RECONOS: AN OPERATING SYSTEM APPROACH FOR RECONFIGURABLE COMPUTING

Andreas Agne
University of Paderborn

Markus Happe
Ariane Keller
ETH Zürich

Enno Lübbers
Intel Labs Europe

Bernhard Plattner
ETH Zürich

Marco Platzner
Christian Plessl
University of Paderborn

THE RECONOS OPERATING SYSTEM FOR RECONFIGURABLE COMPUTING OFFERS A UNIFIED MULTITHREADED PROGRAMMING MODEL AND OS SERVICES FOR THREADS EXECUTING IN SOFTWARE AND THREADS MAPPED TO RECONFIGURABLE HARDWARE. BY SEMANTICALLY INTEGRATING HARDWARE ACCELERATORS INTO A STANDARD OS ENVIRONMENT, RECONOS ALLOWS FOR RAPID DESIGN-SPACE EXPLORATION, SUPPORTS A STRUCTURED APPLICATION DEVELOPMENT PROCESS, AND IMPROVES THE PORTABILITY OF APPLICATIONS BETWEEN DIFFERENT RECONFIGURABLE COMPUTING SYSTEMS.

Today's high-density field-programmable gate arrays (FPGAs) allow for implementing very complex circuits. Still, reconfigurable computing applications are rarely mapped exclusively to the FPGA accelerator. 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 in order to separate concerns and achieve concurrent or asynchronous processing. In software systems, the operating system (OS) standardizes these communication and synchronization mechanisms and provides abstractions for encapsulating the execution units (processes and threads), communication, and synchronization.

Reconfigurable computing systems still lack an established OS 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 designer's productivity, and prevent portability of applications between different reconfigurable computing systems.

The ReconOS operating system, programming model, and system architecture offers unified OS services for functions executing in software and hardware and a standardized interface for integrating custom hardware accelerators. ReconOS leverages the well-established multithreading programming model and extends a host OS with hardware thread support. These extensions let the hardware threads interact with software threads using the same standardized OS 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 OS 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 successively explore the hardware/software design space.
- The portability of applications between different reconfigurable computing systems is improved by using defined OS interfaces for communication and synchronization instead of low-level platform-specific interfaces.
- The unified appearance of hardware and software threads from the application's perspective allows functions to move between software and hardware during runtime, which supports the design of adaptive computing systems that exploit partial reconfiguration.

We discuss the evolution of operating systems for reconfigurable computing and how ReconOS relates to this heritage in the "Operating Systems for Reconfigurable Computing" sidebar.

Programming model

The key idea of ReconOS is to extend the multithreading programming model across the hardware/software interface. In multithreaded 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 programming model's objects; 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 multithreaded software applications. A major advantage that developers can draw from the ReconOS approach is that these abstractions can be used not only 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 Figure 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 the lock `count_mutex`. Using standard APIs for message passing and synchronization, the software thread accesses OS services in an expressive, straightforward, and portable way. As an additional benefit, such a thread description manages to clearly separate thread-specific processing from OS calls.

Figure 2 shows a ReconOS hardware implementation of the same thread, partitioned into similar thread-specific logic and OS interactions. While the thread-specific user logic contains the hardware thread's datapath and is limited only by available FPGA resources, the OS 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 OS calls from within hardware modules. The developer specifies the OSFSM using a standard VHDL state machine description, as shown in Figure 3. For accessing OS functions in this state machine, ReconOS provides a VHDL library that wraps all OS calls with VHDL procedures. The OSFSM's transitions are guarded by an OS-controlled signal `done` (line 47), so that blocking OS calls—such as `mutex_lock()`—can temporarily inhibit the execution of a hardware thread.

Operating Systems for Reconfigurable Computing

The introduction of the partially reconfigurable Xilinx XC6200 FPGA series in the mid 1990s and, later on, the JBits software library for bitstream manipulation, inspired researchers to investigate dynamic resource management for reconfigurable hardware. Early works 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 intermodule routing.[1-3] Although these works suggested centralizing resource management in a runtime layer for convenience, integration with a software operating system (OS) was not a predominant design goal. The few projects that resulted in implementations used first-in, first-out (FIFO) interconnects 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, such as a multimedia appliance using multitasking in hardware,[4] several researchers concurrently pushed the idea of treating hardware tasks as independent execution units[5-7] equipped with similar access to OS functions as their software peers. Around 2004, these projects fundamentally changed the concept of reconfigurable hardware operating systems because the emerging prototypes turned hardware modules into threads or processes and offered them a set of OS functions for intertask communication and synchronization. These approaches can be considered the first operating systems directly dedicated to reconfigurable computing.

