Reconfigurable Hardware OS Prototype

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 detailed 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, partial reconfiguration, multitasking

1 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 [7], mobile systems [5], 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

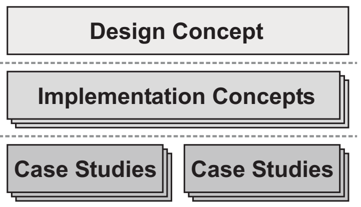

Figure 1: Design Layers

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 a as much as possible device-independent way. At runtime, the operating system resolves resource conflicts and ensures connectivity between the application objects. 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 peri-

ods 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 [11]. Multitasking, task preemption, and scheduling was investigated by Simmler et al. [9], Brebner/Diessel [2], and Walder et al. [10], respectively. In [6], 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, as depicted in Figure 1.

In Section 3, 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 4 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. We address the most critical issues and limitations when realizing an operating system on this technology.

Finally, Section 5 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, DES decryption, audio streaming, and display functions.

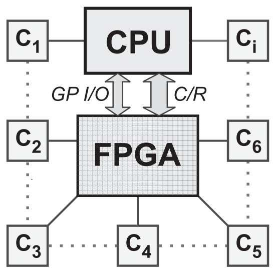

Figure 2: System architecture (concept)

2 Target System

In this work, we consider the system architecture conceptualized in Figure 2. A partially reconfigurable SRAM-based FPGA resource is coupled to a host CPU by two bidirectional channels:

- The configuration and readback port (C/R) enables the CPU to completely control the FPGA's configuration. During system runtime full and partial bitstreams can be downloaded to the FPGA, and full and partial device states can be read back.
- The interface port provides a number of general-purpose signals (GP I/O) between the host CPU and the FPGA. This port allows for communication between host and FPGA application functions.

Both CPU and FPGA are further connected to a number of external components C_1 to C_i, such as memories and I/O devices. In this work we do not address arbitration issues between CPU and FPGA to access shared resources. Hence, the components in our system architecture are either connected to the CPU or the FPGA. The architectural concept of Figure 2 covers many implementations. The system can be realized by several devices or as integrated Configurable System on a Chip (CSoC).

3 Design Concepts

Task and resource management are central issues in any operating system. Tasks are the smallest units of execution. Developers compose their applications from a set of interacting tasks. While these abstractions are well-established in software, they are still in discussion for hardware operating systems. This section first proposes design concepts for resource and task abstractions. Then, a set of operating system modules is presented and the design flows for building applications and the operating system itself are outlined. Finally, we discuss quality metrics that can be used to evaluate implementations of reconfigurable hardware operating systems.

3.1 Reconfigurable Surface Partitioning

The reconfigurable device, offers a number of configurable logic units and routing elements. We propose to partition this 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 OS frame is static and remains stable during the system's operation.
- 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.

3.2 Hardware Tasks

Hardware tasks are user circuits with the following characteristics that have to be exposed to the operating system:

- The *size* is the area requirement of the task's circuit, given in number of reconfigurable units.
- The *clock range* gives the frequencies at which the task can run. Design tools usually report an upper bound for the clock rate. A task may, however, require a specific clock rate, for example to derive a timer object that relates events to physical time. A task might further require a clock rate in a certain interval to preserve timing requirements of I/O devices or memory.

A task has a number of *required cycles* to execute, which might or might not be known in advance. The actual execution time is determined by the number of cycles and the clock rate at which the task runs.

Generally, the user area will be large enough to accommodate several tasks at once, enabling true parallel processing. As hardware tasks are dynamically loaded into the user area, we cannot ensure that a certain task always gets allocated to the same area. Consequently, we require task *relocatability*. The use of position-dependent resources limits the relocatability of tasks. Modern FPGAs have special resources such as block RAM and multipliers. Block RAMs are memory resources and are controlled by the operating system, as any other external memory.

Software tasks use kernel calls to make use of operating system services. Similarly, hardware task need a means to issue service requests to the reconfigurable hardware operating system. Therefore every task has a *standard task interface (STI)*. All tasks in the system implement the same standard task interface.

More efficient systems can be built if tasks may be *preempted* and later on resumed. Preemption of hardware tasks requires the readback and of the task's circuit and the subsequent extraction of the task's context. The context of a task is defined by the state of all its state-holding elements. To allow the context extraction, a preemptible task must provide a list of all its state-holding elements to the operating system. For preemptible tasks, we further define *time critical sections*. A time critical section denotes a time period during which a task must not be preempted.

