Chapter "ReconOS" in Book "FPGAs for Software Programmers"

Andreas Agne, Marco Platzner, Christian Plessl,
Markus Happe, and Enno Lübbers

1 Introduction

A growing number of applications originating in and integrating diverse fields like pattern recognition, machine learning, large-scale data analytics, or communication systems features a diverse mix of control-intensive and highly parallelizable regular data-centric operations. This heterogeneity of the composition of application components increasingly prohibits the exclusive use of a single acceleration technology. Thus, in the case of fine-grained reconfigurable computing, applications are rarely mapped exclusively to an FPGA accelerator. Instead, individual application parts amenable to parallel execution, customization, and deep pipelining are often implemented as custom hardware to improve performance or energy-efficiency. Other parts, especially code that is highly sequential or difficult to implement as custom hardware, are executed in software mapped to a CPU. This decomposition of applications into separate, communicating parts that require synchronization among them is also widely used in pure software systems for achieving a separation of concerns and concurrent or asynchronous processing. In software systems the operating system standardizes these communication and synchronization mechanisms and provides abstractions for encapsulating the unit of execution (processes, threads), communication, and synchronization.

Reconfigurable computing systems still lack an established operating system foundation that covers both software and hardware parts. Instead, communication and synchronization are usually handled in a highly system and application-specific way, which tends to be error prone, limit the productivity of the designer, and prevent portability of applications between different reconfigurable computing systems.

The ReconOS operating system, programming model and system architecture offers unified operating system services for functions execut-

Published in: *FPGAs for Software Programmers*
 DOI: 10.1007/978-3-319-26408-0_13

ing in software and hardware and a standardized interface for integrating custom hardware accelerators. ReconOS leverages the well-established multi-threading programming model and extends a host operating system with support for hardware threads. These extensions allow the hardware threads to interact with software threads using the same, standardized operating system mechanisms, for example, semaphores, mutexes, condition variables, and message queues. From the perspective of an application it is thus completely transparent whether a thread is executing in software or hardware. The availability of an operating system layer providing symmetry between software and hardware threads provides the following benefits for reconfigurable computing systems:

- The application development process can be structured in a step-by-step fashion with an all-in-software implementation as a starting point. Performance-critical application parts can then be turned into hardware threads one-by-one to explore the hardware/software design space successively.
- The portability of applications between different reconfigurable computing systems is improved by using defined operating system interfaces for communication and synchronization instead of low-level platform-specific interfaces.
- The unified appearance of hard and software threads from the application's perspective allows for moving functions between software and hardware during runtime, which supports the design of adaptive computing systems that exploit partial reconfiguration.

The evolution of operating systems for reconfigurable computing and how ReconOS relates to this heritage is discussed in Section 9.

2 Programming Model

The key idea of ReconOS is extending the multi-threading programming model across the hardware/software interface. In multi-threaded programming, applications are composed of objects such as threads, message queues, and semaphores, each of which has a strictly defined interface and purpose. The application's functionality is partitioned into *threads*, which in our case can be either blocks of sequential software or parallel hardware modules. Threads communicate and synchronize using one or more of the remaining objects of the programming model: for example, they can pass data using *message queues* or *mailboxes*, explicitly coordinate execution through *barriers* or *semaphores*, or implicitly synchronize access to shared

resources by locking and unlocking mutually exclusive locks (*mutexes*). These objects and their interactions are widely used in well-established APIs for programming multi-threaded software applications. One of the major advantages developers can draw from the ReconOS approach is that these abstractions can not only be used for software threads but also for optimized hardware implementations of data-parallel functions—the hardware threads—without sacrificing the expressiveness and portability of the application description.

Consider the example software thread sketched in Listing 1. The thread receives packets streaming in via ingress mailbox `mbox_in`, processes them in a user-defined way, sends the processed packets to egress mailbox `mbox_out`, and updates a packet counter stored in a shared variable protected by lock `count_mutex`. Using standard APIs for message passing and synchronization, the software thread accesses operating system services in an expressive, straightforward, and portable way. As an additional benefit, such a thread description manages to clearly separate thread-specific processing from operating system calls.