Soon after these first operating systems were developed, designers found that promoting hardware tasks to peers of software threads while carrying over a manually managed local memory architecture was too restrictive. Thus, researchers began studying how hardware tasks can autonomously access the main memory. For reconfigurable operating systems that build on a general-purpose OS, such as Linux, this meant that virtual memory had to be supported. The first approaches solve this challenge by creating a transparently managed local copy of the main memory and modifying the host OS to handle page misses on the CPU.[8,9] To improve the efficiency of accessing main memory, especially for nonlinear data access patterns, ReconOS later pioneered a hardware memory management unit for hardware modules that translates virtual addresses without the CPU.[10]

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 choose UNIX processes,[11] Hthreads[12] and ReconOS use a lightweight threading model to represent hardware modules. More recently, SPREAD began integrating multithreading and streaming paradigms,[13] while FUSE focuses on a closer, more efficient kernel integration of hardware accelerators.[14]

Compared to other approaches leveraging the threading model—especially Hthreads, which focuses on low-jitter hardware implementations of OS services—ReconOS, with its unified hardware/software interfaces, lets us offer an essentially identical and rich set of OS services to both software and hardware threads. ReconOS doesn't require any change to the host OS, which leads to three key benefits: a comparatively simple tool flow for building applications, improved portability and interoperability through

Consequently, the OSFSM in VHDL closely mimics the sequence of OS 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 OS calls.

ReconOS architecture

The ReconOS runtime system architecture provides the structural foundation to support the multithreading programming model and its execution on CPU/FPGA platforms. Figure 4 shows a conceptual view of a typical system that is decomposed into the 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 runtime environment consists of hardware components that provide 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 standard OS kernels, and a step-by-step design process starting with a fully functional software prototype on a desktop.

References

1. G. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," *Proc. Int'l Workshop Field-Programmable Logic and Applications* (FPL 96), LNCS 1142, 1996, pp. 327-336.
2. K. Compton et al., "Configuration Relocation and Defragmentation for Reconfigurable Computing," *Proc. Int'l Symp. Field-Programmable Custom Computing Machines* (FCCM), 2000, pp. 279-280.
3. K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design and Test of Computers,* vol. 17, no. 1, 2000, pp. 68-83.
4. V. Nollet et al., "Designing an Operating System for a Heterogeneous Reconfigurable SoC," *Proc. 17th Int'l Symp. Parallel and Distributed Processing,* 2003, doi:10.1109/IPDPS.2003.1213320.
5. D. Andrews et al., "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro,* vol. 24, no. 4, 2004, pp. 42-53.
6. C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks," *IEEE Trans. Computers,* vol. 53, no. 11, 2004, pp. 1392-1407.
7. N.W. Bergmann et al., "A Process Model for Hardware Modules in Reconfigurable System-on-Chip," *Proc. 19th Int'l Conf. Architecture of Computing Systems,* LNCS, vol. 81, no. 3894, 2006, pp. 205-214.
8. M. Vuletic, L. Pozzi, and P. Ienne, "Seamless Hardware-Software Integration in Reconfigurable Computing Systems," *IEEE Design & Test of Computers,* vol. 22, no. 2, 2005, pp. 102-113.
9. P. Garcia and K. Compton, "A Reconfigurable Hardware Interface for a Modern Computing System," *Proc. Int'l Symp. Field-Programmable Custom Computing Machines* (FCCM 07), 2007, pp. 73-84.
10. A. Agne, M. Platzner, and E. Lübbers, "Memory Virtualization for Multithreaded Reconfigurable Hardware," *Proc. Int'l Conf. Field Programmable Logic and Applications* (FPL 11), 2011, pp. 185-188.
11. H.K.-H. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH," *ACM Trans. Embedded Computing Systems,* vol. 7, no. 2, 2008, article 14.
12. D. Andrews et al., "Achieving Programming Model Abstractions for Reconfigurable Computing," *IEEE Trans. Very Large Scale Integration Systems,* vol. 16, no. 1, 2008, pp. 34-44.
13. Y. Wang et al., "A Partially Reconfigurable Architecture Supporting Hardware Threads," *Proc. Int'l Conf. Field-Programmable Technology* (FPT 12), 2012, pp. 269-276.
14. A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," *Proc. IEEE 19th Ann. Int'l Symp. Field-Programmable Custom Computing Machines* (FCCM 11), 2011, pp. 170-177.

offer an interface to the hardware, the OS, and the application's software threads.

