Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations

Herbert Walder and Marco Platzner
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

Abstract ***In this paper, we approach the rather new area of reconfigurable hardware operating systems in a top-down manner. First, we describe a design concept that defines basic abstractions and operating system services in a device-independent way. Then, we refine this model to an implementation concept on the Xilinx Virtex XCV-800 technology. The implementation concept proposes a multitasking environment that executes relocatable hardware tasks, uses a memory management unit translating task requests to internal and external memory accesses, and relies on device drivers and triggers to connect to external I/O. Finally, we present a prototypical implementation and an application case study. The application consists of a set of dynamically loaded and executed networking and multimedia tasks such as IP packet processing, AES decryption, and audio stream decoding.***

Keywords: Hardware Operating System, FPGA, multitasking

I. Introduction

Embedded systems may consist of a large variety of different processing elements, memories, I/O devices, sensors, and actors. The processing elements split into software-programmable CPUs, fixed-function hardware (ASICs), and (re)programmable hardware. Until recently, such systems used programmable hardware mainly as ASIC replacements with much shorter time-to-market and the capability of hardware updating after system deployment.

Today, the increasing densities and reconfiguration modes of SRAM-based field-programmable gate arrays (FPGAs) and configurable systems on a chip (CSoCs) advocate more dynamic uses of these components. Many promising application domains for *reconfigurable embedded systems*, such as wearable computing [6], mobile systems [4], and network processors [3], combine high performance demands with frequent application changes. The dynamics in such systems is caused by user requests and packet flows in the communication networks.

As a consequence of treating reconfigurable devices as dynamic processing elements, the introduction of *reconfigurable hardware operating systems* follows. Much like real-time operating systems for CPUs, a reconfigurable hardware operating system offers a set of services to the application developer. At design time, an application is composed out of objects such as tasks, buffers, semaphores, and timers. The main abstraction is the *hardware task* which captures an application functionality in an as much as possible device-independent way. At runtime, the operating system resolves resource conflicts and ensures connectivity between the application objects. The use of a reconfigurable hardware operating system results in a number of benefits, as it

- *increases productivity:* The operating system objects and the task abstraction facilitate the re-use of tested and reliable code and circuitry. This can considerably speed up development cycles and shorten time-to-market.
- *increases portability:* Porting applications is greatly simplified by operating systems that are available for different target platforms. Recompilation/resynthesis often results in a functionally correct ported version, allowing the developer to concentrate on performance figures.
- *eases system re-partitioning:* The system can be repartitioned between different hardware components or even between hardware and software components. A task that was previously implemented in software running on a CPU could be mapped to an FPGA to increase its performance. New objects can be inserted or existing objects can be removed while the rest of the system remains unchanged. This allows to fix bugs and to enhance and customize applications. In much shorter periods of time, repartitioning can be used to balance the load.
- *simplifies debugging:* Debugging dynamically reconfigured hardware or communicating hardware and software objects is a challenge for its own. An operating system supports debugging with monitoring and triggering facilities that give insight into the interaction between the application objects.

A reconfigurable hardware operating system forms an abstraction that hides the details of the underlying technology from the developer. As any abstraction in system design, the gained productivity is paid for by an overhead in runtime and area/memory.

Reconfigurable hardware operating systems are a rather new line of research. The first description of hardware multitasking is due to Brebner [1]. Recently, Wigley et al. discussed OS services including device partitioning, placement and routing [10]. Multitasking, task preemption, and scheduling was investigated by Simmler et al. [7], Brebner/Diessel [2], and Walder et al. [8], respectively. In [5], Mignolet et al. introduce *relocatable tasks* which can be executed either in software or in hardware, depending on the available resources and the

performance required.

Many questions have not been addressed yet. This holds for conceptual, algorithmic, as well as practical issues.

Due to the many open ends in this field, it is difficult to compare existing work or even to separate conceptual issues from implementation details and limitations of currently available FPGA technology.

To foster a more structured view of reconfigurable hardware operating systems, we follow a top-down approach in this paper.

