF provides an interface to two bus systems, the system memory bus (PLB) and an OS control bus (DCR). Further, the OSIF requires an interrupt line to the CPU's interrupt controller and features optional ports to connect to FIFO cores. The OSIF itself is built from several modules whose functions are described in the following.

4.1.1 *Thread Supervision and Control.* ReconOS provides hardware threads with a hardware API that comes in the form of a function library that specifies VHDL functions and procedures like `reconos_sem_post()` or `reconos_thread_exit()` (see Table I). A designer can use these procedures inside the thread's OS synchronization state machine to sequentially call operating system functions, much like a software thread uses functions from the operating system's C-API. As a consequence, every state of the OS synchronization

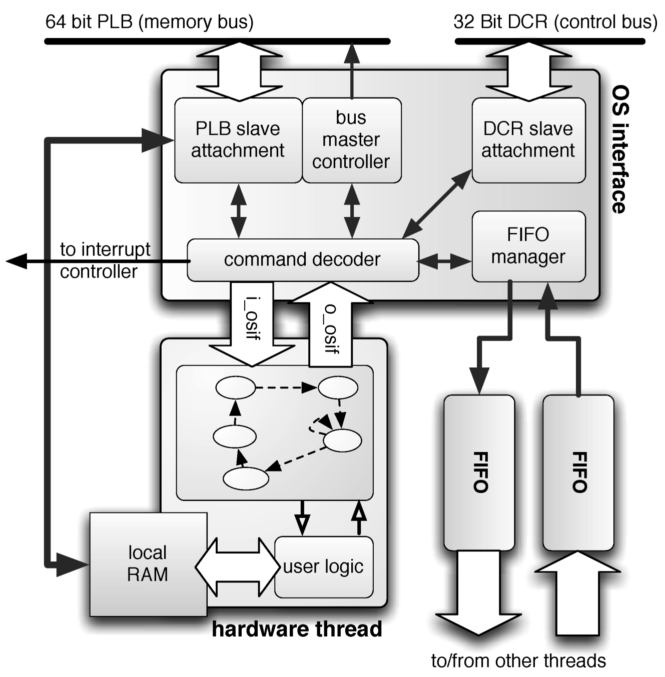


Fig. 4. OSIF overview and interfaces.

Table II. OSIF Communication Records

Signal		Description
		`osif2task`
command	[0:7]	requested OS call code
data	[0:31]	OS call arguments
request		request strobe
error		error flag
		`task2osif`
data	[0:31]	return value of OS call
step	[0:3]	current step of multicycle command
valid		indicates success of call
busy		system buses are busy
blocking		set while executing blocking OS calls

state machine may contain at most one VHDL system call. The VHDL procedures are purely combinational and communicate with the OSIF through a set of incoming and outgoing signals, which are assembled in the `osif2task` and `task2osif` records shown in Table II.

The mechanisms that govern the OS call request-response interactions between the OSIF and the hardware thread are controlled by the *command decoder* module. This module receives OS call requests from the hardware thread, decodes them and initiates the appropriate processes to fulfill that request. This may involve, for example, raising an interrupt with the system CPU, initiating a bus master transfer, or feeding data into a FIFO.

Since the operating system executing on the CPU cannot process OS calls within one clock cycle, the OSIF needs a means to suspend state transitions of the thread's OS synchronization state machine. This is achieved by having the OS synchronization state machine routinely check input signals from the `osif2task` record before setting its next state (see `reconos_ready()` in Listing 2). This way, the OSIF can block the part of the hardware thread that interacts with the operating system, which effectively implements the semantics of blocking calls in VHDL.

The OSIF distinguishes between two conditions that can suspend state transitions: *busy* and *blocking*. The hardware thread is held in the *busy* state as long as there are pending bus transfers as a result of a thread's request. On the other hand, a thread enters the *blocked* state after calling an OS function that can lead to thread blocking, for example `reconos_sem_wait()`. For the hardware thread, this distinction is arbitrary. The OSIF, however, manages blocking and busy internally in different ways. The *blocking* signal is a settable and resettable register that is indirectly controlled by the CPU, while the *busy* signal is set asynchronously by the PLB and DCR modules (see Section 4.1.2).

One of the purposes of the provided VHDL library is to make writing the OS synchronization state machine as easy and straightforward as possible. Thus, we want to avoid any complicated handshaking between state machine code and the OSIF—the command decoder must be able to transparently suspend the thread's state machine without requiring the thread designer to explicitly check for handshaking signals in every transition. Hence, the *busy* or *blocking* signals must be asserted in the same clock cycle as the thread's *request* signal. This is achieved by clocking the command decoder's state machine on the falling edge of the clock, which avoids possible combinational loops and keeps all handshake signals clock-synchronous.

Some of the supported OS calls require more than one 32-bit data argument. An example for such a call is a single-word memory access (`reconos_write()`), which needs both an address and a data argument. Other calls produce a return value, which the hardware thread needs to wait for (e.g., `reconos_mbox_get()`). Neither of these calls can be completed in a single clock cycle. Furthermore, these calls need to interact with the OSIF across multiple clock cycles, ruling out simply delaying the state transition until the call completes and then resuming with the next call.

To address these issues, the command decoder implements a mechanism for *multi-cycle commands*. In the case of a single call requiring different actions in subsequent clock cycles, the VHDL procedure is evaluated for more than one clock cycle, and only if all steps are completed successfully, the OS synchronization state machine transitions to the next state. Every multicycle VHDL API procedure takes one additional argument, `completed`. This argument, implemented as a VHDL variable, returns *false* as long as not all steps have been completed. Only in the last step, `completed` is set to *true*, which then prompts the state transition. Thus, a multicycle command induces additional state that keeps track of the currently executing step of the command. This state is kept by the OSIF and transmitted via the `step` signal inside the `osif2task` record

