In Proceedings of the 8th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, July 2008

Communication and Synchronization in Multithreaded Reconfigurable Computing Systems

Enno Lübbers and Marco Platzner
University of Paderborn
Email: {enno.luebbers, platzner}@upb.de

***Abstract*—This paper describes an approach to provide communication and synchronization services to hardware threads being executed on reconfigurable devices under the control of a software-based operating system. This work aims at enabling hardware circuits to be modeled as active, independently executing threads with access to all operating system services, instead of passive coprocessors that can simply be called from software.**

Communication and synchronization in a multithreaded reconfigurable system poses a challenge on two different levels: First, hardware circuits need to communicate and synchronize on a low level with the operating system kernel. Second, inter-thread communication and synchronization as the basic services provided by the multithreaded programming model need special consideration when applied to reconfigurable devices. We describe the concepts and methods to provide these services as implemented in the ReconOS operating system, and quantify the efficiency of the proposed techniques.

I. Introduction

Design methodologies for embedded systems involving reconfigurable logic have not kept pace with the increasing complexity of modern platform FPGAs. The prevalent design technique of modeling hardware accelerators as passive coprocessors to the system CPU is not able to exploit the full potential of modern devices in a scalable way.

At the same time, the advent of multi-core processors both in the desktop and embedded computing domains aggressively promotes the use of the multithreaded programming model as a means of segmenting computations into a set of independent execution units. In the software world, there exist a multitude of abstractions and established APIs such as OpenMP, POSIX threads and others, which allow application designers to express the parallelism inherent in the applied algorithms in terms of independent threads of execution. This programming model is flexible enough to be supported across many application domains, from distributed high-performance computing clusters, multi-core desktop CPUs, over gaming consoles to embedded systems employing multi-core DSPs. Extending this programming model to the design of reconfigurable systems will significantly reduce design complexity, support reusability, facilitate design space exploration as well as promote portability.

The operating system ReconOS [1] builds on top of an existing (real-time) operating system, modifying it to support hardware threads. ReconOS uses and extends the existing, established API of embedded real-time operating systems as an abstraction layer to enable transparent thread-to-thread communication regardless of the hw/sw partitioning. Software threads run directly on the CPU, while hardware components access the operating system services through an RPC-like mechanism using the same set of APIs that is available to software threads.

The main contribution of this paper is the description of concepts and methods used by ReconOS to provide flexible synchronization and communication mechanisms for multithreaded hardware/software systems on both the level of the programming model as well as the low-level implementation of the hardware/software interface.

The remainder of this paper is structured as follows: Section II reviews related work. A general overview of the ReconOS architecture is given in Section III. Section IV details the low-level synchronization and communication mechanisms that connect hardware threads to the operating system, while Section V describes the high-level abstractions for synchronization and communication provided by the programming model. Experimental results acquired from a prototype implementation are presented and analyzed in Section VI. Section VII summarizes the results of this work and points to ongoing and future work.

II. Related Work

In recent years, considerable research has been carried out in the field of operating systems for reconfigurable computing. Brebner [2] was the first to discuss the notion of hardware multitasking and supporting reconfigurable devices with operating system services. In the following years, most efforts focused on single problems and techniques to manage hardware processes as resources. Examples are the integrated resource management and scheduling by Danne et al. [3], and the services for task relocation and preemption using partial reconfiguration by Hinkelmann et al. [4]. However, these approaches did not try to integrate hardware processes as independent execution units into an operating system. An example for such an integrated approach is shown in Walder et al. [5], where a hybrid system composed of a CPU and reconfigurable logic manages tasks that execute in both software and hardware and access the same operating system resources.