In Section II, we start out with the discussion of a *design concept*. The design concept describes the reconfigurable hardware operating system on a rather high level of abstraction, focusing on general aspects such as models, modules, and required operating system functions and services. The design concept does not address implementation or technology specific details. Section III refines the design concept to an *implementation concept*. The implementation concept maps the objects of the operating system to a specific technology, while it still maintains application independency. We have chosen Xilinx Virtex as target technology and demonstrate a complete implementation of an operating system. Finally, Section IV presents a prototypical case study. The case study implements a control/data-flow application running on our reconfigurable hardware operating system. The application executes networking and multimedia tasks, such as IP packet processing, AES decryption, audio streaming, and display functions.

II. Design Concepts

A. Reconfigurable Surface Partitioning

Any reconfigurable device offers a number of configurable logic units and routing elements. We propose to partition the reconfigurable area into two kinds of regions:

- The *OS frame* accommodates circuits that constitute the runtime part of the operating system. These circuits are required to establish operating system functions such as memory access, communication, and I/O.
- The *user area* is devoted to accommodate the application functions, implemented in form of hardware tasks. Tasks are dynamically allocated to the user space, executed there, and removed upon completion.

B. Hardware Tasks

Hardware tasks represent the central elements in a reconfigurable hardware operating system environment. A set of cooperating HW-tasks form the applications running on the system (see Figure 3).

C. OS-Modules

Figure 1 presents a set of modules that provide the operating system services. The modules are partitioned between the host CPU and the FPGA. The CPU modules can be categorized into three levels. The highest level of operating system modules are responsible for task, resource, and time management:

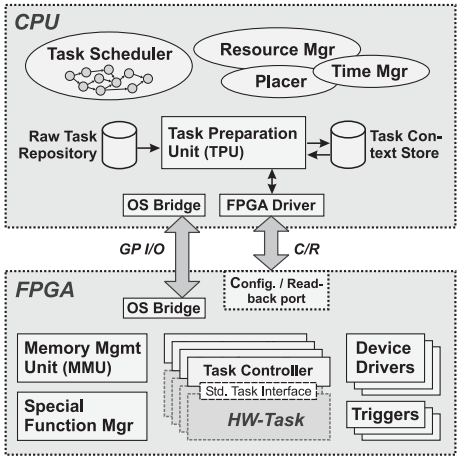

Fig. 1. OS modules running on CPU and FPGA

- *Task Scheduler:* The task scheduler decides which task has to be executed next, among all tasks ready to run. It receives events that are generated by different sources during runtime, e.g., by queues, timers, device drivers, and triggers.
- *Resource Manager:* This module keeps track of all dynamically assigned resources, such as user area, queues, triggers, and device drivers. Task scheduling and resource management are strongly coupled, which is in contrast to scheduling software tasks on single processors.
- *Time Manager:* This module offers time-based services to tasks, such as one-shot and periodic timer events.

The intermediate level of operating system modules performs the handling of task bitstreams and tasks states:

- *Raw Task Repository:* This repository stores task circuits in their raw form, i.e., in a position-independent form which is generated by the task design flow.
- *Context Store:* This module holds the contexts that have previously been extracted from preempted tasks.
- *Task Preparation Unit (TPU):* The TPU generates and analyzes partial bitstreams that represent the tasks. Following services are provided: *Task relocation* takes a raw task and a position in the user area and generates a partial bitstream that can be downloaded to the FPGA. *Context extraction* takes a read back partial bitstream and extracts the context. *Context insertion* takes a raw task and its previously stored context and generates a partial bitstream.

The lowest level of operating system modules on the CPU deal with communication and configuration:

- *OS Bridge:* Since both CPU and FPGA accommodate part of the operating system, a communication channel between the two devices is required. The operating system modules use this channel to exchange commands and data.
- *FPGA Driver:* This driver provides device-independent configuration and readback services to the TPU. The

services comprise full and partial configuration as well as full and partial readback. Physically, the driver connects to the FPGA's configuration and readback port.

The list of modules mapped to the FPGA includes:

- *Task Controller:* The task controller is the operating system's interface to a user task. Each loaded task has one task controller assigned that supplies the task with control signals and provides connectivity to OS elements.
- *Memory Management Unit (MMU):* The MMU offers memory services to the tasks, such as FIFO queues with specific access modes (blocking/non-blocking), private memory blocks, or shared memory blocks. The memory structures are implemented with the FPGA's internal memories and externally connected memory devices.
- *Special Function Manager:* The special function manager offers services to tasks, based on hardware functions implemented in the FPGA, such as multiplyers.
- *Device Drivers / Triggers:* Device drivers implement circuits that control external devices and offer services to user tasks. Encapsulating access to external devices in device drivers offers similar advantages as in software: the access functions are independent of the actual I/O device and mutual exclusion issues can be resolved. Further, time-critical I/O protocols are handled by permanently resident optimized driver functions.
  A trigger is basically a special form of a device driver that is used for rather simple external devices, e.g. switches, that can only generate events, which are then routed to operating system modules.