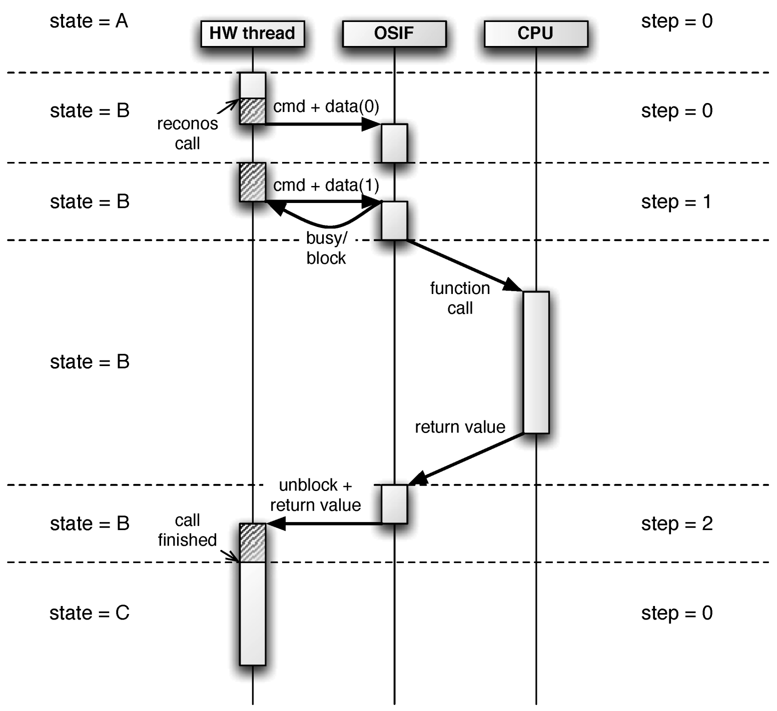


Fig. 5. Multicycle command example.

to the VHDL procedure, which uses it to perform the appropriate function for this step.

An example of this mechanism is depicted in Figure 5. Here, an OS call taking two arguments and returning a third value is requested, requiring three steps to complete. On entering state B of the OS synchronization state machine, the hardware thread invokes the appropriate VHDL procedure, which transmits the first (state B, Step 0) and second (state B, Step 1) argument. The OSIF then blocks the thread's OS synchronization state machine by setting the busy and/or blocking signals and relays the OS call to the CPU, where the associated delegate thread executes it. Upon returning from the software OS call, the OSIF unblocks the hardware thread and passes the return value in state B, Step 2, where it is stored by the same VHDL procedure that invoked the call. Since Step 2 is the last step of this command, the `completed` variable is set, prompting the OS synchronization state machine to enter state C, Step 0 in the next clock cycle.

This mechanism is highly flexible and largely transparent to the thread developer. It does, however, require some additional VHDL code to check for the `completed` variable (as shown in Listing 2).

4.1.2 *OS Call Relaying.* OS services that are not provided by the OSIF directly (such as memory or FIFO accesses) are relayed to the OS kernel running on the CPU. Once the command decoder receives such a request from the hardware thread, it places the command and associated arguments in software-accessible registers on the DCR bus, and raises an interrupt with the CPU. This interrupt is forwarded to the software delegate thread associated with the hardware thread (see Section 4.2), which retrieves the command and arguments from the registers and executes the software OS call on behalf of the hardware thread. Any return values are placed in the OSIF's DCR registers, which pass the values on to the hardware thread.

This mechanism provides maximum flexibility, since virtually every call that is possible from a software thread can now be requested by a hardware thread as well. However, OS call relaying comes with a considerable overhead, which is quantified in Section 6.1. On every relayed OS call, the CPU needs to process an interrupt, switch to the associated delegate's context, and access the DCR bus registers before actually executing the call. During this time, the hardware thread's OS synchronization state machine remains suspended. Nonetheless, it must be noted that the parallel user processes inside the thread may continue their execution.

4.1.3 *Data Communication Routing.*

Due to the substantial overhead involved in relaying OS requests to software, all high-throughput data communications should be handled in hardware without involving the CPU. In the ReconOS OSIF, this is realized in two variants, which provide the basis for any efficient, high-bandwidth thread-to-thread communication.

—*Bus Master Access.* By utilizing the OSIF's PLB interface, a hardware thread has direct access to any memory location and bus-connected peripheral in the system. Using the bus master controller (see Figure 4), it is even possible to transfer bursts of data to and from memory. To request a burst write, the hardware thread must first store the data to transfer in the thread-local RAM. Then, the thread's OS synchronization state machine calls a `reconos_write_burst()` procedure. This prompts the bus master controller to initiate a PLB bus transfer from the local RAM, which is mapped into the system memory space, to the target address in main memory. Similarly, a thread can request a burst read transaction, which will place data from main memory in the local RAM.

—*Hardware FIFOs.* The bus access facilities provided by the OSIF permit the hardware thread to achieve high data transfer rates to and from main memory. While this mechanism represents an improvement over the indirect communication methods provided by the OS call relay technique, their performance can suffer considerably when several threads, the CPU, or other peripherals contend for the bus.

To allow for bus-independent thread-to-thread data communication, the ReconOS run-time environment provides dedicated FIFO buffers implemented in hardware. Two threads connected by such a FIFO module can transfer data without interrupting the CPU or increasing bus load. When a hardware thread signals a pending read or write access to such a FIFO, the OSIF's command decoder passes the request to the *FIFO manager* (see Figure 4), which controls the handshake lines of the FIFO modules. In the event of a write request to a full FIFO or a read request to an empty FIFO, the FIFO manager can also

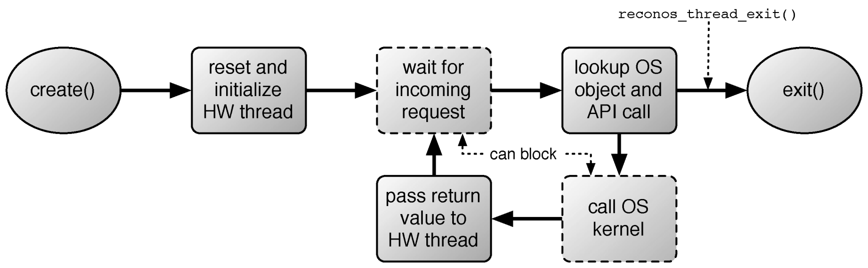

reconos_thread_exit()
create()
reset and initialize HW thread
wait for incoming request
lookup OS object and API call
exit()
can block
pass return value to HW thread
call OS kernel

Fig. 6. Execution flow of a delegate thread.

suspend the hardware thread's OS synchronization state machine, thus providing blocking *get*/*put* operations on FIFOs. Details on the performance of this message passing mechanism can be found in Section 6.2.