A key component for multithreading across the hardware/software boundary is the *delegate thread,* a lightweight software thread that interfaces between the hardware thread and the OS. When a hardware thread needs to execute an OS function, it relays this request through the OSIF to the delegate thread using platform-specific (but application-independent) communication interfaces. The delegate thread then executes the desired OS 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 a thread's execution mode—that is, whether it runs in software or hardware. While the delegate mechanism causes a certain overhead for executing OS calls, the resulting simplicity of switching thread implementations between software and hardware greatly facilitates system generation and design space exploration.

The ReconOS concept is rather general and has been ported to several FPGA families, main CPU architectures, and host operating systems (see the "ReconOS Versions and Availability" sidebar). For the rest of this article, we describe the implementation of

```
extern mutex_t *count_mutex;           // mutex protecting packet counter
extern mqd_t mbox_in,                  // ingress packets
             mbox_out;                 // egress packets

void *thread_a_entry( void *count_ptr ) {
   data_t buf;                         // buffer for packet processing

   while ( true ) {
      buf = mbox_get ( mbox_in );      // receive new packet
      process ( buf );                 // process packet
      mbox_put ( mbox_out, buf );      // send processed packet
      mutex_lock ( count_mutex );      // acquire lock
      ( (count_t) *count_ptr )++;      // update counter
      mutex_unlock( count_mutex );     // release lock
   }
}
```

Figure 1. Example of a stream processing software thread using operating system services. The software thread accesses OS services in an expressive, straightforward, and portable way.