Figure 1 shows a ReconOS hardware implementation of the same thread, partitioned into similar thread-specific logic and operating system interactions. While the thread-specific *user logic* contains the hardware thread's data path and is only limited by available FPGA resources, the operating system interactions of a hardware thread are captured by the *OS synchronization finite state machine* (OSFSM). Together with the *OS interface* (OSIF), this state machine enables seamless operating system calls from within hardware modules. The developer specifies the OSFSM using a standard VHDL state machine description, as shown in Listing 2. For accessing operating system functions in this state machine ReconOS provides a VHDL library that wraps all operating system calls with VHDL procedures. The transitions of the OSFSM are guarded by an OS-controlled signal `done` (line 47), so that blocking operating system calls—such as `mutex_lock()`—can temporarily inhibit the execution of a hardware thread.

Consequently, the OSFSM in VHDL closely mimics the sequence of operating system calls within the equivalent software thread: it reads a packet from a mailbox, passes it to a separate module to be processed, writes the processed packet back to another mailbox, and increments a thread-safe counter. The description of the actual user logic, however, may well differ from the software realization, as this is the area where the fine-grained parallel execution of an FPGA-optimized implementation can realize its strengths—unhindered by the necessarily sequential execution of operating system calls.

Listing 1: Example of a stream processing software thread using OS services.

```
1  extern mutex_t *count_mutex;
   // mutex protecting packet counter
2  extern mqd_t mbox_in,
   // ingress packets
3                mbox_out;
   // egress packets
4
5  void *thread_a_entry( void *count_ptr ) {
6    data_t buf;
   // buffer for packet processing
7
8    while ( true ) {
9      buf = mbox_get( mbox_in );
   // receive new packet
10     process( buf );
   // process packet
11     mbox_put( mbox_out, buf );
   // send processed packet
12     mutex_lock( count_mutex );
   // acquire lock
13     ( (count_t) *count_ptr )++;
   // update counter
14     mutex_unlock( count_mutex );
   // release lock
15   }
16 }
```

3 ReconOS Architecture

The ReconOS run-time system architecture provides the structural foundation to support the multi-threading programming model and its execution on CPU/FPGA platforms. Figure 2 shows a conceptual view of a typical system that is decomposed into application software, OS kernel and hardware architecture. The application's software threads are usually executed on the main CPU alongside the host OS kernel that encapsulates APIs, libraries, and all programming model objects as well as lower level functions such as memory management and device drivers. The ReconOS run-time environment consists of hardware components that pro-

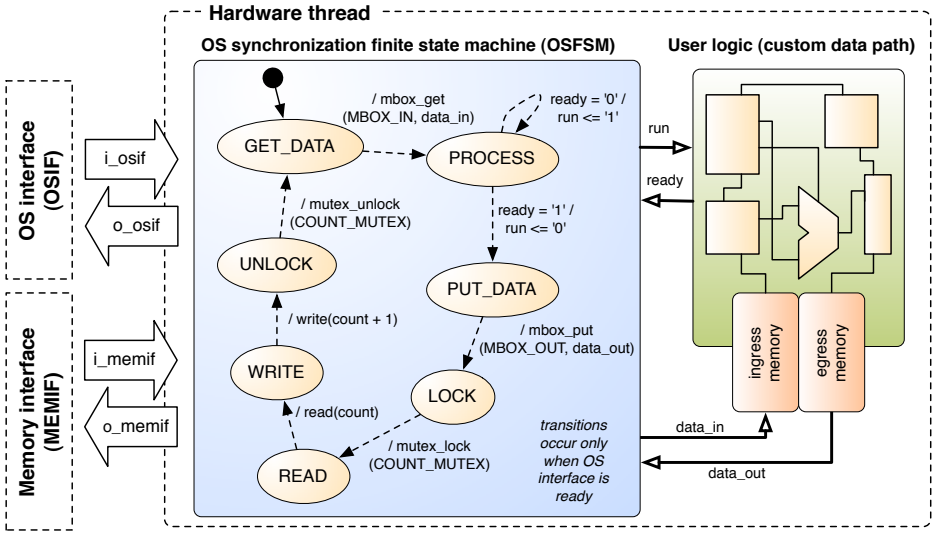

Figure 1: A ReconOS hardware thread comprises the OS synchronization finite state machine and the user logic implementing the data path.

vide interfaces, communication channels, and other functionality such as memory access and address translation to the hardware threads. Additionally, the runtime system comprises software components in the form of libraries and kernel modules that offer an interface to the hardware, the operating system, and the application's software threads.