Recently, some authors presented extensions of Linux-based operating systems that facilitate communication between software threads and hardware processes. Kosciuszkiewicz et al. [6] build on top of an existing Linux operating system kernel and try to model hardware

tasks as a drop-in replacement for software tasks, maximizing transparency. 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 PicoBlaze processors. Going a step further, Bergmann et al. [7] wrap arbitrary hardware circuits in software wrappers, so called *ghost processes*, which are similar to ReconOS' *delegate threads* (see Section III-A) in that they provide a transparent interface for interactions from the kernel and other threads. This approach also uses FIFOs for communications, which are mapped into the Linux file system [8]. So et al. [9][10] 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 singular existing operating system objects to ease communication. That most approaches choose a FIFO as the preferred means of communication hints at its merits for thread-to-thread communication. However, not all OS interactions of threads can be efficiently served with unidirectional communication channels. 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 hybrid reconfigurable hardware/software systems.

The hthreads project [11] follows a very similar approach to ReconOS. Hardware threads are able to access various OS functions through a dedicated hardware thread interface. hthreads is based on the POSIX pthreads programming model for both hard- and software threads and implements the OS components managing synchronization and task scheduling as hardware IP cores, sacrificing the flexibility of a software OS kernel for exceptionally low response time and jitter [12]. The existing implementation, however, while employing a sophisticated inter-thread memory model [13], is limited in its data transfer throughput due to the employed bus topology.

III. System Architecture

This section outlines the ReconOS system architecture, including the software architecture, the hardware platform, and hardware threads. An example with two software and two hardware threads is shown in Figure 1. A more detailed overview over the ReconOS system can be found in [1].

A. Software Architecture

ReconOS uses the eCos [14] kernel for most of its software implemented operating system services, such as the scheduler or the synchronization mechanisms. Software threads are created and executed as regular eCos threads on the system CPU. They access the individual OS services through standard C system calls. eCos provides both a native API as well as compatibility APIs, most importantly POSIX.

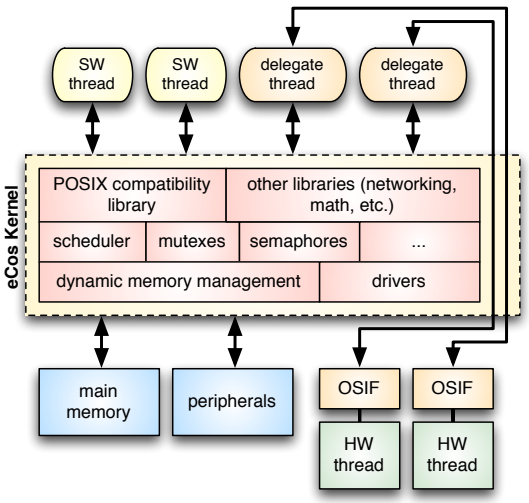

Fig. 1. ReconOS system architecture

In ReconOS, a dedicated software thread, the *delegate*, is associated with every hardware thread. The delegate listens for incoming OS requests from its hardware thread and executes the system calls on its behalf. All OS interactions for hardware threads (with a few exceptions) are managed by delegates, which guarantees totally transparent inter-thread communication. No thread needs to know whether its communication partners are executed in hardware or software. While this approach provides great flexibility and extensibility, it comes with an overhead due to interrupt processing and context switching.

B. Hardware Platform

Designed for modern platform FPGAs such as the Xilinx Virtex-II Pro/Virtex-4 families, the ReconOS hardware architecture provides the low-level communication channels which connect the software part of an application, running on the system's CPU, with the hardware threads, located in the FPGA's configurable fabric, and other peripherals. Using standard design tools together with ReconOS-specific automation tools, the ReconOS hardware architecture builds on the CoreConnect bus topology to link the system components together. Two bus systems are employed: The Processor Local Bus (PLB) is used for most data communications in the system and connects the CPU, main memory, system peripherals, and all hardware threads. To prevent prolonged memory accesses from interfering with latency-sensitive OS calls, the latter are routed through a separate low-overhead bus, the Device Control Register (DCR) bus. This mechanism provides bounded access times from the CPU to the hardware threads regardless of memory bus load. At the same time it offers the flexibility to remove the resource-intensive PLB bus interface from hardware threads that do not need to access the PLB while retaining the capability to call all supported OS functions.