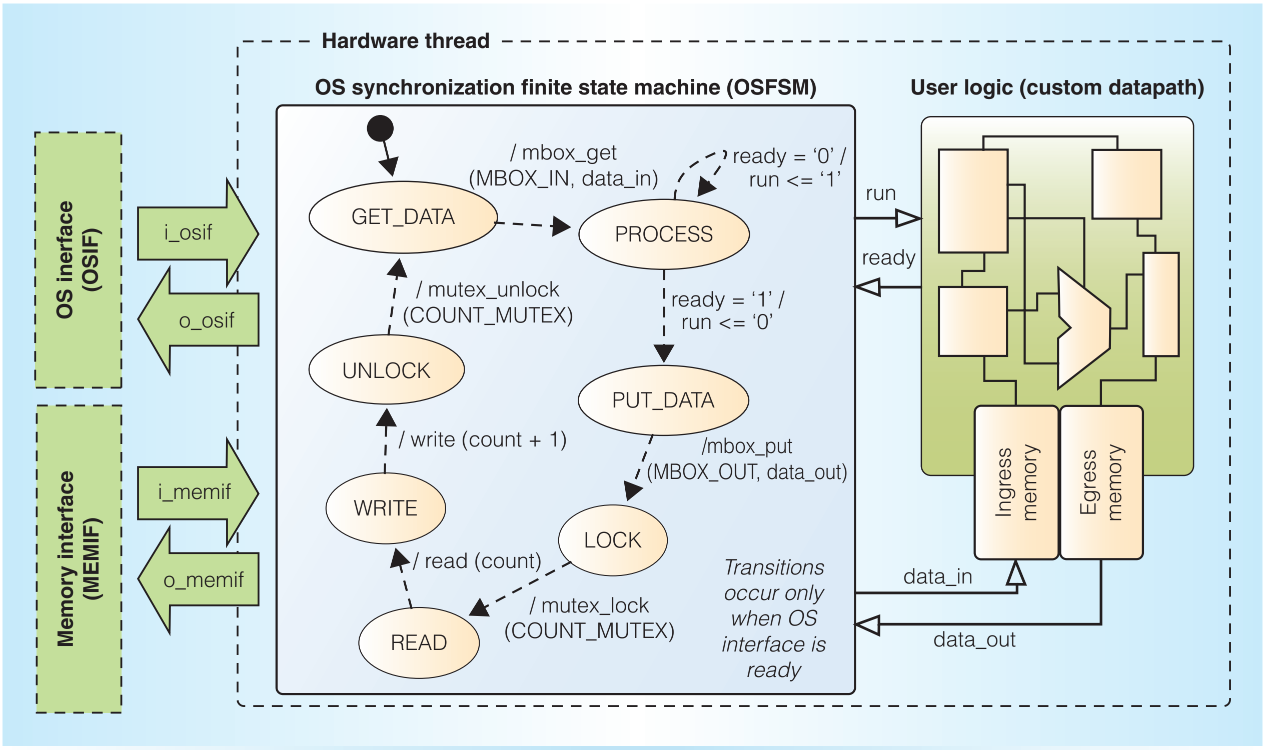

Figure 2. A ReconOS hardware thread comprises the OS synchronization finite state machine and the user logic implementing the datapath. Together with the OS interface (OSIF), the OS synchronization finite state machine enables seamless OS calls from within the hardware thread. The memory interface (MEMIF) provides the hardware thread with access to the ReconOS memory subsystem.

```
OSFSM: process (clk, reset)
  variable ack: boolean;
begin

  if reset = '1' then
    state <= GET_DATA;
    run <= '0';
    osif_reset (o_osif , i_osif);
    memif_reset (o_memif, i_memif);
  elsif rising_edge (clk) then

   case state is

     when GET_DATA =>
       mbox_get (o_osif,i_osif,MB_IN,data_in,done);    -- receive new packet
       next_state <= COMPUTE;

     when COMPUTE =>
       run <= '1';                                      --  process packet
       if ready = '1' then
         run <= '0';
         next_state <= PUT_DATA;
       end if;

     when PUT_DATA =>
       mbox_put (o_osif,i_osif,MB_OUT,data_out,done); -- send processed packet
       next_state <= LOCK;

     when LOCK =>
       mutex_lock (o_osif,i_osif,CNT_MUTEX,done);      -- acquire lock
       next_state <= READ;

     when READ =>
       read (o_memif,i_memif,addr,count,done);
       next_state <= WRITE

     when WRITE =>
       write (o_memif,i_memif,addr,count + 1,done);    -- update counter
       next_state <= UNLOCK;

     when UNLOCK =>
       mutex_unlock (o_osif,i_osif,CNT_MUTEX,done);    -- release lock
       next_state <= GET_DATA;

   end case;

   if done then state <= next_state; end if;

 end if;
end process;
```

Figure 3. OS synchronization finite state machine (OSFSM) for a stream processing hardware thread. In order to simplify coding the OSFSM, ReconOS provides a VHDL library with procedures that wrap OS calls.

ReconOS v3, which is the most recent version of ReconOS targeting Xilinx Virtex-6 FPGAs and using 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 OS call signaling, such as `mutex_lock()` in Figure 3.

Technically, the VHDL procedures implement further state machines that are nested

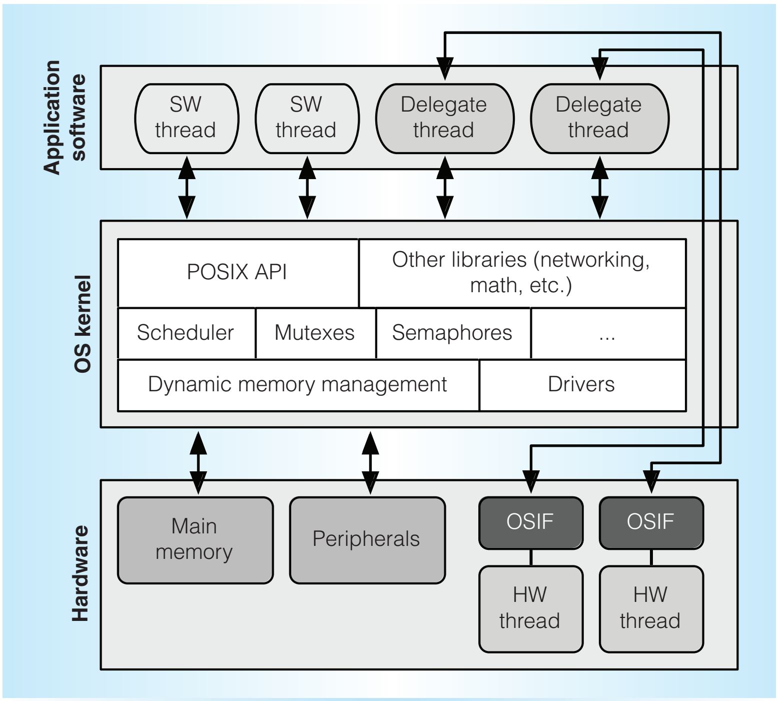

Figure 4. Conceptual overview of the ReconOS system architecture. Software threads interact directly with the OS kernel, while hardware threads connect through an OS interface (OSIF) and delegate threads.