3.3 OS-Modules

Figure 3 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:

- *Task Scheduler*
  The task scheduler decides which task has to be executed next, among all tasks ready to run. This decision is based on some scheduling policy. Our design concept supports a wide range of scheduling policies. Scheduling can be off-line or on-line. An off-line schedule is suitable for statically-defined applications and is reflected by a rather simple task sequence table. On-line schedulers are priority-driven and split into non-preemptive or preemptive schedulers. The task scheduler receives events that are generated by different sources during runtime, e.g., by queues, timers, device drivers, and triggers.
- *Resource Manager*
  This modules keeps track of all dynamically assigned resources, such as user area, queues, triggers, and device drivers. Once the task scheduler decides to execute a task, the resource manager is responsible to provide a suitable free space in the user area and all other resources the task requires. Task scheduling and resource management are strongly coupled, which is in contrast to scheduling software tasks on single processors.
- *Time Manager*
  This modules 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

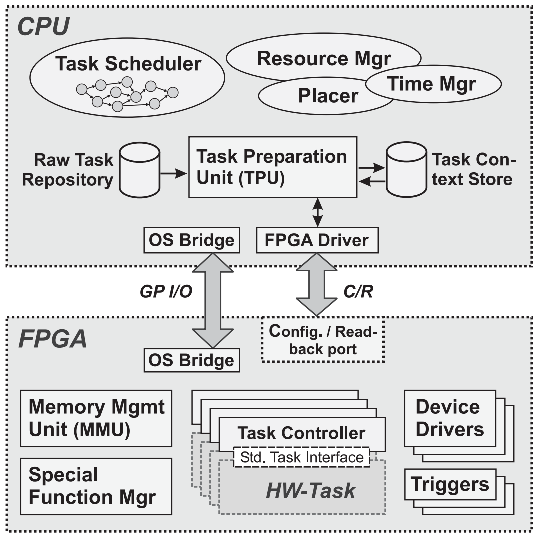

Figure 3: OS Modules running on CPU and FPGA

is generated by the task design flow. Before a raw task can be downloaded to the FPGA, it must be relocated to a specific location in the user area.

- *Context Store*
  This modules 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 readback partial bitstream and extracts the context. *Context insertion* takes a raw task and its previously stored context and generates a partial bitstream. For example, resuming a task in a preemptive scheduling scenario requires to call the context insertion service and, subsequently, the task relocation service. Another service is *parameter insertion* which takes a raw task and inserts a set of parameters at predefined locations. This service allows for load-time customization of tasks.

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

- *OS Bridge*
  Since both CPU and FGPA 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. The OS bridge provides a device-independent command interface. Physically, the communication channel is mapped to the GP I/O. port (see Figure 2).
- *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 FPGAs configuration and readback port.

The list of modules mapped to the FPGA includes:

- *Task Controller*
  The task controller is the operating system's counterpart of the standard task interface and forms the border between OS frame and user area. 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 offer services to tasks, based on hardware functions implemented in the FPGA, such as multiplyers.
- *Device Driver / Trigger*
  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. This is an important issue for a hardware operating system as loading a driver function on demand with the same (rather long) reconfiguration latency as a user task could easily violate the timing constraints.
  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.

3.4 Programming Model and Design Flow

The hardware operating system we discuss in this paper requires that an application is decomposed from 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 the 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, or the maximum size of a task. Our design concept proposes an *OS builder* which is a design flow that generates both the OS frame and the task template for a given target. The OS builder is used in two different ways: to port the hardware operating system to a different target, and to customize the runtime parts of the operating system to an application. The latter use is crucial for many cost-sensitive embedded systems. There, all OS modules and services that are not needed to implement the application are removed to save reconfigurable area.