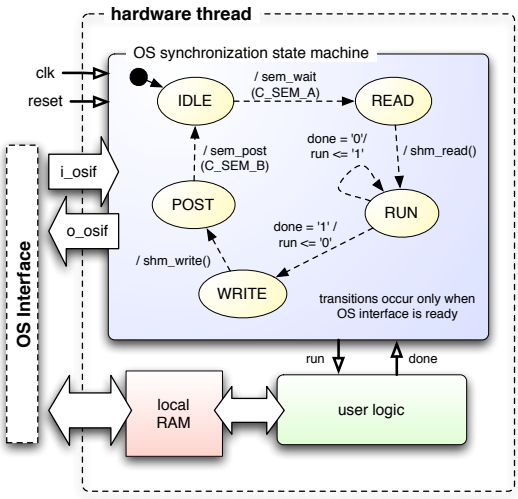

Fig. 2. Hardware thread structure

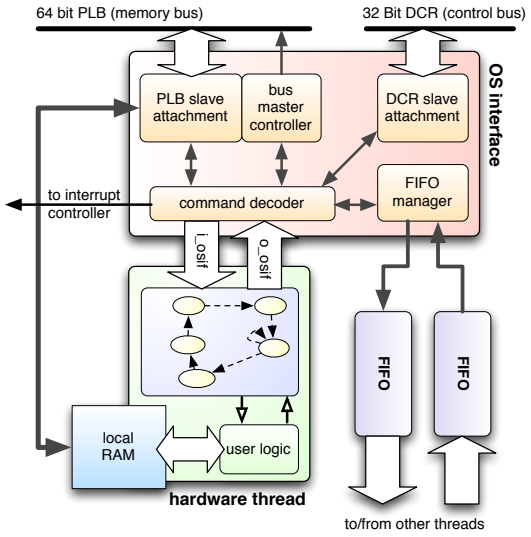

Fig. 3. OSIF overview and interfaces

C. Hardware Threads

A typical ReconOS hardware thread is implemented in VHDL, which may be either hand-written or generated from a higher level language or abstraction. Figure 2 shows the general structure of a hardware thread, comprising the user logic, the OS synchronization state machine, and an optional local memory.

To enable access to OS services, ReconOS provides hardware threads with a strictly defined "hardware API", which closely resembles the software API provided by eCos. All calls to this hardware API are encapsulated in the OS synchronization state machine that communicates with and is partly controlled by the OS interface (OSIF). Specifically, the OSIF can prevent the OS synchronization state machine from transitioning, thus effectively providing the semantics of blocking function calls to hardware. Nonetheless, hardware threads are not limited to the sequential execution dictated by the centralized OS management. Usually, the OS synchronization state machine controls the user logic, a set of processes executing in parallel performing the thread's actual processing tasks. Thus, a designer can not only use hardware threads to exploit coarse-grained thread-level parallelism by having several hardware and possibly a software thread executing truly in parallel, but also take advantage of more fine-grained parallelism by placing multiple processes in the user logic.

Hardware threads have also access to a local memory. This memory is primarily used for buffering main memory accesses and increasing throughput by using burst transactions. Optionally, the hardware thread may use the memory as a local data storage.

IV. Hardware/Software 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 operating system interface (OSIF). An overview of the OSIF's structure and its interfaces to the hardware thread, the system buses and the FIFO cores is given in Figure 3. The OSIF is built from several modules that together perform the following tasks:

A. Thread Supervision and Control

ReconOS provides hardware threads with a hardware API that comes in the form of a function library which specifies VHDL functions and procedures like `semaphore_post()` or `thread_exit()`. 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 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 I.

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. 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

TABLE I
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 multi-cycle command
valid		indicates success of call
busy		system buses are busy
blocking		set while executing blocking OS calls

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 `semaphore_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 IV-B).

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 (`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. `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 simply 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 multi-cycle 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 multi-cycle command induces additional state which 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 to the VHDL procedure, which uses it to perform the appropriate function for this step.