A key component for multi-threading across the hardware/software boundary is the *delegate thread*, which is a light-weight software thread that interfaces between the hardware thread and the operating system. When a hardware thread needs to execute an operating system function, it relays this request through the operating system interface (OSIF) to the delegate thread using platform-specific (but application-independent) communication interfaces. The delegate thread then executes the desired operating system functions on behalf of its associated hardware thread. Hence, from the OS kernel's point of view, only software threads exist and interact, while the hardware threads are completely hidden behind their respective delegate threads. From the application programmer's point of view however, the delegate threads are hidden by the ReconOS runtime environment and only the application's hardware and software threads exist. This delegate mechanism together with the unified thread interfaces gives ReconOS exceptional transparency regarding the execution mode of a thread, i.e., whether it runs in software or hardware. While the delegate mechanism causes a certain overhead for executing OS calls, the result-

Listing 2: OS synchronization FSM for a stream processing hardware thread.

```
1  OSFSM: process(clk, reset)
2    variable ack: boolean;
3  begin
4
5    if reset = '1' then
6      state <= GET_DATA;
7      run <= '0';
8      osif_reset (o_osif , i_osif);
9      memif_reset(o_memif, i_memif);
10   elsif rising_edge(clk) then
11
12   case state is
13
14     when GET_DATA =>
15       mbox_get(o_osif,i_osif,MB_IN,data_in,done);
16       next_state <= COMPUTE;
17
18     when COMPUTE =>
19       run <= '1';
20       if ready = '1' then
21         run <= '0';
22         next_state <= PUT_DATA;
23       end if;
24
25     when PUT_DATA =>
26       mbox_put(o_osif,i_osif,MB_OUT,data_out,done);
27       next_state <= LOCK;
28
29     when LOCK =>
30       mutex_lock(o_osif,i_osif,CNT_MUTEX,done);
31       next_state <= READ;
32
33     when READ =>
34       read(o_memif,i_memif,addr,count,done);
35       next_state <= WRITE
36
37     when WRITE =>
38       write(o_memif,i_memif,addr,count + 1,done);
39       next_state <= UNLOCK;
40
41     when UNLOCK =>
42       mutex_unlock(o_osif,i_osif,CNT_MUTEX,done);
43       next_state <= GET_DATA;
44
45   end case;
46
47   if done then state <= next_state; end if;
48
49   end if;
50 end process;
```

ing simplicity of switching thread implementations between software and hardware greatly facilitates system generation and design space explo-

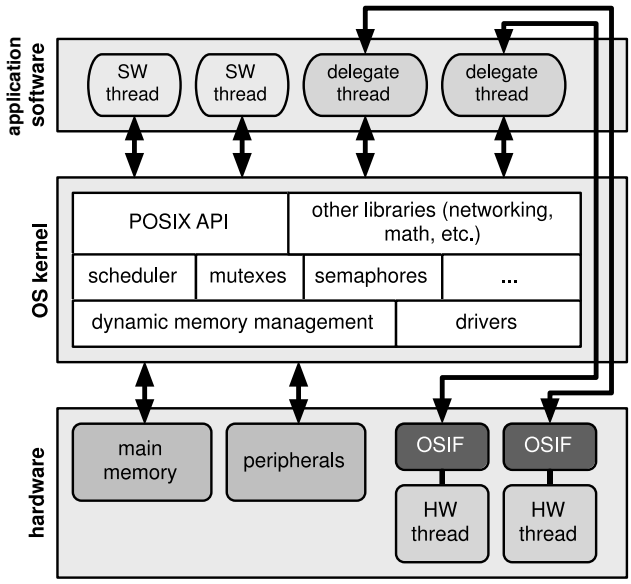

Figure 2: Conceptual overview of the ReconOS system architecture. Software threads interact directly with the OS kernel, while hardware threads connect through an OSIF and delegate threads.

ration.

The ReconOS concept is rather general and has been ported to several FPGA families, main CPU architectures, and host operating systems (see Section 10). For the remainder of this article we describe the implementation of ReconOS v3, which is the most recent version of ReconOS targeting Xilinx Virtex-6 FPGAs and utilizing a MicroBlaze/Linux environment.