within the OSFSM and access the OSIF through the two first-in, first-out (FIFO) buffers, `i_soif` and `o_osif`. Figure 5 outlines the relationship between the OSFSM, the nested state machine implementing the `mutex_lock` procedure, and the two FIFO buffers. Synchronization between the nested state machines and the OSFSM is controlled via the handshaking signal `done`. For communicating with the delegate thread, we use a 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 the delegate thread has sent an acknowledgement and, optionally, 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 6 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 2, 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 OS, ReconOS implements a full-featured memory management unit (MMU), including a translation look-aside buffer, which can autonomously translate addresses using the Linux kernel's page tables.[1] Hardware threads use FIFO buffers to communicate with the memory subsystem; one outgoing and one incoming FIFO buffer per hardware thread. Requests for memory transactions are encoded and written to the outgoing FIFO buffer, followed by data in the case of a write request. In the case of a read request, data become available on the incoming FIFO buffer 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 FIFO buffers, and automatically transfer data to and from local memory elements within the hardware thread.

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 multithreading approach offered by ReconOS simplifies the development process, which is typically structured in three steps. First, the developer prototypes the application's functionality in multithreaded software using, for example, the Pthreads library on Linux. This first software-based implementation allows for functional testing. Second, the multithreaded software is ported to the embedded CPU on the targeted platform FPGA, such as a MicroBlaze running Linux. The developer can then use profiling to identify the

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 OS running on PowerPC CPUs embedded in Xilinx Virtex-2 Pro and Virtex-4 FPGAs. Version 2 improved on the original by providing first-in, first-out (FIFO) interconnects between hardware threads; adding support for the Linux OS; 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 toward 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 FPGAs. A port to the new Xilinx Zynq platform will be released in early 2014. ReconOS is open source; the source code and further information is available at http://www.reconos.de.

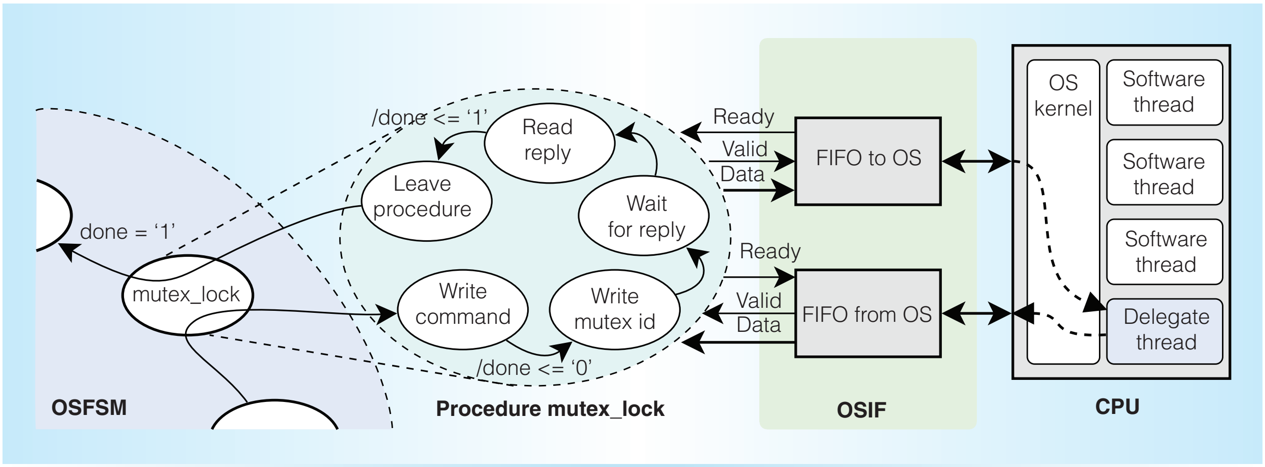

Figure 5. A finite state machine nested within the operating system's synchronization finite state machine handles the communication between the hardware thread and the OS (via the OSIF and delegate thread). The OSIF contains two first-in, first-out (FIFO) buffers that connect the hardware thread with the CPU. The OS relays the hardware thread's request to the respective delegate thread, where the request is carried out.

application's potential for parallel execution—that is, 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 lets the developer evaluate different mappings of threads to hardware and software and to quickly assess the overall performance on the target system.