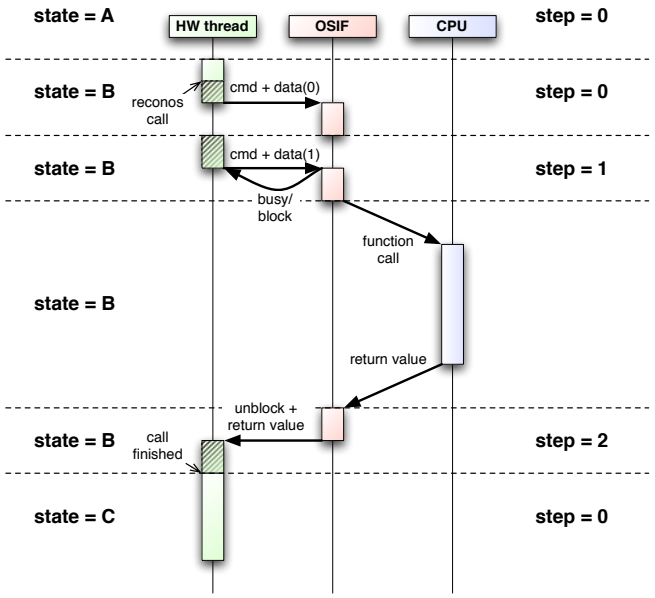

Fig. 4. Multi-cycle-command example

An example of this mechanism is depicted in Figure 4. 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.

B. 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, 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, it should only be used for relatively infrequent synchronization calls, since there is considerable overhead involved. 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. However, it must be noted that the parallel user processes inside the thread may continue their execution.

C. 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 OS interface, this is realized in two variants, which provide the basis for any efficient, high-bandwidth thread-to-thread communication:

1) 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 3), 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 burst RAM. Then, the thread's OS synchronization state machine calls a `write_burst()` procedure. This prompts the bus master controller to initiate a PLB bus transfer from the local burst RAM (which is mapped into the system's 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 burst RAM.

When transferring data between hardware and software threads through shared memory, it is up to the designer to take care of cache coherency. ReconOS provides cache control functions to help handle these issues.

2) 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 suffers considerably when several threads (and other peripherals or the CPU) are contending for bus access.

To allow 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 3), 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 suspend the hardware thread's OS synchronization state machine, thus providing blocking `get()/put()` operations on FIFOs.

V. Thread Synchronization and Communication

An application is divided into several threads, which can potentially execute in parallel. These thread objects form the basic structure of the multi-threaded programming model. Communication and synchronization require a thread to interact with operating system primitives by calling appropriate OS functions. In the embedded software world most real-time operating systems, among them eCos and variants of Linux, support a very similar set of operating system objects for the synchronization and communication of independently executing components of an application. It is one of the major goals of the ReconOS project to extend this established programming model to the hardware domain and thus form a common abstraction layer for all processes executing in a hybrid hardware/software system.

In a single processor system, communication and synchronization services rely on the well-known hardware/software interface provided by instruction-set programmable processors. In contrast, establishing such services in a reconfigurable computer featuring true coarse-grained parallelism through multithreading poses a much greater challenge. Depending on the actual hardware/software mapping, different implementations of OS services might be favored. While ReconOS consequently provides the same semantics for OS services, the designer has to be aware of the fact that the performance will vary with the chosen hardware/software mapping. Overall, we have implemented the following OS services for synchronization and communication:

A. Semaphores and Mutexes

If two or more threads share or access the same resource, it is necessary to synchronize their execution. Usually, this is done by semaphore or mutex (mutual exclusion) primitives. ReconOS provides both primitives as software implementations, which can be accessed by both software and hardware threads. As such synchronizations usually happen quite infrequently, the overhead involved with calling OS kernel functions from hardware threads is considered acceptable.

