Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2016, Article ID 2478907, 11 pages
http://dx.doi.org/10.1155/2016/2478907

Review Article

Operating System Concepts for Reconfigurable Computing: Review and Survey

Marcel Eckert, Dominik Meyer, Jan Haase, and Bernd Klauer

Faculty of Electrical Engineering, Helmut Schmidt University, Hamburg, Germany

Correspondence should be addressed to Marcel Eckert; eckert@hsu-hh.de

Received 7 September 2016; Revised 4 November 2016; Accepted 7 November 2016

Academic Editor: Michael Hübner

One of the key future challenges for reconfigurable computing is to enable higher design productivity and a more easy way to use reconfigurable computing systems for users that are unfamiliar with the underlying concepts. One way of doing this is to provide standardization and abstraction, usually supported and enforced by an operating system. This article gives historical review and a summary on ideas and key concepts to include reconfigurable computing aspects in operating systems. The article also presents an overview on published and available operating systems targeting the area of reconfigurable computing. The purpose of this article is to identify and summarize common patterns among those systems that can be seen as de facto standard. Furthermore, open problems, not covered by these already available systems, are identified.

1. Introduction

Reconfigurable computing is a rapidly emerging computing paradigm, not only in research but yet also in real applications. This is indicated by the following facts: Intel recently bought Altera, several companies including AMD, ARM, Huawei, IBM, Mellanox, Qualcomm, and Xilinx have joined into consortiums to define *Cache Coherent Interconnect for Accelerators* (CCIX [1]) or to harmonize the industry around heterogeneous computing (HSA Foundation [2]).

Reconfigurable computing allows introducing new flexible computing architectures but also contains some challenges. The first of all is the productivity gap: designing problem specific hardware is something different than writing software, targeting a conventional computer architecture. The differences arise from the underlying methodologies and the required time to perform a design iteration step; software is compiled very fast whereas hardware synthesis is a very time consuming process.

This productivity gap hinders reconfigurable computing to spread faster. Fortunately, several approaches are emerging to tackle the gap: *High-Level-Synthesis* is an approach to raise the level of abstraction, at which hardware is designed. It is comparable with the paradigm shift in software design, when the first high level languages were introduced to overcome the task of implementing software entirely in assembler.

Another approach to overcome the productivity gap is standardization. By providing a well-defined set of interfaces and protocols, it will be easier to enable more developers to design hardware against these unified models. Standardization can be done at different levels; the lowest one is defining a standardized set of interfaces and protocols at the hardware level (see CCIX consortium [1]). Another important level is the operating system level. By providing standardized interfaces at the level of the operating system, a unified view is made available not only for developers but also for the users. The user does not need to care about the details; everything is just working out of the box.

Related to reconfigurable computing, the idea of integrating reconfigurable components into the operating system is as nearly as old as reconfigurable computing itself, starting at the end of the 1990s. This article will provide an overview on the different aspects, needed to care about, when integrating reconfigurable computing into an operating system. Furthermore, this article will also present an overview and a summary on different operating system implementations that have been published in the past. The purpose of this retrospective analysis is to understand common patterns currently used and identify open or remaining problems that need further or maybe renewed research interest. Therefore a detailed analysis and benchmark driven comparison of the presented system are not within the scope of this article.

2. Reconfigurable Computing: Challenges for Operating Systems

According to [3], *one of the major tasks of an operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, and consistent abstractions to work with instead*. In other words, the two main tasks of an operating system are *abstraction* and *resource management*.

Abstraction is a powerful mechanism to handle complex and different tasks (hardware) in a well-defined and common fashion. One of the most elementary OS abstractions is a process. A *process* is a running application that has the perception (provided by the OS) that it is running on its own on the underlying virtual hardware. This can be relaxed by the concept of *threads*, allowing different tasks to run concurrently on this virtual hardware to exploit task level parallelism. To allow different processes and threads to coordinate their work, *communication* and *synchronization* methods have to be provided by the OS.

In addition to abstraction, *resource management* of the underlying hardware components is necessary because the virtual computers, provided to the processes and threads by the operating system, need to share available physical resources (processors, memory, and devices) spatially and temporarily.

With the introduction of run-time reconfigurable FPGAs in the late 1990s, the idea of integrating and supporting them in operating systems was developed and rapidly gained research interest. One of the first papers is [4] (1996): *The overall intention is to make the FPGA hardware available as an extra resource to operating systems. In doing so, the FPGA hardware can be viewed variously as having the nature of processors, memory or input/output devices, or some combination of these.* The above-mentioned paper also stated two general usage scenarios for reconfigurable logic, which are in use till today:

(i) Usage as accelerator to support a general purpose processor by performing computational functions (sea of accelerators).

(ii) Usage as cooperating parallel processing elements, interacting with their neighbors (parallel harness model).

For these usage models, Wigley and Kearney ([5] (2001) and [6] (2002)) identified key OS services to support reconfigurable computing within an operating system:

(a) Application loading

(b) Partitioning

(c) Memory management

(d) Scheduling

(e) Protection and I/O

As time evolved and thereby size and speed of FPGAs increased an additional usage model was proposed (around mid-2000s):

(iii) Usage as additional but independent processing element that is capable of not only performing computational functions, but also executing usage of the FPGAs' own tasks.