3.5 Operating System Benchmarking

A design concept for a reconfigurable hardware operating system must include the evaluation of the OS performance. Such an evaluation is necessary to motivate the use of an operating system, to characterize single OS functions, and to compare different operating system implementations. An evaluation consists of a well-defined set of *benchmark* applications together with a description of the benchmarking procedure to derive quantitative data. While in the current state of our work there exists neither the application set nor the benchmarking procedure, we feel that following *quality metrics* should be included:

- *area overheads:* These metrics measure the amount of reconfigurable resources and memory that are needed to implement the runtime modules on the FPGA and the CPU, respectively.
- *runtime overheads:* A number of metrics relate to the times needed for task relocation, context insertion/extraction, task configuration, memory accesses, etc. Another important metric is the interrupt latency, i.e., the time between an external event and the start of the task loaded in response.
- *fragmentation:* The *external* fragmentation denotes the partitioning of free space in the user area which is a result of the resource manager's quality. A low external fragmentation keeps the free space in a few big blocks, whereas a high external fragmentation keeps many small blocks scattered over the user area. A low fragmentation is desirable as it leads to an improved system performance.

  The *internal* fragmentation for a mapped task denotes the difference between the task template area and the actual task area. Uniform task templates greatly simplify placement and scheduling at the cost of a high internal fragmentation. Differently-sized task templates reduce internal fragmentation but require more involved resource managers and schedulers.

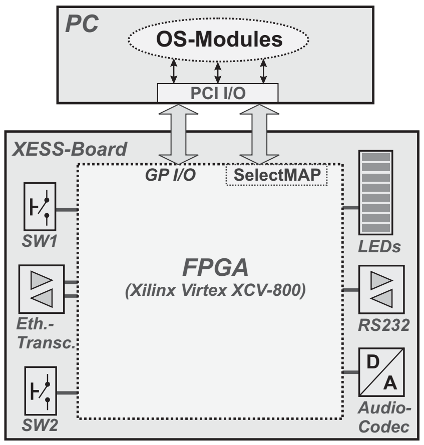

Figure 4: Implementation Platform

4 Implementation Concepts

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

4.1 Target Technology and -Platform

We have chosen the XESS XSV-800 board [12] 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 RAMDAC, display elements, user switches and an audio codec. Figure 4 shows the block diagram of the XESS board and indicates which components we use in our system.

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 FGPA's configuration and readback port (SelectMAP). In parallel, a number of wires implement general purpose I/O between PC and FPGA. The PC substitutes the host CPU, referred in Section 2. In our system, all external components are connected to the FPGA.

Xilinx' *ISE Foundation 5.1* in combination with the *Modular Design* package [13] served as development environment for circuit / bitstream generation; all PC-software was created with *MS Visual C++* using library functions to access the PCI I/O-card.

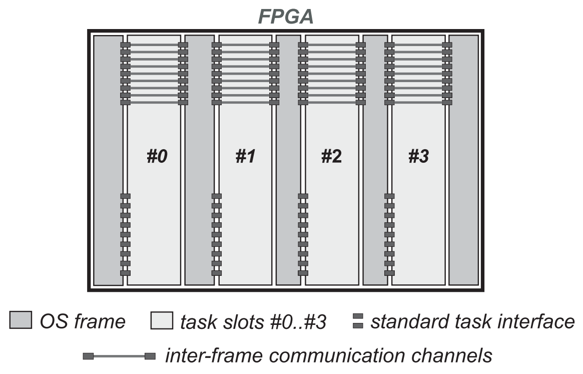

Figure 5: OS frame parts and task slots

4.2 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 3.1, is also done column oriented.

We split the FPGA surface into several vertically ordered *OS-frame* parts and *task slots* (see Figure 5).This structure ensures reconfiguration of task slots, without affecting neighboring OS-frame parts. A number of *inter-frame communication channels (IFCC)* make communication between OS-frame parts possible.

4.3 HW-Tasks and Task Slots

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. The STI forms a convention, that all tasks must implement.
- *Inter-frame communication channels (IFCC):* A number of IFCCs need to pass through tasks to ensure OS-frame connectivity. Tasks must implement these lines.

To speed up the creation of slot-compatible tasks, the OS provides *task templates* with predefined STI und IFCC, as depicted in Figure 6. Consequently, the task development process just comprises in embedding the task's functionality into the *task circuit area*.

The STI as well as IFCC endpoints are realized with location invariant *bus macros*, a new design element of Xilinx' Modular Design package [13].