B. Shared Memory

Both software and hardware threads have direct access to the system's main memory: the CPU can use its PLB interface to initiate memory transfers, while the OSIF provides hardware threads with single-word or burst access to the PLB and, thus, to main memory. After exchanging access information, such as a pointer to a defined memory area, threads can use main memory to transfer data quite efficiently. In contrast to the synchronization primitives outlined above, memory access from hardware threads is a bus master operation and does

not interrupt the CPU. To avoid data corruption through non-atomic operations, however, access to shared memory regions must be protected by mutexes or semaphores. Again, this kind of synchronization occurs infrequently enough so that the processing overhead remains tolerable.

C. Message Queues

For stream-processing applications which often line up several threads in a processing chain, it is desirable to be able to efficiently pass data from one thread to the next. To that end, the ReconOS programming model provides an OS mailbox primitive which can be used to send data to a logical buffer, the mailbox, from where it can be read by another thread. The ReconOS mailbox functions as a message queue with FIFO semantics, the corresponding calls for writing and reading may also block if the buffer is full or empty, respectively. Thus, the mailbox primitive provides communication and synchronization services at the same time.

Normally, the memory block that implements a mailbox object is located in main memory. This mapping can be used perfectly to establish mailboxes between any software and hardware threads in ReconOS. Driven by the great importance of efficient data exchange for streaming applications, we have further implemented direct FIFO objects between hardware threads (see Section IV-C2). Consider the situation where a hardware thread A generates data and forwards it to hardware thread B using a mailbox. In this case, the use of a hardware FIFO to transfer data neither requires to interrupt the CPU nor does it increase the load on the memory bus.

We have extended the ReconOS OS interface to transparently support such direct thread-to-thread FIFO communications with the existing programming model. When using the OS mailbox primitives, the individual hardware threads do not need to know whether the mailbox is implemented in hardware or software. To achieve this, every OSIF keeps track of which mailboxes are mapped to the connected FIFOs. If a thread invokes an OS call accessing such a mailbox, the OSIF directly routes that access to the FIFOs via its FIFO manager, which handles the handshaking and also blocks the reading or writing thread, if necessary. If the accessed mailbox is not mapped to a local FIFO, the call is forwarded as usual to the CPU.

VI. Experimental Results

To show the feasibility and quantify the efficiency of the proposed synchronization and communication mechanisms, we have conducted measurements on a prototype implementation. Figure 5 displays the system architecture of the prototype which has been implemented on a Xilinx XC2VP30 FPGA, with the PPC CPU running at 300 MHz and the bus, hardware threads, and FIFOs running at 100 MHz. A single OS interface requires 1147 slices. An estimated 65 % of these are taken up by the PLB bus interface alone; the largest parts of the remaining logic are used for the command decoder (21 %), DCR bus interface (6 %) and FIFO manager (3 %). Two sets of measurements are described - one reporting the execution times of operating system calls involving synchronization

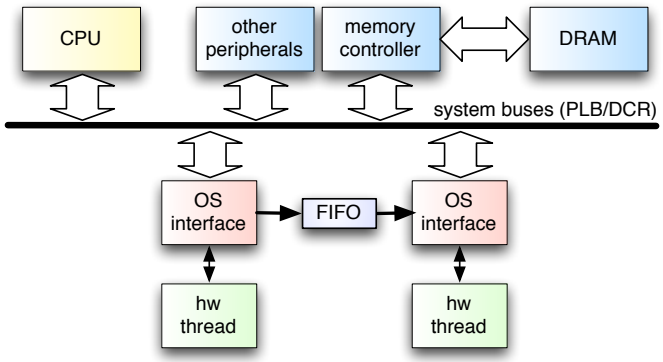

Fig. 5. Hardware architecture with thread-to-thread FIFOs

TABLE II
Measurements (synchronization primitives)

	with data cache		without data cache	
Configuration	semaphore	mutex	semaphore	mutex
SW → SW	3.39	4.53	29.05	43.99
SW → HW	4.71	6.29	30.49	47.43
HW → SW	10.74	14.49	82.90	100.23
HW → HW	11.90	14.60	83.13	101.49

All values are given in μs

primitives serviced in software, the other determining the attainable communication time and bandwidth of the hardware-implemented communication primitives, using both shared memory and mailbox operations.