The above identified key OS services are also required for this usage model, but another was identified to be important, especially for calculation performance reasons:

(f) Communication and synchronization

The remainder of this section presents the different ideas and proposal for integrating reconfigurable computing into operating systems according to the above OS services related taxonomy.

2.1. Partitioning. The problem of partitioning reconfigurable logic can be solved as a resource management task inside the operating system. FPGAs consist of numerous small configurable blocks (so called *Configurable Logic Block (CLB)*, to implement logic functions and registers) and a configurable routing network between them. Nowadays, FPGAs also contain dedicated functional elements (DSPs, BlockRAM) to save *CLB*s to instantiate such frequently used elements. In the following, *CLB*s, BlockRAMs, DSPs are summarized as *functional resources* and the configurable routing network elements are summarized as *routing resources*. At the operating systems level, reconfigurable logic is usually not managed on the fine grained level of CLBs. At operating system level *reconfigurable areas* are managed at the granularity level of *reconfigurable modules*. The reconfigurable areas are constituted by a number of adjacent functional and associated routing resources. *Instantiating* a reconfigurable module inside a reconfigurable area means configuring the functional and routing resources so that the reconfigurable area provides the functionality of the reconfigure module. Different models, how those reconfigurable areas are manageable by an operating system and how reconfigurable modules can be instantiated on them, were proposed. Care has to be taken on terminology when comparing different systems. Different systems of different authors sometimes use the same terms for different architecture styles which might lead to misunderstanding. For the reason the following definitions are given (based on [7]) and used within this article.

As a first solution the reconfigurable logic is divided into several reconfigurable areas, called *islands*. A reconfigurable module can be instantiated inside such a reconfigurable area. It is not intended to allocate one reconfigurable module across more than one reconfigurable area.

This solution is commonly used in research proposals today because it is supported by partial reconfiguration design flows of commercial tools. The number of reconfigurable resources inside a reconfigurable area has to satisfy the footprint of the "largest" reconfigurable module that might be instantiated in it. Based on this worst case dimensioning of the reconfigurable area, for all "smaller" reconfigurable modules there will be unused reconfigurable resources. This is called *internal fragmentation* and it is one of the main disadvantages of the "Island" solution.

The islands solution can be applied in several ways, first by taking advantage of dynamic partial reconfiguration or second by using several FPGAs, where each FPGA represents an entire island [8].

In opposition to the island solution, *1D slots and 2D grids architectures* were proposed. In these solutions, reconfigurable areas are placed on a one-dimensional array of the so called slots or a two-dimensional grid. Reconfigurable modules can be instantiated over adjacent reconfigurable areas according to their reconfigurable resources requirements. This solution helps to reduce the effects of internal fragmentation but requires defragmentation techniques based on relocation to reduce *external fragmentation*.

The 1D slot solution (discussed in [7, 9, 10]) is not natively integrated in conventional commercial synthesis tools. However, as most FPGAs are configured on a column or row based mechanism, it is possible to adopt those synthesis tools for 1D slot solutions, as available partial reconfiguration design flows generate configuration files for columns of an FPGA. The relocation of reconfigurable modules remains a problem.

The 2D grid solution was examined and discussed in several early publications, [5, 11, 12] (around 2000). However, as this approach is not supported by commercial design flows for FPGAs, the islands style solution is dominant in current publications. Nevertheless, "newer" publications based on the 2D grid solutions can be found occasionally ([13] (2008) and [14] (2011)).

2.2. Application Loading/Abstraction. The intention behind *reconfigurable computing* is to use reconfigurable resources to gain an increase in computational performance, reduce power consumption, or minimize hardware costs. These gains are achieved by transferring parts of or even an entire application from a software solution to a dedicated hardware solution. For the operating system, it is necessary to have an internal representation for these hardware supported functionalities that allows controlling them and also provides the possibility of integrating/using them in conventional software applications.

If the reconfigurable module acts as a slave of the main processing units (CPUs) of an architecture and therefore is fully dependent and controlled by a CPU it is called an *accelerator unit*. The usage of accelerator units is a common way of using reconfigurable logic to improve applications computational performance and/or power consumption today and is supported by toolchains of commercial FPGA vendors. Inside the operating system, the accelerator unit is controlled like a conventional device with the abstraction mechanisms of a device driver. Several publications exist on how to define a standardized interface for those accelerator units and how they can be handled in a standardized way by providing libraries, for example, HybridOS (2007) [15] and LEAP (2011) [16, 17].

If the reconfigurable module acts as processor equivalent which is interacting with the CPUs by means of communication and synchronization it is called *hw-application*. The usage of hw-applications is a hot topic in current research issues regarding operating systems and reconfigurable computing. Depending on the application area and memory coupling mechanisms, the terms hw-task, hw-process, and hw-thread can be found in publications, emphasizing the application area or method. The term *hw-task* can be found in several real time oriented proposals. If hw-application is strictly separated from its software counterparts and communication is usually implemented by message passing mechanisms it is called *hw-process* (BORPH (2007) [8]). On the contrary, hw-application is called *hw-thread*, if the hw-application is memory coupled and directly interacts with its software counterparts. This requires sophisticated communication and synchronization methods. Several publications using this abstraction for hw-applications extend the well-known *POSIX thread* model.