ReconOS tool flow

Figure 7 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 a 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 the kernel image, respectively. On the hardware side, components such as the OS and

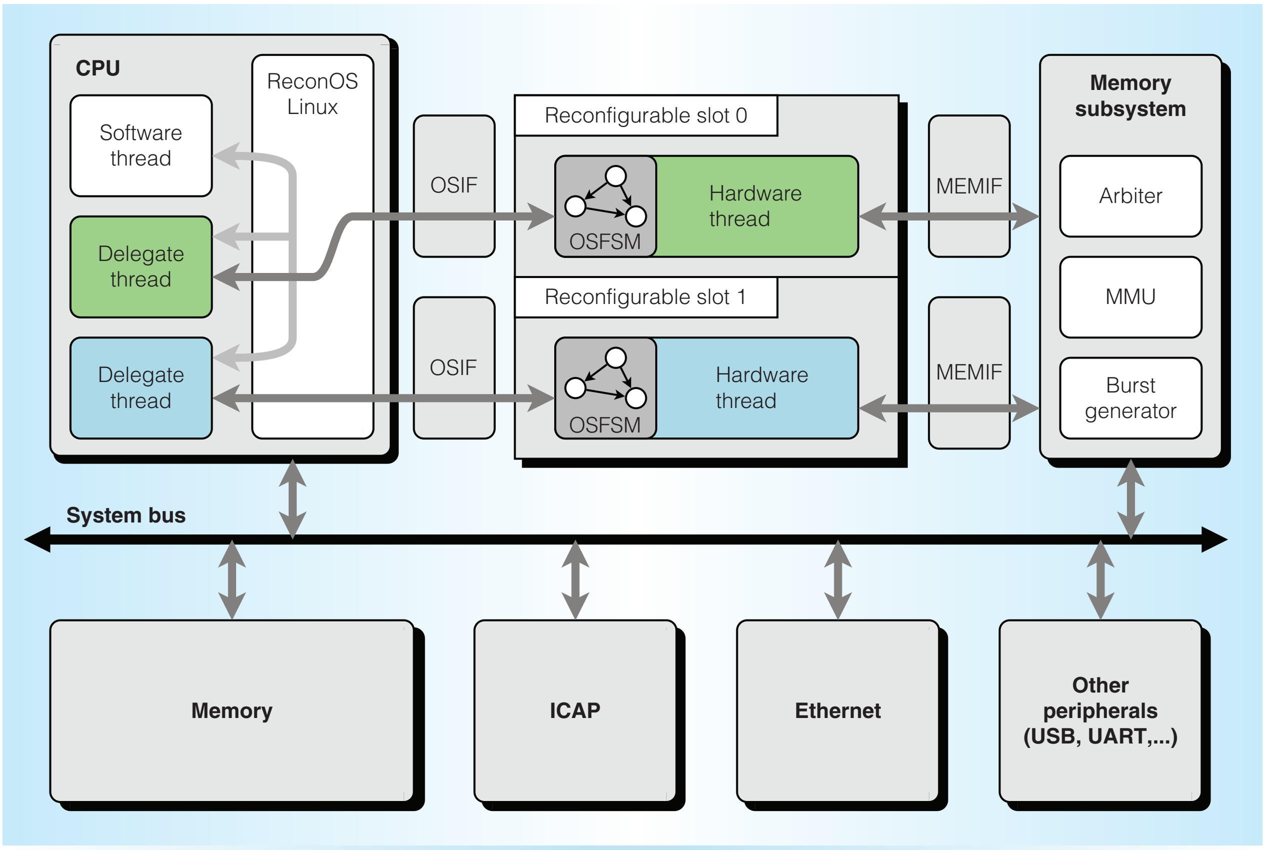

Figure 6. Example of a ReconOS hardware architecture with a CPU, two reconfigurable hardware slots, a memory subsystem, and various peripherals. Hardware threads reside in reconfigurable hardware slots and can access the OS kernel on the CPU via the OSIF and system memory via the MEMIF.

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—for example, using `rthread_create()` instead of `pthread_create()`. Such a decision for software or hardware can even be made during runtime (see the "Applications of ReconOS" sidebar).

Case study: Video object tracker

To illustrate the benefits of the ReconOS approach, we present a particle-filter-based video object tracker for continuous estimation of an object's position and size in a video sequence.[2] 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.