To assist developers with creating the OSFSM for a hardware thread, ReconOS provides a library that wraps convenient VHDL procedures around the operating system call signaling, e.g., `mutex_lock()` as used in Figure 2. Technically, the VHDL procedures implement further state machines that are nested within the OSFSM and access the two FIFOs `i_osif` and `o_osif` to connect to the OSIF. Figure 3 outlines the relationship between the OSFSM, the nested state machine implementing the `mutex_lock` procedure and the two FIFOs. Synchronization between the nested state machines and the OSFSM is controlled via the handshaking signal `done`. Towards the delegate thread, we use a communication protocol that encodes an OS request as a sequence of words comprising a function identifier and a call-specific number of parameters. The encoded request is written to the outgoing FIFO `o_osif`. For a hardware thread, a function call is completed when an acknowledgement has been sent by the delegate thread and, op-

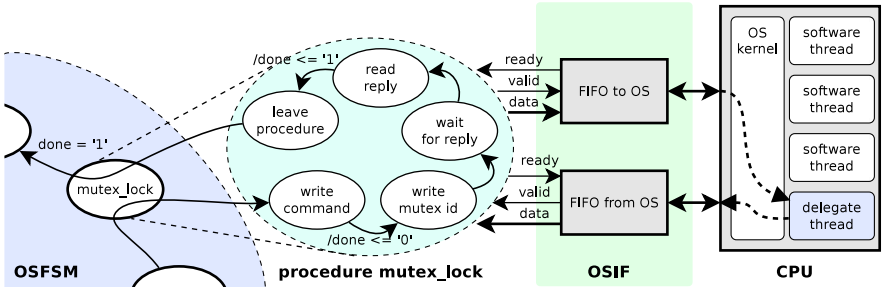

Figure 3: A finite state machine nested within the OS synchronization finite state machine handles the communication between the hardware thread and the OS (via OSIF and delegate thread). The OSIF contains two FIFOs that connect the hardware thread with the CPU. The operating system relays the hardware thread's requests to the respective delegate thread where the request is carried out.

tionally, a return value has been read from the incoming FIFO `i_osif`.

Hardware threads reside in *reconfigurable slots*, which are predefined areas of reconfigurable logic equipped with the necessary communication interfaces. Figure 4 shows an instance of a ReconOS hardware architecture with a CPU, two reconfigurable slots, the memory subsystem and various peripherals. Besides communicating with the OS kernel on the host CPU, hardware threads residing in reconfigurable slots can also access the system memory. To that end, a hardware thread uses its *memory interface* (MEMIF) shown in Figure 1 to connect to the ReconOS memory subsystem. The memory subsystem arbitrates and aligns the hardware threads' memory requests and can handle single word as well as burst accesses. To support Linux with virtual addressing as host operating system, ReconOS implements a full-featured memory management unit (MMU), including a translation lookaside buffer, that can autonomously translate addresses using the Linux kernel's page tables [1]. Hardware threads use FIFOs to communicate with the memory subsystem; one outgoing and one incoming FIFO per hardware thread. Requests for memory transactions are encoded and written to the outgoing FIFO followed by data in the case of a write request. In the case of a read request, data become available on the incoming FIFO upon completion of the memory transfer. Similar to the communication with the OS, we provide a library of VHDL procedures to conveniently handle memory operations. These procedures encode the requests, synchronize with the memory FIFOs, and automatically transfer data from/to local memory elements within the hardware thread.

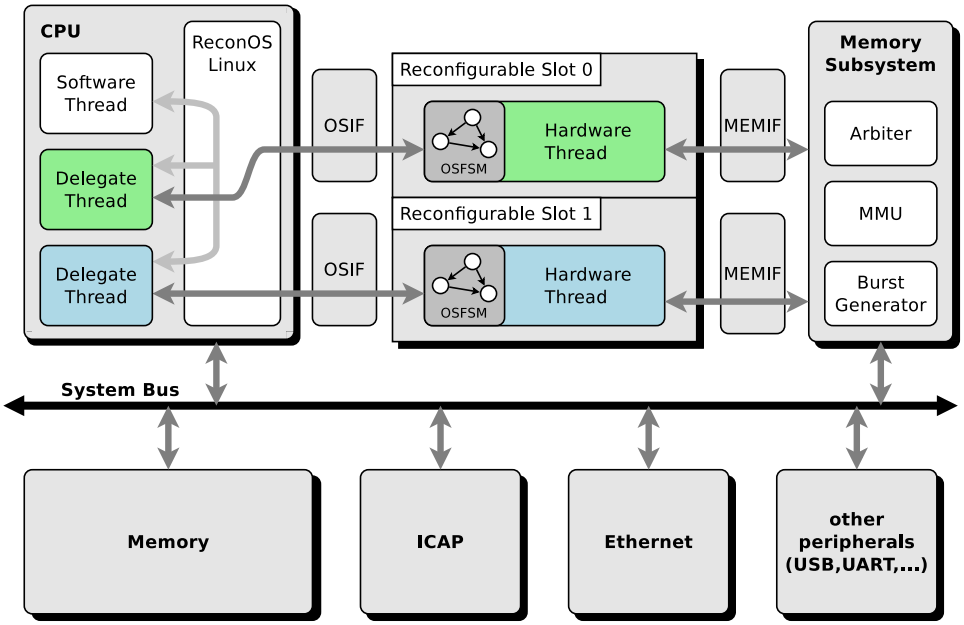

Figure 4: Example instance of an ReconOS hardware architecture with a CPU and two reconfigurable hardware slots