4.2 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 transparency. 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 the hardware thread. It then waits for an incoming OS request from the hardware to execute. The basic execution flow of a delegate thread is depicted in Figure 6.

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 (see Section 3.4). 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.

5. IMPLEMENTING RECONOS

We have implemented the programming and execution models described in Sections 3 and 4 on top of two wide-spread operating system kernels: eCos and

Table III. Comparison eCos/Linux

	eCos	Linux	Linux nommu
virtual memory (requires MMU)	no	yes	no
run-time loading of object code (spawning of new processes)	no	yes	yes
real-time characterization of system functions	yes	partial[1]	partial[2]
user code has access to privileged instructions (executes in supervisor mode)	yes	no	no

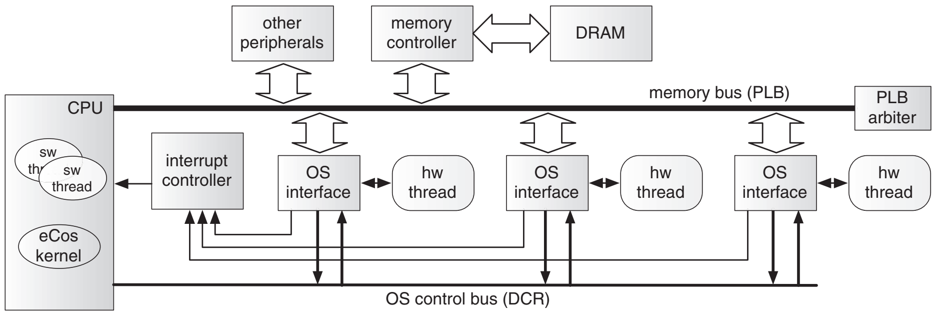


Fig. 7. ReconOS hardware architecture with three hardware threads.

Linux. In the following, we present the ReconOS hardware architecture and the prototype systems. While eCos is designed primarily for embedded systems with limited resources, Linux is targeted at a wider range of application areas and target platforms. As a consequence, both systems provide a somewhat disparate feature set, shown in Table III, which also influences the way our programming model is implemented. Many differences, however, can be hidden from the application programmer behind the POSIX API, which is supported by both eCos and Linux. Since hardware threads are written with the separate ReconOS API that is similar to POSIX, we can run software and hardware threads on both operating system kernels with little to no changes to the source code.

5.1 Hardware Architecture

The ReconOS hardware architecture, shown in Figure 7, is built on top of the IBM CoreConnect bus topology available for Xilinx FPGAs. The basic system architecture is independent from the employed host operating system, which is executed in software on the system CPU.

Hardware threads are connected to the system via their OS interfaces, which, in turn, are connected to the system's buses. ReconOS systems employ two different buses: the processor local bus (PLB) and the device control register bus (DCR). The 64-bit PLB is used for high-throughput data transfers. Both

[1]Through third-party extensions (e.g., RTAI and RTLinux).
[2]In RTAI-68k-nommu.

Table IV. ReconOS Prototype Implementations

Prototype (ReconOS/–)	eCos-PPC	Linux-PPC	Linux-MicroBlaze
operating system	eCos	Linux	Linux
based on kernel	eCos-VIRTEX4[3]	2.6-virtex[4]	2.6-nommu[5]
CPU	PowerPC 405	PowerPC 405	MicroBlaze 4.0
FPGA	XC2VP30	XC2VP30	XC4VSX35
CPU clock	300 MHz	300 MHz	100 MHz
PLB/DCR bus clock	100 MHz	100 MHz	100 MHz
MMU	no	yes	no

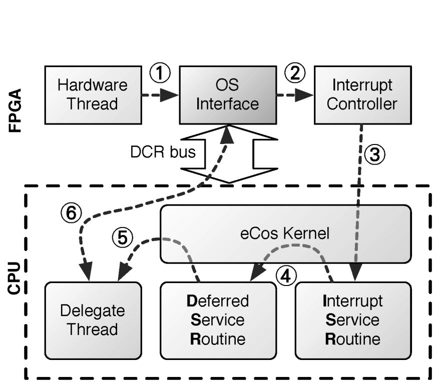
the CPU cache subsystem and the hardware threads use it to access main memory and system peripherals. All control communication between the OS kernel on the CPU and the threads' OS interfaces is routed across a separate 32-bit DCR bus. The separation of control and data communications provides several benefits:

—OS control communications do not obstruct data communications on the memory bus, thus reducing the PLB's arbitration overhead and latency.
—Vice versa, memory communications, especially bursts, can not interfere with OS communications. This reduces the latency of OS calls, which is paramount to the use of ReconOS in real-time environments.
—OS interfaces for hardware threads that do not need direct access to system memory can be synthesized without the PLB interface, thereby greatly reducing the area footprint. Such threads are typical for many signal processing applications that arrange filter stages in pipeline form, connected by hardware FIFOs (see Section 4.1.3).

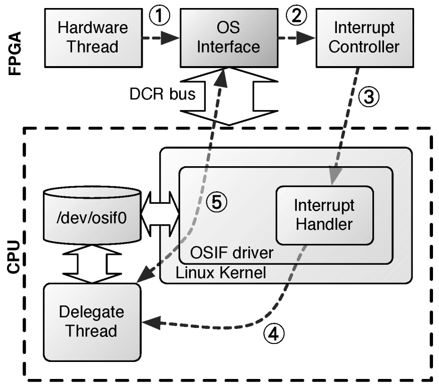
Based on the hardware architecture shown in Figure 7 and two OS kernels, we have created three ReconOS prototypes: ReconOS/eCos-PPC and ReconOS/Linux-PPC on a XC2VP30 FPGA, and ReconOS/Linux-MicroBlaze on a XC4VSX35 FPGA. The main features of these prototypes are listed in Table IV. External SDRAM is used for both the operating system and shared thread memory. The prototypes also include I/O peripherals, such as serial ports, Ethernet interfaces, and general-purpose I/O, all of which are managed by the software operating system kernel. Both the OS interfaces and the hardware threads run at the system's bus clock, which is 100MHz for all prototypes.