As the hw-application solution was and still is in focus of current reconfigurable computing operating systems research publications, the remainder of this article will focus on this solution.

2.3. Placement, Scheduling, and Preemption. In conventional operating systems, a task (or process or thread) is subject to scheduling and placement. This task involves the questions: when (scheduling) and where (placement) is a task executed? Can a started task be preempted by the operating system? is a schedule calculated at system creation time or can it be adapted at run-time?

In the scope of reconfigurable computing, *placement* answers the question, onto which reconfigurable area(s) hw-application is configured. Depending on the underlying architecture (islands or slots/grids) the possibilities differ. For island based architectures, the only important aspect is as follows: does the hw-application fit into all or only some reconfigurable areas and which of those areas are available (not occupied by other hw-applications)? For slots/grids based architecture another degree of freedom is available: As the hw-application can cover some adjacent reconfigurable areas, a rearrangement of the new hw-application is needed and/or the currently configured hw-applications need to be replaced and rerouted, due to external fragmentation issues. Examples for placement and scheduling strategies targeting slot/grid based architectures can be found in Bazargan et al. (2000) [18], Teich et al. (2001) [19], Steiger et al. [20], Pellizzoni and Caccamo (2007) [21], Koch et al. (2009) [22], and Ahmadinia et al. (2010) [23]. However most of these works either are theoretical or evaluated their proposals by simulation only.

The question when hw-application is configured into the reconfigurable area(s) is subject of the operating system task *scheduling*. An important aspect to consider here is to take into account the required reconfiguration time. It might be beneficial to delay the instantiation of hw-application: an island might be available in the future, due to another hw-application finishing its execution and the afterwards available island allows reducing internal fragmentation. In other words, in the future, an island might be available, where the needed hw-application (resource footprint) "fits" better in terms of unused reconfigurable resources.

The above discussion focused on executing hw-application in hw. However, some publications propose the idea to allow an application to be executed either as sw on a CPU or in hw. The idea is to extend the executable file format to even hold configuration data and give the scheduler the freedom to execute an application in HW or SW. OS4RS, BORPH, CAP-OS, and RTSM are examples implementing this approach.

The first operating systems for conventional computer architectures implemented a *single-tasking* strategy. Once a single task was started, it held the processing element (CPU) until it finished. As an enhancement, *multitasking* was developed to allow the temporal sharing of the processor, first by *cooperative multitasking* where a task has to release the processor voluntarily and finally by *preemptive multitasking*, where a task is forced to release the processor after a given time, coordinated by the schedule of the operating system. In a multitasking environment, when releasing and assigning a task from the processor, a context switch occurs.

A reconfigurable area can, in theory, also be used by hw-applications in those three ways. To allow multitasking, the "context" of hw-applications needs to be extracted, saved, and restored. The context of a software application is given by the contents of its memory and processor registers. For hw-application the context is given by its memory contents and all storing elements (flip-flops, block RAMs, and FIFOs) of the occupied reconfigurable area(s). As extracting the current content of all memory elements of a reconfigurable area is not that easy on commercial FPGAs, the single-tasking approach is commonly used. However, some publications provide a dedicated mechanism for "context" extraction and restoration of their reconfigurable modules to allow multitasking (cooperative (ReconOS [24], OS4RS [25]) or preemptive (RTSM [26])) approaches. Further discussions of details on possible hw-context switching strategies can be found in [27, 28].

A schedule can be calculated statically at compiling/synthesis time. This is only applicable, when the number, execution, and reconfiguration time and occurrence of hw-applications are known before execution, which might be given for real time applications. A more dynamic scheduling is also possible and several hw-application scheduling strategies have been summarized, compared, and evaluated in [29, 30].

2.4. Communication and Synchronization. In conventional operating systems the applications (tasks, processes, and threads) can communicate with each other by means of shared memory or message passing methods. However, message passing is an operating system abstraction usually implemented on the basis of shared memory, where the operating system handles synchronization issues for the applications.

When operating systems have to support reconfigurable computing things get more complicated. Communication between sw- and hw-applications has to be provided, and also dedicated hw-to-hw-application communication might be required or is beneficial for performance, because it bypasses a time consuming hw-to-sw-to-hw communication.

How communication is enabled mainly depends on the underlying architecture (bus based versus NoC-based, one channel for all or dedicated point-to-point channels, message passing based versus shared memory based). However the operating system is in charge of managing and controlling those communication possibilities and providing sw-developers with unified abstraction.

Furthermore, synchronization mechanisms need to be extended onto reconfigurable hw. This requires providing mutexes or semaphores in hardware.

To provide a standardized communication and synchronization scheme is a requirement for higher productivity at design phase and computational performance at run-time. Therefore, several publications focus on this topic, without providing full operating system capabilities. Hence, some of these frameworks are summarized in Section 3.2.

2.5. Protection and I/O. Operating systems provide mechanisms to separate applications against each other, so that applications cannot influence others or even the operating system itself. This is supported by the hardware, firstly by implementing different privilege levels and secondly by memory management units. In conventional architectures the memory management protection mechanism is extended to I/O with IOMMUs. The reason is the possibility of an *intelligent* I/O device (embedded systems inside another system) to access main memory by means of DMA and therefore corrupt the memory contents of an application or the OS. Following the same argumentation, the MMU mechanism has also to be applied to hw-applications.