4 Application Development with ReconOS

Over the years, ReconOS has been used to implement several applications on hybrid CPU/FPGA systems. These experiences have confirmed that the hybrid multi-threading approach offered by ReconOS simplifies the development process, which is typically structured in three steps: First, the developer prototypes the application's functionality in multi-threaded software using, for example, the Pthreads library on Linux. This first software-based implementation allows for functional testing. Second, the multi-threaded software is ported to the embedded CPU on the targeted platform FPGA, e.g., a MicroBlaze running Linux. The developer can now use profiling to identify the application's potential for parallel execution, i.e., those threads that could benefit from the fine-grained parallelism of a hardware realization, and those code segments that are amenable to a coarser-grained parallel implementation with multiple threads. The third step includes creating the hardware threads and the ReconOS system architecture. At this point, ReconOS easily allows the developer to evaluate different mappings of threads to hardware and software and to quickly assess the overall performance on the target system.

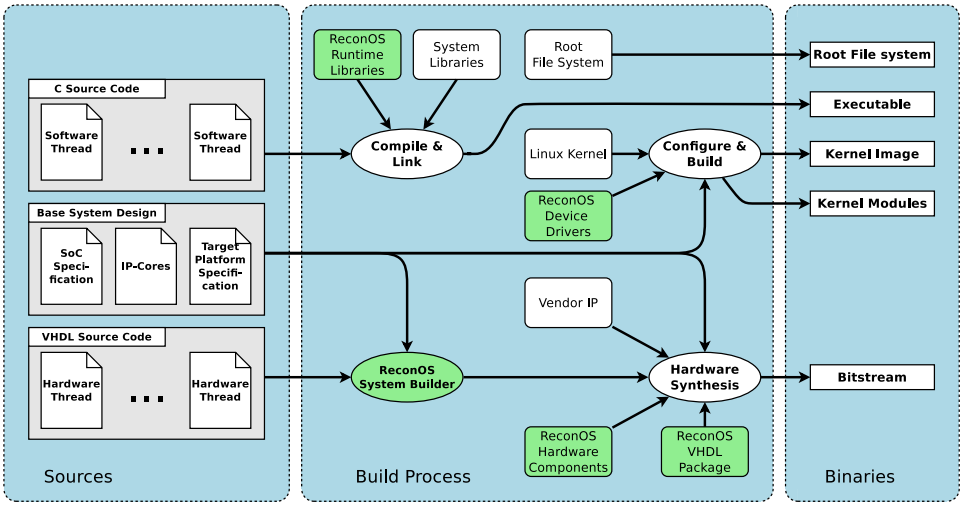

Figure 5: Tool flow for assembling a ReconOS system on a Linux target. ReconOS-specific steps are colored green.

5 ReconOS Tool Flow

Figure 5 captures the ReconOS v3 tool flow. The required sources comprise the software threads, the hardware threads and the specification of the ReconOS hardware architecture. We code software threads in C and hardware threads in VHDL, using the ReconOS-provided VHDL libraries for OS communication and memory access. An automatic synthesis of hardware threads is not part of the ReconOS project; developers are, however, free to use any hardware description language or high-level synthesis tool to create hardware threads. ReconOS extends the process for building a reconfigurable system-on-chip using standard vendor tools. On the software side, the delegate threads and device drivers for transparent communication with hardware threads are linked into the application executable and kernel image, respectively. On the hardware side, components such as the OS and memory interfaces as well as support logic for hardware threads are integrated into the tool flow. The ReconOS System Builder assembles the base system design and the hardware threads into a reference design and automatically connects bus interfaces, interrupts, and I/O. The build process then creates an FPGA configuration bitstream for the reference design using conventional synthesis and implementation tools.

During design space exploration, the developer will create both hardware and software implementations for some of the threads. Switching between these implementations is a matter of replacing a single thread instantiation statement, e.g., using `rthread_create()` instead of `pthread_create()`.