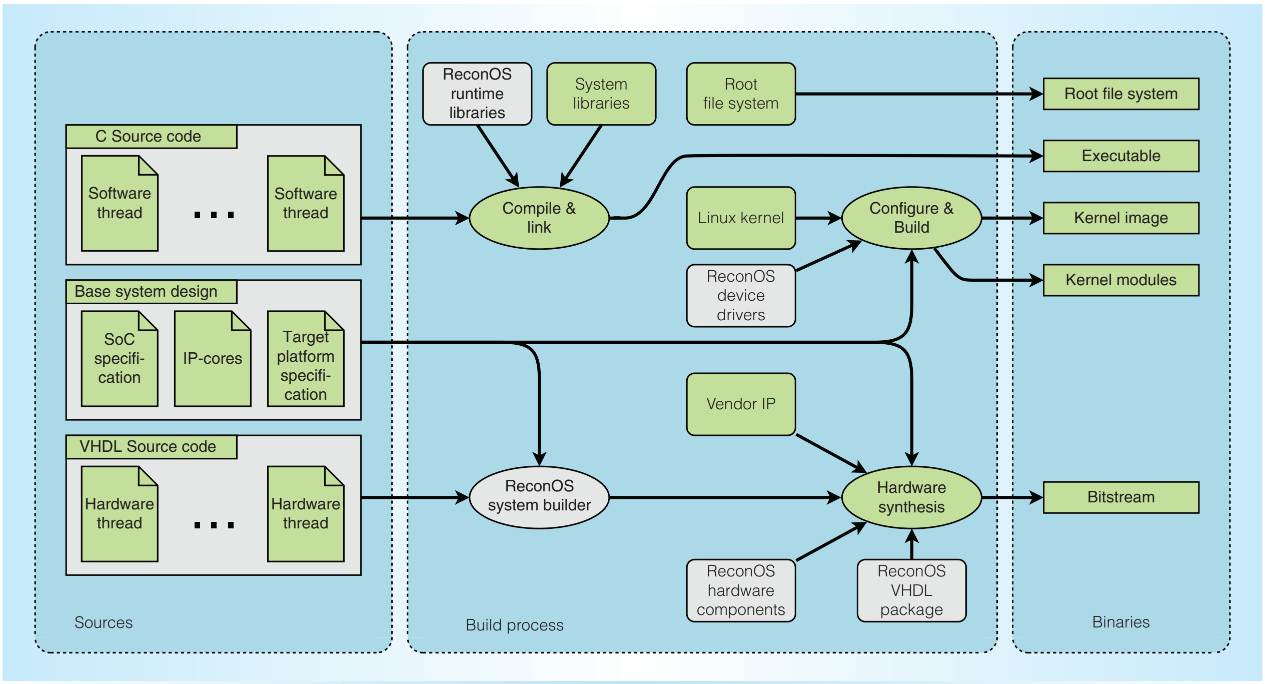

Figure 7. Tool flow for assembling a ReconOS system on a Linux target. ReconOS-specific steps are colored gray.

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 allows for exploiting FPGA resources in unconventional ways—for example, by loading hardware threads on demand, moving functionality between software and hardware, or even multitasking hardware slots by time-multiplexing. ReconOS supports DPR by dividing the architecture into a static part and a dynamic part. The static part contains the processor, the memory subsystem, OS interfaces, memory interfaces, 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 partial bitstreams for each desired hardware thread/slot combination. Time-multiplexing of hardware slots is supported through cooperative multitasking.[1]

We 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 computational resources availability. The developed architecture[2] 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 created self-adaptive implementations of the particle filter presented in the main article 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 predefined band. In the EPiCS project 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 on the basis of 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 computing context. For example, ReconOS is currently being evaluated for use in high-speed data acquisition and particle physics applications. In current work, we also are studying how ReconOS can be ported to x86-based server systems that attach FPGA accelerator cards via PCI Express.

References

1. E. Lübbers and M. Platzner, "Cooperative Multithreading in Dynamically Reconfigurable Systems," *Proc. Int'l Conf. Field Programmable Logic and Applications* (FPL 09), 2009, pp. 551-554.
2. A. Keller et al., "Reconfigurable Nodes for Future Networks," *Proc. Workshop Network of the Future,* 2010, pp. 372-376.