Another important aspect related to protection is malware. sw-applications can be infected by malware. In reconfigurable computing, malware can also be part of hw-application. This problem has already been identified by Wigley and Kearney in 2002 [6]. However, only little research effort has been spent on this issue in the past.

3. Proposed and Published Systems

In the following several frameworks and operating system extensions targeting issues for reconfigurable computing (as discussed in the previous section) are presented. For each system, the underlying hardware architecture and used/proposed operating system concept is given. A detailed comparison of the systems is beyond the scope of this article. However, it is highlighted in the systems short summary when one of the systems references provides a qualitative comparison of selected systems.

3.1. Early Ancestors. The systems discussed in this subsection present some of the first implementations and ideas on operating system support for reconfigurable computing.

3.1.1. Brebner (1996). Based on Xilinx XC6200 FPGA Brebner [4] was one of the first to present the idea of an operating system supporting reconfigurable computing. Brebner coined the term "virtual hardware" to compare the exchange possibilities of reconfigurable hardware with "virtual memory." Therefore he also named reconfigurable areas as *Swappable Logic Unit (SLU)* and identified the allocation of the required *SLU*s as the operating systems main responsibility.

3.1.2. Compton et al. (2000). Compton et al. [11, 31] provided ideas on 2D grid partitioning including reallocation and defragmentation targeting Xilinx XC6200 FPGA. Compton also discussed several scheduling algorithms to reduce the reconfiguration time overhead.

3.1.3. OS4RC (2001). Diessel, Wigley, and Kearny (University of South Australia) published several articles on operating systems for reconfigurable computing [5, 6, 12, 32, 33]. In the later ones, they mention developing *operating system for reconfigurable computing (OS4RC)*. Unfortunately, only the intention to implement this operating system is extensively discussed and simulated ideas can be found.

3.2. Accelerator API Frameworks. In this section, some frameworks are presented that define standardized interface and/or provide APIs to provide convenient access to reconfigurable computing to the software developer.

3.2.1. HybridOS (2007). HybridOS was developed at University of Illinois (USA). *HybridOS specifically targets the application integration, data movement, and communication overheads for a CPU/accelerator model when running a commodity operating system* [15]. HybridOS extends Linux with a number of standardized APIs and therefore provides *accelerator virtualization*. The discussed mechanisms to interact with the accelerator hardware are character device driver access, DMA access method, and accelerator direct mapping [34]. The proposed access mechanisms are evaluated against each other, based on an example application for JPEG-compression.

3.2.2. LEAP (2011). Latency-intensitive Environment for Application Programming (LEAP) was developed in corporation of Intel and the Massachusetts Institute of Technology. It is presented in [16, 17].

LEAP provides both basic device abstractions for FPGAs and a collection of standard I/O and memory management services. LEAP's fundamental abstraction is communication. By combining communications and memory primitives with an extensible compilation infrastructure, LEAP is able to provide libraries that rival software in their scope and simplicity [17]. Leap builds upon the theory of a latency insensitive design. *The key idea behind latency insensitive design is separating communication and computation. Communication is governed by an abstract protocol, whose main characteristic is to be insensitive to latencies from the underlying channel [35].*

In summary, the standardized design elements of LEAP are intended to increase design productivity.

3.2.3. RIFFA (2012). Reusable Integration Framework For FPGA Accelerators (RIFFA) was developed at University of California (USA) and its first version is presented in [36]. A second version, removing restraints of the initial version, is presented in [37].

RIFFA provides communication and synchronization for FPGA accelerated applications using simple and self-defined C/C++ interfaces serving as a standardized API for both hardware and software.

3.3. Applications in Hardware: Operating Systems. In the following, several proposed operating systems targeting reconfigurable computing are presented on their own by a short summary. Relevant references for the dedicated systems are also included. As opposed to the previous section these systems are true operating systems and not only API frameworks. Finally, Table 1 gives a summary of these operating systems and a discussion of the implemented concepts is given at the end of this section.

3.3.1. OS4RS (2003). The most recent publications on *operating system for reconfigurable systems (OS4RS)* are Mignolet et al. [25] and Nollet et al. [38] and were developed at KU Leuven (Belgium). *OS4RS* extends a real time operating system extension for Linux (RTAI). The operating system extensions are targeting an equally sized multi-island reconfigurable architecture (a reconfigurable island is called *tile* by the authors), which additionally provide communication channels among those islands based on a *Network-on-Chip (NoC)*.

OS4RS allows dynamically scheduling tasks to a general purpose processor (ISP-instruction set architecture processor) or a reconfigurable island. The authors propose a 2-level scheduler and also present the possibility of relocating tasks, based on a check-pointing mechanism. The *OS4RS* authors extend the format of executable files to allow the execution of a task both on a general purpose processor and on reconfigurable hardware.

Evaluation is done on the basis of video-processing examples.

3.3.2. HThreads (2005). HThreads was developed at the University of Kansas (USA). The initial ideas are described in [39, 40]. Implementation details are given in [41, 42]. Several further publications not given here present individual aspects of HThreads in more detail. According to the authors, *HThreads is a computational architecture which aims to bridge the gap between regular programmers and powerful but complex reconfigurable devices.* It therefore allows compiling C-code to hw-thread.