A single OS interface requires 1,147 slices, which amounts to about 8.4%/7.5% of the logic resources of the employed XC2VP30 and XC4VSX35 FPGAs, respectively. Roughly two thirds of the OS interface slices are taken up by the PLB bus interface, while the remaining logic is mainly used for the command decoder (267 slices) and the DCR bus interface (56 slices). The hardware architecture has been assembled and synthesized using Xilinx ISE 9.1, Xilinx EDK 9.1, and custom tools for automated OS interface and thread instantiation.

[3]Mind NV [2008]
[4]Secret Lab Technologies Ltd. [2008]
[5]PetaLogix [2007]



(a) eCos



(b) Linux

Fig. 8. Communication between hardware thread and delegate thread.

5.2 ReconOS/eCos

The eCos [eCosCentric 2008] 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. Currently, this limits applying eCos to Xilinx FPGAs of the Virtex-II Pro, Virtex-4 FX, and Virtex-5 FXT families.

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 kernel code allows ReconOS delegate threads (and, by extension, the associated hardware threads) to take advantage of all 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 8(a). 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 calls requested by the OSIF are signalled to the CPU's interrupt controller via a dedicated interrupt line (2). In eCos, interrupt processing is split into two steps to minimize interrupt latency. First, a very small *interrupt service routine* (ISR) 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* (DSR) (4) for execution. The latter is scheduled by the regular eCos scheduler so as not to interfere with the low-level interrupt processing. As the last step before the actual delegate thread is invoked, the DSR posts a semaphore (5), which the delegate is waiting on, essentially signaling 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. Section 6.1 evaluates the timing overhead of this OS call sequence.

5.3 ReconOS/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 ReconOS targets to include FPGAs without an embedded CPU core. The MicroBlaze can be synthesized with or without an MMU. For our MicroBlaze prototype, we have opted for the omission of the MMU, which simplifies memory transfers between software and hardware threads.

While offering a wide set of configurable options, the Linux kernel does not allow to reduce its memory footprint as much as the eCos kernel. 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 exceeds 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 privileged instructions. To make the OSIF registers accessible to the delegate, we have thus 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 8(b). 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 for the eCos kernel. 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. The timing overhead of this sequence is also analyzed in Section 6.1.

Data transfers between software and hardware threads are complicated by the fact that Linux usually employs virtual memory. This means that blocks of shared memory used to transfer data to or from hardware threads are not necessarily contiguous. Then, hardware threads operate on physical addresses, while software threads use virtual addresses that are translated by the MMU. Moreover, it is not possible for user applications to flush or invalidate the processor's caches in order to maintain cache coherency. Our current implementation of ReconOS/Linux-PPC therefore, uses a separate, uncached memory buffer that is advertised to the kernel as a memory mapped device. Section 7 outlines more transparent solutions that integrate more easily with the ReconOS programming model.

6. EXPERIMENTAL RESULTS AND MEASUREMENTS

The ReconOS programming and execution models have been experimentally verified using several prototypes based on the implementations described in Section 5. In the following, we present quantitative performance results of operations on operating system primitives on both host operating systems, which provides valuable pointers on the costs and overheads involved with individual thread interactions. Then, we discuss another set of experiments that has been conducted to evaluate the throughput of the different communication mechanisms available to ReconOS threads. Lastly, we focus on two more elaborate application case studies to analyze the impact of the operating system overheads on real-world implementations. The results demonstrate the applicability, feasibility, and portability of the proposed multithreaded programming model.

6.1 Operating System Overheads

To enable quantitative measurements on the performance of operating system calls, we have run a set of benchmarks on the three prototype implementations listed in Table IV. 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 I serve as representative examples, as most other supported API calls are either based on them or are not considered timing critical.

The threads measure the raw execution time of single API calls to lock (unlock) a mutex or post (wait on) a semaphore, respectively, as well as a measure we call the *turnaround time*. The turnaround time is defined as the time it takes from one thread releasing a mutex (posting a semaphore), to the next thread acquiring a lock (receiving the semaphore).

Table V. Performance of ReconOS Synchronization Primitives

	eCos/PPC	Linux/PPC	Linux/MicroBlaze
mutex (raw OS calls)			
SW lock	83	821	9,178
SW unlock	171	551	9,179
HW lock	959	7,769	35,855
HW unlock	679	2,636	22,360
mutex (turnaround)			
SW → SW	453	8,821	83,657
SW → HW	629	9,824	90,515
HW → SW	1,449	14,371	121,673
HW → HW	1,460	14,102	126,668
semaphore (raw OS calls)			
SW post	73	598	13,180
HW post	695	1,972	22,116
semaphore (turnaround)			
SW → SW	305	9,094	203,221
SW → HW	528	9,575	207,824
HW → SW	908	12,291	145,924
HW → HW	1,114	12,196	154,013

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

The experiments have been run with different combinations of software and hardware threads. The results are shown in Table V.

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 behavior 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 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 effect can be observed only on the Linux/MicroBlaze prototype and will be further investigated.

While the synchronization overhead incurred by the operating system is not negligible, its impact on system performance remains within reasonable bounds, as the application case studies in Section 6.3 will show.