Such a decision for software or hardware can even be taken during runtime, see Section 8.

6 Case Study: Video Object Tracker

To illustrate the benefits of the ReconOS approach, we present a particle-filter based video object tracker [9] for continuos estimation of an object's position and size in a video sequence. A particle filter is a robust technique for video object tracking because it maintains several estimates (particles) for the position and size of the tracked object. The filter iterates over video frames and processes the particles in three consecutive stages: 1) sampling estimates where the object might have been moved; 2) importance weights all estimated particles by comparison with the observed next video frame; 3) resampling eliminates low-weighted particles and duplicates high-weighted ones to create the particle set for the next filter iteration.

For our implementation we start with an existing video object tracker [10] implemented in C. First, we transform the monolithic code into a multi-threaded implementation on a desktop using POSIX Pthreads under Linux. Each filter can be naturally turned into a software thread and the particles, grouped into chunks, are forwarded between the filter stages via message boxes. Since the particles are independent and thus can be processed in parallel, each of the stages is represented by multiple thread instances exploiting data parallelism. Second, we port our multi-threaded software implementation from the desktop to the CPU embedded in a Xilinx FPGA. Video data is streamed from the desktop to the FPGA via Ethernet. Overall, this step requires very little effort because both platforms offer the same OS and APIs. Third, we profile the execution times of all filter stages and confirm that the execution times strongly depend on the input data because the filter computes color histograms in variable-sized regions of interest, in which the tracked object is searched. We identify two functions that are typically performance-critical, color histogram computation (observation, o) and color histogram comparison (importance, i) and implement hardware thread versions for both functions.

Using the hardware threads for observation and importance as well as the multi-threaded software implementation, we perform a swift design space exploration measuring the required computational effort for a given video sequence using hardware/software mappings with different resource requirements. Figure 6 shows the required computational effort in execution time per frame of various mappings for tracking a soccer player. The tracker employing four hardware threads, two for observa-

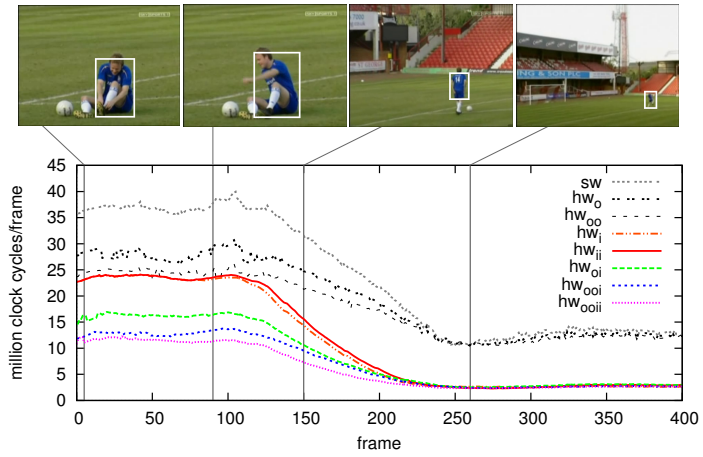

Figure 6: Design space exploration for a video object tracker: The graph shows the computational effort for tracking vs. time in video frames for a specific video (taken from [10]). The individual curves represent ReconOS implementations with different hardware/software mappings, where *sw* denotes an all-in-software system, and curves labelled with *hw* denote systems with one to four threads of type observation (o) and importance (i) running in reconfigurable hardware.

tion and two for importance (mapping hw_{ooii}), achieves the highest performance. Clearly the required effort decreases when the object moves into the background. There, mapping hw_i with a single hardware thread for importance achieves comparable performance results.

7 Conclusion

Among the existing operating system approaches for reconfigurable computers, ReconOS stands out by providing a deep semantic integration of hardware accelerators into an operating system environment while leveraging standard operating system kernels. Hardware threads can access a rich set of operating system functions, making them essentially identical to software threads with respect to operating system interaction. Consequently, hardware threads can easily be exchanged for software threads and vice versa, which allow for rapid design space exploration at design time and even migration of function across the hardware/software border at run-time. The use of standard operating system kernels in ReconOS leads on to a structured design process starting with a, possibly monolithic, software implementation and to improved portability. Our experience shows that these features can significantly lower the entry barrier for

reconfigurable computing technology.