The developed operating system targets real time applications and provides a standardized API for accessing hw-applications. This API is based on a thread model. The underlying hardware provides static hw-threads, connected to the system bus. So HThreads does not use dynamic and partial reconfiguration. Communication is based on a memory mapped register interface. The HThreads authors identified three major tasks for their operating system: management, scheduling, and synchronization. HThreads implements all of them in hardware. The operating system implements preemptive priority, round robin, and FIFP scheduling algorithms [41]; it also provides hw-based mutexes and semaphores.

3.3.3. BORPH (2007). Berkley Operating system for ReProgrammable Hardware (BORPH) was developed at University of California at Berkeley (USA). It was presented as dissertation [8].

BORPH builds upon an island style architecture, where each island is represented by an entire FPGA. In *BORPH*

Table 1: Operating systems for reconfigurable computing and their implemented OS aspects.

	Architecture			Scheduling						
	Reconfigurable area style	Interconnection	Application abstraction	Schedule creation	hw-sw-interchangeable	Preemption	Real time focus	Communication/synchronization	Example/benchmark applications	Note
OS4RS (2003)	Islands	NoC	Task	Run-time	x	Check-pointing	x	NoC-based		
HThreads (2005)	Static islands	Bus	PThread	Design time	—	—	x	Memory mapped		
BORPH (2007)	Islands (FPGA)	P2P	Unix-process	Run-time	x	—	—	Message passing (OS kernel)	Signal and video processing	Linux extension extends executable format
ReconOS (2007)	Islands	Bus	Delegated PThread	Run-time	—	x	x	x	Video object tracking	Linux support
CAP-OS (2010)	Islands	NoC	Task	Run-time	x	—	x	NoC-based	Image processing	
R3TOS (2010)	Grid	NoC	PThread	Run-time	—	—	x	NoC-based	SDR, JPEG-compression	Focus on reliability
FUSE (2011)	Islands	Bus	PThread	Run-time	—	—	—	Device driver mechanisms	Image processing, encryption	Loadable kernel modules (no need for a standard HW-IF)
SPREAD (2012)	Islands	Bus + P2P	Delegated PThread	Run-time	x	x	—	Bus based + P2P	De-/encryption	
RTSM (2015)	Islands (inhomogeneous)	Bus	Delegated PThread	Run-time	x	x	x	Bus based, Vivado HLS IF	Image processing	

hw-application follows the Unix-process model and therefore no implicit shared memory among software-processes and/or hardware-processes exists. As most of the other reconfigurable computing operating systems it is a Linux extension. *BORPH* OS kernel provides standard Unix services, such as file system access, to the hardware-processes.

Application loading is implemented on the basis of a new executable file format (BOF: *BORPH* object file). A BOF extends the conventional .elf format by adding the required configuration information for the possible reconfigurable areas it can be executed on. *BORPH* therefore provides the option to choose where to execute a process (allocation on a conventional processor or in one of the available reconfigurable areas.) However, *BORPH* does not provide a mechanism to swap (preempt) a running hw-process. sw-to-hw communication is enabled by a dedicated *hybrid message passing system call interface*. Example applications include signal and video processing.

3.3.4. ReconOS (2007). ReconOS has been developed at University of Paderborn (Germany) and is presented in [43]. Despite the fact that its first version was already presented in 2007 it is still under development as version 3 is presented in [24, 44] (2014).

ReconOS (version 1) builds on top of eCos, a widely used real time operating system. Beginning with version 2 of ReconOS, Linux support is also available. The underlying architecture is island style based, where CPU and reconfigurable areas (called *slots* in ReconOS terminology) are interconnected via a bus. One of the latest publications [24] introduced an ReconOS extension that uses a 2D grid style architecture. Each hw-thread is handled via a standardized hw-interface (OSIF: operating system interface) at physical level and by usage of a *delegate* sw-thread in the operating system. ReconOS does not require any change to the host OS. 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 programmers point of view, however, the delegate threads are hidden by the ReconOS run-time environment, and only the applications hardware and software threads exist. ReconOS includes hw-to-sw and hw-to-hw communication and synchronization.

As presented in a recent publication [45], ReconOS also implements a preemptive multitasking mechanism. Schedule creation is dynamically done and adoptable at run-time in ReconOS.

In [44] a video object tracking application is presented as evaluation example for ReconOS.

3.3.5. CAP-OS (2010). Configuration Access Port Operating System (CAP-OS) was developed by Fraunhofer IOSB (Germany) and Karlsruhe Institute of Technology (Germany) and is presented in several publications by Goehringer et al. [46–49]. The *CAP-OS* is used for run-time scheduling, task mapping, and resource management on a *Run-time Adaptive Multiprocessor System-on-Chip (RAMPSoC)* [50]. Task graphs and partial bitstreams (island style based architecture with an interconnecting *NoC*) are created at design time for a *RAMPSoC*. *CAP-OS* and *RAMPSoC* are targeting real time applications and try to ease the design tool flow. *CAP-OS* thereby hides the complexity of the underlying *RAMPSoC*. The scheduling algorithm included in *CAP-OS* takes into account the time required for reconfiguring a module and real time demands of the tasks. CAP-OS key functionalities are run-time scheduling of the tasks, resource allocation, and configuration management. These functionalities are realized based on the task graph and the corresponding partial bitstreams, generated at design time with the software toolchain. The hw-tasks are not preemptive. hw-to-hw and hw-to-sw communication is managed by *CAP-OS* on the basis of the *NoC* provided by the underlying *RAMPSoC*. A task can be executed either in software on a processor or in hardware as a hardware accelerator.

