Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 789134, 40 pages
http://dx.doi.org/10.1155/2013/789134

Research Article

Rainbow: An Operating System for Software-Hardware Multitasking on Dynamically Partially Reconfigurable FPGAs

Krzysztof Jozwik,[1] Shinya Honda,[1] Masato Edahiro,[1] Hiroyuki Tomiyama,[2] and Hiroaki Takada[1]

[1] *Graduate School of Information Science, Nagoya University, C3-1 (631) Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan*
[2] *Department of VLSI System Design, College of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-Higashi Kusatsu, Shiga 525-8577, Japan*

Correspondence should be addressed to Krzysztof Jozwik; kjozwik@acm.org

Received 28 February 2013; Revised 4 July 2013; Accepted 8 July 2013

Academic Editor: Michael Hübner

Copyright © 2013 Krzysztof Jozwik et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Dynamic Partial Reconfiguration technology coupled with an Operating System for Reconfigurable Systems (OS4RS) allows for implementation of a hardware task concept, that is, an active computing object which can contend for reconfigurable computing resources and request OS services in a way software task does in a conventional OS. In this work, we show a complete model and implementation of a lightweight OS4RS supporting preemptable and clock-scalable hardware tasks. We also propose a novel, lightweight scheduling mechanism allowing for timely and priority-based reservation of reconfigurable resources, which aims at usage of preemption only at the time it brings benefits to the performance of a system. The architecture of the scheduler and the way it schedules allocations of the hardware tasks result in shorter latency of system calls, thereby reducing the overall OS overhead. Finally, we present a novel model and implementation of a channel-based intertask communication and synchronization suitable for software-hardware multitasking with preemptable and clock-scalable hardware tasks. It allows for optimizations of the communication on per task basis and utilizes point-to-point message passing rather than shared-memory communication, whenever it is possible. Extensive overhead tests of the OS4RS services as well as application speedup tests show efficiency of our approach.

1. Introduction

The research on Dynamically Partially Reconfigurable (DPR) Field-Programmable Gate Arrays (FPGAs) is motivated by their superior flexibility, when compared to traditional FPGAs and Application-Specific Integrated Circuits (ASICs), as well as their potential to increase overall system performance and reduce dynamic power consumption by adapting to varying processing requirements of a system. DPR technology allows to partially change contents of the initial FPGA's configuration at run-time, without disturbing operation of the rest of the system [1]. It leads to a concept of virtualization of hardware resources where a small, however, dynamically reconfigurable array, multiplexing its hardware resources in time, may give an illusion of hosting circuitry by far exceeding its real capacity [2]. While the idea of virtual hardware managed by an OS was first proposed by Brebner in [3, 4], Wigley and Karney [5] defined a set of properties an Operating System for Reconfigurable Systems (OS4RS) should have. These properties have been refined later on as the research in this field progressed [6].

The main objective of the OS4RS is to provide an abstraction layer boosting development of applications composed of both software (SW) and hardware (HW) tasks. The HW task [7]/thread [8] can be thought of as a flow of execution running on an FPGA, sharing the reconfigurable resources with other HW tasks in a time-multiplexed manner. However, the concept of the HW task is different than that of an FPGA-based HW accelerator. The HW tasks are dynamic objects, just like the SW tasks in the traditional OSes [9, 10]. They may compete for reconfigurable computing resources with other HW tasks, request OS services such

as communication, synchronization, or even activate other HW tasks. To accomplish that, the OS4RS must provide the HW tasks with an adequate interface to access those services. The OS4RS is responsible for control over all the aspects related to execution of SW tasks, that is, state management, dispatching, scheduling, and their intertask communication, just like a conventional OS does. In addition to that, it is responsible for all the aspects related to execution of HW tasks [6], that is, their allocation and deallocation on the FPGA, scheduling, state management, and intertask communication within and across SW and HW domains. While performing all these activities, the OS4RS must keep in mind applications' execution time, response times, and utilization of the FPGA resources, meeting requirements imposed on executed applications.

Apart of meeting the aforementioned basic requirements, the OS4RS may additionally provide some of the following more advanced functions, that is, *HW task preemption* by allowing suspension and restoration of HW task's execution, *Dynamic Frequency Scaling (DFS)* of HW task's clock (HW task clock scaling), *HW task migration* by allowing relocation of HW tasks on the FPGA, and *SW-HW task morphing* by allowing the tasks to migrate between SW and HW domains during their execution. All these functions aim at improving performance and reducing power consumption of an application running on top of the OS4RS.

Concept of the HW task and the SW-HW Multitasking OS4RS recently gained wider popularity in the industrial world and research community due to promising results of High-Level Synthesis (HLS) technology [11]. When combined with it, the concepts of the HW task and OS4RS make it easier for software developers with weak or no background in hardware design to take advantage of the computing potential of reconfigurable hardware. SW-HW partitioned applications can be entirely written in the same way, using high-level description language like C and then mapped to either Central Processing Unit (CPU) or FPGA depending on the performance and power consumption requirements imposed on the application. Yet, further optimizations on the software or hardware side could be allowed by supporting assembly or Hardware Description Language (HDL) as a language to describe SW and HW tasks, respectively.

The contributions to the state-of-the-art OS4RS made by this work can be grouped into three following research topics.

(i) *OS4RS Architecture*. It proposes a complete and novel model and implementation of the OS4RS architecture supporting preemptable and clock-scalable HW tasks. The proposed OS4RS is composed of a conventional off-the-shelf Real-Time Operating System (RTOS) kernel which provides the SW multitasking functionality and created reusable extension, called Rainbow, which adds the HW multitasking capability.

(ii) *Scheduling Mechanism*. It proposes a novel, lightweight software-based scheduling mechanism allowing for timely and priority-based reservation of reconfigurable resources, which aims at use of HW task preemption only at the time it brings benefits to the total performance of a system. The architecture of the scheduler and the way it schedules allocations and deallocations of the HW tasks on the FPGA result in shorter latency of Application Programming Interface (API) calls, thereby reducing the overall OS overhead.

(iii) *Intertask Communication*. It shows a novel model and implementation of an easily scalable channel-based inter-task communication and synchronization suitable for SW-HW multitasking with preemptable and clock-scalable HW tasks. The model allows for optimizations of the communication on per task basis and takes advantage of more efficient point-to-point message passing rather than shared-memory communication, whenever it is possible.

The remaining contents of this paper are organized as follows. In Section 2, related research works are described, and work presented in this paper is compared with them. Section 3 gives a general overview of the developed OS4RS and its architecture. Section 4 introduces the base OS kernel used in our implementation of the OS4RS, whereas Sections 5, 6, 7, and 8 show implementation details of HW tasks' reconfiguration, scheduling, management and intertask communication services provided by the developed extension. Section 9 presents results of extensive evaluation of the OS4RS's services and application speed-up tests. Finally, Section 10 concludes the work and reveals area of prospective research.

2. Related Research

Several previous research works were devoted to design of the OS4RS or its certain services. While some of the works provide only a means of run-time management of HW accelerators [12–17], the others provide a proper support for HW tasks [18–21]. This is accomplished by supplying the HW tasks with a dedicated OS interface. BORPH [18] is a Linux extension which targets multi-FPGA platforms where each HW task is implemented on a separate FPGA. The HW tasks are modeled with a Simulink [22] and provided with an equivalent API to the one that SW tasks have. ReconOS [19, 20] targets DPR FPGAs where HW tasks are implemented as reconfigurable modules on a common FPGA. It is implemented as an extension for Linux and eCos [10] and utilizes a unified POSIX-compliant API for SW and HW tasks. HThreads [21] presents another OS which is a Linux extension; thus it is also based on a POSIX-compliant API. An interesting aspect of that work is architecture of the OS interface for HW tasks which gives them support for dynamic memory allocation and recursive execution of functions. That work, however, lacks the support for DPR. An interesting work is shown in [23] where tasks can migrate between software and hardware domains during their execution, that is, the work supports software-hardware morphing.

In this paper, we present a model of a portable and lightweight OS4RS built as an extension to existing OS kernel. The presented OS4RS treats HW tasks as active computing objects. Moreover, we show a complete implementation of this model which fully utilizes DPR. The implementation is

based on Toyohashi OPen Platform for Embedded Real-time Systems (TOPPERS) [24] Advanced System Profile (ASP) RTOS kernel compliant with an uItron [9] API specification. The uItron specification is widely used in the Japanese industry, research communities, and the European automotive industry.

Several works were devoted to scheduling and placement of HW tasks (or reconfigurable hardware accelerators) on the FPGA [25–31]. In [25], an online scheduling and placement algorithm targeting 1D and 2D model of reconfigurable area is presented. The work considers nonpreemptable HW tasks which execute independently of each other. On the contrary, works in [26–30, 32] present scheduling strategies for reconfigurable area divided into fixed slots. As modern FPGA architectures are becoming increasingly heterogeneous in terms of their resources, this model of a reconfigurable area seems to be more applicable. [26] presents a mixed offline/online scheduling strategy, where the Control Data Flow Graphs (CDFGs) representing applications are analyzed offline by means of customized list scheduling techniques, and then extracted parameters are used to optimize the scheduling result at run-time. The online scheduling step is implemented in hardware as dedicated logic. [27] presents another hardware implementation of a run-time scheduler utilizing information extracted from the CDFGs. Similarly to work presented in [26]. Configuration Access Port OS (CAP-OS) [31] shows a mixed offline/online scheduling strategy utilizing information extracted from the CDFGs and is implemented on Xilkernel OS [33] running on a microprocessor. In the approaches presented in [26, 27, 31], HW tasks comprising the CDFGs execute in a nonpreemptive manner till their completion. Work in [32] presents a non-preemptive scheduling strategy where HW tasks may voluntarily relinquish the allocated resources at certain points of their execution. Works in [28, 29] present a Deadline-Monotonic- (DM-) based scheduling approach for real-time systems, where HW tasks' reconfigurations rather than executions are scheduled. The work does not consider data dependencies between HW tasks. The works in [28, 29] and CAP-OS [31] do not support preemption of the HW tasks' execution but allow for termination of their reconfigurations.

This paper presents a simple preemptive priority-based scheduling mechanism with certain customizations allowing for HW task prefetching and timely reservation of reconfigurable resources. Similarly to works presented in [26–30, 32], it targets systems where the reconfigurable area is divided into fixed regions. Unlike the works in [28, 29, 31], this work allows for true preemption of the HW task's execution phase. The concept of reservation of reconfigurable resources has been already introduced in [29, 30]. In [29], one or more slots are reserved at the design time, exclusively for a given set of high-priority tasks, and then used by these tasks only. In [30], reconfigurable slots, called tiles, are reserved (locked) by the most frequently executed HW tasks. Unlike those works, the work presented in this paper allows for timely reservation of the resources, based on blocking time of HW tasks, which can be adapted to match the latency characteristics of a given system. The timely reservation of the resources will allow for preemption of the HW task only after a specified time from the point the HW task blocks. While the presented scheduling mechanism is implemented mostly in software, it allows for overlapped execution of the scheduler and HW tasks' reconfigurations. The presented work is suitable for both, set of independent HW tasks as well as set of HW tasks with data dependencies described, for example, in form of CDFGs.

In the first case, the overall ability of preemption may be used to prevent HW tasks from starving for reconfigurable resources in case another task has been executing for a very long time. Ability of preemption after a specified time may be used to allow execution of another HW task, while the currently running task blocked, waiting for data for a specified time. Yet, at other times, the preemption and related reconfiguration overhead can be avoided. It can be beneficial especially in systems with memory virtualization and systems with networking where data delivery latencies may be variable and higher than reconfiguration times present in today's DPR FPGAs. Furthermore, it may be also used as a deadlock recovery measure.

In the second case, where there are data dependencies between some HW tasks, the set of tasks could be analyzed offline and divided into set of independent CDFGs. These could be further analyzed and appropriate setting of HW tasks priorities and sequence of their activations generated in form of API calls.

This part is assumed to be done by an offline tool and is considered orthogonal to the work presented in this paper. While the prefetching capability presented in this work may allow for efficient execution of such CDFGs, the timely reservation of reconfigurable resources may be used to allow execution of another CDFG, while the currently executing one is waiting for data.

Several works presented inter-task communication mechanisms targeting SW-HW multitasking. Works in [8, 34] utilize POSIX-like communication and synchronization mechanisms known from traditional SW multitasking OSes, such as message queues, semaphores, and apply them to the SW-HW multitasking domain. In both works, HW tasks request the communication and synchronization services via a dedicated OS interface module. In [34], the inter-task synchronization objects are located in software and are accessed by delegate SW tasks executed on behalf of the HW tasks. In [8], the corresponding objects are implemented in hardware. ReconOS [34] additionally implements HW-HW message passing communication with FIFO buffers which are directly connected to the OS interface for HW tasks. While the additional fixed FIFO buffers avoid software processing overhead, the approach lacks scalability. Although the hardware implementation given in [8] significantly reduces the processing overhead, it only allows for shared memory communication which does not seem to be suitable for all HW-HW inter-task communication.

The work in [35] shows an inter-task communication mechanism for SW and HW tasks based on additional software-hardware codesigned virtualization extension for Linux. The extension uses the concept of memory paging. The hardware side of the extension consists of specialized

modules interfaced by the HW tasks. The modules contain a local storage for pages and perform address translation for the HW task's memory accesses. The software side of the extension handles copying of pages during misses as well as their speculative prefetching to the local memory. Thanks to the created extension, SW and HW tasks may communicate with each other transparently, without knowing whether their communication partner is located in hardware or software. This results in easier programming and better application's portability. While it is an interesting approach, it only supports shared memory communication and does not consider run-time reconfiguration of HW tasks.

An interesting work is shown in [36] where FPGA's Configuration Memory (CM) is used for communication between HW tasks. The data to be transferred is read from a sending HW task's buffer through the configuration port of the FPGA and stored in some different location in the CM which corresponds to a buffer of the receiving HW task. While this approach avoids signal routing issues related to HW Task's relocation, it creates a significant overhead for communication.

In works [18, 37, 38], SW and HW tasks communicate through FIFO buffers. FUSE [12] shows implementation of an SW-HW communication interface for dynamically reconfigurable HW accelerators. In that work, the communication drivers located on the software side can be loaded dynamically, thereby providing a means of their run-time customization. However, the hardware side of the interface is fixed, and the work does not show implementation in which the accelerators are dynamically reconfigured. One of the interesting works on channel-based communication for SW-HW Multitasking systems is presented in [39]. In this work, the tasks communicate through dynamically allocated channels managed by a hardware-based communication manager. Although performance advantages of the hardware implementation are mentioned, the work does not allow for optimizations based on a type of communication. Moreover it does not actually support blocking communication semantics as our work does.

We show a complete model and implementation of a channel-based intertask communication and synchronization allowing for Point-2-Point (P2P) communication between the tasks, similarly to [39]. Unlike [18, 37–39], it allows for optimization of the communication based on a given pair of tasks. This is achieved by, firstly, matching the type of the channel to communication requirements and, secondly, by allowing the channel interfaces to be allocated and deallocated together with HW tasks. Unlike [39], this work allows for truly blocking communication semantics. Unlike the previous works, the presented inter-task communication and synchronization mechanism is suitable for clock-scalable and preemptable HW tasks.

Dynamic Frequency Scaling (DFS) [40, 41] makes it easier to balance the trade-off between the performance and power consumption of the resources held by an HW task. It has been widely used in single and multicore CPUs as a method to balance the trade-off between low power consumption and high performance depending on current processing demands of applications. In the field of FPGAs, previous works either only discussed this mechanism in general [42, 43] without the context of HW multitasking or only assumed its presence in their scheduling mechanisms without showing any implementation [31]. The works in [44, 45] show implementation of a HW task utilizing the regional clocking resources available in Xilinx FPGAs in order to enhance HW task relocation and provide means of discrete clock division. It is, however, not a substitute for continuous clock-scaling functionality, but rather its enhancement.

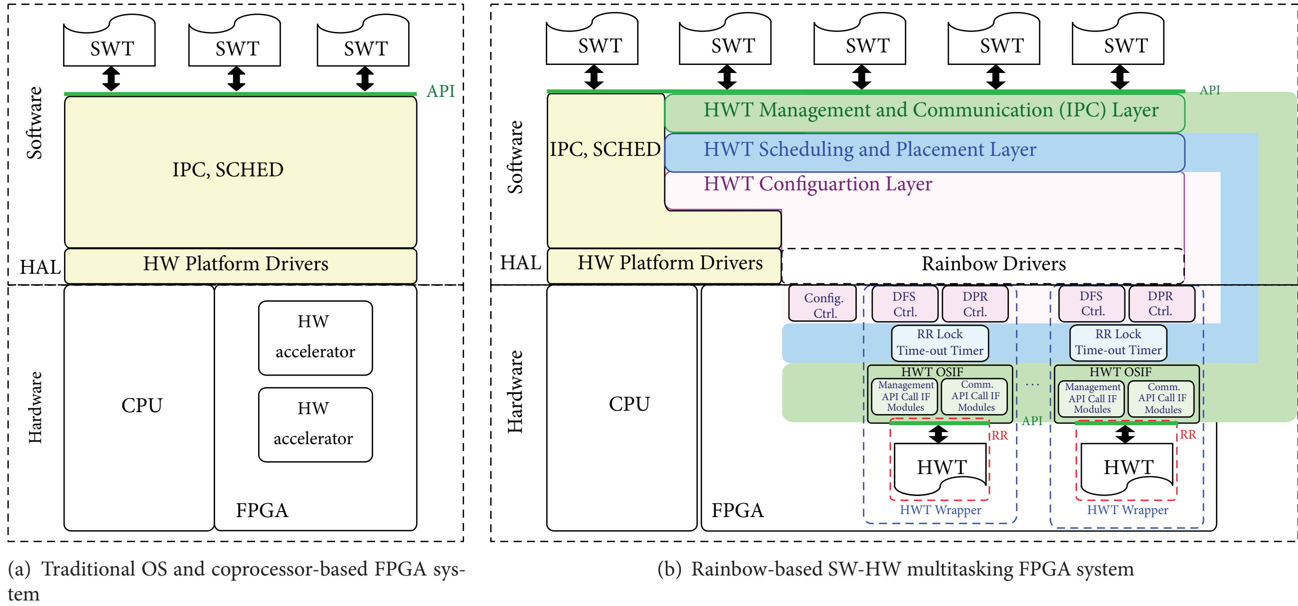
This work proposes a complete model and implementation of the OS4RS supporting preemptable and clock-scalable HW tasks. The DFS presented in this work is supported by all services provided by the OS4RS, that is, management, scheduling, reconfiguration, and inter-task communication and synchronization.