8 Sidebar: Applications of ReconOS

ReconOS defines a standardized interface for hardware threads, which simplifies exchanging them, not only at design time but also during runtime using *dynamic partial reconfiguration* (DPR). DPR allows for exploiting FPGA resources in unconventional ways, for example, by loading hardware threads on demand, moving functionality between software and hardware, or even multi-tasking hardware slots by time-multiplexing. ReconOS supports DPR by dividing the architecture in a static and a dynamic part. The static part contains the processor, the memory subsystem, OSIFs, MEMIFs, and peripherals. The dynamic part is reserved for hardware threads, which can be reconfigured into the hardware slots. Our DPR tool flow builds on Xilinx PlanAhead and creates the static subsystem and the partial bitstreams for each desired hardware thread/slot combination. Time-multiplexing of hardware slots is supported through cooperative multi-tasking [13].

Researchers at ETH use ReconOS to implement *adaptive network architectures* that continuously optimize the network protocol stack on a per-application basis to cope with varying transmission characteristics, security requirements, and compute resources availability. The developed architecture [12] autonomously adapts itself by offloading performance-critical, network processing tasks to hardware threads, which are loaded at runtime using dynamic partial reconfiguration.

Another line of research also leverages the unified software/hardware interface and partial reconfiguration to create *self-adaptive and self-aware* computing systems that autonomously optimize performance goals under varying workloads. For example, we have created self-adaptive implementations of the particle filter presented in Section 4 that start and stop additional threads on worker CPUs and in reconfigurable hardware slots to keep the resulting frame rate for the video object tracker within a pre-defined band. In the EPiCS project [1] funded by the European Commission, we even advance the autonomy of computing systems and enable them to optimize for diverse goals such as performance, energy consumption and chip temperature based on the current quality-of-service requirements, workload characteristics and system state.

So far ReconOS has been used in embedded systems where the CPU and the hardware cores are implemented in Xilinx platform FPGAs. The general approach of ReconOS is equally attractive in a *high-performance*

[1] `http://www.epics-project.eu`

computing context. For example, ReconOS is currently being evaluated for use in high-speed data acquisition and particle physics applications [2]. In current work [3] we also are studying how ReconOS can be ported to x86-based server systems that attach FPGA accelerator cards via PCIe.

9 Sidebar: Operating Systems for Reconfigurable Computing

The introduction of the partially reconfigurable Xilinx XC6200 FPGA series in the mid 1990's and, later on, the JBits software library for bitstream manipulation inspired researchers to investigate dynamic resource management for reconfigurable hardware. Early works, e.g.,[6], [7], [4] drew an analogy between tasks in software and so-called virtual or swappable hardware modules and studied fundamental operations such as scheduling; placement, relocation and defragmentation; slot-based device partitioning and reconfiguration schemes; and inter-module routing. Although these works suggested to centralize resource management in a runtime layer for convenience, an integration with a software OS was not a predominant design goal. The very few projects that resulted in implementations used FIFOs or shared memory to interface reconfigurable hardware modules with other parts of an application running in software. However, the nature of these hardware modules was still that of a passive coprocessor, which was fed with data from software tasks.

After the development of more sophisticated prototypes, e.g., a multimedia appliance using multitasking in hardware [14], several researchers, e.g.,[2], [16], [5], concurrently pushed the idea of treating hardware tasks as independent execution units, equipped with similar access to operating system functions as their software peers. Around 2004, these projects fundamentally changed the concept of reconfigurable hardware operating systems since the emerging prototypes turned hardware modules into threads or processes and offered them a set of operating system functions for inter-task communication and synchronization. These approaches can be considered the first operating systems directly dedicated to reconfigurable computing.

Soon after these first operating systems have been developed it was found that promoting hardware tasks to peers of software threads while carrying over a manually managed local memory architecture was too restrictive. Thus, researchers have studied how hardware tasks can autonomously access the main memory. For reconfigurable operating sys-

[2] `http://openlab.web.cern.ch/ice-dip`

[3] `http://sfb901.uni-paderborn.de`

tems that build on general purpose OS such as Linux, this meant that virtual memory had to be supported. The first approaches, e.g., [17], [8], solve this challenge by creating a transparently-managed local copy of the main memory and modifying the host operating system to handle page misses on the CPU. To improve the efficiency of accessing main memory, especially for non-linear data access patterns, ReconOS has later pioneered a hardware memory management unit [1] for hardware modules that translates virtual addresses without the CPU.