Evaluation is based on image processing tasks.

3.3.6. R3TOS (2010). Reliable Reconfigurable Real Time Operating System (R3TOS) was developed in cooperation of the University of Edinburgh (UK) and the IKERLAN-IK4 Research Alliance (Spain). As the name implies this reconfigurable computing operating system is targeting real time applications with focus on reliability. *R3TOS* idea is briefly presented in [51] and the resulting implementation is examined in [52] in detail. Task definitions and their interactions are described using parallel software programming syntax (e.g., POSIX threads), but the body of some of the tasks (hardware tasks) is implemented in hardware. The underlying architecture is based on a 2D grid. Grid tiles are named *computation regions* which are interconnected by a NoC to allow intertask communication and synchronization. Reliability is achieved by identifying defective hardware regions and rearranging affected reconfigurable modules so that they do not further use these malfunctional regions. *R3TOS* uses a nonpreemptive *earliest deadline first (EDF)* policy to schedule, named finishing-aware EDF (FAEDF), and takes into account required reconfiguration times.

The proposed operating system is evaluated with a Software Defined Radio application for video transmitting (JPEG-compression).

3.3.7. FUSE (2011). Front-end USEr framework (FUSE) was developed at Simon Fraser University in Burnaby (Canada) and is presented in [53]. *FUSE* uses an API based on the POSIX thread standard and integrates it with PetaLinux OS. It is based on an island style architecture. The reconfigurable areas are attached to the system bus. According to the authors, *FUSE objective is to provide a framework for OS abstraction of the underlying architectural configuration to run hardware tasks, as opposed to a new scheduling algorithm*. The OS abstraction is provided by applying a loadable kernel module with each reconfigurable module which provides a standardized sw-interface to the corresponding sw-task which interacts with the hw-accelerator. Therefore, in *FUSE* there is no need for a standardized interface for the accelerators at the hardware level.

Example applications for evaluation of *FUSE* are JPEG-compression, 3DES encoding/decoding, and image filtering (Sobel based edge detection).

3.3.8. SPREAD (2012). SPREAD is based on an island style architecture with a well-defined interface at the hw-level. It is presented in [30, 54]. The reconfigurable areas are connected to the system bus, but also have DMA channels. The well-defined interface not only covers control but also communication (dedicated data-streams between reconfigurable areas are available), as the system is targeting streaming applications. It adopts the PThreads model and uses sw-thread counterparts (stubs) for managing the hw-task inside the operating system. SPREAD also allows for switching a thread between its hw and sw implementation at run-time and therefore implements hw-context switching mechanisms. Evaluation is done on the basis of several crypto algorithms (AES, DES, and 3DES) and compares FUSE, BORPH, ReconOS, and SPREAD.

3.3.9. RTSM (2015). Run-Time System Manager (RTSM) was developed at the Technical University of Crete (Greece) and is presented in [26]. The RTSM manages physical resources employing scheduling and placement algorithms to select the appropriate hw processing element (PE), that is, a reconfigurable area, to load and execute a particular hw-task, or to activate a software-processing element (CPU) for executing the SW version of a task. hw-tasks are implemented as reconfigurable modules, stored in a bitstream repository.

RTSM is based on an differently sized island style architecture. The reconfigurable areas are connected to the system bus. *RTSM* allows relocating hw-task among the reconfigurable areas by context switching techniques. The proposed scheduling algorithm allows including tasks with deadlines and tries to minimize internal fragmentation. A further approach for minimizing internal fragmentation is to put more than one reconfigurable module into a reconfigurable area.

Evaluation is based on an edge detection algorithm.

3.4. Discussion. Table 1 gives a summary of the above presented operating systems targeting reconfigurable computing.

Roughly all of the presented systems use islands style based hardware architecture. The only exception is *R3TOS* which uses a 2D grid style architecture. The interconnection of the hw-applications is based on either *NoC* or bus structure. One exception is *BORPH* which uses a P2P connection scheme, due to the fact that its islands are separate FPGAs. Another exception is *RTSM* which primarily uses a bus structure for control but also provides dedicated streaming communication channels between the hw-applications to speed up streaming applications.

With the exception of *BORPH*, which uses a process abstraction for the hw-applications, all other presented systems rely on a PThread-based abstraction or, when focusing on real time applications, a task (*OS4RS* and *CAP-OS*) abstraction. Some of the systems allow the operating system to either start an application as sw or instantiate and execute it in hw. Preemptive multitasking is only fully supported and implemented by *SPREAD*, *RTSM*, and recently *ReconOS*. However, *OS4RS* provides some kind of cooperative multitasking.

Unfortunately, the task *resource management* was discussed mostly theoretically and experimentally by simulation (see Early Ancestors). The reason for this is simple: current design flows only allow easily defining island style architectures (partitioning problem). Furthermore, sophisticated mechanisms to suspend and resume hw-applications in form of exchangeable/preemptive hardware are very hard to implement (scheduling/placement problem). The resource management therefore simply decreases to the following question for the available operating systems: "Is there an unused island? If one is available, then occupy it till the application finishes."