3. OS Architecture

3.1. Architecture Model: Overview. Instead of building the OS4RS completely from scratch, we decided to use an already available OS kernel with a well-established API as a base and create a HW multitasking extension for it. This was done to utilize already available SW multitasking support and to facilitate future reuse of already available applications. The HW multitasking extension is general enough, so that a variety of existing SW multitasking RTOS kernels [10, 24, 33] could be used as its base.

The software-hardware codesigned extension, called Rainbow, follows a layered architecture in both software and hardware, which improves code reuse and portability and helps in grouping different OS services. An abstracted view of the developed OS4RS architecture is shown in Figure 1(b). The *HW Task Management and Communication Layer* manages HW tasks' high-level execution state, reflecting the API calls made by interacting SW and HW tasks, as well as inter-task communication and synchronization. The state of SW tasks is managed independently, by the base OS kernel. The *HW Task Scheduling and Placement Layer* is responsible for scheduling allocations, deallocations, and preemptions of HW tasks as well as process of their clock dynamic frequency scaling. Finally, the *HW Task Configuration Layer*, which is specific to a given FPGA architecture, manages the related low level aspects of these activities.

The software part of the extension is executed on a CPU together with the base OS kernel, whereas hardware part is implemented on an FPGA. Hardware part is comprised of a *Configuration Controller* which belongs to the *HW Task Configuration Layer* and *HW Task Wrappers* whose modules logically belong to all three layers. The *Configuration Controller* gives a physical means of allocation and preemption of HW tasks by providing an interface to access the FPGA's CM. The *HW Task Wrapper* contains a *Dynamic Partial Reconfiguration (DPR) Controller* which also belongs to the same layer as the *Configuration Controller*. It handles the low-level physical aspects of the DPR technology related to HW task's allocation and preemption [46–48]. The *Dynamic Frequency Scaling (DFS) Controller* is another module which belongs to the *HW Task Configuration Layer*. It provides access to clock frequency scaling feature of HW tasks. The *Reconfigurable*



(a) Traditional OS and coprocessor-based FPGA system

(b) Rainbow-based SW-HW multitasking FPGA system

Figure 1: OS architecture model: Traditional versus Rainbow-extended.

Region (RR) Lock Timer, being part of the *HW Task Scheduling and Placement Layer*, is used to implement a mechanism of timely reservation of reconfigurable resources, described in Section 6.2. Finally, the *Management API Call Interface Modules* and the *Communication API Call Interface Modules* belong to the *HW Task Management and Communication Layer*. They comprise a *Hardware Task Operating System Interface (HWT OSIF)* which is an interface for HW tasks to access the OS services.

Figure 1 shows the relationship between SW tasks and HW tasks in a traditional and Rainbow-extended OS. In the traditional OS, SW tasks use HW platform drivers provided by the *Hardware Abstraction Layer (HAL)* to access the FPGA resources. These drivers have to be customized on per HW accelerator basis, thus making porting difficult. In the OS4RS based on the Rainbow extension, the HW tasks are treated as active objects which have access to the same OS services as SW tasks do. The services are accessed by means of a high-level API, which improves application portability. The Rainbow extension uses HAL drivers internally, only to bind together its software and hardware components.

There are two types of service calls the SW and HW tasks can make: *Management API Calls* and *Communication API Calls*. The *Management API Calls* give the tasks an ability to directly control their state and the state of other tasks. It includes calls for task activation, termination, task-dependent synchronization, for example, suspension and resumption, as well as clock management functionality for the HW tasks, that is, DFS calls. The *Communication API Calls* give the means of communication between tasks and task-independent synchronization, that is, the one implemented by communication primitives.

When compared to the traditional OSes, the inter-task communication implemented by Rainbow is based on a concept of channel adopted from Electronic System-Level (ESL) Design Methodology [49]. The channels are abstract objects encapsulating communication between two computing processes. In our case, the channels are referenced by their Identifiers (IDs) and provide the tasks with an easy interface to access communication and synchronization resources of the OS.

3.2. A Reconfigurable Hardware Architecture Model. The developed OS4RS assumes the hardware model of a reconfigurable architecture shown in Figure 2. SW tasks execute on a CPU. HW tasks are represented by configuration bitstreams stored in the *HW Task Repository* and take a form of digital circuits once configured in the *Reconfigurable Region (RR)* by means of the *Configuration Controller*.

Depending on placement restrictions imposed on HW tasks, we can distinguish different models of reconfigurable area [6]. 2D model is the most flexible and does not impose any restrictions on the placement of the tasks, provided that they do not overlap. In 1D model vertical dimension is fixed, but allows unconstrained placement in horizontal dimension. Finally, in fixed-slots model tasks can be only placed at predefined positions. An additional, fixed-regions model can be thought of as a generalized fixed-slots model in which not all of the regions are of the same dimensions. This may be a disadvantage by limiting task relocation to compatible regions only, but on the other hand allows them to better match their associated regions, thereby alleviating the problem of region's internal fragmentation.

Due to increasing heterogeneity of FPGAs, accompanied by limitations of frame-based Configuration Memory's architecture and Partial Reconfiguration (PR) technology, the last two models are by far the most suitable for recent FPGA devices and their supporting design tools. The fixed-regions model is currently widely used by Xilinx and Altera and has been also adopted in our model of reconfigurable

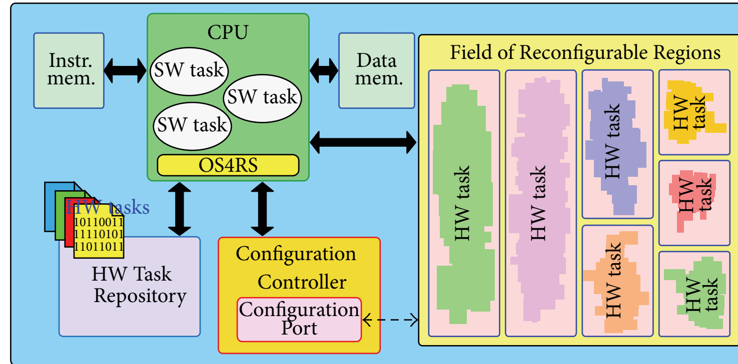


Figure 2: Reconfigurable architecture model.

architecture. In this model, HW tasks are prepared offline and loaded at run-time to predefined regions. The HW tasks have also their state which can be saved and restored if the preemption mechanism is available. Furthermore, they can migrate between compatible regions if the bitstream relocation [36, 50–53] is supported.

3.3. Rainbow Extension: Software Structures Overview. The details of the architecture of the software side of the Rainbow extension are given in Figure 3.

The key structures of the software side of the extension are the arrays of *HW Task Control Blocks (HWTCBs)* and *HW Task Region Control Blocks (HWTRCBs)* which span all three layers.

Entries in the arrays of *HWTCBs* and *HWTRCBs* correspond to the HW tasks and RRs with given IDs. The HW tasks' IDs are currently considered separately from those of SW tasks, managed by the base OS. As the task IDs are passed to *Management API Calls*, this requires that different API calls are used for SW and HW tasks. This is not the case with *Communication API Calls* which use a common ID space for all channels. The common ID space for both SW and HW tasks would allow for entirely unified API for SW and HW tasks. It is left as a part of future work.

3.3.1. HW Task Configuration Layer. Part of the *HWTCB* located in this layer holds information such as memory address and size of the bitstream(s) needed to configure a given task as well as initialize, save, and restore its state. It is represented by the *Bitstream Descriptors* and the *Physical State Descriptors* initialized during OS4RS boot-up, while the bitstreams representing the HW tasks are read from an external nonvolatile storage and stored in a faster volatile memory serving as a bitstream repository.

3.3.2. HW Task Scheduling and Placement Layer. The key structure related to scheduling is an array of *HWTCBs* queues ordered by priority, that is, ready queues. Each RR, where HW tasks are allocated, has an independent array of ready queues associated with it. It is located in the *HWTRCB*.

This layer implements a preemptive scheduling with timely and priority-based resource reservation, called *RR Locking*, described in details in Section 6. The hardware side of this layer implements the *Time-out Timer* used in the *RR Locking*. The *HWT Dispatcher* shown in Figure 3 is an additional SW task which is started whenever a scheduling decision results in requests for allocation, deallocation, or clock management. The *Dispatch Flags* entry in the *HWTCB* is used to pass information about actions to be performed between the scheduler and the *HWT Dispatcher*.

3.3.3. HW Task Management and Communication Layer. Each *HWTCB* in this layer stores the HW task state and extended information, such as, for example, ID of the system it was made for, retrieved from the configuration bitstream upon OS initialization. It also holds control information of the *Management API Call Module* such as whether the HW task was blocked as a result of executing the API call (*API Call IF Mgmt Flags*), and the return value that should be passed to it when it is released from blocking and allocated again. The *API Call IF Mgmt Flags* are used to control operation of the Finite State Machine (FSM) located in the *Management API Call Module*. The control information entries are updated by the *Management API Call Interrupt Handler* and *After Physical HWT Activation Callback*.

The *Management API Call Interrupt Handler* is executed whenever a HW task makes a *Management API Call*. The *Before Physical HWT* Inactivation and the *After Physical HWT Activation* callbacks are functions of the *HW Task Management and Communication Layer* registered and executed within the *HW Task Configuration Layer*. The first callback contains code which is to be executed just before the HW task is deallocated or its clock is suspended, whereas the second one contains the code to be executed just after the HW task is allocated or its clock is switched on.

The *Config Lock* field of the *HWTCB* is used to implement a configuration locking mechanism used by the inter-task communication. The idea behind it is to prevent possible HW task deallocations when the transfer over the channel connecting the task is in progress.

A key structure in the *HW Task Management and Communication Layer* is the *Channel Control Table* which controls the communication and synchronization over channels. The table is updated by the *Comm Event Interrupt Handlers*, SW Task API calls, *Copy Task,* and *Callbacks* which act together

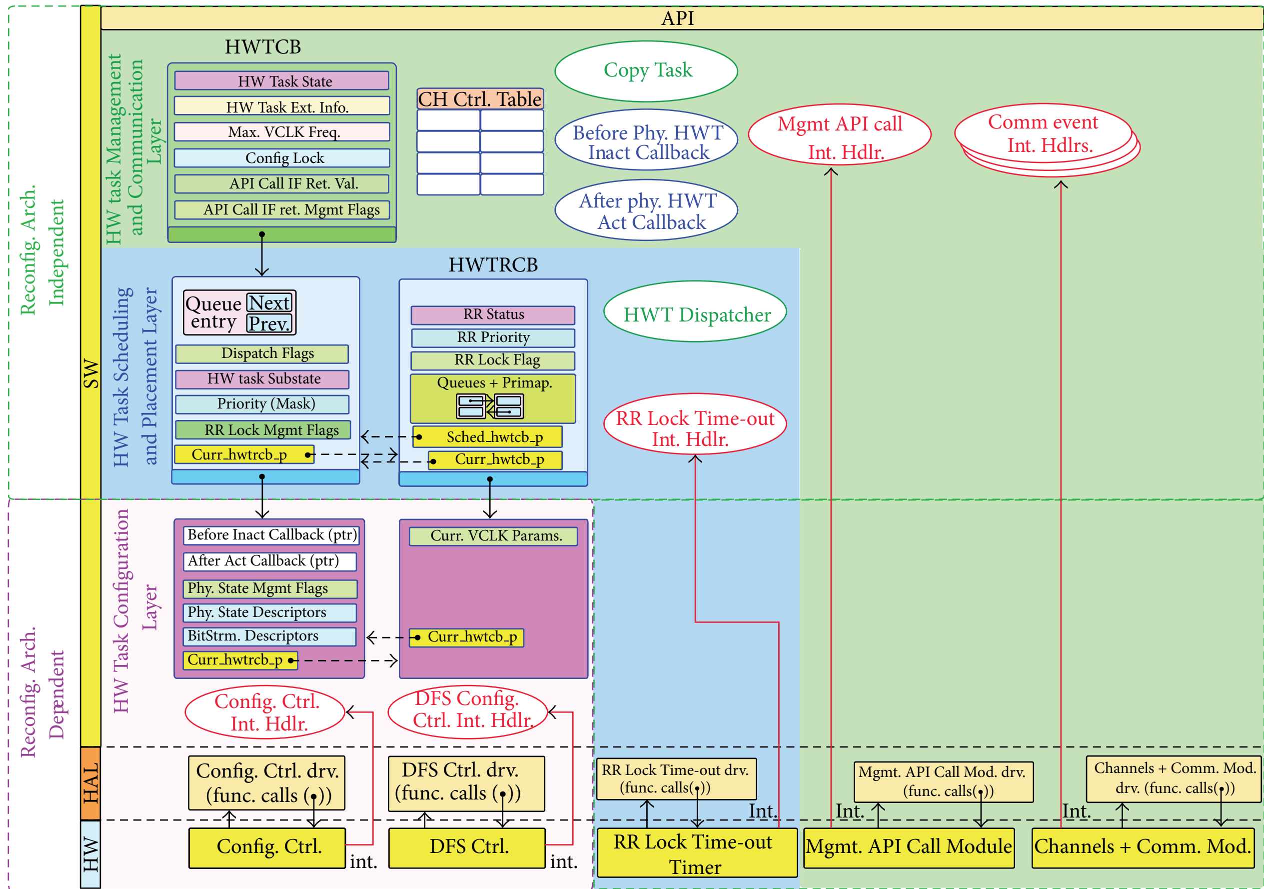


Figure 3: Rainbow extension—software structures and software-hardware interface.

in order to provide communication services to SW and HW tasks.

The *Comm Event Interrupt Handlers* are interrupt handlers executed in response to channel events which are generated by the hardware side of the OS4RS whenever HW tasks access the communication channels. SW Task API calls are communication calls provided by the OS4RS which are executed by the SW tasks. The *Copy Task* is an additional SW task executed whenever a direct point-to-point (P2P) communication between the HW tasks is not possible. All of them are described in details in Section 8.

3.4. *Rainbow Extension: Hardware Components Overview*

3.4.1. Configuration Controller. The Configuration Controller provides physical means of allocation and preemption of HW tasks. It accomplishes this job by talking to the configuration port of the FPGA and transferring configuration bitstreams representing HW tasks between their storage memory and FPGA's CM.

3.4.2. HW Task Wrapper. The *HW Task Wrapper* can be divided into static part, that is, *Static Region (SR)*, and reconfigurable part, that is, *Reconfigurable Region (RR)*. Logically, the RR serves as a container where HW modules, representing HW tasks, can be plugged in. Physically, once the whole system is programmed on an FPGA, the RR becomes an area on the FPGA where HW tasks are allocated and deallocated. Figure 4 shows the *HW Task Wrapper* and the *HW Task* allocated in the RR.

To alleviate timing issues during placement and routing of a DPR system, the boundary between the SR and the RR must be designed in an appropriate way. More specifically, all signals crossing the SR-RR boundary should be registered by inserting FFs before they are passed to the other side of the boundary. In the developed *HW Task Wrapper*, the SR side of the boundary is always registered, whereas the RR side is registered on per task basis.

The *HW Task Wrapper* is composed of two planes: a *Data Plane* and a *Control Plane*. The *Data Plane* contains an *HW Task Communication Module* and a *Data Interconnect Interfacing Logic* module, whereas the *Control Plane* is composed of an *OS Control Module* and a *Ctrl Interconnect Interfacing Logic* module. The *OS Control Module* is further divided into an *RR Lock Time-out Timer*, a *Translation Table (TT)*, a *DFS Controller*, a *DPR Controller*, an *HW Task Communication Event Control*, and an *HW Task Management API Call Module*.

The *HW Task Communication Module*, the *HW Task Communication Event Control*, the *TT*, and *Communication*

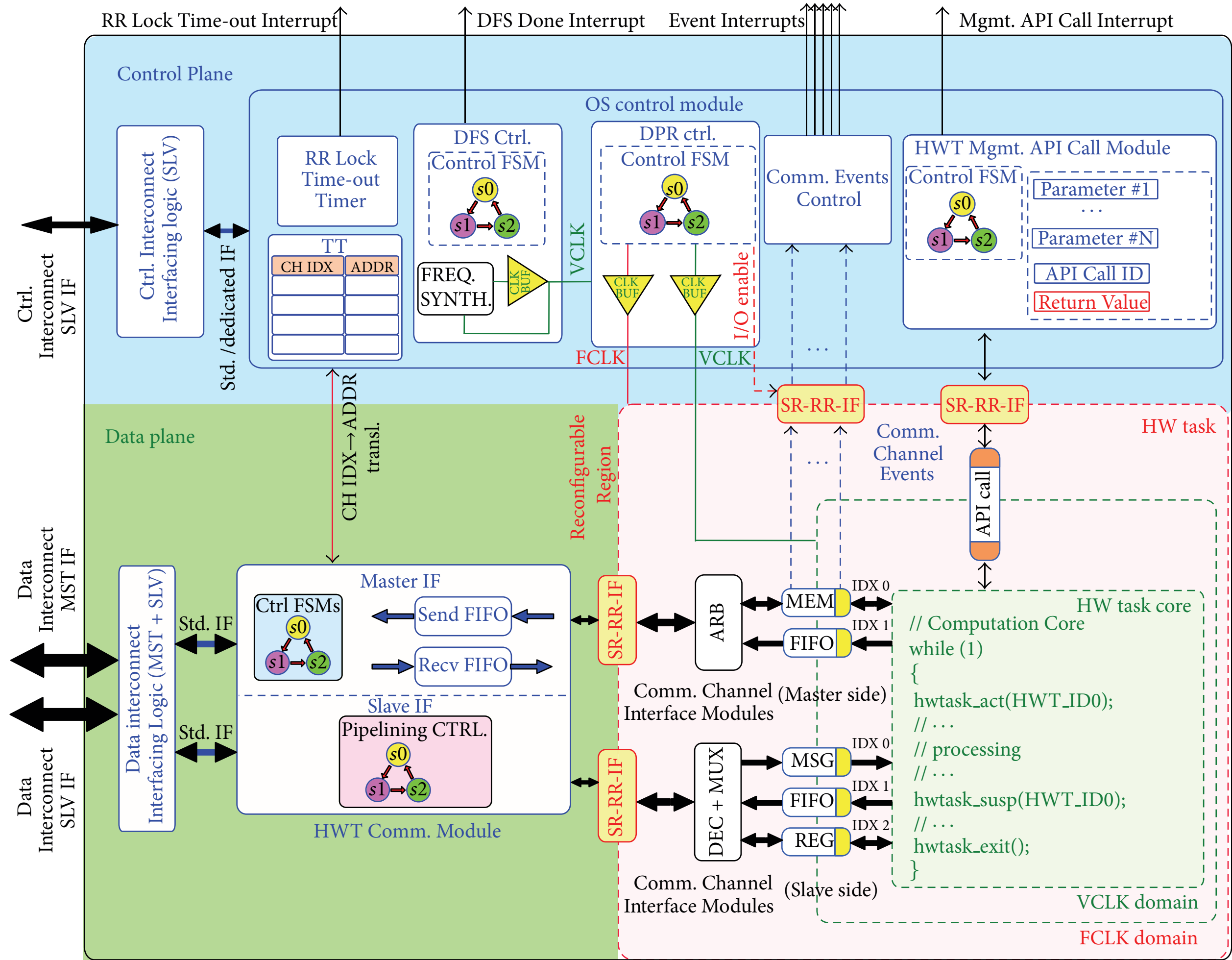


Figure 4: HW Task and HW Task Wrapper.