Current research projects on operating systems for reconfigurable computing differ mainly with respect to whether a hardware module is turned into a process, a thread or a kernel module, and in the richness of OS services made available to reconfigurable hardware. While projects such as BORPH [15] choose UNIX processes, Hthreads [3] and ReconOS use a light-weight threading model to represent hardware modules. More recently, SPREAD [18] started to integrate multithreading and streaming paradigms, while FUSE [11] focuses on a closer, more efficient kernel integration of hardware accelerators.

Compared to other approaches leveraging the threading model, especially Hthreads that focuses on low-jitter hardware implementations of operating system services, ReconOS with its unified hardware/software interfaces allows us to offer an essentially identical and rich set of OS services to both software and hardware threads. ReconOS does not require any change to the host OS, which leads to a comparatively simple tool flow for building applications, to an improved portability and interoperability through standard OS kernels, and to a step-by-step design process starting with a fully functional software prototype on a desktop.

10 Sidebar: ReconOS Versions and Availability

ReconOS has been actively developed since its inception in 2006. Since then it has gone through three major revisions and has been ported to several operating systems and hardware platforms. The first version of ReconOS used the eCos operating system running on PowerPC CPUs embedded in Xilinx Virtex-2 Pro and Virtex-4 FPGAs. Version 2 improved on the original version by providing FIFO interconnects between hardware threads, adding support for the Linux operating system, and offering a common virtual address space between hardware and software threads. Version 3, which was released in early 2013, is a major overhaul that streamlines the hardware architecture towards a more lightweight and modular design. It brings ReconOS to the Microblaze/Linux and Microblaze/Xilkernel architectures and has been used extensively on Virtex-6 FP-

GAs. A port to the new Xilinx Zynq platform will be released soon. ReconOS is open source. The source code and further information is available at `http://www.reconos.de`.

References

[1] A. Agne, M. Platzner, and E. Lubbers. Memory virtualization for multithreaded reconfigurable hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 185–188. IEEE Computer Society, September 2011.

[2] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro*, 24(4):42–53, July 2004.

[3] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, 2008.

[4] K. Bazargan, R. Kaster, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, 17(1):68–83, 2000.

[5] Neil W. Bergmann, John A. Williams, Jie Han, and Yi Chen. A process model for hardware modules in reconfigurable system-on-chip. In *Int. Conf. on Architecture of Computing Systems, Workshop Proceedings*, volume 81 of *Lecture Notes in Informatics*, pages 205–214, Bonn, Germany, March 2006. Gesellschaft fr Informatik (GI).

[6] Gordon Brebner. A virtual hardware operating system for the xilinx xc6200. In *Proc. Int. Workshop Fiel-Programmable Logic and Applications (FPL)*, pages 327–336, 1996.

[7] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for reconfigurable computing. In *Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 279–280, 2000.

[8] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–84. IEEE Computer Society, April 2007.

[9] Markus Happe, Enno Lübbers, and Marco Platzner. A Self-adaptive Heterogeneous Multi-core Architecture for Embedded Real-time Video Object Tracking. *Journal of Real-Time Image Processing*, pages 1–16, 2011. 10.1007/s11554-011-0212-y.

[10] Rob Hess. Particle Filter Object Tracking - C code. `http://blogs.oregonstate.edu/hess/code/particles`, May 2013.

[11] Aws Ismail and Lesley Shannon. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2011.

[12] Ariane Keller, Bernhard Plattner, Enno Lbbers, Marco Platzner, and Christian Plessl. Reconfigurable Nodes for Future Networks. In *Proceedings of the IEEE Globecom Workshop on Network of the Future (FutureNet)*, page 372376. IEEE, 2010.

[13] Enno Lübbers and Marco Platzner. Cooperative Multithreading in Dynamically Reconfigurable Systems. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2009.

[14] V. Nollet, P. Coene, D. Verkest, S. Vernalde, , and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc,. In *Reconfigurable Architectures Workshop, Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2003.

[15] H. K.-H. So and R. Brodersen. A Unified Hardware/software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH. *IEEE Transactions on Computers*, 7(2):1–28, 2008.

[16] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.

[17] Miljan Vuletic, Laura Pozzi, and Paolo Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design & Test of Computers*, 22(2):102–113, 2005.

[18] Ying Wang, Jian Yan, Xuegong Zhou, Lingli Wang, Wayne Luk, Chenglian Peng, and Jiarong Tong. A Partially Reconfigurable Architecture Supporting Hardware Threads. In *International Conference on Field-Programmable Technology (FPT)*, 2012.