Managing the available reconfigurable areas inside the operating system is an argument for taking advantage of dynamic and partial reconfiguration, as the 2D grid partitioning solution would allow minimizing internal and external fragmentation. This will only work efficiently by the usage of dynamic and partial reconfiguration.

According to the presented systems, a limited number of typical benchmark applications exist: image and video processing, data encryption and decryption, and data compression and decompression. What is still missing is some kind of a standardized benchmarking suite to evaluate the reconfigurable computing operating systems and their underlying architecture in a more comparable way. Currently, the dedicated evaluations focus on the targeted application area of the proposed systems.

One important mechanism of operating systems is security. Within the current and passed research proposals security issues were not discussed. The reason for this might be the focus of the presented systems, embedded real time systems. In this domain, security is not that big issue compared to system targeting more general purpose computing areas. However, more efforts have to be spent into security problems for reconfigurable computing, as the hardware itself now also can be subject to infection with malware.

Another still open problem is standardization, which would allow for a faster development and a better portability and (re)usability of hardware applications. However, as the big industrial players are starting to cooperate in defining standards (CCIX [1] or HSA Foundation [2]) we might see enhancements in the near future.

4. Conclusion

In this article we presented a summary of ideas to integrate reconfigurable computing into an operating system. Furthermore, several implemented systems are presented. Based on these systems summary and discussion on the implemented concepts are given.

Several common patterns are identified. hw-applications usually use a PThread-based abstraction model; the hw-applications themselves are presented as delegate (sw-)threads inside operating system; preemptive multitasking is used by the newest systems; partitioning is usually implemented on top of islands style based architecture; typical benchmarks include image and video processing, data encryption and decryption, and data compression and decompression.

However, there is still room for improvement, especially to exploit the possibilities of dynamic and partial reconfiguration. Furthermore some concepts, like security, were rarely discussed/investigated in the past but should gain more interest in the future.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] http://www.ccixconsortium.com/.

[2] http://www.hsafoundation.com/.

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, Prentice Hall Press, Englewood Cliffs, NJ, USA, 2014.

[4] G. Brebner, "A virtual hardware operating system for the xilinx xc6200," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, R. Hartenstein and M. Glesner, Eds., vol. 1142 of *Lecture Notes in Computer Science*, pp. 327–336, Springer, Berlin, Germany, 1996.

[5] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 249–250, IEEE, Rohnert Park, Calif, USA, March 2001.

[6] G. B. Wigley and D. A. Kearney, "Research issues in operating systems for reconfigurable computing," in *Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA '02)*, 2002.

[7] D. Koch, C. Beckhoff, and J. Teich, "Recobus-builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 119–124, IEEE, Heidelberg, Germany, September 2008.

[8] H. So and B. University of California, *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*, University of California, Berkeley, Calif, USA, 2007, https://books.google.de/books?id=A3jVa48owLgC.

[9] J. Angermeier and J. Teich, "Heuristics for scheduling reconfigurable devices with consideration of reconfiguration overheads," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–8, April 2008.

[10] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A bus-based SoC architecture for flexible module placement on reconfigurable FPGAs," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 234–239, Milano, Italy, September 2010.

[11] K. Compton, J. Cooley, S. Knol, and S. Hauck, "Configuration relocation and defragmentation for fpgas," Tech. Rep., 2000.

[12] O. Diessel, H. Eigindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," *IEE Proceedings: Computers and Digital Techniques*, vol. 147, no. 3, pp. 181–188, 2000.

[13] F. Redaelli, M. D. Santambrogio, and S. Ogrenci Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 97–102, Cancun, Mexico, December 2008.

[14] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada, "Rainbow: an OS extension for hardware multitasking on dynamically partially reconfigurable FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, pp. 416–421, December 2011.

[15] J. Kelm, I. Gelado, K. Hwang et al., "Operating system interfaces: bridging the gap between cpu and fpga accelerators," in *Proceedings of the International Symposium on FPGAs*, vol. 48, Monterey, Calif, USA, February 2007.

[16] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: automatic memory and cache management for reconfigurable logic," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pp. 25–28, ACM, Monterey, Calif, USA, March 2011.

[17] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA operating system," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL '14)*, pp. 1–8, Munich, Germany, September 2014.

[18] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[19] J. Teich, S. P. Fekete, and J. Schepers, "Optimization of dynamic hardware reconfigurations," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 57–75, 2001.

[20] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[21] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[22] D. Koch, C. Beckhoff, and J. Teich, "Minimizing internal fragmentation by fine-grained two-dimensional module placement for runtime reconfigurable systems," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 251–254, April 2009.

[23] A. Ahmadinia, J. Angermeier, S. P. Fekete et al., "ReCoNodes-optimization methods for module scheduling and placement on reconfigurable hardware devices," in *Dynamically Reconfigurable Systems*, pp. 199–222, Springer, Berlin, Germany, 2010.

[24] A. Wold, A. Agne, and J. Torresen, "Relocatable hardware threads in run-time reconfigurable systems," in *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14–16, 2014. Proceedings*, D. Goehringer, M. Santambrogio, J. Cardoso, and K. Bertels, Eds., vol. 8405 of *Lecture Notes in Computer Science*, pp. 61–72, Springer, Berlin, Germany, 2014.