D. Programming Model and Design Flow

The hardware operating system we discuss in this paper requires that an application is decomposed into tasks and objects such as buffers, timers, etc. While the task functionality has to be defined by the developer, the interaction with the operating system modules must follow predefined patterns. Therefore, we provide the developer with i) a *task template* including a standard task interface used to request OS services and ii) a *task design flow* that generates raw (position-independent) tasks. The task template itself depends on the actual implementation details of the target system, e.g., the set of supported queues and device drivers, and the maximum size of a task.

III. Implementation Concepts

In this section, we describe the implementation of a reconfigurable operating system based on Xilinx Virtex technology.

A. Target Technology and -Platform

We have chosen the XESS XSV-800 board [11] as implementation platform. This board integrates a Virtex XCV-800 SRAM based FPGA and a variety of different I/O devices, i.e. Ethernet and RS232 transceivers, video digitizer and RAM-DAC, display elements, user switches and an audio codec. We have modified the board's configuration controller to allow for full and partial configuration and readback. A PC equipped with a PCI I/O card directly connects to the configuration controller and thus provides bidirectional access to the FPGA's configuration and readback port. In parallel, a number of wires implement general purpose I/O between PC and FPGA. Xilinx' *ISE Foundation 5.1* in combination with the *Modular Design* package [12] serves as development environment for circuit / bitstream generation; all PC-software has been created with *MS Visual C++* using library functions to access the PCI I/O-card.

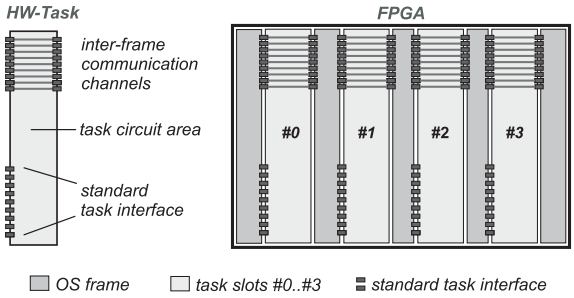

Fig. 2. Task template / OS frame parts / task slots

B. FPGA Surface Partitioning

Xilinx Virtex devices allow for partial reconfiguration and readback during runtime. However, these operations are limited to vertical chip-spanning columns. Due to this constraint, the partitioning of the reconfigurable area, as described in Section II-A, is also done column-oriented.

We split the FPGA surface into several vertical parts, the *OS-frame* parts and *task slots* (see Figure 2). This structure ensures reconfiguration of task slots without affecting neighboring OS-frame parts. A number of *inter-frame communication channels (IFCC)* enables communication between OS-frame parts.

C. Hardware Tasks / Task Slots / Standard Task Interface

Task slots are placeholders for HW tasks. A task slot defines obligatory guidelines for HW tasks:

- *Width:* Tasks to be loaded into a task slot must not exceed the slot's width.
- *Standard task interface (STI):* The only way for tasks to interact with outside elements is across the standard task interface.
- *Inter-frame communication channels (IFCC):* A number of IFCCs need to pass through tasks to ensure OS-frame connectivity. Tasks must implement these channels.

The STI as well as IFCC endpoints are realized with location invariant *bus macros*, a new design element of Xilinx' Modular Design package. All running tasks are controlled by the OS via their standard task interface (STI). Tasks must provide a minimal set of functions which can be invoked by the OS through dedicated STI pins:

- *RST (Reset):* This signal initializes a task circuit after it has been loaded into a slot.

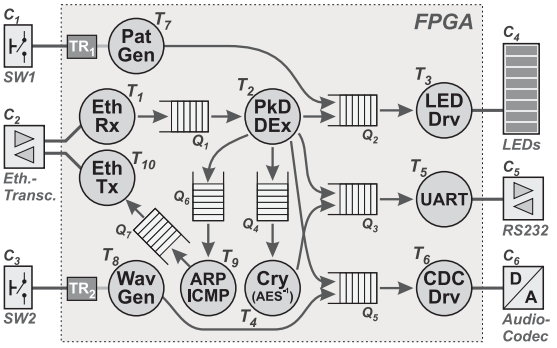

Fig. 3. DFG of the case study application

- *ENA (Enable):* This signal starts and stops task execution.
- *FIN (Finished):* This signal is used by the task to indicate its completion. Then, the OS controller generates an event which might be used by the scheduling module.