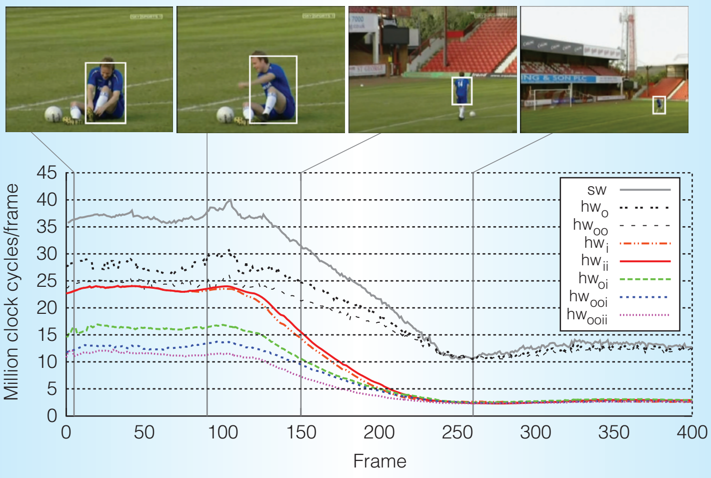

Figure 8. Design-space exploration for a video object tracker: The graph shows the computational effort for tracking versus time in video frames for a specific video (taken from Hess[3]). The individual curves represent ReconOS implementations with different hardware/software mappings, where "sw" denotes an all-in-software system, and curves labeled "hw" denote systems with one to four threads of type observation (*o*) and importance (*i*) running in reconfigurable hardware.

For our implementation, we start with an existing video object tracker implemented in C.[3] First, we transform the monolithic code into a multithreaded implementation on a desktop using Posix Pthreads under Linux. Each filter stage can be naturally turned into a software thread, and the particles, grouped into chunks, are forwarded between the filter stages via message boxes. Because the particles are independent and thus can be processed in parallel, each stage is represented by multiple thread instances exploiting data parallelism. Second, we port our multithreaded 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 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 multithreaded 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 8 shows the required computational effort in execution time per frame of various mappings for tracking a soccer player. The tracker that achieves the highest performance is the one that employs four hardware threads, two for observation and two for importance (mapping *hwooii*). Clearly, the required effort decreases when the object moves into the background. There, mapping *hwi* with a single hardware thread for importance achieves comparable performance results.

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

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901), the International Graduate School of Dynamic Intelligent Systems, and the European

Union Seventh Framework Programme under grant agreement 257906 (EPiCS).

References

1. A. Agne, M. Platzner, and E. Lübbers, "Memory Virtualization for Multithreaded Reconfigurable Hardware," *Proc. Int'l Conf. Field Programmable Logic and Applications* (FPL 11), 2011, pp. 185-188.
2. M. Happe, E. Lübbers, and M. Platzner, "A Self-Adaptive Heterogeneous Multi-core Architecture for Embedded Real-Time Video Object Tracking," *J. Real-Time Image Processing,* vol. 8, no. 1, 2013, pp. 95-110.
3. R. Hess, "Particle Filter Object Tracking," blog, May 2013, http://blogs.oregonstate.edu/hess/code/particles.

Andreas Agne is a PhD student in the Computer Engineering Group at the University of Paderborn. His research interests include reconfigurable computing and operating systems for heterogeneous multicore architectures. Agne has a Diploma in computer science from the University of Paderborn.

Markus Happe is a senior researcher at the Communication Systems Group at ETH Zürich. His research interests include networking architectures, self-adaptation strategies, and reconfigurable systems. Happe has a PhD in computer science from the University of Paderborn.

Ariane Keller is a PhD student in the Communication Systems Group at ETH Zürich. Her research interests include computer architectures for self-organizing networks. Keller has a Diploma in electrical engineering from ETH Zürich.

Enno Lübbers is a senior researcher at the Intel Open Lab in Munich, which is part of Intel Labs Europe. His research interests include adaptive systems and heterogeneous architectures for high-performance, embedded, and safety-critical applications. Lübbers has a PhD in computer engineering from the University of Paderborn.

Bernhard Plattner is a full professor of computer engineering in the Department of Information Technology and Electrical Engineering at ETH Zürich, where he leads the Communication Systems Group. His research interests include self-organizing networks, mobile and opportunistic networking, and practical aspects of information security. Plattner has a PhD in computer engineering from ETH Zürich.

Marco Platzner is professor of computer engineering in the Department of Computer Science at the University of Paderborn. His research interests include reconfigurable computing, hardware-software codesign, and parallel architectures. Platzner has a PhD in telematics from Graz University of Technology.

Christian Plessl is assistant professor of custom computing in the Department of Computer Science at the University of Paderborn. His research interests include parallel and reconfigurable computer architectures, high-performance computing, and adaptive computing systems. Plessl has a PhD in computer engineering from ETH Zürich.

Direct questions and comments about this article to Christian Plessl, University of Paderborn, Department of Computer Science, Warburger Str. 100, 33098 Paderborn, Germany; christian.plessl@uni-paderborn.de.