[25] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 986–991, Munich, Germany, March 2003.

[26] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware task scheduling for partially reconfigurable fpgas," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Huebner, and P. C. Diniz, Eds., vol. 9040 of *Lecture Notes in Computer Science*, pp. 487–498, Springer, Berlin, Germany, 2015.

[27] K. Rupnow, W. Fu, and K. Compton, "Block, drop or roll(back): alternative preemption methods for RH multi-tasking," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 63–70, April 2009.

[28] K.-J. Shih, H.-Y. Sun, and P.-A. Hsiung, "Dynamic hardware-software task switching and relocation mechanisms for reconfigurable systems," in *Proceedings of the IET International Conference on Frontier Computing. Theory, Technologies and Applications*, pp. 157–162, Taichung, Taiwan, August 2010.

[29] P. G. Zaykov, G. K. Kuzmanov, and G. N. Gaydadjiev, "Reconfigurable multithreading architectures: a survey," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 9th International Workshop, SAMOS 2009, Samos, Greece, July 20–23, 2009. Proceedings*, vol. 5657 of *Lecture Notes in Computer Science*, pp. 263–274, Springer, Berlin, Germany, 2009.

[30] Y. Wang, X. Zhou, L. Wang et al., "SPREAD: a streaming-based partially reconfigurable architecture and programming model," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2179–2192, 2013.

[31] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck, "Configuration relocation and defragmentation for run-time reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 209–220, 2002.

[32] O. Diessel and G. Wigley, "Opportunities for operating systems research in reconfigurable computing," Tech. Rep. ACRC-99-018, University of South Australia, Adelaide, Australia, 1999.

[33] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," in *Proceedings of the 6th Australasian Computer Systems Architecture Conference (ACSAC '01)*, pp. 130–137, January 2001.

[34] J. H. Kelm and S. S. Lumetta, "HybridOS: runtime support for reconfigurable accelerators," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA '08)*, pp. 212–221, ACM, Monterey, Calif, USA, February 2008.

[35] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[36] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: a reusable integration framework for FPGA accelerators," in *Proceedings of the 20th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '12)*, pp. 216–219, IEEE, Toronto, Canada, April-May 2012.

[37] M. Jacobsen and R. Kastner, "RIFFA 2.0: a reusable integration framework for FPGA accelerators," in *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications (FPL '13)*, pp. 1–8, Porto, Portugal, September 2013.

[38] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 7, IEEE, April 2003.

[39] D. Andrews, D. Niehaus, R. Jidin et al., "Programming models for hybrid FPGA-CPU computational components: a missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[40] D. Andrews, D. Niehaus, and P. Ashenden, "Programming models for hybrid CPU/FPGA chips," *Computer*, vol. 37, no. 1, pp. 118–120, 2004.

[41] D. Andrews, W. Peck, J. Agron et al., "hthreads: a hardware/software co-designed multithreaded RTOS kernel," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '05)*, pp. 331–338, September 2005.

[42] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: a computational model for reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 885–888, Madrid, Spain, August 2006.

[43] E. Lübbers and M. Platzner, "Reconos: an RTOS supporting hard- and software threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 441–446, IEEE, Amsterdam, The Netherlands, August 2007.

[44] A. Agne, M. Happe, A. Keller et al., "ReconOS: an operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[45] M. Happe, A. Traber, and A. Keller, "Preemptive hardware multitasking in reconOS," in *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13–17, 2015, Proceedings*, vol. 9040 of *Lecture Notes in Computer Science*, pp. 79–90, Springer, Berlin, Germany, 2015.

[46] D. Gohringer and J. Becker, "High performance reconfigurable multi-processorbased computing on FPGAs," in *Proceedings of the IEEE International Symposium on Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–4, Atlanta, Ga, USA, 2010.

[47] D. Göhringer, M. Hübner, L. Hugot-Derville, and J. Becker, "Message passing interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC," in *Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS '10)*, pp. 357–364, July 2010.

[48] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker, "CAP-OS: operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–8, April 2010.

[49] D. Göhringer, S. Werner, M. Hübner, and J. Becker, "RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC," in *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL '11)*, pp. 181–184, IEEE, Chania, Greece, September 2011.

[50] D. Gohringer, M. Hubner, V. Schatz, and J. Becker, "Runtime adaptive multi-processor system-on-chip: RAMPSoC," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, pp. 1–7, Miami, Fla, USA, April 2008.

[51] X. Iturbe, K. Benkrid, A. T. Erdogan et al., "R3TOS: a reliable reconfigurable real-time operating system," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '10)*, pp. 99–104, IEEE, Anaheim, Calif, USA, June 2010.

[52] X. Iturbe, K. Benkrid, C. Hong et al., "R3TOS: a novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1542–1556, 2013.

[53] A. Ismail and L. Shannon, "FUSE: front-end user framework for O/S abstraction of hardware accelerators," in *Proceedings of the 19th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, pp. 170–177, May 2011.

[54] Y. Wang, J. Yan, X. Zhou et al., "A partially reconfigurable architecture supporting hardware threads," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '12)*, pp. 269–276, IEEE, Seoul, Republic of Korea, December 2012.