6.2 Communication Primitives

In a second experiment, we have analyzed the attainable throughput for the communication primitives available to ReconOS threads. Two threads perform a sequence of data transfers, subsequently reading and writing data from and to main memory, as well as reading and writing data from and to a mailbox. Several configurations of the test have been run, using hardware and software threads, and with mailboxes mapped either to hardware FIFOs or to eCos

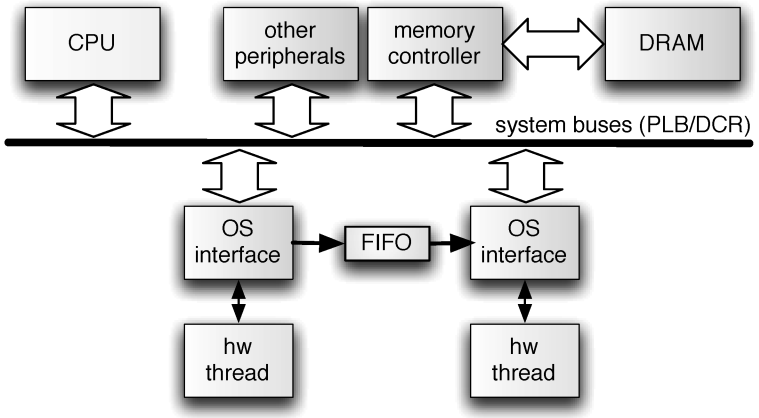


Fig. 9. Hardware architecture with a thread-to-thread FIFO.

software mailboxes. The throughput of ReconOS communication primitives for hardware threads depends primarily on the specifics of the hardware architecture (memory bus, hardware FIFOs), which is identical for the ReconOS/eCos and ReconOS/Linux prototypes. For this test, the ReconOS/eCos prototype employing a PowerPC processor was selected.

Figure 9 shows the architecture used for testing the configuration with two hardware threads and a hardware FIFO. The first hardware thread reads 8kB of data from main memory into its local RAM. It then uses the ReconOS mailbox calls to transfer this data to the hardware FIFO, one 32-bit word at a time. Simultaneously, the second hardware thread reads from the hardware FIFO, also by using the ReconOS mailbox API. Once this data transfer is completed, the second thread writes the data back to main memory.

The hardware FIFOs are implemented as parametrizable IP cores that can be easily instantiated and connected via the Xilinx EDK, or by the ReconOS build system. To transfer one word of data to or from a FIFO, a hardware thread needs three cycles. This includes all handshaking between hardware thread and the OSIF's command decoder as well as between the OSIF's FIFO manager and the FIFO core. The additional FIFO manager increases the OSIFs area requirements only by 64 slices or 5%.

During the experiment, we have measured the times for reading and writing the data from and to main memory, and the times for writing and reading the data to and from the mailboxes. For comparison, we have also measured the times for data transfer between hardware and software threads using ReconOS' message queue primitives. Since software threads do not possess local memory, the memory read/write tests for software threads have been combined into a single *memcopy* test. The results are shown in Table VI.

While the hardware FIFOs only achieve 66% to 74% of the memory bus (PLB) in terms of raw throughput, one has to keep in mind that in order to transfer data from one thread to another, two memory transactions have to occur: First, the sending thread needs to write to shared memory, before the receiving thread can read the data. When using hardware FIFOs, reading and writing can occur

Table VI. Performance of ReconOS Communication Primitives

	With Data Cache		Without Data Cache	
Operation	[μs]	[MB/s]	[μs]	[MB/s]
MEM→HW (burst read)	45.74	170.80	46.41	168.34
HW→MEM (burst write)	40.54	192.71	40.55	192.66
MEM→SW→MEM (memcopy)	132.51	58.96	625.00	12.50
HW→HW (mailbox)	61.42	127.20	61.42	127.20
SW→HW (mailbox)	58,500	0.13	374,000	0.02
HW→SW (mailbox)	58,510	0.13	374,000	0.02
SW→HW (message queue)	472.00	16.55	2,166.79	3.61
HW→SW (message queue)	482.31	16.20	2,160.69	3.62

All operations were run for 8 kBytes of data.

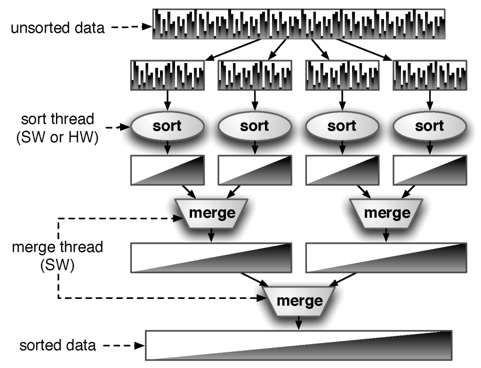
concurrently. Considering this, an 8kB data transfer via hardware FIFOs is about 40% faster than a transfer of the same data via shared memory. Also, the transfer via mailboxes is implicitly synchronized, while two threads exchanging data via shared memory need explicit synchronization, for example, via mutexes or semaphores.

The previous figures show that for applications able to chain several hardware threads together for data processing, the hardware FIFOs provide improved performance and reduced bus load over shared memory. Importantly, hardware FIFOs fully maintain transparency and flexibility using the ReconOS programming model abstractions. For mailbox-based data transfers across the hardware/software boundary, we currently use regular eCos software mailboxes with data structures located in shared memory. It should be noted that mailbox-based data transfer between hardware and software threads is thus rather inefficient. On the other hand, direct shared memory communication with several orders of magnitude better performance comes at the cost of explicit synchronization and cache coherency issues. A compromise between transparency and performance is established by the ReconOS message queue primitives, which map directly to POSIX message queues and hide the details of shared memory access and explicit cache management from the user. In Section 7, we propose a modification to our hardware FIFOs that will further alleviate the performance bottleneck for synchronized communication across the HW/SW boundary.

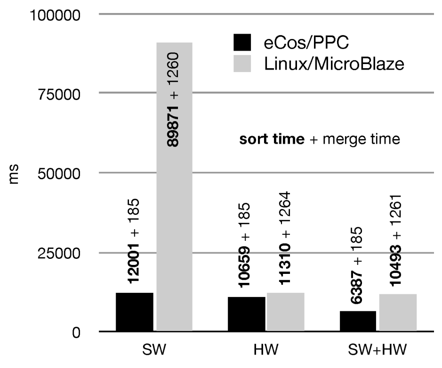
6.3 Application Case Studies

To analyze the real-world implications of the ReconOS overheads on the overall system performance and to demonstrate the feasibility of hardware/software system design based on the ReconOS programming model, we present two more elaborate case studies, a sorting and an image processing application.