A. Synchronization Primitives

In the first experiment, synchronization operations are performed between two threads, using either semaphores or mutexes, with one thread waiting for a synchronization primitive and the other releasing (in case of a mutex) or posting (in case of a semaphore) it. For both tests, all four configurations of possible execution contexts (hardware/software) have been measured. The results can be seen in Table II.

Since all synchronization operations in ReconOS use eCos kernel functions implemented in software, the pure software-based tests show the lowest synchronization latencies, with operations initiated from hardware threads showing around three times higher execution times, due to the necessary interrupt processing. However, since designers are likely to use hardware threads for data-dominated processing tasks, leaving control-intensive operations to software threads, the higher latencies have little impact on overall application execution times.

B. Communication Primitives

In this experiment, two threads perform a sequence of 8 kByte 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 software mailboxes.

During this experiment, the following periods of time are measured: the times for reading and writing the data from and

TABLE III
MEASUREMENTS (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 (mbox read)	61.42	127.20	61.42	127.20
HW→HW (mbox write)	61.45	127.14	61.45	127.14
SW→HW (mbox read)	58500	0.13	374000	0.02
HW→SW (mbox write)	58510	0.13	374000	0.02

All operations were run for 8 kBytes of data

to main memory, and the times for writing and reading the data to and from the mailboxes. Since software threads do not possess local memory apart from the processor's registers, the memory read/write tests for software threads has been combined into a single *memcopy* test. The results are shown in Table III.

While the hardware FIFOs only achieve 66% to 74% of the 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 concurrently. Considering this, an 8 kBytes data transfer via hardware FIFOs is about 40% faster than a transfer of the same size via shared memory. Also, the transfer via mailboxes is implicitly synchronized, while two threads exchanging data via shared memory need explicit synchronization, e.g., via mutexes or semaphores.

The above 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. Further, hardware FIFOs fully maintain transparency and flexibility using the ReconOS programming model abstractions. It should however be noted that for mailbox-based data transfers across the hardware/software boundary we currently use regular eCos software mailboxes. For this purpose shared memory should be preferred as it still yields several orders of magnitude better performance. Section VII proposes a modification to the hardware FIFOs that will alleviate this problem.

VII. CONCLUSION AND FUTURE WORK

In this paper we have shown efficient mechanisms for communication and synchronization in multithreaded reconfigurable systems on two levels. First, we have detailed the hardware/software interface allowing hardware circuits access the same OS services as software threads, as implemented in the ReconOS system. Second, we have shown how our programming model allows multiple threads to synchronize and communicate transparently on a high abstraction level, regardless of their respective execution contexts (hardware/software). We have implemented a fully functional prototype and demonstrated the feasibility of our interfaces and programming model with experimental measurements.

Ongoing work focuses on alleviating or removing some of the limitations that still exist in the current implementation of the hardware communication primitives. Extending the existing hardware FIFO IP cores with a bus interface will allow hardware threads without direct connection to the hardware FIFO to communicate across the bus without having to resort to expensive software OS calls. Further, software threads can talk directly to the FIFOs, providing a fast mechanism for hardware/software communication.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.

[2] G. J. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," in *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL)*. London, UK: Springer-Verlag, 1996, pp. 327–336.

[3] K. Danne, R. Muehlenbernd, and M. Platzner, "Executing Hardware Tasks on Dynamically Reconfigurable Devices under Real-time Conditions," in *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 541–546.

[4] H. Hinkelmann, A. Gunberg, P. Zipf, L. S. Indrusiak, and M. Glesner, "Multitasking Support for Dynamically Reconfigurable Systems," in *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL)*, August 2006, pp. 219–224.

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

[6] K. 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Baijot, S. Warn, and D. Andrews, "Supporting High Level Language Semantics within Hardware Resident Threads," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 1. IEEE, August 2007, pp. 98–103.

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