Channel Interface Modules located in the HW Task logically belong to a group of *Communication API Call Interface Modules*. The HW Task Management *API Call* Module and the API Call channel located in the HW Task logically belong to another group called *Management API Call Interface Modules*. Both groups form the *HWT OSIF* and described before. All *HW Task Wrapper's* modules are briefly described below.

HW Task Communication Module. The *HW Task Communication Module* and the *Data Interconnect Interfacing Logic* module give access for master- and slave-side *Communication Channel Interface Modules* to the physical interconnect. The *HW Task Communication Module* contains additional FIFO buffers and pipelining logic. The FIFO buffers are required by the master interface of the *Data Interconnect Interfacing Logic* module in order to allow burst transfers. The pipelining logic is required by the slave interface of the *Data Interconnect Interfacing Logic* module to allow for burst transfers to and from the storage located within the RR as transfers over the SR-RR boundary undergo additional latency.

Translation Table (TT). The communication channels are accessed through the channel interface modules by providing their IDs. These IDs are assigned globally and used by all channels in the system. Locally, within each HW task, channel interface modules are assigned index numbers. The index numbers are used to facilitate mapping of channel interfaces on both sides of the channel to their location in the address space.

The *Translation Table* is used to translate the channel index, provided by the master-side interface, into address of the slave-side interface located in another HW task. Thanks to local index addressing of channels, rather than global ID addressing, the size of the table can be reduced to the maximum number of indices allowed per HW task.

Communication Events Control. This module is used to collect the events from all channel interface modules located within a currently allocated HW task and notify the software side of the OS about their occurrence. When it happens, a dedicated interrupt handler, that is, the *Comm Event Interrupt Handler* mentioned previously, is executed. It contacts the

Communication Events Control to retrieve the index of the channel generating the event.

HW Task Management API Call Module. The *HW Task Management API Call Module* serves as a middle man between the HW task and the software side of the OS4RS, whenever the HW task makes a *Management API Call*. It is described in Section 7.2.

DFS Controller. The *DFS Controller* is used to implement the very feature of dynamic clock frequency scaling. It consists of the main part, being the *Frequency Synthesizer* module itself and the wrapping logic built around it. As the *Frequency Synthesizer* module, a Digital Clock Manager (DCM) [54] or Phase-Locked Loop circuits common in modern FPGAs may be used. The wrapping logic provides the OS4RS with an interface to control the process of frequency scaling.

DPR Controller. The *Dynamic Partial Reconfiguration (DPR) Controller* is used to control the low-level aspects of reconfiguration and preemption of HW tasks. This includes isolation of the SR-RR boundary during reconfiguration and preemption as well as clock suspension.

RR Lock Time-out Timer. The *RR Lock Time-out Timer* is used in implementation of the developed scheduling mechanism. It is a simple count-down counter, started and stopped by the OS4RS, which generates an interrupt once its reaches value zero.

3.4.3. HW Task: Architecture. From the architectural point of view, the *HW Task*, shown in the bottom right corner of Figure 4, is composed of a *HW Task Core*, *Communication Channel Interface Modules* (referred to as channel interface modules), and the remaining HW task glue logic. The *HW Task Core* represents a computational part of the *HW Task*, which can be either synthesized from a high-level description using an HLS tool or already provided in form of an HDL description.

The *HW Task Core* communicates with an outside world through ports using standardized signal protocols. The protocol can be either nonblocking or blocking. The channel interface modules, shown as white-yellow ovals connected to the *HW Task Core*, are implemented keeping these simple and standardized protocols in mind. Depending on the side of channel where they are located, the channel interface modules can be of two types: master side and slave side. The type defines which module is responsible for starting the transaction over the physical interconnect.

The channel interface modules are part of the *HW Task* and reconfigured along with it. The channel interface modules implement local storage space for the data to be sent or received by the *HW Task Core* over the physical interconnect. They are also the key components interacting with the software side of the OS4RS during initialization and completion of the data transfer over the channel.

The slave-side interfaces of the communication channels are accessed from outside the *HW Task Wrapper* as a part of one contiguous address space defined by the *HW Task Wrapper*. This allows for a fixed interface between the static part of the *HW Task Wrapper* and the RR containing the *HW Tasks*, irrespective of the number and type of channels *HW Tasks* use. The fixed interface is required by the current DPR design flows provided by the major FPGA manufacturers, that is, Xilinx [46] and Altera [47]. As a result, the multiplexers and address decoding logic must be a part of the *HW Task*'s glue logic.

If the number of master-side interface modules is bigger than one, the *HW task* glue logic will also contain a lightweight arbiter controlling access of those interface modules to the *HW Task Communication Module*, in turn, giving them access to the physical interconnect.

The *API Call* channel, shown in Figure 4, is a special kind of channel. It is used by the *HW Task Core* at the time of making the *Management API Call*.

The *HW Task* works in two, potentially unrelated clock domains, that is, *Fixed Clock (FCLK)* domain and *Variable Clock (VCLK)* domain. The *FCLK* domain is controlled by a system-wide clock, whose frequency is fixed during operation of the system. On the other hand, the clock in the *VCLK* domain can be dynamically adapted depending on the performance and power consumption needs of an application, provided that the maximum frequency of operation of the related logic allows for it. The *HW Task Core* works in the *VCLK* domain, whereas the glue logic works in the *FCLK* domain, just as the *HW Task Wrapper* does. The channel interface modules work partly in the *FCLK* and partly in the *VCLK* domain. They encapsulate the details of clock domain crossing to the *HW Task* glue logic and the *HW Task Core*.

3.4.4. HW Task: Architecture's Pros and Cons. The ability of the *HW Task Core* to operate at different frequencies solves many issues and lets designs avoid the limitations of the DPR technology. The *HW Task Cores* which can work at much higher frequencies than the rest of the system are not restricted by it anymore. On the other hand, the *HW Task Cores* which may have problems in achieving frequencies defined by the rest of the system will also not impose any limitations on the system itself. Although in the cases above, bridge logic could be used on the interconnect, the fixed clock in the RR would still make the slowest HW Task limit the performance of the other tasks that may be allocated to this RR. The downside of this approach is, however, increased latency of data communication.

A fundamental question about the architecture of the proposed *HW Task*, that may arise, concerns placement of the channel interface modules within the RR. Since it is better to transfer data directly between HW tasks rather than using shared memory as an additional middle man, the channel interfaces and the related storage are not placed outside the *HW Task Wrapper*. It will be especially beneficial in case of Network-On-Chip (NoC) where latencies associated with transferring data between nodes may be high.

The *HW Task Wrapper* contains logic common for all the HW Tasks. Common local storage for all the channel interface modules can be a bottleneck. Also, number and type of used channels are specific to each HW task and

thus may vary among different tasks. Since we want the communication to be customized on per HW task basis, the channel interface modules are not placed in the *HW Task Wrappers*.

Although, in Figure 4, slave-side interface modules share the access to the interconnect, multiple other HW tasks may prefetch the data to storage contained in these modules, one after another, while the *HW Task Core* is executing. This way the communication and execution of the whole application composed of SW and HW tasks may be overlapped. Besides, more interfaces could be provided depending on the performance requirements of a given system. Since the current state of channels constitutes the context of the executing HW Task, the placement of the interface modules within the RR also facilitates HW task's preemption. The developed architecture does not support HW task preemption when the actual transfer over the channel is in progress, that is, the preemption is possible only once the whole data transferred over the channel has been stored in the storage element of the interface module.

Another question may concern the decision about the location of the boundary of *FCLK* and *VCLK* domains. Its location within the *HW Task* rather than at the SR-RR boundary was dictated by performance reasons. For improved throughput of streaming data communication across two unrelated clock domains, asynchronous FIFOs provide significant advantage over the double-flopping synchronization method. While the latter one could be easily adopted to match the requirements of the SR-RR interface, it is not the case with the asynchronous FIFOs. For this reason, we decided to place the clock domain boundary within the *HW Task*.

A downside of the proposed approach may be increased size of HW tasks and, as a result, their longer reconfiguration time. However, our evaluation results in Section 9.4 show that, in many cases, the increase in logic area due to the additional logic is not high when compared to the size of the computing *HW Task Cores*.

4. Base OS Kernel

As a base OS kernel for the implementation of the OS4RS, we used the Toppers ASP RTOS kernel [24]. It mainly targets highly reliable small-scale real-time embedded systems, although memory protection extension is also available, making it applicable to larger scale systems. Toppers ASP kernel is of a small size which results in low memory usage. It follows a one object linking model, where both the kernel and the application are merged together and run in a privileged CPU mode.

ASP kernel provides basic services for the application composed of SW tasks, such as task state management, scheduling, and inter-task communication and synchronization. It implements a preemptive-priority scheduling and a Task Control Block (TCB) model, where special structures, that is, TCBs, are used by the kernel to store information related to task's execution state, its priority, and so forth. The kernel provides basic communication/synchronization primitives for SW tasks, that is, data queues, mailboxes, semaphores, and event flags, to name a few [9]. It also provides means of task-dependent synchronization, to allow an SW task to block or unblock another task by executing a system call.

The kernel is highly portable as the large part of it is CPU architecture independent. Only small part related to low-level system initialization, interrupt processing, and task dispatching depends on a given processor's architecture. Although a wide gamma of CPU architectures is currently supported by the ASP kernel, the Power PC (PPC) 405 processor being part of the FPGA used in our tests was not. For this reason, we had to port the architecture-dependent part of the kernel to the PPC 405 processor's architecture.

5. HW Task Configuration and Clock Scaling

The *HW Task Configuration Layer* implements a model of a HW task state accessible by means of the Configuration Port Access (CPA) approach presented in [48, 55]. In this approach, the state is described by complete configuration frames rather than separate bits within the frame. While it may lead to redundancy, it is a much more portable approach, which leads to more efficient implementation of a preemption mechanism. In this approach a mapping between the state-containing frames, stored in the bitstreams, and those in FPGA's Configuration Memory (CM) is established and used afterwards by the save and restore functions. We developed a tool, called Bitformatter, whose role is to analyze the configuration bitstream representing a given HW task and to extract all the state-related information, creating a sort of extension containing the information about the aforementioned mapping of the state frames. The extended bitstream follows a developed DBF (Dynamically partially reconfigurable system's Bitstream Format).

The DBF file is divided into six sections, as shown in Figure 5. *Main Header* corresponds to the header of the original configuration file, *State Data Descriptor Table (SDDT) Header* contains information about the *SDDT* itself, whereas each *State Data Descriptor (SDD)* is composed of three-word elements. The first one is the offset of the first frame which contains the state information, calculated versus beginning of the configuration bitstream. Knowing the base address of the configuration bitstream, we can calculate the address of the state-related frame within the HW task bitstream repository. The second word in the *SDD* indicates the number of words in the state frames. Finally, the third word in the *SDD* is the start address of the first frame in the CM. *Primary Configuration Bitstream* section represents the original configuration bitstream that may be slightly modified for the purpose of preemption [48, 55]. *FF Init Data* bitstream is used to reinitialize and restore the state of all tasks, whereas *Mem Init Data* bitstream only is used to reinitialize the state of a memory-based HW tasks.

5.1. Configuration Controller and HTIC Bus

5.1. Configuration Controller and HTIC Bus. ICAP-DMA, presented in [48, 55], is a reconfiguration/readback controller which also controls the physical aspects of HW Tasks' reconfiguration and preemption. The controller's architecture allows for burst transfers of both configuration and readback data, which results in significantly improved reconfiguration

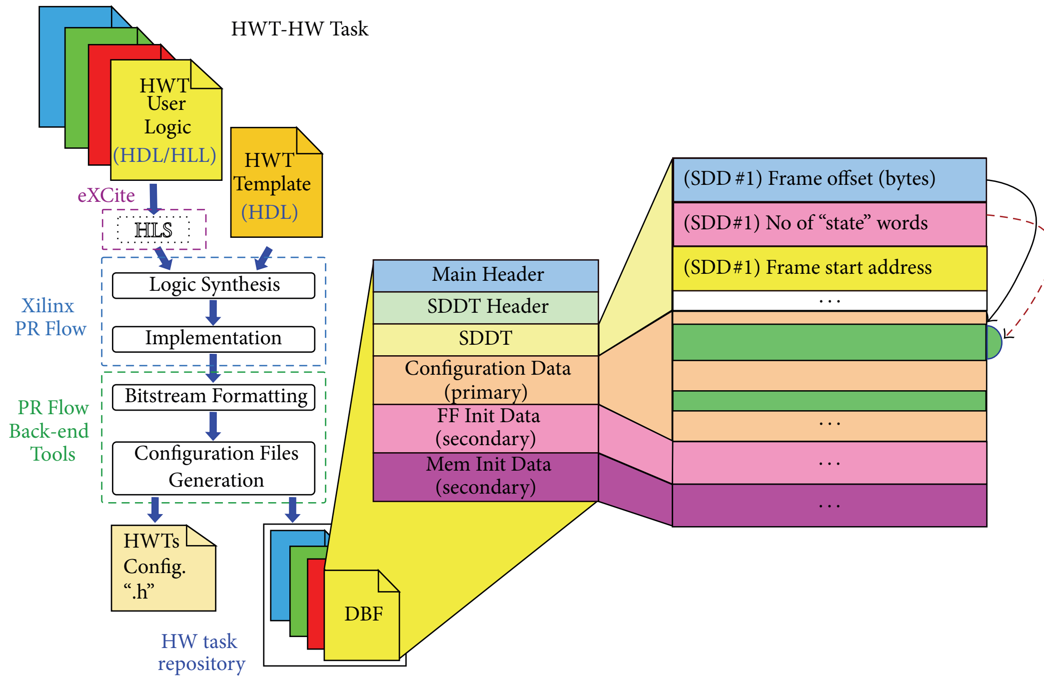


Figure 5: DBF file format.

and preemption times. Moreover, it allows for buffering of all requests related to reconfiguration and preemption, thereby reducing CPU utilization. Management of the physical aspects of the reconfiguration and preemption is done via the *HTIC* bus which connects the *ICAP-DMA* with the *DPR Controller* inside the *HW Task Wrapper*.

The *HTIC* bus has a simple, one master (*ICAP-DMA*), multiple slaves (*HW Task Wrappers*) topology and is a write only bus. Every *HTIC* bus slave interface has a unique RR ID assigned at a system design time and used later on when being addressed by the *ICAP-DMA*.

5.2. DPR Controller. In the developed method [48, 55], saving of the HW task's state is based on readback of the configuration bitstream from the FPGA's CM and its filtering, in order to retrieve the task's state. Restoring of the state is accomplished by configuring the CM with the bitstream containing previously saved state and assertion of the reset signal. The last one brings back all the storage elements constituting the HW task to their previously saved state. As the mechanism presented in [48, 55] was only suitable for synchronous logic, some additions were needed to make it work with the clock-scalable HW tasks.

The *DPR Controller* is composed of three modules: a *Parent Control Module* and two *Child Modules*. The *Parent Module* and one of the *Child Modules* work in the *FCLK* domain, whereas the other *Child Module*, in the *VCLK* domain. Communication between the parent and its children is based on sending request signals and receiving feedbacks. These handshaking signals have to be additionally synchronized for the case of *Parent* and the *VCLK Child Module*. After the control Finite State Machine (FSM) in the *Parent Module* sends the request, it waits for feedback, before moving to the next state. In order to support dynamic frequency scaling, the clock in the *VCLK Child Module* has to be suspended whenever the DCM generating the clock is being reconfigured. For this reason, the *DPR Controller* is not used at that time.

In order to make sure that upon restoring of the task's state both *FCLK* and *VCLK* domains come out of reset simultaneously, the *Parent Module* has to adhere to the following order of steps. First, it requests both *Child Modules* to assert reset, then to switch on clocks, so that the both domains are reset, then switch off the clocks back, and finally deassert the reset. Since the *Parent Module* waits for the feedback when following the aforementioned procedure, we ensure that the reset is asserted for long enough to reset the domain with much slower clock than the static part. Apart of switching of resets and clocks, the *DPR Controller* is also responsible for enabling and disabling the SR-RR interface as well as enabling and disabling memory accesses within the HW task. The corresponding steps are not given for the sake of simplicity.

5.3. DFS Controller. Our implementation of the *DFS Controller* is based on the DCM module available on Virtex-4 FPGAs, which we used as the *Frequency Synthesizer*. The wrapping logic built around the DCM contains a control FSM which directly interfaces the Dynamic Reconfiguration Port (DRP) of the DCM [54]. Further details on DCMs and different ways of interfacing them can be found in [42, 43, 54].