6.3.1 *Sorting Application.* We have implemented a multithreaded sorting algorithm with ReconOS and mapped it to different host operating systems and underlying hardware architectures to demonstrate the portability of applications based on the ReconOS programming model. A list of 2^{18} unsorted 32-bit integers is sorted, using a combination of bubble sort and merge sort; the basic concept is depicted in Figure 10. First, the data is divided into 128 chunks, which



(a) Concept of sort application



(b) Execution times (in ms)

Fig. 10. Sort case study.

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 simultaneous synchronization and passing of buffer addresses. The application has been run on two prototype platforms, one running ReconOS/eCos on a PowerPC, the other running ReconOS/Linux on a MicroBlaze. Both systems use exactly the same application code for both software and hardware threads. Three tests have been 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 10. In this figure, two times are given for each test and architecture, the first (bold) value denotes the time spent sorting, while the second corresponds to 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 thread interacts with the OS synchronization primitives infrequently enough so that the performance penalty due to additional software processing remains within acceptable limits. This is a typical scenario: An application designer will likely use the precious hardware resources for data-centric computations with relatively infrequent 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 marginal.

6.3.2 *Image Processing Application.* A second application running on the ReconOS/eCos prototype demonstrates the iterative design approach made possible by the multithreaded programming model. Here, the transparent

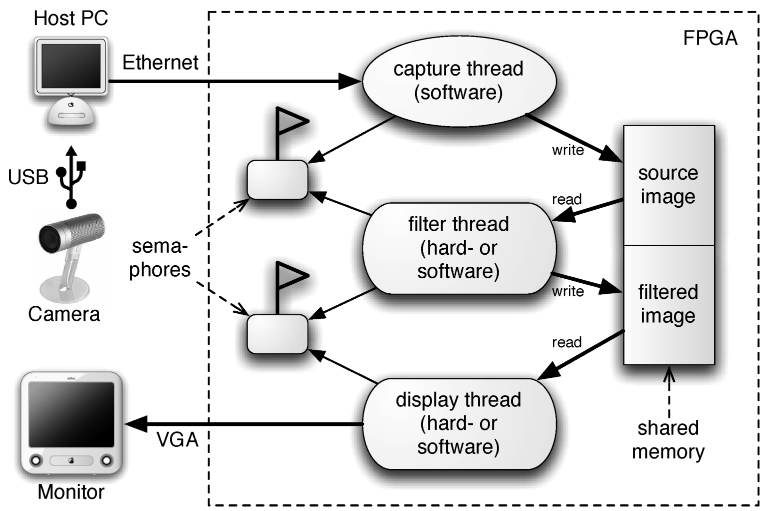


Fig. 11. Image-processing application.

interthread communication and synchronization regardless of the actual execution context facilitates design-space exploration. In this application, grayscale image data is acquired from a web cam 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 11. Data is passed between the threads through shared memory, while semaphores synchronize access to the memory.

The application was first implemented and tested purely in software, where all three threads are scheduled in sequence, as shown in Figure 12(a). Then, as a first try at optimization, 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.

Table VII 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. If we execute all three threads in sequence, as shown in Figure 12(b), the theoretical speed-up for this configuration amounts to 1.4 for a 3×3 filter, and 2.7 for a 5×5 filter. In practice, these speed-ups will not be reached due to the overhead of the OS.

Although, at this point, the application utilizes the FPGAs fine-grained parallelism by performing the convolution filter in hardware, the potential of thread-level parallelism is not yet exploited. Therefore, the next optimization step has been to implement the display thread, which postprocesses the filtered image for display, in hardware. Because the display and capture threads can

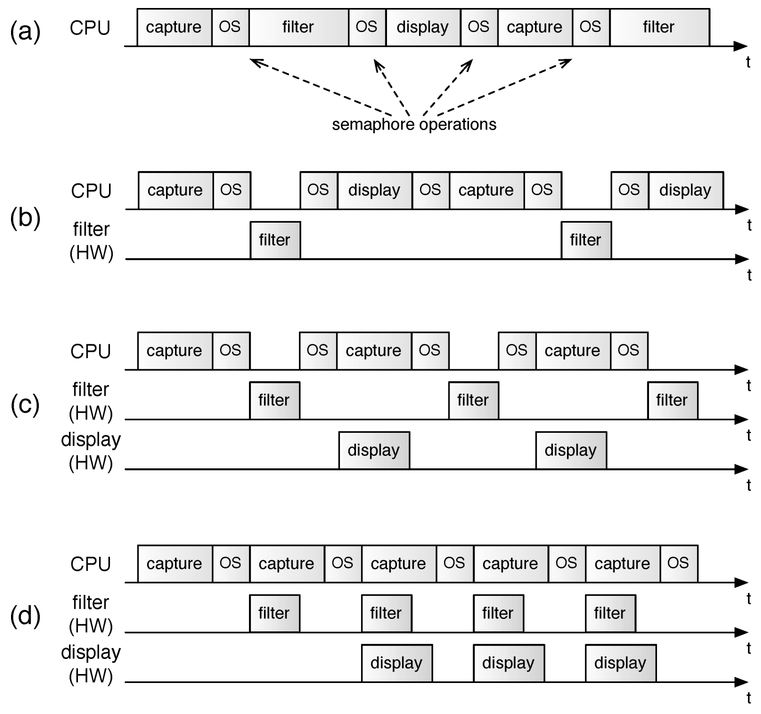


Fig. 12. Thread-level parallelism for different configurations of the image processing application.

Table VII. Raw Execution Times [ms/Frame]

Thread	Software	Hardware
capture	16.0	—
filter 3 × 3	23.9	6.0
filter 5 × 5	86.6	7.6
display	22.5	3.1

now be run in different execution contexts (CPU and FPGA), they do not have to be executed in sequence anymore, which is shown in Figure 12(c). To further improve the thread-level parallelism, double buffering of the image data has been introduced. This allows all three threads to run truly concurrently and is depicted in Figure 12(d).