On the other hand, tasks can make use of services offered by the OS. For example, tasks can

- read from / write to FIFO buffers and check the buffer states,
- send to / receive from device drivers, and
- check triggers.

IV. Case Study

To put our concepts into practice, we have implemented a case study based on the XESS prototyping board. The case study executes a control/data-flow application that performs networking and multimedia tasks in an OS environment.

A. Functional Description

From a user's point of view, the case study consists of several applications that are activated by the user or incoming Ethernet packets (as shown in Figure 3). The system can

- generate an artificial audio signal,
- display light patterns on the LED bar,
- receive UDP packets via the Ethernet interface and forward their payload to either the RS232 interface or the audio codec (audio streaming, 12kHz, 16bit, mono). Data for the RS232 output can be plain-text or, optionally, AES encrypted;
- reply to incoming ARP (address resolution protocol) and ICMP (Internet control message protocol) echo requests.

B. OS Disposition

A Xilinx Virtex XCV-800 contains 84 columns $\times$ 56 rows of configurable logic blocks (CLBs). In this case study, we created an OS frame that consists of three OS-frame parts, OS_{left}, OS_{center} and OS_{right}, as well as two task slots S_0 and S_1. The widths of the three OS-frame parts amount to 8, 10, and 8 CLBs. OS_{left} and OS_{right} are located at the FPGA's left and right edges, since all BRAM cells need to be captured by the OS-frame. The task slots have a width of 28 CLBs each. The standard task interface (STI) consumes 12 bus macros, thus providing 48 signals. A total of 44 inter-frame communication channels (IFCCs) are established in each task slot to provide inter-frame communication. This layout results in an OS area overhead of about 33.3% of the XCV-800's size.

C. Runtime Observations

The case study application shown in Figure 3 is fully operational. There are five tasks that can be loaded on demand and two task slots, which requires a minimal scheduling activity. The file lengths of the partial bitstreams stored in the raw task repository ranges from 84.2kB (for T_8) to 182.8kB for the most complex task (T_4, AES decryption). The task activation times, including download and reset/enable commands, are between 290ms for small tasks and 510ms for large tasks (AES). The PCI I/O card turned out to be the bottleneck in the configuration path.

A more detailed report on the implementation concepts and the case study application presented in this paper can be found in [9].

V. Acknowledgements

This work was supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99.

References

[1] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 327–336. Springer, 1996.

[2] G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. In *Proceedings of the 11th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 182–191. Springer, 2001.

[3] S. Chakraborty, M. Gries, S. Künzli, and L. Thiele. Design Space Exploration of Network Processor Architectures. In *Network Processor Design: Issues and Practices, Volume 1*, pages 55–89. Morgan Kaufmann Publishers, October 2002.

[4] IMEC Interuniversity Micro Electronic Center, T-ReCS Gecko, `http://www.imec.be`.

[5] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, V. S., and R. Lauwreins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 986–991. IEEE Computer Society, March 2003.

[6] C. Plessl and et al. Reconfigurable Hardware in Wearable Computing Nodes. In *Proceedings of the 6th International Symposium on Wearable Computers (ISWC)*, pages 215–222. IEEE Computer Society, October 2002.

[7] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA Coprocessors. In *Proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 121–130. Springer, 2000.

[8] H. Walder and M. Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.

[9] H. Walder and M. Platzner. Reconfigurable Hardware OS Prototype. Technical Report TIK Nr. 168, Swiss Federal Institute of Technology (ETH), Zurich, April 2003.

[10] G. Wigley and D. Kearney. Research Issues in Operating Systems for Reconfigurable Computing. In *Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms(ERSA)*, pages 10–16. CSREA Press, Juni 2002.

[11] Xess Corporation, XSV-800 Xilinx Virtex Prototyping Board, `http://www.xess.com`.

[12] Xilinx Inc., Advanced Design Techniques, Modular Design, `http://www.xilinx.com`.