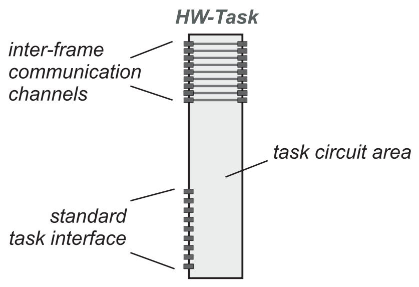

Figure 6: HW-Task / Task Template

4.4 Standard Task Interface

All running tasks are controlled by the OS via thier 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):* After loading the task into a slot, the task circuit needs to be initialized. This is done by applying a logical '1' to this pin.
- *ENA (Enable):* Starts (logical '1') or stops (logical '0') task execution.
- *FIN (Finished):* A task signalizes its completion by applying a logic '1' to this pin, whereupon the OS controller generates an event. This event can cause the schedling module to load the next ready task into the slot.

On the other hand, tasks can make use of services offered by the OS, as for example

- reading from / writing to FIFO buffers and check their states,
- sending to / receiving form device drivers, or
- checking triggers.

The STI implements a number of pins to provide access to these services.

4.5 OS Frame Insides

As a part of the OS-frame, the *OS-Bridge* module implements a synchronous bidirectional communication channel, connecting to the OS modules running on the PC. This channel is used by several OS modules to send and receive commands for the purpose of controlling OS elements. The following 4 commands are available:

- *OSConfig():* Allows for resetting the board, the OS-frame or single OS-elements (as buffers or device drivers), as well as initializing or reparametrizing OS-elements;
- *TS0Config(), TS1Config():* Controls STI pins of the OS-controller of task slot 0, and 1, respectively. In addition, these commands control multiplexers which connect buffers to STIs.

- *ReadState():* Reads the status of OS-elements, e.g. *FIN*-pins of task slots, status information of buffers, device drivers and triggers.

Furthermore, a dedicated interupt line is available to signalize events from FPGA to PC.

The XCV-800 FGPA includes a total amount of 114.7kbit Block RAM, divided in 28 portions (SelectRAM cells). A *Memory Management Unit (MMU)* forms 8 FIFO-buffers out of it and puts them at task's disposal. The depth of these FIFOs ranges between 512 and 2048 8bit words. Tasks can read and write FIFOs by using the STI.

4.6 Device Drivers and Triggers

Device driver circuits are physically connected to external components and perform low-level controlling of the devices. In our system, we implemented one device driver for each component, Ethernet transceiver, RS232 transceiver, audio codec and LED bar.

In contrast, buttons like *SW1* and *SW2* don't need much controlling, hence, no device drivers are needed. Triggers generate events upon pressing and releasing buttons.

All device drivers and triggers mentioned above are implemented in the OS-frame.

5 Case Study

To put our concepts into practice, we have implemented a case study, using the XESS prototyping board [12] and some of its components. The case study executes a control/data-flow application, performing networking and multimedia tasks in an OS environment.

5.1 Functional Description

From a users point of view, the application

- generates an artificial audio signal, when *SW1* is pressed;
- displays some pattern on the LED bar, when *SW2* is pressed;
- receives UDP packets via the Ethernet interface and forwards their payload either to the RS232 interface or to the audio codec (audio streaming, 12kHz, 16bit, mono). Data for the RS232 output can be plain-text or AES encrypted;
- replies to incoming ARP (adress resolution protocol) and ICMP (internet control message protocol) echo requests received via the Ethernet interface.

The incoming IP/UDP packets in our case study are generated and sent by another PC connected to the LAN.

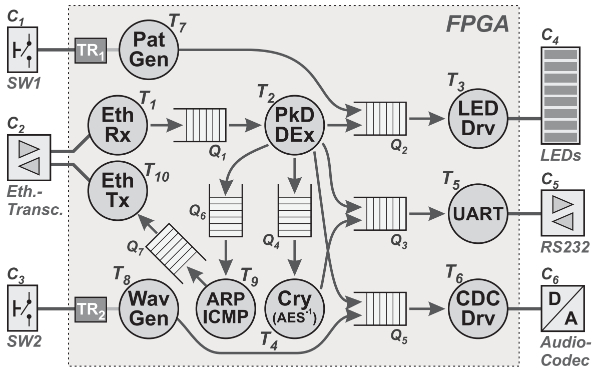

Figure 7: DFG of case study

5.2 Tasks

The tasks T_1 to T_{10} execute the following functions:

- (T_1) *Eth Rx (Ethernet receiver):* Receives packets form Ethernet transceiver; verifies frame checksum; forwards packet to Q_2 if MAC- and IP-adresses match (MAC- and IP-adresses are hard wired inside the task).
- (T_2) *Paket Discriminator / Data Extractor:* Checks if packet is UDP; extracts application data and puts it into the appropriate queue (Q_2, Q_3, Q_4 or Q_6), depending on the UDP port number, which identifies the application.
- (T_3) *LED Driver:* Reads data from Q_2 and displays bit patterns on the LED bar.
- (T_4) *Crypto AES-Decoder:* Decrypts data received from Q_4 and forwards the results to Q_3.
- (T_5) *Asynchronous Serial Driver:* Reads data from Q_3 and generates an asynchronous serial signal, with a baudrate of 115kbit/s.
- (T_6) *Codec Driver:* Gets data from Q_5 and forwards it to the audio codec.
- (T_7) *LED Pattern Generator:* Generates some bit patterns and writes them into Q_2.
- (T_8) *Sound Generator:* Generates a audio waveform and is written into Q_5.
- (T_9) *ARP / ICMP:* Reads ARP / ICMP requests out of Q_6, creates the appropriate reply packet and forwards it to Q_7.
- (T_{10}) *Ethernet Sender:* Reads data from Q_7, puts Ethernet packets together (frame header, payload and checksum) and sends them using the Ethernet transceiver.

Since T_1, T_3, T_5, T_6 and T_{10} directly connect to I/O devices and implement time-critical tasks, they are mapped into the OS frame and act as device drivers. However, T_2, T_4, T_7, T_8 and T_9 perform non time-critical tasks and can be loaded into slot S_0 or S_1, as soon as the scheduler receives a corresponding event, that requests the task. Events are generated either by FIFO buffers Q_1 to Q_7 or triggers TR_1 or TR_2.

Priorities of the loadable tasks descend in the order: T_7, T_8, T_2, T_4, T_9.

The scheduler follows a non-preemptive, priority based policy to determine the next task to be loaded.

5.3 OS Disposition

A Xilinx Virtex XCV-800 devices includes an array built of 84 columns and 56 rows of configurable logic blocks (CLBs). For this case study, we created an OS frame, consisting of 3 OS-frame parts, OS_{left}, OS_{center} and OS_{right}) and two task slots S_0 and S_1. The widths of the three OS-frame parts amount to 8, 10, 8 CLB columns, the task slots have a width of 28 CLB columns each. 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.

Each standard task interface (STI) consumes 12 bus macros (=48 pins).

A total amount 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%.

5.4 Observations and Measurements

The case study application works well. However, since there are only two task slots available, but as much as 5 tasks are loadable into slots, not all functions can run at the same time. In addition, the audio-streaming part needs some improvements in data flow control. Queue underruns in Q_5 interrupting the audio stream occur quite often.

The file length 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).

Task activation time, including download phase and reset/enable commands, amounts to about 290ms for small tasks, and up to 510ms for large tasks (AES). The PCI I/O card turned out to be the bottleneck in the configuration path.

6 Conclusion and Further Work

In this paper we used a top-down approach to discuss issues in reconfigurable hardware operating systems. We started with a design concept that describes operating system abstractions and services in a device-independent way. Then we refined the design concept to an implementation concept based on the Xilinx Virtex XCV800 technology. We showed how the entities of the design concept map to this specific technology. Finally, we presented a prototype running a networking and multimedia application. Our experience with this prototype demonstrates the feasibility of hardware multitasking and the concept of hardware operating systems in general.

Further work will include the application of the implementation concept to larger FPGAs, the creation of OS frames with more than two task slots, and the implementation and experiments with differently-sized task slots, as reported in [10].

7 Acknowledgements

This work was supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99. We would further like to thank Andres Erni, Stefan Reichmuth [4] and Michael Ruppen [8] for their valuable work on getting the case study running.

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] A. Erni and S. Reichmuth. Inter-Task-Communication in Reconfigurable Hardware OS. Master's thesis, ETH Zurich, 2003.

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

[6] 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.

[7] 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.

[8] M. Ruppen. Reconfigurable OS Prototype. Master's thesis, ETH Zurich, 2003.

[9] 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.

[10] 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.

[11] 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.

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

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