The wrapping logic also implements a set of registers: *Clock Multiplier*, *Clock Divider*, *Control*, *Status,* and registers for enabling and acknowledging interrupts. First two registers control the frequency of the clock generated by the DCM module with respect to its input clock. *Control* and *Status* are

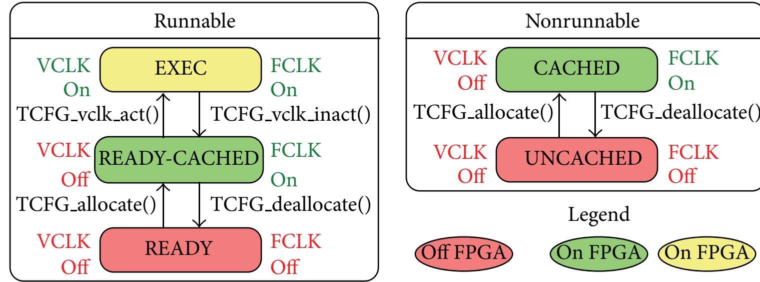


Figure 6: HWT State (HWT Scheduling and Placement Layer perspective).

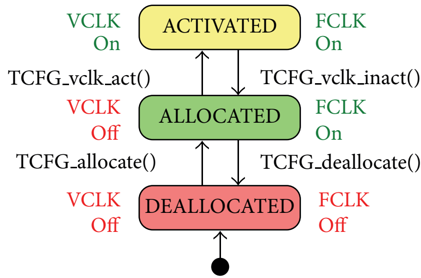


Figure 7: RR State (HWT Scheduling and Placement Layer perspective).

used to request dynamic clock frequency scaling and to check its current progress status. Finally, interrupt-related registers control generation and acknowledgment of the interrupt indicating completion of the scaling operation.

Although logically, the DCM-based *DFS Controller* (*DCM Controller*) is a part of the *HW Task Wrapper*, physically it is connected to it externally but still being controlled by the *HW Task Wrapper*. This is done to simplify application of timing constraints on the *VCLK* generated by the DCM.

6. HW Task Scheduling and Placement

6.1. HW Task and Reconfigurable Region' State. While above the *HW Task Scheduling and Placement Layer* tasks are only seen as *Runnable* and *Non-runnable*, this layer makes an actual decision upon which HW task to allocate and start execution and is responsible for managing caching of HW tasks. HW task's states visible from the perspective of this layer are shown in Figure 6. *READY* and *UNCACHED* are used to describe deallocated HW tasks. HW tasks in *READY-CACHED* and *CACHED* states are allocated, but only their part located in the *FCLK* domain is operating, whereas the computational part, that is, the *HW Task Core,* is off. Finally, in *EXEC* state, both clock domains of the HW task are operating.

While maintaining the state of HW tasks, the *HW Task Scheduling and Placement Layer* is also responsible for managing the state of the RRs where the HW tasks are allocated. The RRs can be in one of three states: *DEALLOCATED*, *ALLOCATED,* and *ACTIVATED*. They are shown in Figure 7. The RR in the first state is ready for new allocations, in the second state is already allocated with some HW task which is currently not executing, and in the third state is allocated with some HW task which is currently executing.

The labels near transition edges in Figures 6 and 7 denote the calls made to the *HW Task Configuration Layer* in order to realize a given state. *TCFG_allocate()* and *TCFG_deallocate()* denote HW tasks' allocation and deallocation requests for a given RR, and they additionally cause the *FCLK* to be switched on and off, respectively. *TCFG_vclk_act()* and *TCFG_vclk_inact()* denote requests for switching the *VCLK* clock in the *HW Task Core* on and off, respectively.

6.2. Preemptive Scheduling with Reservation of Reconfigurable Resources. The *HW Task Scheduling and Placement Layer* maintains *Ready Queues* and a *Priority Map* for each RR. The *Priority Map* is used in speeding up searching of the highest priority task located in these queues. The *Ready Queues* (Figure 8) link together all *runnable* tasks and one task which was granted lock on the RR resource. The latter one is kept in the first entry, in one of the queues even if it becomes *Non-runnable*. This way, tasks which block only for a short time can be kept allocated on the FPGA and restarted without reconfiguration overhead. Whenever a task acquires the lock, its priority is elevated to the priority of the resource, in case it was lower than it, or remains unchanged otherwise. If the RR's priority is set to the highest possible value, no task will be able to preempt that which holds the lock, resulting in a nonpreemptive scheduling. Setting priority of the RR to any lower value will allow some critical task to preempt the locked task. The lock will be restored when the critical task finishes execution.

A HW task may request the lock by means of *Rbow_hwt_req_RR_lock()* API call; however, granting of the lock will occur only when it is selected by the scheduler as shown in Figure 8. The lock may be requested with an additional time-out value or *EXIT REL FLAG* which makes the HW task releases the lock upon completion of its execution. The timely lock, which expires after a specified amount of time, may be used as a deadlock [56] recovery measure. It may be also used to prevent unnecessary preemptions in certain phases of execution of an application, still taking advantage of the preemption while waiting for some data from external source. Whenever a task which was granted the lock with time-out becomes *NON-RUNNABLE*, that is, idle, a *RR Lock Time-out Timer*, located in the *HW Task Wrapper*, will be started. When the counter's value

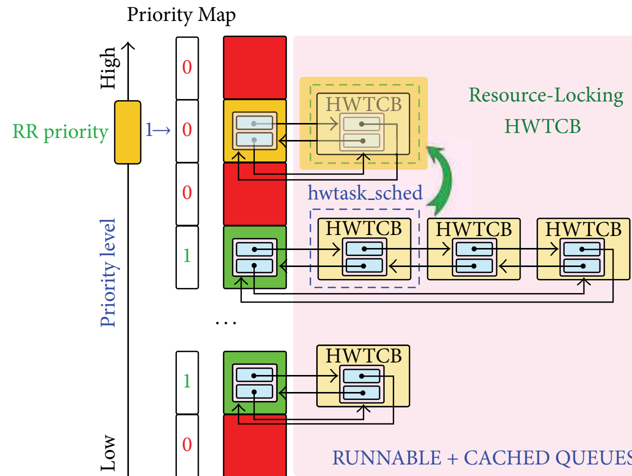


Figure 8: HWT Ready Queues and RR locking.

reaches zero, the executed interrupt handler will release the lock, possibly making the other *RUNNABLE* task allocated.

6.3. Delayed Execution of Configuration and Preemption Requests. The processing done by the *HW Task Scheduling and Placement Layer* can be divided into two steps. The first one is done at the caller's side; that is, it is implemented by a function of the *HW Task Scheduling and Placement Layer* called by the upper layer as a result of executing an API call. This processing consists of state management and scheduling. The second step is done by the *HW Task Dispatcher* task, which is started in the first step whenever the allocation, deallocation, or clock management is requested.

The main job of the *HW Task Dispatcher* is to process all these requests and call appropriate functions of the *HW Task Configuration Layer* to realize them. The *HW Task Dispatcher* is implemented as a highest priority task in the base OS in order to make the necessary allocations, deallocations, or clock management operations start with a minimum delay.

The interaction between the scheduler and the *HW Task Dispatcher* is based on the idea of pointers to the *HWTCB* of the scheduled task and currently allocated task as well as additional *Dispatch Flags*. The pointer corresponding to the scheduled task is maintained and assigned by the scheduler, whereas the pointer corresponding to the currently allocated task is assigned by the *HW Task Dispatcher*. When the *HW Task Dispatcher* starts, it compares the pointers and starts allocations and deallocations accordingly. When it completes, it assigns the scheduled task's pointer to the currently allocated task's pointer. The *Dispatcher Flags* are used to signal additional requests related to clock management, while the aforementioned pointers are equal.

Since the allocations, deallocations, and clock management operations may take some time to be completed, the state maintained by the *HW Task Scheduling and Placement Layer* can be pending for that period of time. The state and a *Pending Flag* are set in the first step, mentioned above, whereas the actual realization of that state and clearing of that flag are done within the *HW Task Dispatcher*.

The main advantage of the aforementioned division in processing is that API calls which do not result in requests for allocation, deallocation, or clock management are very short. Moreover, when the requests are actually being processed, the SW and HW tasks may continue their execution; that is, their processing is overlapped with the processing performed by the *HW Task Configuration Layer*; it is shown in Figure 9. In the figure, the *Send Request to Configuration Controller* corresponds to processing done by the *HW Task Configuration Layer*. It sends the reconfiguration and readback requests to the *Configuration Controller* and then blocks, waiting for the processing to be completed. This interaction is based on semaphore, acquired by the *HW Task Configuration Layer* and released by the dedicated *Configuration Controller* interrupt handler. In case of clock scaling requests, the *HW Task Configuration Layer* talks to *DFS Controller* module (DCM Controller), following similar procedure as in case of *Configuration Controller* (ICAP-DMA). The *Before Inact Callback* and the *After Act Callback* refer to function callbacks of the upper layer described previously in Section 3.3.3

The *Init Req Processing* corresponds to accessing of the structures related to the request, shared between the *HW Task Dispatcher* and the scheduler. At this time, also the decision about which functions of the *HW Task Configuration Layer* to call is made. This decision is based on checking the status of the *Dispatch Flags* and comparison of the pointers of the scheduled and currently allocated task, described before. In the *Final Req Processing*, the assignment of the pointer corresponding to the currently allocated task and clearing of the *Pending Flag* is conducted. The latter one is performed, provided that no further requests have been made, while the processing which has now completed was in progress.

The actions in the *Init Req Processing*, the *Final Req Processing*, and those in the *Callbacks* are performed in the state of disabled interrupts, enforcing exclusive access to the shared OS4RS structures.

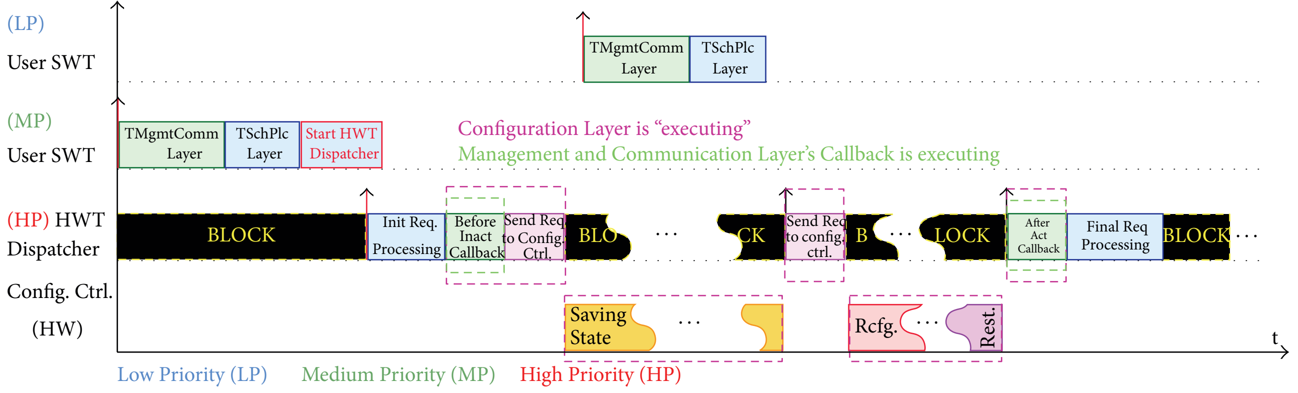


FIGURE 9: Delayed execution of configuration requests.

7. HW Task Management

7.1. HW Task Management API. The Table 1 shows the *Management API* Calls for SW and HW tasks. We provided API Calls for SW tasks as C preprocessor macros which reference the functions of the *HW Task Management and Communication Layer*. We also provided implementation of these calls for HW tasks coded in C and synthesized using eXCite [57] HLS tool. These are converted into states of the control FSM accessing ports of the synthesized *HW Task Core* module, mentioned in Section 3.4. Outside the *HW Task Core*, the ports are connected to the interface of the *Management API Call Channel*, described later on in Section 7.2.

The calls with∗ can be only executed by the SW task caller. The restriction for *Rbow_swt_exit()* is natural as it is a voluntary exit call to be made only by the SW tasks. The restriction for the other calls comes from the fact that the calls made by the HW task caller are actually executed on its behalf by a dedicated interrupt handler. Currently, the base OS implementing SW multitasking does not provide support for these calls made within the interrupt context. Extension of the base OS is a part of a future work.

While the majority of calls in the Table 1 are typical for conventional OSes [9, 24], the other ones have been specifically designed for the HW tasks. *Rbow_hwt_acq_RR_lock* and *Rbow_hwt_rel_RR_lock()* are used to reserve and release the reconfigurable resources for a given HW task (RR Locking), described in Section 6.2. The *Rbow_hwt_prefetch()* call is used in HW task prefetching, that is, putting it into a *PREFETCH* state, described in Section 7.3. The actual activation of the task put into the *PREFETCH* state is done when the *Rbow_hwt_prefetch_act()* call is executed. Another special call, *Rbow_hwt_updt_freq()*, is used to dynamically update the clock frequency of a HW task. If the new requested frequency is within the range of frequencies supported by the HW task and the *Frequency Synthesizer* module itself, the clock frequency scaling will be requested from the *HW Task Configuration Layer*. The new frequency is calculated using the clock frequency multiplier (*MULT*) and divider (*DIV*) parameters passed to the call as well as the input clock frequency of the *Frequency Synthesizer* module, given as a system-wide constant.

TABLE 1: Management API.

SW tasks	HW tasks
—	Rbow_hwt_prefetch (HWT ID)
—	Rbow_hwt_prefetch_act (HWT ID)
Rbow_swt_act (SWT ID)	Rbow_hwt_act (HWT ID)
Rbow_swt_term (SWT ID)*	Rbow_hwt_term (HWT ID)
Rbow_swt_exit ()*	Rbow_hwt_exit (HWT ID)
Rbow_swt_chg_pri (SWT ID, NEW PRI)*	Rbow_hwt_chg_pri (HWT ID, NEW PRI)
Rbow_swt_susp (SWT ID)*	Rbow_hwt_susp (HWT ID)
Rbow_swt_resm (SWT ID)*	Rbow_hwt_resm (HWT ID)
Rbow_swt_rotate_queue (PRI)	Rbow_hwt_rotate_queue (RR ID, PRI)
—	Rbow_hwt_acq_RR_lock (HWT ID, EXIT REL FLG, TMOUT VAL)
—	Rbow_hwt_rel_RR_lock (HWT ID)
—	Rbow_hwt_updt_freq (HWT ID, MULT, DIV)
—	Rbow_hwt_wait_state (HWT ID)
Rbow_ch_lock_acq (CH ID)	Rbow_ch_lock_acq (CH ID)
Rbow_ch_lock_rel (CH ID)	Rbow_ch_lock_rel (CH ID)

*Only supported by the SWT caller.

Some of calls may result in long HW task's allocation or clock scaling process. As mentioned in Section 6, these are done in background, while other tasks continue their execution. As a result, by the time the API call returns to the caller, and the allocation or clock scaling will be still in progress. This is done to allow the caller for further execution without waiting for the time-consuming process to complete. In some situations, it may be required to synchronize the further task's execution with the completion of the time-consuming reconfiguration. This is where the *Rbow_hwt_wait_state()* call comes into play. This call will only make the caller wait if the previous call resulted in any of the aforementioned time-consuming processes and otherwise

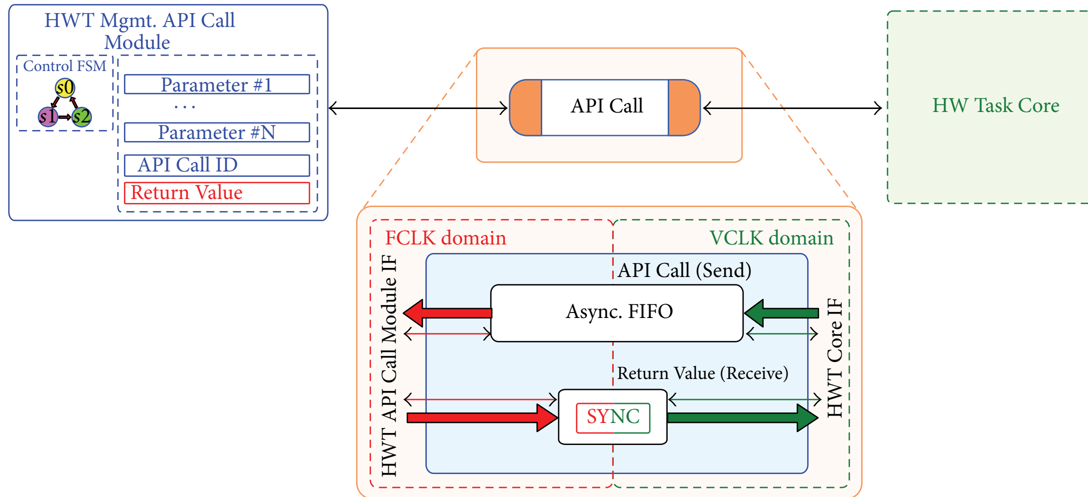


FIGURE 10: Management API Call Channel.

will return immediately. It is currently supported only by the SW task caller. It should be noted that communication calls, presented later in this paper, do not require this additional synchronization call as they always block if the conditions for the communication are not met.

Finally, the *Rbow_ch_lock_acq()* and *Rbow_ch_lock_rel()* calls are used in tandem with the RR Locking and only when the HW task is involved in communication. Although they are related to communication, they logically belong to the group of *Management API Calls*. The first call acquires a lock on a channel, whereas the second one releases it. The channel lock is an idea, described in details in Section 8.4.1, which optimizes the communication by reducing OS4RS processing overhead.

7.2. Processing of HW Task Management API Call

7.2.1. Management API Call Channel. A *Management API Call Channel*, shown in Figure 10, is a special kind of channel used by the *HW Task Core* to pass the information about the *Management API Call* it makes, that is, API call ID and its parameters, to the *Management API Call Module*, located in the *HW Task Wrapper*, and then get the return value of the call when it completes. The API call ID and the parameters are first buffered in the FIFO structure of the channel and then read by the *Management API Call Module*. The buffer is used to decrease the latency of sending the API call-related information across the asynchronous clock domains. The return value is passed to the *HW Task Core* using a standard double flopping synchronization scheme.

7.2.2. Processing of Management API Call. While SW task callers make the OS4RS service calls by directly calling the API functions of the Rainbow extension, this process is different for HW tasks. It is shown in Figure 11.