The image processing application has been run with differently sized blocks of data. Larger block sizes reduce the system call overhead for semaphore synchronization, but require more shared memory. Table VIII lists the resulting performance figures in frames per second for different Laplacian kernel sizes and software and hardware thread configurations. The *configuration* column indicates whether the threads (capture—laplace—display) have been run in software or hardware—the letters in parenthesis correspond to the configurations shown in Figure 12. For the SW-HW-HW* configurations, double-buffering has been enabled.

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

Table VIII. System Performance in [Frames/s]

w	Configuration		Block Size [image lines] 4	8	20	40	80
3	SW-SW-SW	(a)	14.4 (1.00)	15.5 (1.00)	16.1 (1.00)	16.2 (1.00)	16.3 (1.00)
	SW-HW-SW	(b)	15.5 (1.08)	17.6 (1.14)	18.6 (1.16)	**19.0 (1.17)**	18.9 (1.16)
	SW-HW-HW	(c)				23.5 (1.45)	23.4 (1.44)
	SW-HW-HW*	(d)				25.5 (1.57)	25.2 (1.55)
5	SW-SW-SW	(a)	8.1 (1.00)	8.3 (1.00)	8.5 (1.00)	8.5 (1.00)	8.5 (1.00)
	SW-HW-SW	(b)	15.3 (1.89)	17.0 (2.05)	18.4 (2.16)	**18.6 (2.19)**	18.6 (2.19)
	SW-HW-HW	(c)				23.2 (2.73)	23.0 (2.71)
	SW-HW-HW*	(d)				25.4 (2.99)	25.1 (2.95)

Figures in parenthesis denote relative speed-ups.

exploit more fine-grained parallelism and delivers an almost constant performance. Also, we see that the resulting overall speed-ups of the sequential (SW-HW-SW) configuration (marked in bold) are quite close to the theoretically achievable speed-ups of 1.4 and 2.7 mentioned earlier. This points to an acceptable overhead of the ReconOS system calls.

Naturally, the performance of the application could be further improved by additional low-level optimizations. However, the case study serves 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.

7. CONCLUSION AND OUTLOOK

The increasing complexity of reconfigurable platforms calls for novel approaches to model and implement both software and hardware parts of an application in a portable and scalable way. ReconOS offers such a way by extending the multithreaded programming model from its established software domain toward reconfigurable circuits, creating a common abstraction layer for both software and hardware. A common set of communication and synchronization primitives is made available to both software and hardware threads in order to leverage the full potential of today's reconfigurable computers while retaining portability across different hardware and software execution platforms.

In this article, we have detailed the ReconOS programming model, as well as its execution environment. Then, we have shown prototype implementations running on different host operating systems and hardware platforms. Finally, we have analyzed and discussed experimental measurements on OS call overheads and communication performance and have demonstrated the applicability of our approach on two more elaborate case studies.

Ongoing and future work focuses on three main aspects:

—*Partial Reconfiguration*. While in our current ReconOS prototypes hardware threads are statically configured, the partial reconfiguration capabilities of modern Xilinx FPGAs allow for dynamic loading. We are working on the integration of these mechanisms into our run-time environment to enable ReconOS to replace inactive or terminated hardware threads with active ones and thus increase resource utilization. This involves a supporting hardware infrastructure, which is already in place, as well as the investigation of suitable scheduling techniques. We plan to leverage our existing work on scheduling (see Danne et al. [2006, 2007]) and extend the host operating system's schedulers to deal with loadable hardware threads.

—*Communication Primitives*. While the existing set of operating system objects is sufficient to model the thread interactions of quite complex multithreaded applications, we see potential for improvement especially with the performance-sensitive communication services. The performance benefits of hardware FIFOs for message-based communication, as detailed in Section 4.1.3, will be extended also to data exchanges across the hardware/software boundary. We are currently investigating an extension to the hardware FIFOs that allows hardware threads not directly adjacent to a particular FIFO as well as software threads running on the CPU to access the FIFO. Such hardware FIFOs will further reduce contention on the memory bus and CPU load, which is currently impeding performance of hardware-software communication using message boxes.

—*Virtual Memory*. As mentioned in Section 5.3, communication between hardware and software threads can be significantly complicated if the host operating system uses virtual memory. In the presence of virtual memory, we see three ways to implement shared memory as a means to communicate across the hardware/software boundary: (a) by using a separate, uncached memory buffer, which is advertized to the kernel as a memory mapped device; (b) by allocating a 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 mirrors the CPU's TLB, similar to Vuletic et al. [2005]. The first two options require a separate device driver and 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 future research.

REFERENCES

AGRON, J., PECK, W., ANDERSON, E., ANDREWS, D., KOMP, E., SASS, R., BAIJOT, F., AND STEVENS, J. 2006. Run-time services for hybrid CPU/FPGA systems on chip. In *Proceedings of the 27th International Real-Time Systems Symposium (RTSS'06)*. IEEE, Los Alamitos, CA, 3–12.

ANDERSON, E., PECK, W., STEVENS, J., AGRON, J., BAIJOT, F., WARN, S., AND ANDREWS, D. 2007. Supporting high-level language semantics within hardware resident threads. In *Proceedings of the 17th International Conference on Field-Programmable Logic and Applications (FPL'07)*. IEEE, Los Alamitos, CA, 98–103.

BAZARGAN, K., KASTNER, R., AND SARRAFZADEH, M. 2000. Fast template placement for reconfigurable computing systems. *IEEE Des. Test Comput. 17*, 1, 68–83.

BERGMANN, N. W., WILLIAMS, J. A., HAN, J., AND CHEN, Y. 2006. 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. Springer, Berlin, 205–214.

Brebner, G. 1997. The swappable logic unit: A paradigm for virtual hardware. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, Los Alamitos, CA, 77–86.

Brebner, G. J. 1996. A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'96)*. Springer-Verlag, Berlin, 327–336.

Burns, J., Donlin, A., Hogg, J., Singh, S., and de Wit, M. 1997. A dynamic reconfiguration run-time system. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*. IEEE, Los Alamitos, CA, 66–75.

Compton, K., Li, Z., Cooley, J., Knol, S., and Hauck, S. 2002. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. VLSI Syst. 10*, 3, 209–220.