Firstly, the *HW Task Core*, being part of the HW task, sends the ID of the call and its parameters to the *API Call* channel. The control FSM of the *Management API Call Module* constantly checks for presence of data in the channel. When the *HW Task Core* sends the data, the control FSM detects it and notifies the software side of the OS4RS about it by generating an interrupt. As a result, a dedicated interrupt handler, called *Management API Call Interrupt Handler*, is executed which grants the request in the *Management API Call Module*. This starts prefetching of the data from the channel to the internal storage of the *Management API Call Module*. The software side of the OS4RS waits till the prefetching is over and then reads the ID and parameters of the call from the *Management API Call Module*. The API call made by the HW task is then executed on its behalf and return value sent back to the *API Call* channel. This completes the calling process for the HW task caller. However, if the call made by the HW task results in its preemption or blocking, then the return value is saved, at this time, in the *HWTCB* and passed to the *HW Task Core* later on, upon its allocation.

7.3. A HW Task State. Figure 12 shows the HW task state diagram from the perspective of the *HW Task Management and Communication Layer*. *ACT, TERM, EXIT, SUSP, RESUME, WAIT, and REL_WAIT* labels near the transition edges indicate a corresponding action requested from the *HW Task Management and Communication Layer*, which results in the state transition. *ACT* corresponds to task activation, *TERM* corresponds to its forced termination, and *EXIT* corresponds to voluntary exit, whereas the *SUSP, RESUME* correspond to blocking and resumption as a result of execution of the task-dependent synchronization calls. Finally, *WAIT* and *REL_WAIT* denote actions requested as a result of communication of the HW task over channels.

The *PREFETCH* state being part of the *RUNNABLE* macrostate is a special state. In this state, the SR-RR interface, mentioned in Section 3.4, is enabled, but only the part of the HW task operating in the *FCLK* clock domain is switched on. On the other hand, the computational part, that is, the *HW Task Core*, is off. For this reason, the channel interface modules and their storage elements are accessible from

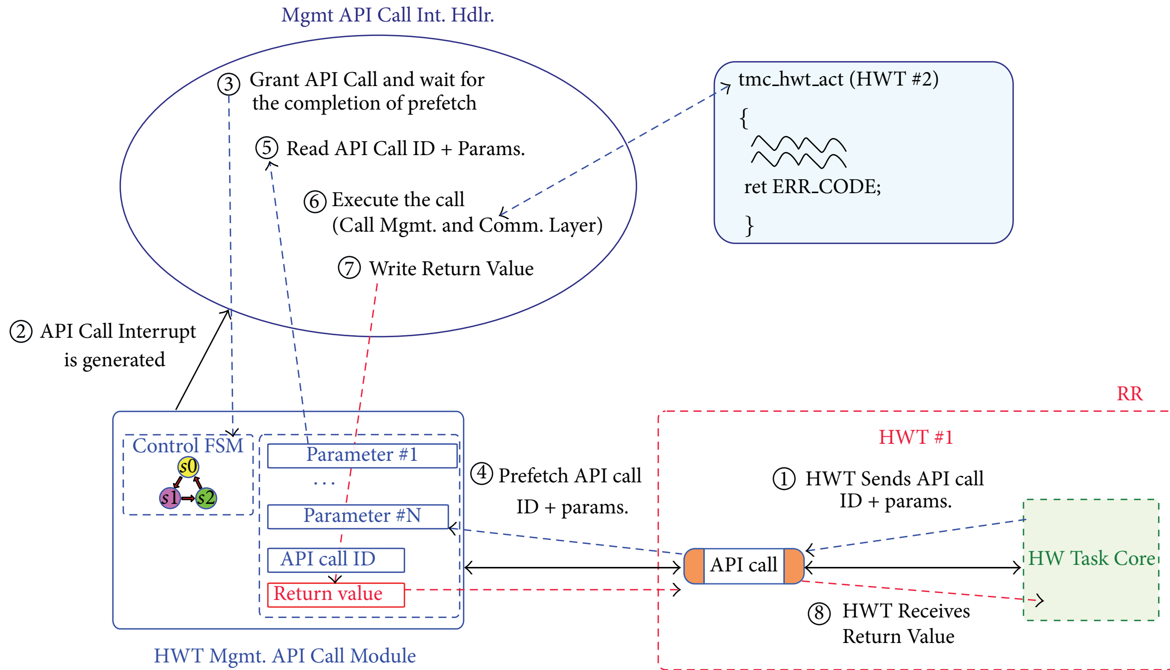


Figure 11: Processing of Task Management API Call by an HWT caller.

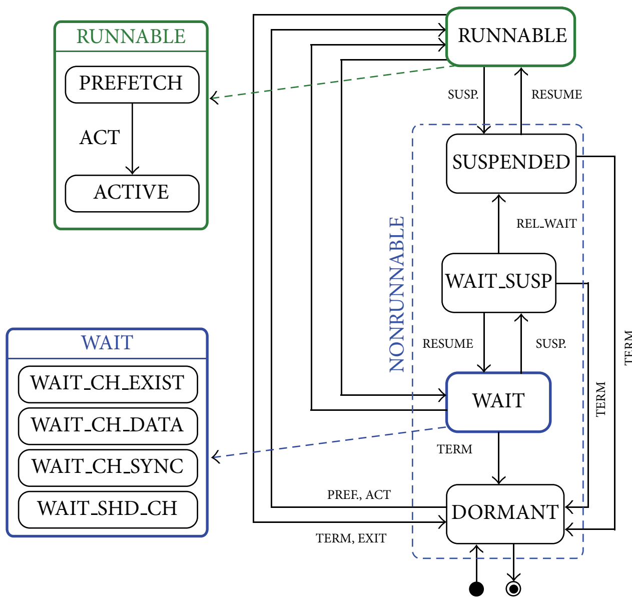


Figure 12: HWT State (HWT Management and Communication Layer perspective).

outside of the HW task. These conditions allow other tasks to prefetch data to the channel interface modules before the *HW Task core* starts executing and so cannot possibly block on any API calls and be deallocated.

The *WAIT* macrostate is composed of *WAIT_CH_EXIST*, *WAIT_CH_DATA*, *WAIT_CH_SYNC*, and *WAIT_SHD_CH* states. These states indicate waiting condition on the communication channel. The first one indicates that the HW task is waiting for the other side of the channel, that is, its interface module, to become allocated. The second one means that the HW task either waits for the storage element being part of the channel interface on the other side of the channel to be empty, so that the data can be written to it, or waits for the storage element on its side of the channel to filled, so that the data can be read from it. The third one is used when the HW task communicates over the Message (MSG) channel, where a sending HW task is blocked and put to *WAIT_CH_SYNC* till the other side executes the complementary API call and receives the data. Finally, the *WAIT_SHD_CH* state is used whenever the P2P communication over the channel is not possible and an intermediate shared memory, called *Shadow Channel,* has to be used. As the communication over the *Shadow Channel* takes more time than the P2P communication, it may happen that the HW task will try accessing the channel again, before the previous communication completes. If that happens, the HW task will be put into *WAIT_SHD_CH* state. Details on the intertask communication and synchronization are given in Section 8.

8. Intertask Communication and Synchronization

8.1. Overview. The inter-task communication and synchronization implemented by the presented OS4RS is based on the concept of channel. In our opinion, it is more suitable for SW-HW multitasking systems than previous approaches which directly follow the model of communication and synchronization based on data queues and semaphores, known from traditional SW multitasking OSes.

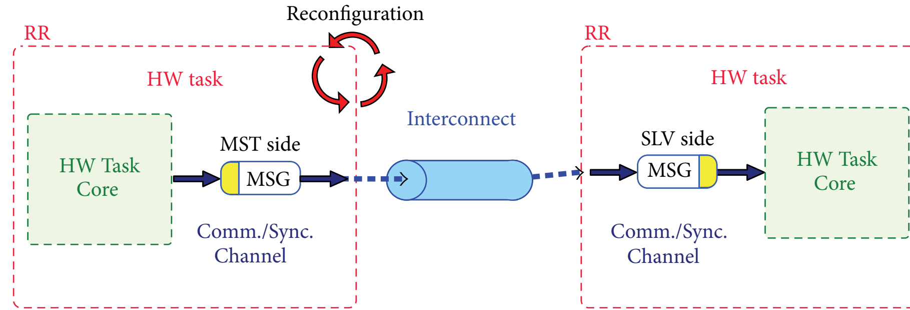


Figure 13: Channel with reconfigurable interfaces.

The channel provides interface for tasks to communicate with other tasks, located either in software or hardware. The data sent to the channel is transfered over the physical interconnect provided by the hardware platform before it can be received by another task. Multiple channels can share the same interconnect.

We have implemented four types of channels which can be used by SW and HW tasks. These are *FIFO*, *MSG*, *Shared Memory (SHARED MEM),* and *Register (REG)* channels. The *FIFO* channel implements FIFO like, unidirectional, P2P communication. In this style of communication, the receiver blocks only if the sender has not sent the data yet (FIFO empty condition), while the sender is able to send as much data as is the depth of the channel, before it blocks (FIFO full condition). The *MSG* channel implements a unidirectional, P2P, rendezvous style of communication where the receiver and the sender block till their communicating partner makes a corresponding *Communication API Call* and sends or receives the data. The *SHARED MEM* channel implements a nonblocking, bidirectional communication based on the storage located externally to tasks. Finally, the *REG* channel implements a nonblocking communication and is used for polling-based synchronization of SW and HW tasks communicating over the *SHARED MEM* channel.

8.2. HW Task Wrapper: Implementation Details. As mentioned previously in Section 3.4, HW Tasks access communication services of the OS4RS via a dedicated interface, that is, the *HWT OSIF*, which consists of the *HW Task Communication Module*, the *HW Task Communication Event Control*, and the *Translation Table* modules located in the *HW Task Wrapper*.

The proposed communication model has been implemented on top of IBM CoreConnect buses. The Processor Local Bus (PLB) is used for data transfers, whereas the Device Control Register (DCR) bus is used to control the HW task's OS interface. Nevertheless, it is also suitable for other types of interconnects. The benefits of the model will be especially visible in case of NoC, which promotes P2P communication and where communication latencies between nodes may be high.

Channel interfaces use ID addressing globally and index addressing locally, within the HW task. The maximum number of channel indices allowed in the HW task was set to 16 which made it possible to reduce the size of the *Translation Table* to 16 entries. The table was implemented with LUT-RAMs allowing for index to address translation within one clock cycle after the translation request is sent by the *HW Task Communication Module*.

8.3. Hardware Side View

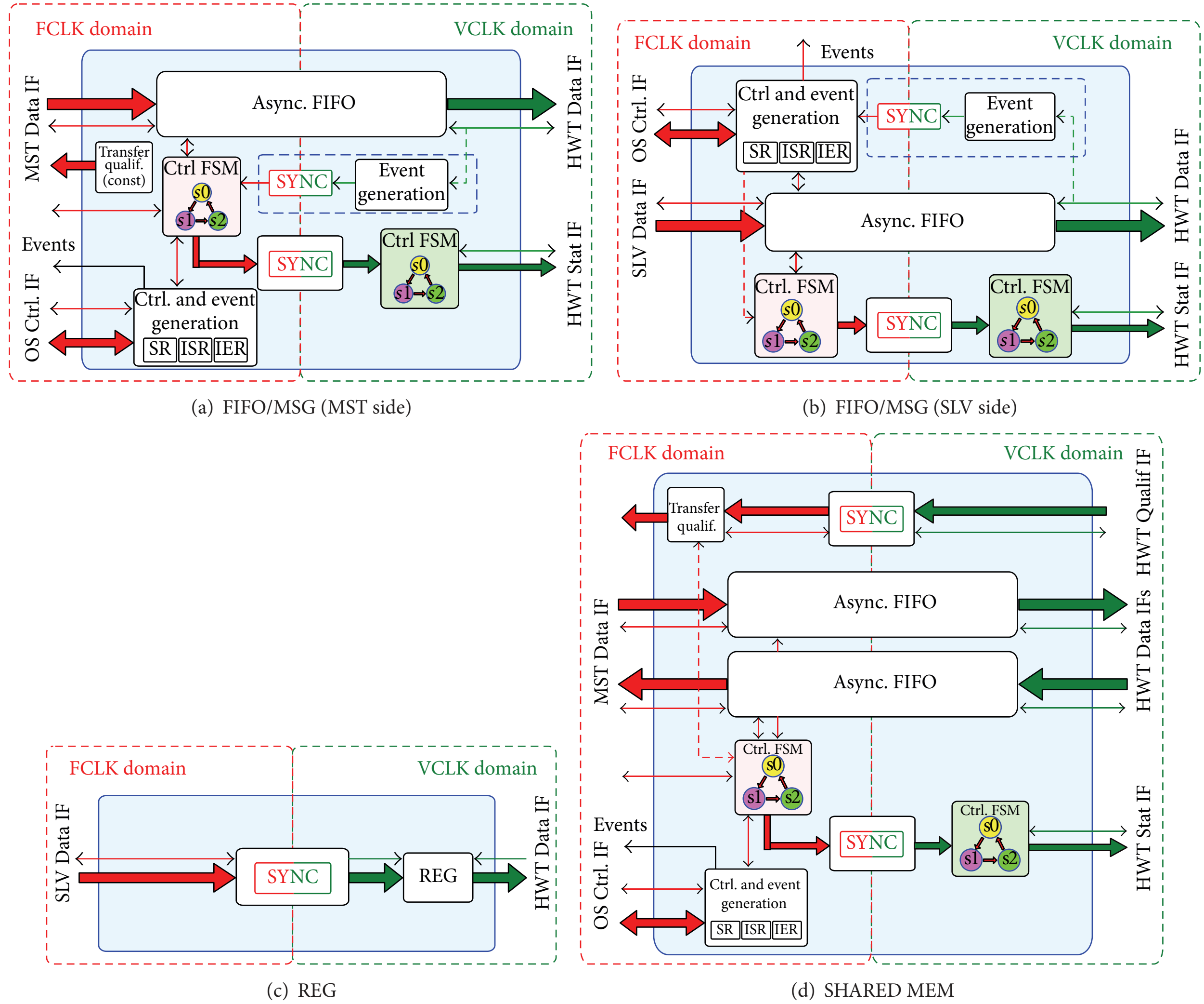
8.3.1. Channel with Reconfigurable Interfaces. The channel interface used by a HW task is a hardware module which lets the task access the channel with an easy protocol while taking care of the intricacies of conducting the actual communication over the channel. The channel interface has its local storage used to buffer the data received over the channel, before it can be accessed by the task or data that is written by the task, before sending it over the channel.

One of the ideas used in the developed inter-task communication and synchronization mechanism is the one of a *reconfigurable channel interface*, shown in Figure 13. The *reconfigurable channel interface* is an interface allocated and deallocated together with the HW task which uses it.

As interfacing modules, containing local storage, are part of the HW tasks, the P2P communication between two already configured HW tasks is possible. Moreover, thanks to this specific location of the channel interfaces, their number, type, and size of their local storage can be defined based on communication needs of the task. As in the developed OS4RS, the *HW Task Core* and the remaining part of the HW Task use separate clocks, prefetching of data to the channel interfaces is still possible, while the computational part is halted.

8.3.2. Hardware-Side Channel Interface Modules. The channel interfaces for HW tasks are implemented by the hardware modules presented in Figure 14, referred to as *channel interface modules* or simply *channel interfaces*. The *channel interface modules* have two sides: the interconnect side and the *HW task Core* side. For the presented modules, the interconnect side is shown on the left side of the module, whereas the *HW task Core* side is shown on the right side of it.

Depending on the type of the interconnect interface, there are two major types of interfacing modules for *FIFO* and *MSG* channels. These are master-side and slave-side modules.



(a) FIFO/MSG (MST side)

(b) FIFO/MSG (SLV side)

(c) REG

(d) SHARED MEM

FIGURE 14: Hardware-side Channel Interface Modules.

SHARED MEM channel interface modules and *REG* channel interface modules are only available as master- and slave-side modules, respectively. The modules for *FIFO*, *MSG* and *REG* channels can be additionally divided based on a direction of the channel, that is, whether they are connected to the sender or receiver. We will refer to them as read-side (receive-side) modules and write-side (send-side) modules. This terminology will be used in the rest of the text. The exception is the interface modules for the *SHARED MEM* channel which is bidirectional.

Channel interface modules operate partly in the *FCLK* domain and partly in the *VCLK* domain. The interface between the asynchronous domains is implemented by means of the asynchronous FIFO and double-flopping synchronization primitives. As such, channel interface modules encapsulate communication across the clock domains.

As mentioned before, all channel interface modules implement a local storage for data. *FIFO*, *MSG*, and *SHARED MEM* channels utilize the asynchronous FIFO, whereas the *REG* channels implement a register. The register is placed in the *HW task Core*'s clock domain to allow one clock cycle to access latency by the *HW task Core*. It is required by the nonblocking access protocols used by the *HW task Cores* implemented by HLS tools. Depth of the asynchronous FIFO is always set to the integer power of two. In order to handle different FIFO depths, additional programmable empty and full flags are provided. Asynchronous FIFOs with depths of less than four are always implemented with depth equal to four. This restriction is imposed by the FIFO design.

Our design of the asynchronous FIFO is based on that presented in [58]. When compared to that design, the clock domain crossing is implemented by means of synchronous FFs in order to support the developed HW task preemption scheme utilized by the OS4RS. The *DPR Controller* module ensures that the reset signal is asserted for long enough to reset the FFs in the slower clock domain.

In case of HW-HW communication, the depth of asynchronous FIFO located in *FIFO* and *MSG* channel interface modules is set to the same value on both sides of the channel. It is referred to as a depth of the channel. This is

done to simplify processing of the *Channel Communication Events* (channel events). The *Channel Communication Events* are interrupts generated by the channel interface modules, whenever the *HW Task Core* starts accessing the channel or state of the local storage implemented by the interface module changes.

The depth of the channel defines the unit of communication over the *FIFO* and *MSG* channels that is, tasks cannot communicate over the channel with finer data granularity. In case of *SHARED MEM* channel, the depth of the buffers is set up depending only on performance requirements of the communication, and it does not define the unit of communication over the channel. The tasks are allowed to define the size of the data to be transfered at each time they want to communicate over the *SHARED MEM* channel. This is achieved by passing an extra argument to the communication API call.

Except of the *REG* channel interface module, all interface modules implement an *OS Control IF* port, used by the software side of the OS4RS to control operation of the modules. This includes generation of the channel events and granting access to the physical interconnect. In Figure 14, parts of the modules related to control which are enclosed in blue dotted lines are only present in read-side modules, whereas signals drawn with red dashed line are only present in write-side interface modules of the *MSG* channel. The *REG* channel does not support events since it implements non-blocking communication and is used only for polling-based synchronization of the *SHARED MEM* channel.