Danne, K., Mühlenbernd, R., and Platzner, M. 2007. Server-based execution of periodic tasks on dynamically reconfigurable hardware. *IET Comput. Digital Tech. 1*, 4, 295–302.

Danne, K. and Platzner, M. 2006. An EDF schedule test for periodic tasks on reconfigurable hardware devices. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*. ACM, New York.

Diessel, O., ElGindy, H., Middendorf, M., Schmeck, H., and Schmidt, B. 2000. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proc. Comput. Digital Tech. 147*, 3, 181–188.

Duchene, M. and Hanna, D. 2005. Implementing parallel algorithms on an FPGA directly from multi-threaded Java using flow-paths. In *Proceedings of the 48th Midwest Symposium on Circuits and Systems*. IEEE, Los Alamitos, CA, 980–983.

eCosCentric. 2008. eCos. http://ecos.sourceware.org/.

IEEE and The Open Group. 2004. The Open Group Base Specifications Issue 6, IEEE Std. 1003.1. http://www.opengroup.org/onlinepubs/009695399/

Jean, J. S. N., Tomko, K., Yavagal, V., Shah, J., and Cook, R. 1999. Dynamic reconfiguration to support concurrent applications. *IEEE Trans. Comput. 48*, 6, 591–602.

Kalte, H. and Porrmann, M. 2005. Context saving and restoring for multi-tasking in reconfigurable systems. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'05)*. IEEE, Los Alamitos, CA, 223–228.

Kosciuszkiewicz, K., Morgan, F., and Kepa, K. 2007. Run-time management of reconfigurable hardware tasks using embedded Linux. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'07)*. IEEE, Los Alamitos, CA, 209–215.

Lee, E. A. 2006. The problem with threads. *IEEE Comput.*, 33–42.

Lübbers, E. and Platzner, M. 2007. ReconOS: An RTOS supporting hard- and software threads. In *Proceedings of the 17th International Conference on Field-Programmable Logic and Applications (FPL'07)*. IEEE, Los Alamitos, CA, 441–446.

Lübbers, E. and Platzner, M. 2008a. A portable abstraction layer for hardware threads. In *Proceedings of the 18th International Conference on Field-Programmable Logic and Applications (FPL'08)*. IEEE, Los Alamitos, CA.

Lübbers, E. and Platzner, M. 2008b. Communication and synchronization in multi-threaded reconfigurable computing systems. In *Proceedings of the 8th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'08)*. CSREA Press.

Merino, P., Lopez, J. C., and Jacome, M. 1998. A hardware operating system for dynamic reconfiguration of FPGAs. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications (FPL'98)*. Springer-Verlag, Berlin, 431–435.

Mignolet, J.-Y., Vernalde, S., Verkest, D., and Lauwereins, R. 2002. Enabling hardware-software multi-tasking on a reconfigurable computing platform for networked portable multimedia appliances. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, 116–122.

Mind, NV. 2008. Release of the eCos port to the Xilinx Virtex4 ML403 board. http://www.mind.be/?page=ML403.

Nollet, V., Coene, P., Verkest, D., Vernalde, S., and Lauwereins, R. 2003. Designing an operating system for a heterogeneous reconfigurable SoC. In Reconfigurable Architectures Workshop (RAW), *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Los Alamitos, CA.

PECK, W., ANDERSON, E., AGRON, J., STEVENS, J., BAIJOT, F., AND ANDREWS, D. 2006. Threads: A computational model for reconfigurable devices. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Los Alamitos, CA, 885–888.

PELLIZZONI, R. AND CACCAMO, M. 2007. Real-time management of hardware and software tasks for FPGA-based embedded systems. *IEEE Trans. Comput. 56*, 12, 1666–1678.

PETALOGIX. 2007. Petalinux. http://developer.petalogix.com/

QUADROS SYSTEMS, INC. 2007. RTXC 3.2 real-time kernel. http://www.quadros.com/products/operating-systems/rtxc-32/

SECRET LAB TECHNOLOGIES LTD. 2008. Linux on Xilinx Virtex. http://wiki.secretlab.ca/index.php/Linux_on_Xilinx_Virtex

SHIRAZI, N., LUK, W., AND CHEUNG, P. 1998. Run-time management of dynamically reconfigurable designs. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications (FPL'98)*. Springer-Verlag, Berlin, 59–68.

SIMMLER, H., LEVINSON, L., AND MÄNNER, R. 2000. Multitasking on FPGA Coprocessors. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'00)*. Springer-Verlag, Berlin, 121–130.

SO, H. K.-H. AND BRODERSEN, R. W. 2006. 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, Los Alamitos, CA, 349–354.

STEIGER, C., WALDER, H., AND PLATZNER, M. 2004. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput. 53*, 11, 1392–1407.

TEICH, J., FEKETE, S., AND SCHEPERS, J. 2000. Optimization of dynamic hardware reconfigurations. *J. Super-Comput. 19*, 1, 57–75.

VULETIĆ, M., POZZI, L., AND IENNE, P. 2005. Seamless hardware-software integration in reconfigurable computing systems. *Des. Test Comput. IEEE 22*, 2, 102–113.

WALDER, H. AND PLATZNER, M. 2003. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*. CSREA Press, 284–287.

WIGLEY, G. AND KEARNEY, D. 2001. The Development of an operating system for reconfigurable computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA.

WILLIAMS, J. A., BERGMANN, N. W., AND XIE, X. 2005. FIFO communication models in operating systems for reconfigurable computing. In *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE, Los Alamitos, CA, 277–278.

WIND RIVER. 2007. VxWorks 6.x. http://www.windriver.com/products/run-time_technologies/Real-Time_Operating_Systems/VxWorks_6x/

XIE, X., WILLIAMS, J., AND BERGMANN, N. 2007. Asymmetric multi-processor architecture for reconfigurable system-on-chip and operating system abstractions. In *Proceedings for the International Conference on Field-Programmable Technology (ICFPT'07).* IEEE, Los Alamitos, CA, 41–48.

Received June 2008; revised January 2009; accepted February 2009