The *FIFO* and *MSG* channel interface modules provide the *HW tasks Cores*, which are connected to them, with two ports: *HWT Data IF* and *HWT Stat IF*. The first one is used to transfer data over the channel, whereas the latter one to receive the status of a transfer and pass it as a return value to the communication API call. *SHARED MEM* channel has an additional *HWT Qualif IF* port used by the *HW Task Core* to provide address offset and size of the transfer. The *REG* channel only contains the *HWT Data IF* port.

OS control part in the write side of the *MSG* channel implements a special *Unblock* flag. After the *HW task Core* sends the data over the channel, it requests the status of performed communication. If the *Unblock* flag is not set, the status read request will not be acknowledged, and the *HW task Core* will be kept waiting. This is used by the OS4RS to delay execution of the *HW task Core* till the other side of the channel starts receiving the data, thereby implementing a rendezvous style of communication.

8.4. Software Side View

8.4.1. OS4RS Structures and Concepts

Channel Control Table. The *Channel Control Table* is the main control structure used by the developed channel-based communication/synchronization model. The table is indexed by the channel ID and is composed of constant and variable parts. The constant part is generated by the design flow tool based on channel configuration information. This information includes channel index (mentioned in Section 3.4), type of channel, whether it is used for SW-SW, SW-HW, or HW-HW communication, channel depth, whether master side is located in software or hardware, direction of the channel, that is, whether master-side is the write-side or the read-side of the channel, and IDs of the communication and synchronization objects provided by the base OS, and The last ones are used to implement SW-SW communication and to implement synchronization between SW and HW side in SW-HW communication. For SW-SW communication, data queues and semaphores are used, whereas for SW-HW communication, event flags are used. The main reason for using the event flags was that several waiting conditions, described later, required by the SW-HW communication could be implemented by separate bits of the event flag.

Since each entry of the table has to handle all possible types of channels, the constant part of the table also contains information about the address of the memory region implementing *SHARED MEM* channel, *Shadow Channel,* and their status flags.

The variable part is updated by the *Communication Agents*, described later on, while communication in the channels progresses. The main fields of this part are related to the current RR location of slave- and master-side channel interface modules and their status, that is, whether they are allocated and whether they are empty or filled. Moreover, it contains the access status of the HW and SW tasks using the channel. A given task may have started the access (*ACCESS_START* condition), may have blocked if the channel interface module on the other side of the channel was not allocated yet (*WAIT_EXIST* condition), may have blocked on writing if the buffer on the other side of the channel was not empty (*WAIT_DATA* condition), may have blocked on reading if the buffer in the channel interface module it is connected to was not filled with the data yet (*WAIT_DATA* condition), may have blocked on accessing the *MSG* channel if the data it had sent was not yet received by the other task (*WAIT_SYNC* condition), or may have blocked on writing if the temporarily setup *Shadow Channel* was still in use (*WAIT_SHD_CH* condition). The access status field allows to control the progress of communication over the channel without a knowledge of the execution state of SW tasks, which is managed by the base OS kernel. It also allows implementation of a concept of *Channel Lock* used to optimize the communication over the channel. All the fields in the variable part are updated dynamically in order to ensure proper access to the channel.

Channel Lock. The *Channel Lock* is an idea which prevents the HW tasks from being put into waiting state even if its access to the channel would result in it. In other words, the *Channel Lock* prevents the *HW Task Scheduling and Placement Layer* from being called. The whole communication is handled by the *HW Task Management and Communication Layer* which utilizes the channel access status flags in the *Channel Control Table* to control the progress of the communication. An executing HW task which communicates over the locked channel appears as if it never blocked. This is useful when the

communication over the channel is made in form of short, but frequent data transfers which could otherwise result in high overhead of calling the underlying *HW Task Scheduling and Placement Layer*. However, a special caution has to be made when developing an application which utilizes these calls in order to avoid deadlock [56] situations.

Configuration Lock. Configuration Lock is used to prevent deallocation of a HW task when the transfer over the channel connected to the task is in progress. The lock is acquired just before the start of the transfer over the channel and is released upon completion of the transfer.

The *Configuration Lock* is implemented using the event flag. The ID of the event flag is stored in the *HWTCB*. The lock will cause the *Before Physical HWT Inactivation Callback*, to block upon execution of the event flag's wait function. It will make the *HW Task Dispatcher*, in whose context the callback is executed, block, resulting in switching to another task. The *HW Task Dispatcher* will be released from waiting on the event flag, when the call releasing the lock, that is, setting the event flag, is executed.

8.4.2. Shadow Channel. Shadow Channel is a region in a shared memory which serves as a temporary data storage for transfers between tasks if the P2P communication over *MSG* or *FIFO* channel was not possible. Currently, *Shadow Channels* are implemented using an external memory. A large capacity of the memory allowed to have one *Shadow Channel* per each channel in the system. The *Shadow Channel* can be set up by one of the *Communication Agents*, described later. It happens whenever a write side of the channel is not able to send data over the channel because the interface module on the other side is not allocated.

The *Shadow Channel* is managed by a dedicated *Copy Task*, described later, which is started whenever data is to be transfered from or to the *Shadow Channel*.

8.4.3. Communication Agents. Communication Agents are all these processing units of the OS4RS which act together in order to provide the inter-task communication services for SW and HW tasks. To achieve this goal, all *Communication Agents* share access to the *Channel Control Table*. The following processing units are called *Communication Agents*: SW task executing communication *SW-side API calls*, *Channel Communication Event Interrupt Handlers*, *Copy Task,* and *Callbacks*.

SW-Side API Calls. API calls executed by SW tasks transfer data to or from HW tasks after ensuring that the conditions for the transfer are met.

Channel Communication Event Interrupt Handlers. The *Event Interrupt Handlers* are executed as a result of the event interrupts generated by the channel interface modules. They could be divided into master- and slave-side interrupt handlers, depending on the side of the channel they are generated at. Master-side event handlers are only used in HW-HW communication, whereas the slave-side event handlers are used in both SW-HW and HW-HW communications.

The *Data Notify* event handlers are executed as a result of change in status of the buffer located in the channel interface module, for example, when the buffer becomes empty (read-side interface) or the buffer becomes filled with data (write-side interface). The *Data Notify* event handlers update the *Channel Control Table* with the information about the current status of the buffer. This information is used later on to control the transfer over the channel. The status in the table is cleared by the *Communication Agent* which completes the transfer to or from the buffer.

The *Start Access* event handlers are executed whenever a HW task starts accessing the channel; thus they update the *Channel Control Table* with the information about the access. When executed on behalf of the interface module located at the read side of the channel, they block the reading HW task if it started the access before the write side, as the data is not available yet. The blocked HW task is put into the *WAIT_CH_DATA* state then. When executed on behalf of the write side of the channel, they release the read side if it started before and was blocked.

In case of SW-HW communication, the SW task is always on the master side of the channel since it runs on a CPU which has a master interface to the physical interconnect. Consequently, it is the SW task which transfers the necessary data to or from the channel interface located in the HW task.

For HW-HW communication, the *Start Access* event handlers are responsible for starting the physical transfer over the interconnect. They accomplish it by talking to the control FSM located in the master side interface module and granting the transfer. When it happens, the master side interface sends the transfer request to the *HW Task Communication Module* located in the *HW Task Wrapper* and data transfer is started. As the transfer over the physical interconnect is master side oriented, it is always the master side which starts the read or write transfer. For this reason, in order to perform the transfer, the *Master side Start Access* event handlers have to talk to the master interface which generated the event (case of *FIFO*, *MSG,* and *SHARED MEM* channel), whereas the *Slave side Start Access* event handlers have to talk to the master interface on the other side of the channel (*FIFO* and *MSG* channels only). In case of *SHARED MEM* channel, which utilizes shared storage external to HW tasks, the *Start Access* event handler always grants the transfer. In case of *FIFO* and *MSG* channels, the *Start Access* event handlers always make sure that the interface module at the other side of the channel is currently present, before granting the transfer. If it does not and the HW task which triggered the event is at the read side, it will be put into waiting state. However, if the HW task is at the write side of the channel, a *Shadow Channel*, mentioned in Section 8.4.2, will be set up and data safely transfered to it. In case of master-side, the data will be transfered by the channel interface module. In case of slave-side, the *Copy Task* will read the data from the channel interface module and write it to the *Shadow Channel*.

Before the transfer is started, the *Configuration Locks*, described in Section 8.4.1, are acquired to prevent deallocations of the channel interfaces, while the transfer is in

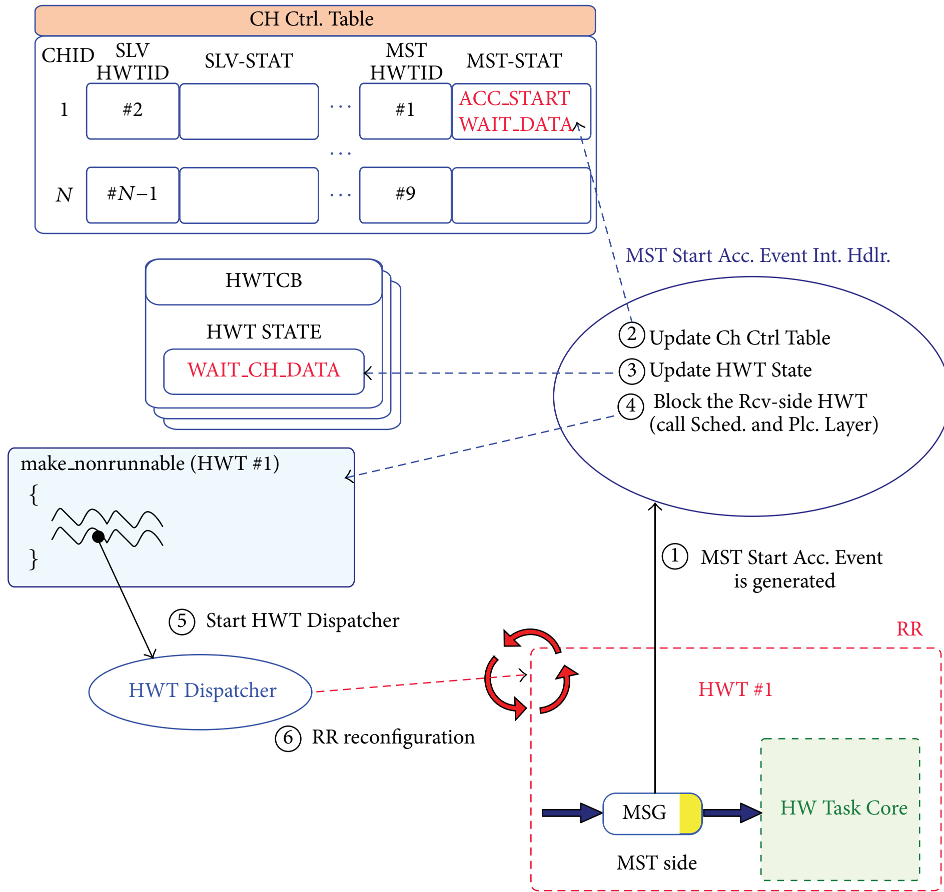


FIGURE 15: Channel event processing example—MSG read-side channel interface.

progress. In case of direct transfers, *Configuration Lock* is acquired for HW tasks located at the both sides of the channel. In case of transfers to or from the *Shadow Channel*, only lock for the related side is acquired. The *Configuration Locks* are released once the transfer has completed, that is, either by the *Copy Task* or the *End Access Event* handler.

The *End Access Event* handler is only generated by the master-side interface modules. Its main role is to signal completion of the transfer over the physical interconnect and release the aforementioned *Configuration Locks*. In case of *MSG* channel, it will also put the HW task located at the write side of the channel into waiting state, that is, *WAIT_CH_SYNC*, if the reader has not yet started accessing the channel.

Figure 15 shows exemplary processing done by the *Master-Side Start Access* event handler. It is generated when the HW task attempts to read data from the *MSG* channel via the channel interface module whose storage buffer is empty; that is, data has not been delivered yet. In the presented case, the channel is not locked, so that the *HW Task Scheduling and Placement Layer* is called. When the corresponding interrupt handler is executed, it will firstly update the *Channel Control Table* with the information about the access; that is, HW Task started access, but is waiting for data. Then, it will update the HW Task execution state located in the *HWTCB* of the *HW Task Management and Communication Layer*. Then, it will block the HW task by calling the appropriate function of the *HW Task Scheduling and Placement Layer*. As a result the *HW Task Dispatcher*, presented in Section 6.3, will be called and the HW Task, together with its channel interface modules, will be deallocated.

Copy Task. *Copy Task* is a background task with one priority level lower than the *HW Task Dispatcher*. Its main job is to transfer the data between the SW tasks or channel interfaces located in the HW tasks and the *Shadow Channel*. The *Copy Task* can be implemented as an SW task executing on a CPU or as an additional Direct Memory Access (DMA) IP. While the latter implementation has obvious performance benefits, the former one can be used if the logic resources provided by the FPGA are limited. In our system, we used the first implementation, due to the aforementioned reason.

Copy Task may be started by other *Communication Agents*. To facilitate searching of the shadow channel which needs to be handled, a *request map* structure is used. The *request map* is composed of an array of 32-bit masks, where each bit in the mask indicates request corresponding to the shadow channel of a given ID. The array is used to allow for more than 32 channels to be supported. Before the *Copy Task* is started, a bit in the *request map* is asserted. When the *Copy*

Task starts, it contacts the *request map* to check which shadow channels need to be handled by it and then does the necessary processing for each channel, one by one.

Callbacks. Callbacks are functions of the *HW Task Management and Communication Layer* which implement processing to be conducted just after the HW task is allocated, or the clock in its computational part, that is, *HW Task Core*, is switched on (*After Physical HWT Activation Callback*), and the processing to be conducted just before the HW task is deallocated, or the clock in its computational part is switched off (*Before Physical HWT Inactivation Callback*). Callbacks are registered in the *HW Task Configuration Layer* upon OS initialization and then executed within the *HW Task Dispatcher*. The processing implemented by the *After Physical HWT Activation Callback* includes actions performed upon allocation of the slave-side and master-side interfaces of the channel, passing the return value of the *Management API Call* which caused the HW task to be deallocated as well as enabling *Management API Call* interrupts and event interrupts. One of the actions performed upon allocation of channel interface modules is releasing the tasks from waiting, which blocked on the channel due to absence of the channel interface module on the other side of the channel. As this module is allocated at this time, this action has to be performed as well. Another action is related to activation of the *Copy Task* if the data could not be previously delivered directly to the channel interface module and had to be stored in the *Shadow Channel*.

The *Before Physical HWT Inactivation Callback* implements processing done before the master- and slave-side channel interface modules are deallocated. This processing is related to updating the *Channel Control Table* with the information about the channel interfaces becoming absent. It also implements waiting on the *Configuration Lock* as well as disabling of the *Management API Call* interrupts and event interrupts.

Enabling and disabling of interrupts is required to prevent generation of the events and notifications of the *Management API Calls* when the OS4RS already acknowledged deallocation.

8.5. SW and HW Task Communication/Synchronization API. Table 2 gives the API of supported communication/synchronization calls. *Rbow_send_mem_ch()* and *Rbow_recv_mem_ch()* represent send and receive calls for the *SHARED MEM* channel. Both calls take as their parameters ID of the channel, offset from the base address of the memory region implementing the channel, size of the data to be sent or received, and the pointer to the buffer from which the data is to be read and sent over the channel or to which the received data is to be written. *Rbow_send_reg_ch()*, *Rbow_recv_reg_ch()*, *Rbow_send_fifo_ch()*, *Rbow_recv_fifo_ch()*, *Rbow_send_msg_ch(),* and *Rbow_recv_msg_ch()* represent send and receive calls for *REG*, *FIFO,* and *MSG* channels, respectively. As their parameters, they only take the channel's ID and the pointer to the buffer. The depth of the *REG* channel is always one, whereas the depth of the *FIFO* and *MSG* channels and the size of the buffer are a constant defined at a design

Table 2: Communication/Synchronization API.

SW/HW tasks
Rbow_send_mem_ch (CH ID, ADDR OFFS, DATA SIZE, &Data)
Rbow_recv_mem_ch (CH ID, ADDR OFFS, DATA SIZE, &Data)
Rbow_send_reg_ch (CH ID, &Data)
Rbow_recv_reg_ch (CH ID, &Data)
Rbow_send_fifo_ch (CH ID, &Data)
Rbow_recv_fifo_ch (CH ID, &Data)
Rbow_send_msg_ch (CH ID, &Data)
Rbow_recv_msg_ch (CH ID, &Data)

time. This constant is stored in the *Channel Control Table* and also passed to the HDL code implementing the HW Tasks before they are synthesized. The depth of the channel passed to the HDL code configures the depth of the asynchronous FIFOs being part of the channel interface modules.

On the software side, we provided these calls in form of C preprocessor's macros which directly reference the functions of the *HW Task Management and Communication Layer*. On the hardware side, we provided implementation of these calls for HW tasks coded in C and synthesized using eXcite [57] HLS tool. The calls are converted into states of the control FSM, being part of the synthesized *HW Task Core* module, which accesses its I/O ports. Outside the *HW Task Core*, the ports are connected to the channel interface modules.

Internally, *FIFO* and *MSG* channel API calls intended for SW tasks contact the *Channel Control Table* in order to deduce the type of communication, that is, SW-HW or SW-SW. Later, the processing corresponding to a given type of communication is performed. As the *REG* channel is only intended for SW-HW inter-task communication, only this very processing is part of the API call's implementation. Finally, the processing done by the calls of the *SHARED MEM* channel is independent of the SW/HW location of the communicating sides.

There are two additional calls which are used to optimize communication over a channel. These are *Rbow_ch_lock_acq()* and *Rbow_ch_lock_rel()* calls which acquire and release the lock on the channel. Since these calls logically belong to the group of *Management API Calls*, they were described with other *Managament API Calls* in Section 7.1.

8.6. SW-SW Communication/Synchronization. Figure 16 shows an abstract view of the channel-based inter-task communication between SW tasks. In this type of communication, the access to the channel is provided through its interfaces implemented by the *Communication API Calls*.

FIFO and *MSG* channel-based inter-task communication and synchronization between SW tasks utilize the communication and synchronization objects already provided by the underlying base OS kernel. These are data queues and semaphores. In the base OS, the data queue can be configured to either provide a means of synchronous or asynchronous communication. The first case is realized by configuring the depth of the queue to zero, whereas the latter one is realized

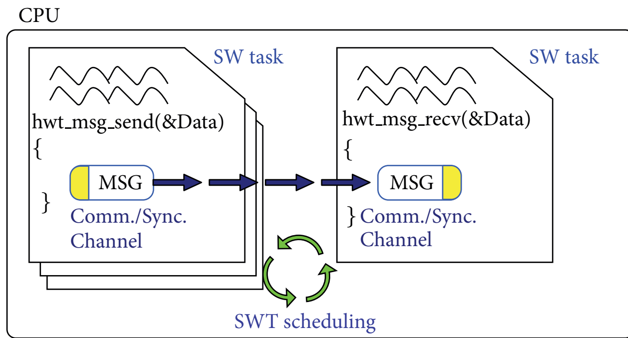


Figure 16: SW-SW channel-based communication.

by setting it to any other value. In the first case, a sender will block till the receiver side starts accessing the data queue and so will the receiver if it tries to read from an empty data queue. In the second case, the sender will be blocked only if the data queue is already full, while it attempted to send more data to it. The semaphore primitive provided by the base OS can be configured as a counting semaphore or a binary semaphore, that is, mutex. For our purposes, only the latter configuration was needed. The data queues and semaphores are referenced by their IDs defined at the time of configuring the base OS kernel.

The *FIFO* channel-based communication API call directly uses the access functions of the data queue object to transfer the data. The ID of the data queue is stored in the *Channel Control Table* and retrieved by the call using the channel ID. The depth of the data queue is configured to be the same as the depth of the channel. The depth also defines how many times the send or receive function of the data queue will be called internally by the the *FIFO* channel API call.

Depending on the depth, the *MSG* channel may be either implemented with the data queue object configured for synchronous operation or data queue object configured for asynchronous operation with an additional semaphore to enforce synchronous operation. If the depth of the channel is equal to one, then the data queue is enough, whereas for bigger depths, the data queue configured for asynchronous operation and an additional semaphore have to be used. Similarly to *FIFO* channel, the depth of the *MSG* channel defines how many times the send or receive function of the data queue will be called internally by the the channel API call before the semaphore is acquired or released.

Depending on the priorities of the communicating SW tasks, the communication through the data queue may result in multiple context switches which add up to the overhead of inter-task communication. In the worst case, when the receiver has already started the access to the channel and the sender has lower priority than the receiver, then at each word sent, the base OS will switch from the sender to the receiver.

Inter-task communication over the *SHARED MEM* channel is implemented using low-level I/O functions to transfer data to and from memory. The address of the region in the memory to write to or read from is known by contacting the *Channel Control Table*.

8.7. SW-HW Communication/Synchronization. Figure 17 shows an abstract view of channel-based inter-task communication between SW and HW tasks. In this type of communication, the access to the channel is provided through its interfaces implemented by the *Communication API Calls* on the SW side and by the channel interface modules on the HW side. The SW-HW inter-task communication is based on interaction between the SW tasks executing the API call and the *Communication Event Interrupt Handlers* executed on behalf of the HW tasks accessing the channel.

When the *FIFO* or *MSG* channel API Call is executed by a SW task, it is first checked if the buffer in the channel interface located in the HW task is filled with data, in case of the receive call, or it is empty, in case of the send call. If these conditions are met, the API call will proceed with execution, otherwise it will block, putting the SW task into waiting state. The blocking is implemented by means of the synchronization objects provided by the underlying base OS (event flags). The SW task will be released by the *Data Notify* event handler.

Then, it is checked if the HW task has blocked before while accessing the channel. If it is the case, the HW task will be released from waiting. Next, it is checked if the channel interface module located on the hardware side is present. If the HW task was previously blocked and deallocated and the previous step released it from waiting, then at this time, the API call will block till the HW task and its channel interfaces are allocated again. The SW task making the API call will be released from waiting by the *Callback* which manages allocation of the channel interface modules.

Finally the transfer to the storage located in the channel interface module is performed. In case *Shadow Channel* is used, its current state is confirmed at the beginning of the API call, potentially resulting in blocking of the SW task. It is required as the transfer over the *Shadow Channel*, managed by the *Copy Task*, may take much longer than the direct transfer. For this reason, it may happen that the software side starts next access, while the previous transfer has not been yet completed. If the SW task blocks on *Shadow Channel*, it will be released by the *Copy task* when it completes the previous transfer. The *Shadow Channel* is only used with *FIFO* SW-HW channels in order to allow the SW caller to continue execution even if the hardware side is not present.

In case of *MSG* channels, the send call blocks after the transfer is completed, if the hardware side has not started the access yet. It will be released by the *Slave-side Start Access* event handler. The read call sets the *Unblock* flag in the channel interface module of the HW task. This flag causes the interface module to acknowledge the request to read status of the transfer, made by the *HW Task Core*. Shall the flag be not set, the *HW Task Core* will be kept waiting for the acknowledge without making further progress in execution.

8.8. HW-HW Communication/Synchronization. Figure 18 shows an abstract view of channel-based inter-task communication between HW tasks. In this type of communication, the access to the channel is provided through its interfaces implemented by the channel interface modules located within

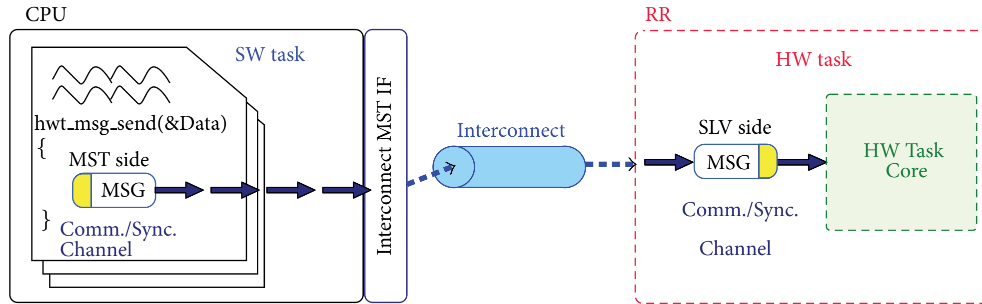


Figure 17: SW-HW channel-based communication.

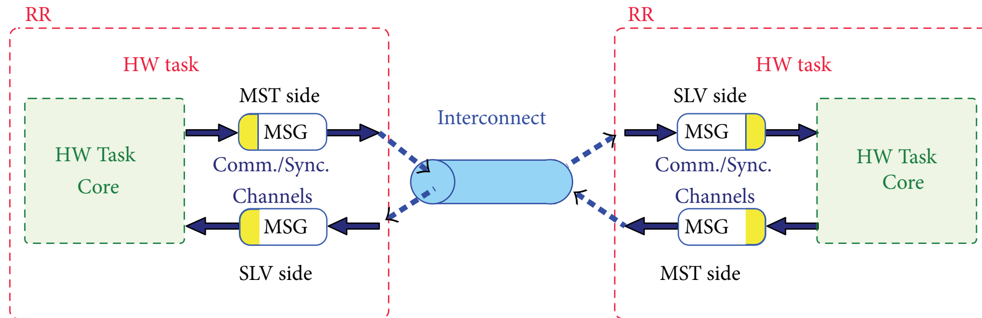


Figure 18: HW-HW channel-based communication.

the HW tasks. The HW-HW inter-task communication is based on interaction between the *Communication Agents*. It is partly managed by the event interrupt handlers, which unlike SW tasks, do not have their state and cannot be blocked. While it could be still handled by the SW tasks; that is, some of the processing done by the event interrupt handlers could be deferred to dedicated SW tasks, it is not done so for performance reasons. As a result of this decision, the control over the communication between the HW tasks is distributed over the *Communication Agents*.

The key role in this processing is done by the *Start Access* event handler. The *Start Access* event handler representing read side will cause the receiving HW task to be blocked if it starts before the write side, and so the data is not available yet. The *Start Access* event handler for the write side of the channel will check the condition for the transfer, that is, if the read side interface exists and if its buffer is empty. In case the read-side interface does not exist, the *Shadow Channel* will be set up, and data will be directed to it. When the read-side becomes allocated, the *Copy Task* will be started, and data will be transfered to it. In case the buffer is not empty, the sending HW task will be put into waiting. The further processing when the buffer becomes empty cannot be done by the *Start Access* event handler as it cannot be put into waiting state. For this reason the processing will be continued either by the *Data Notify* handler or by the *Callback*. The latter one will happen if the write side does not exist when the read side sends notification about the empty buffer. Either of them will release the previously blocked HW task.

The key processing that has to be done by these *Communication Agents* is starting of the transfer. As in the SW-HW case, if at the time the transfer is to be started, the read side does not exist, and the *Shadow Channel* will be used.

Several such cases had to be thoroughly analyzed to make this scheme work.

9. Evaluation Results

9.1. Experimental Setup. As a testing bed for our framework, we used a Xilinx ML410 board hosting a Virtex-4 FX60 speed grade-11 FPGA which was programmed with our system shown in Figure 19. The system was implemented using Xilinx EDK and PlanAhead v14.2 tools. Partial bitstreams representing HW tasks as well as FPGA initialization file were copied onto the compact flash card and used afterwards by the system.

In the presented system, the PPC405 CPU, located on the Virtex-4 FX's fabric, BRAM-based CPU memory, and its controller, the peripherals, and all the buses are running at 100 MHz. CPU instructions are entirely placed in on-chip BRAMs, whereas the data is placed in the external DDR memory which also serves as a bitstream repository. CPU instruction and data caches are on. The reconfiguration controller is connected to the DDR2 memory controller, Xilinx Multi-Port Memory Controller (MPMC) IP core, through a 64-bit PLB bus. The HW Task Wrappers are connected to a separate bus for faster HW-HW communication. As a consequence, the SW-HW communication experienced additional latency as it had to go through an additional PLB2PLB bus bridge. The external timer shown in Figure 19 was used to implement the OS tick timer. Time measurements were conducted using the time base timer being part of the

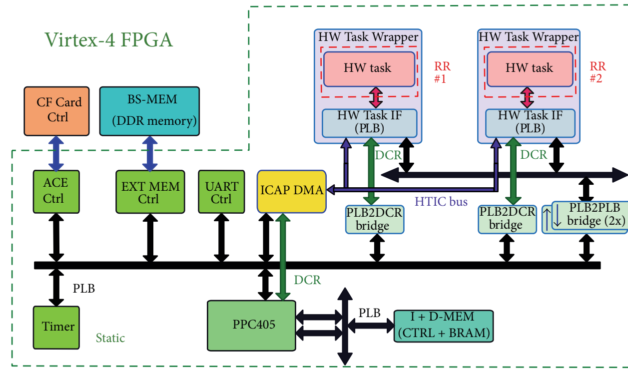


Figure 19: Experimental setup (Rainbow tests).

Table 3: ICAP-DMA—synthesis results.

	With HWT state extraction	w/o HWT state extraction
Slices	1543	1161
FFs	1096	776
LUTs	2223	1723
BRAMs	5	5
ICAP	1	1
Max. Clk freq. (MHz)	158.8	200.8

Table 4: HWT wrapper—synthesis results.

	HWT wrapper (PLB and DCR)	DCM controller
Slices	1706	178
FFs	1721	175
LUTs	2340	310
BRAMs	0	0
DSPs	0	0
DCMs	0	1
BUFGs	2	2
FCLK freq. (MHz)	140.8	276.1
VCLK freq. (MHz)	527.4	315*

*For Virtex-4 speed grade-11 (HF Mode) [59].

PPC405's register set. To interact with this timer, additional inline assembly functions were implemented. The PLB2DCR Bridge being part of the IP repository provided by Xilinx and used to control the HW Task Wrappers had to be modified. Specifically, the bitwidth of its timeout counter had to be extended in order to support HW tasks running at low-frequency clocks.

9.2. HW-Side OS4RS Modules: Synthesis Results. Table 3 gives the synthesis results for the developed configuration controller (*ICAP-DMA*) with Xilinx Synthesis Tool (XST) v14.2 and default synthesis options. *HW Task State Extraction* indicates whether the state extraction module used in HW task preemption is included in the *ICAP-DMA*. If it is not, the corresponding functionality is implemented in software. In our tests, for improved preemption performance, the module was included. In such a configuration, the *ICAP-DMA* occupies about 6% of the total area of the Virtex-4 FX60 used in the experiments.

Table 4 presents synthesis results for the HW Task Wrapper. The table presents the synthesis results separately for the DCM Controller, being logically a part of the HW Task Wrapper, and the rest of the logic contained in the HW Task Wrapper. The reason is that to simplify application of timing constraints for the whole IP, we decided to provide the DCM Controller externally to the HW Task Wrapper, yet making it fully controlled by the HW Task Wrapper. The HW Task Wrapper, including the DCM Controller, takes up about 7.5% of the logic resources available on the FPGA used in tests. Keeping in mind, that Virtex-4 was manufactured a few years ago, the latest FPGAs from Xilinx offer much bigger capacity, and this result is considered satisfactory. Also the maximum clock frequency for the static part of the system, that is, *FCLK* domain is high enough to make the placement and routing for the system running at 100 MHz easier.

9.3. Channels Interface Modules: Synthesis Results. Tables 5 and 6 present synthesis results for the Management API Call channel and communication channel interface modules used by the HW Tasks, synthesized using XST v.14.2 with additional constraints required for the developed CPA-based preemption mechanism. These additional constraints include enforcing the FFs' set and reset signals to be synthesized as a part of set/reset paths [48]. Additionally, the memory extraction style option had to be set to distributed, that is, LUT-RAMs. While technically the developed mechanism supports preemption of BRAMs, we encountered problems in the final system, which are very likely to be on the manufacturer's side. This is why we had to implement all channel interfaces with a distributed memory working as their local storage.

In all presented results, the width of the data bus was set to 32 bits for all channels. The depth of the asynchronous FIFO implementing the storage of the *API Call (Send)* part of the Management API Call channel was configured to four.

Table 5: Management API Call Channel—synthesis results.

Channel	Slices	FFs	LUTs	FCLK freq. (MHz)	VCLK freq. (MHz)
API call (send)	83	28	94	360.9	370
Return value (recv.)	101	174	73	383.7	439.1

Unless indicated otherwise, the values in the table are unitless.

Table 6: Communication Channels' HW Interface Modules—synthesis results.

Interface module	Depth (word)	Slices	FFs	LUTs	FCLK freq. (MHz)	VCLK freq. (MHz)
SHD MEM RD + WR	1	317	242	403	215.4	263.8
	8	342	258	451	205.4	238.6
	32	554	293	730	196.1	213.9
	128	1459	350	1782	196.4	181.6
FIFO (MST-side write)	1	158	89	221	240.8	276.6
	8	173	97	251	210	254
	32	272	115	378	217.7	210.8
	128	721	144	897	182.8	208.2
FIFO (MST-side read)	1	190	130	262	238.1	328.5
	8	211	141	300	228.8	251.6
	32	325	159	455	209.4	227.5
	128	784	188	993	191.1	182.3
MSG (MST-side write)	1	160	96	225	244.7	276.6
	8	170	105	244	244.8	257.8
	32	271	121	375	214.6	211.5
	128	731	140	916	175.5	166.6
MSG (MST-side read)	1	196	135	274	235.3	328.5
	8	212	145	304	227.7	261.8
	32	332	164	468	203.5	234
	128	791	211	1006	214.5	219.5
FIFO (SLV-side write)	1	101	61	119	367.2	333.8
	8	112	74	139	283	283.8
	32	219	84	281	232	226.5
	128	679	102	819	179.4	166.4
FIFO (SLV-side read)	1	109	71	120	343	330
	8	123	87	149	311.8	277
	32	234	99	298	209.3	233
	128	692	123	833	188.5	180
MSG (SLV-side write)	1	104	63	124	363.2	333.8
	8	114	76	144	283	283.7
	32	221	86	286	232	226.5
	128	681	104	824	179.4	166.4
MSG (SLV-side read)	1	115	76	133	336.8	330
	8	130	92	162	311.7	277
	32	240	104	311	209.3	233
	128	698	123	845	172.8	180
REG (write)	1	65	111	80	431.6	370
REG (read)	1	85	148	16	874.1	632.3

Data width = 32.
Unless indicated otherwise, the values in the table are unitless.

The first column of Table 6 denotes the type of channel interface module, whereas the second column represents depth of the storage element used by it. In case of *FIFO*, *MSG,* and *SHARED MEM* channel interface modules, the smallest depth of the storage elements, that is, asynchronous FIFOs, that can configured, is equal to four, as mentioned in Section 8.3.2. This should be kept in mind when analyzing the results for depth equal to one, which actually include modules with the asynchronous FIFO of depth equal to four. For small depths, the major contributor to the logic area is the additional control logic being part of the channel interface module. For bigger depths, the asynchronous FIFO becomes the major part. It is especially visible for the *SHARED MEM* channel interface module which implements the FIFOs for both sending and receiving.

For very small *HW Task Cores* utilizing the presented interface modules with bigger depths, the resource consumption may be considered high. However, for bigger cores synthesized from high-level descriptions, it will not be, in our opinion, the case. Such a comparison for HLS-based cores is conducted in Section 9.4. The results for the *FCLK*, shown in column six, do not seem to pose any limitations, considering common case of 100 MHz for the static part of the system. The worst-case result of *VCLK*, that is, 166.4 MHz for *MSG (SLV-side write)* and *FIFO (SLV-side write)* channel interface modules with depth set to 128 words, shown in column seven, is also considered satisfactory.

9.4. HW Tasks Used in Tests. As HW Task Cores in our tests, we used the image processing (IDCT) core and cryptographic (SHA) core taken from benchmark suite presented in [60] and synthesized to HDL, using YXI eXCite HLS tool [57]. SHA implements a secure hash algorithm with a 160-bit hash function and operates on 512-bit blocks of data. IDCT is an inverse discrete cosine transform which processes an 8×8 matrix of 32-bit words per iteration. These original cores were supplemented with the channel interface modules and related glue logic, described in Section 3.4. The IDCT core first reads data from its input port, processes it, and then writes it to output. The SHA core first reads the size of the data stream to be processed, then reads as many words as indicated by that size. While reading these words, it updates its structures. Finally, when all the words have been read, it writes a 160-bit message digest to the output.

9.4.1. Static Characteristic: Synthesis Results. Table 7 presents synthesis results for the HW tasks used in tests, synthesized with XST v. 14.2. The first column denotes the *HW Task Core* and its channel interface configuration. *IDCT MSG SW-HW and HW-HW* (×64) and *IDCT FIFO SW-HW and HW-HW* (×64) denote HW Tasks using IDCT algorithm in the *HW Task Core* part and implementing two *MSG* or *FIFO* channel interfaces with depth equal to 64 words, and both and configured as slaves. This HW task is used for both SW-HW and HW-HW communications. *IDCT MEM-BC HW-HW* (×64) implements one *SHARED MEM* channel interface and two MSG channel interfaces used for synchronization. These are set to depth equal to one, and the first one is configured as master, whereas the second one is configured as slave. This task is used for testing HW-HW communication. *IDCT MEM-BC SW-HW* (×64) implements one *SHARED MEM* channel interface and two *MSG* channel interfaces used for synchronization. These are set to depth equal to one, and both are configured as slaves. This task is used for testing SW-HW communication. *IDCT MEM-REG SW-HW* (×64) implements one *SHARED MEM* channel interface and two *REG* channel interfaces used for synchronization. This task is used for testing SW-HW communication. *SHA FIFO SW-HW* (×16), *SHA FIFO SW-HW* (×32), and *SHA FIFO SW-HW* (×64) denote HW tasks using the SHA algorithm in the *HW Task Core* part. They implement one word *MSG* channel interface for sending the size of the data stream to be processed and two *FIFO* channel interfaces for data communication, set to depth equal to 16, 32, and 64 words, respectively. All channel interfaces in the *SHA* HW tasks are configured as slaves. These tasks are intended to test SW-HW communication only. Finally, the HW tasks at the bottom of the table, which contain TB in their names, are testbench tasks. They are used to test HW-HW communication with the other HW tasks implementing IDCT algorithm, by sending test vectors stored in their local memory. *IDCT TB MSG HW-HW* (×64) and *IDCT TB FIFO HW-HW* (×64) implement two *MSG* and *FIFO* channel interfaces of depth 64 words, both configured as masters. *IDCT TB MEM-BC HW-HW* (×64) implements one *SHARED MEM* channel interface and two *MSG* channel interfaces used for synchronization. These are set to depth equal to one, and one is configured as master, whereas the other one is configured as slave.

Column three, five, seven, and eleven show the overhead in resource consumption of the preemptable HW task including channel interfaces and the glue logic, with respect to preemptable *HW Task Core* (top row in the cell) and nonpreemptable *HW Task Core* (bottom row in the cell). Column nine shows the corresponding result in baseless units rather than in percent. The last column shows the difference in clock frequency resulting from making the *HW Task Cores* preemptable and adding the channel interfaces and the glue logic. As the original *HW Task Core* operates only in one clock domain which corresponds to *VCLK* domain in the table, there is no column showing difference in *FCLK*.

As it is clear from the table, a big factor increasing the resource utilization and decreasing clock frequency of the created HW tasks is preemption mechanism which, due to faced problems, did not allow for usage of Block RAM (BRAM) resources present on Xilinx Virtex-4 FPGAs. It should be stressed, however, that this is not the limitation of our overall approach but a problem with the design tool chain provided by the manufacturer. If this was solved, the results could be improved.

If we compare the results for the preemptable HW Tasks with the preemptable *HW Task Cores*, we will see that the overhead caused by additional channel interfaces and HW Task glue logic is not prohibitively high. In terms of slices, it is up to 41.66% for IDCT core and up to 68.63% for the SHA core. Furthermore, it should be noted that the *HW Task Cores* used in tests were coded without any additional parallelizing

Table 7: HW tasks—synthesis results.

HW task	Slices	Ovhd. (%)	FFs	Ovhd. (%)	LUTs	Ovhd. (%)	BRAMs	Diff. (unit)	DSPs	Ovhd. (%)	FCLK freq. (MHz)	VCLK freq. (MHz)	VCLK diff. (%)
IDCT MSG SW-HW and HW-HW (×64)	5842	36.27% 78.65%	2035	17.36% 20.63%	9953	30.33% 59.22%	0	0 −2	36	0.00% 38.46%	137.6	66.3	2.00% −19.54%
IDCT FIFO SW-HW and HW-HW (×64)	5827	35.92% 78.20%	2028	16.96% 20.21%	9933	30.06% 58.90%	0	0 −2	36	0.00% 38.46%	137.6	66.3	2.00% −19.54%
IDCT MEM-BC HW-HW (×64)	6083	41.66% 85.40%	2180	24.93% 28.46%	10322	35.03% 64.84%	0	0 −2	36	0.00% 38.46%	116.8	66.8	2.45% −18.83%
IDCT MEM-BC SW-HW (×64)	6023	40.27% 83.57%	2152	23.32% 26.81%	10209	33.56% 63.03%	0	0 −2	36	0.00% 38.46%	133.2	66.8	2.45% −18.83%
IDCT MEM-REG SW-HW (×64)	5979	39.18% 82.06%	2215	26.64% 30.83%	10077	31.66% 60.74%	0	0 −2	36	0.00% 38.46%	133	66.3	3.43% −19.44%
SHA FIFO SW-HW (×16)	5056	66.15% 91.95%	3215	8.87% 10.14%	8566	63.75% 80.49%	0	0 −1	0	0.00% 0.00%	204.6	130.1	−14.65% −15.35%
SHA FIFO SW-HW (×32)	5243	67.03% 103.69%	3234	9.59% 10.60%	8799	64.41% 87.53%	0	0 −2	0	0.00% 0.00%	165.1	130.1	−16.17% −15.35%
SHA FIFO SW-HW (×64)	5543	68.63% 115.35%	3245	9.89% 10.94%	9149	65.59% 94.91%	0	0 −2	0	0.00% 0.00%	151.4	130.2	−16.11% −15.29%
IDCT TB MSG HW-HW (×64)	1259	672.39% 899.21%	656	412.50% 583.33%	1648	499.27% 610.34%	0	0 −1	0	0.00% 0.00%	174.4	164.4	−44.72% −45.09%
IDCT TB FIFOHW-HW (×64)	1249	666.26% 891.27%	644	403.13% 570.83%	1630	492.73% 602.59%	0	0 −1	0	0.00% 0.00%	165	164.4	−44.72% −45.09%
IDCT TB MEM-BC HW-HW (×64)	1400	709.25% 929.41%	746	436.69% 597.20%	1840	538.89% 651.02%	0	0 −1	0	0.00% 0.00%	185.6	183.3	−38.37% −38.78%

Unless indicated otherwise, the values in the table are unitless.

optimizations and synthesized by means of the HLS tool with default options. If those optimizations were used, the logic area of the *HW Task Cores* would be further increased, and the relative overhead of additional logic would be lower.

Although the high logic overhead can be observed for testbench tasks, we should keep in mind that they do not implement any computational algorithm. Instead, they just implement a local memory filled with test vectors and the control FSM which sends the test vectors to the tested HW task and receives the result. Nevertheless, a conclusion we can make here is that for very small circuits the proposed method creates too much overhead to be applicable.

Once a HW task is synthesized, the resulting netlist file has to be passed through the FPGA design flow in order to generate the final configuration bitstream. A special procedure has to be followed in order to make the implementation of the HW tasks with *VCLK* frequency possible. The initial *VCLK* frequency set in the DCM of each *HW Task Wrapper* will define the frequency of all HW tasks allocated in the RR contained in that wrapper. This initial frequency has to be set to maximum clock frequency among all HW tasks. Also the frequency synthesis mode in the DCM has to be appropriately set, based on this clock frequency setting. In order to make the Static Timing Analysis (STA) work for each configuration of the static part of the system and HW tasks, the initial DCM frequency has to be overridden on per HW task basis. In case of our implementation based on Xilinx design flow tools, it is possible by applying an additional clock period constraint to the VCLK clock path located within each HW task and assigning a high priority to it.

9.4.2. Dynamic Characteristic: Bitstream and State Size. Table 8 shows the size of reconfiguration bitstreams as well as reconfiguration/readback and reinitialization times for the tasks shown in column one and previously described in this section. The bistreams were generated by the PR design flow tools and processed with the additional design flow back-end tool, that is, Bitformatter, described in Section 5. It should be noted that bitstream compression option was not used. The primary and secondary configuration bitstreams shown in columns two, three, and four are specific bitstreams generated by the Bitformatter in order to make the preemption possible. Columns five to eight show the reconfiguration, reinitialization and readback times achieved when using these bitstreams. It is assumed that HW tasks are reconfigured as a result of activation; that is, after allocation, they will start their execution form the beginning. The column five corresponds to allocation of the HW task which has not been started yet, and column six corresponds to HW task which has been started before, and so its state needs to be reinitialized, whereas column seven corresponds to reinitialization of already allocated task. As shown in the table, for the tasks used in tests, the allocation can be accomplished in about 2 ms for IDCT and SHA cores and in a bit more than 800 μs for Testbench tasks. On the other hand, the reinitialization without allocation, used in activation of the HW task which held the RR lock and exited without releasing it, can be accomplished in less than 300 μs for IDCT and SHA cores and in a bit more than 100 μs for Testbench tasks. State readback times, presented in the last column, are in the range of about 50–125 μs for the used tasks.

The presented results include overhead of the *ICAP-DMA*'s driver. The bare transfer times accomplished by the developed streaming reconfiguration controller were measured to be 318–333 MB/s for readback and 282–326 MB/s for reconfigurations. Although in Virtex-4 FPGA used in our tests, throughputs of 400 MB/s are theoretically possible, and these were not reachable due to performance restrictions of the memory controller used to access the external memory where the bitstreams are stored.

The preemption time for a given pair of tasks is equal to the sum of deallocation time of the preempted task (shown in column eight) and allocation time of the next task (shown in column five). Considering the obtained results, preemption could be used to improve FPGA utilization whenever HW task stall time, due to delays in data delivery, is in the range of milliseconds. While the HW task waits for data, another HW task could be executed. In case of processing long data streams, which may require even seconds to complete, preemption could be used to improve system responsiveness, when compared to nonpreemptive execution, by allowing a more critical task to start earlier.

One prospective area of applications which could benefit from it is embedded systems with networking, which only store the most essential data locally and larger amount of data in some remote locations. In these systems, it may happen that the packet containing the data of the computing core has not been yet received from its remote storage. Although the packet delivery latencies differ depending on the type of network and its infrastructure, in case of systems communicating over the internet, delays in the range of a few to tens of milliseconds are common.

Second prospective area of application is, for example, FPGA-based accelerator cards for servers and personal computers where memory virtualization with Hard Disk Drives (HDDs) as a secondary storage is used [48]. Although the work on the OS4RS presented in this paper does not consider memory protection and virtualization per se, this very feature is important in medium and large-scale embedded systems, thus can be seen as a future extension of this work.

9.5. SW and HW Task Management API Calls: Performance Results. Figure 20 shows the results of execution of *HW Task Management API Calls* by SW and HW Task callers for the case when the *VCLK* frequency in the HW tasks is set to 100 MHz. The bottom chart is for the case when the *HW Task Dispatcher* did not have to be started as a result of the call, whereas the top chart is for the case when it had to be started. In the tested cases, when the *HW Task Dispatcher* is started, the control is switched from the SW Task executing the call to the *HW Task Dispatcher*. Then, when the *HW Task Dispatcher* blocks waiting for the *HW Task Configuration Layer* to complete reconfiguration, the control is passed back to the caller, which returns. The times presented in both charts were measured by starting the timer before the call was executed and stopped, when it returned. While taking the measurements, the tick-timer and UART interrupts were disabled.

Table 8: HW tasks size of bitstreams and reconfiguration/readback times.

HW task	Bitstream size (KB)			Time (us)			
	Primary bitstrm	Secondary bitstrm (logic)	Secondary bitstrm (mem.)	Reconfig. (alloc. only)	Reconfig. (alloc. + reinit)	Reinit. only	Readback (dealloc)
IDCT MSG SW-HW and HW-HW (×64)	633.38	68.94	20.56	2066.79	2126.57	287.32	119.33
IDCT FIFO SW-HW and HW-HW (×64)	633.38	72.3	19.22	2071.27	2128.88	289.77	118.49
IDCT MEM-BC HW-HW (×64)	633.38	70.62	23.59	2049.29	2126.99	287.67	121.92
IDCT MEM-BC SW-HW (×64)	633.38	69.27	22.58	2057.2	2135.25	296.21	124.79
IDCT MEM-REG SW-HW (×64)	633.38	68.27	19.22	2048.87	2117.05	277.66	119.54
SHA FIFO SW-HW (×16)	633.38	55.5	18.88	2006.66	2063.5	223.62	119.61
SHA FIFO SW-HW (×32)	633.38	51.8	20.23	2011.84	2073.86	234.4	121.36
SHA FIFO SW-HW (×64)	633.38	54.16	19.89	2011	2073.09	233.98	123.11
IDCT TB MSG HW-HW (×64)	229.93	18.55	12.16	789.33	823.26	105.46	50.58
IDCT TB FIFOHW-HW (×64)	229.93	17.88	12.16	780.05	821.42	103.57	51.21
IDCT TB MEM-BC HW-HW (×64)	229.93	18.55	13.17	786.84	826.39	108.61	54.51

The set of the tested calls includes activation of a High Priority (HP) Task, a Low-priority (LP) Task, their termination, suspension, and resumption, task prefetching, activation of the prefetched task, changing of task priority, rotation of the ready queue, acquiring and releasing of the RR lock, and updating of the HW task frequency. In the presented charts, the *Ctxt switch* corresponds to context switch, that is, the HW task preemption.

While allocation times for the HW tasks used in tests were presented in Section 9.4, the measured DCM reconfiguration times were in the range 49.2 μs to 64.1 μs. The measured range is for the best and the worst case where the clock multiplier and divider values passed to the DCM are low and high, respectively.

As we can see from the charts, the execution time of the calls which do not result in activation of the *HW Task Dispatcher* is higher for the HW task caller. The reason for the increased latency is manyfold. The HW task caller's latency includes hardware-side overhead for making the call, additional interrupt response time, processing of a dedicated interrupt handler, and finally sending the return value to the HW task caller. On the other hand, the calls which result in activation of the *HW Task Dispatcher* take less time for the HW tasks. The reason is that the *HW Task Dispatcher* is not activated till the interrupt handler, making the call on behalf of the HW task, returns. Thus, the HW task receives the return value before the *HW Task Dispatcher* is actually started.

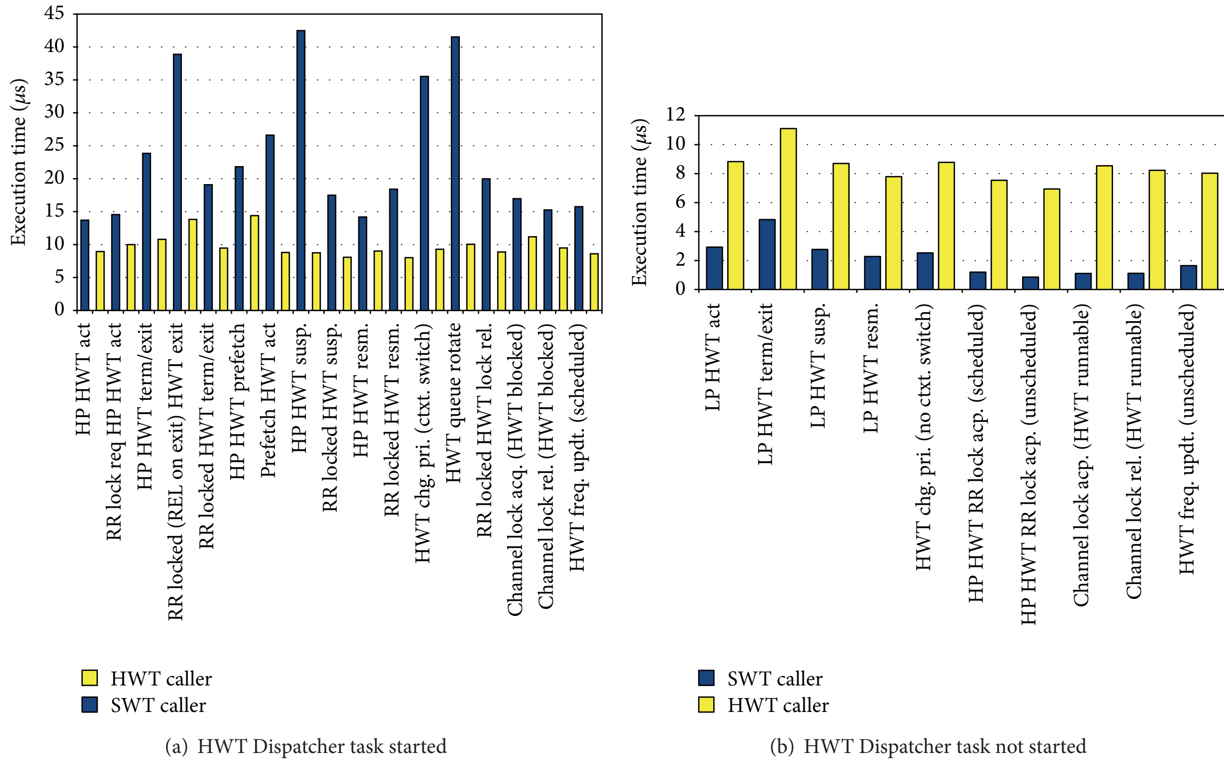
Figure 21 show the results of execution of *SW Task Management API Calls* by the SW and HW Task callers. These calls are handled by the base OS kernel, that is, Toppers ASP.

The tested set of *SW Task Management API Calls* includes, starting from the left on the column charts, activation of an LP task which does not result in the context switch as well as an HP task which results in the context switch, termination of an HP task resulting in the context switch, change of the task priority which results or which does not result in the context switch, rotation of the ready queue, suspension, and resumption of an HP task which results in the context switch. The reason why some results are given only for the SW Task caller is that the remaining calls are not supported in the interrupt handler's context, required to execute the call on behalf of the HW task caller. The measurements were started before executing the API call and taken when the API call returned or the other higher-priority SW task started execution.

As shown in Figure 21, the latency of processing the call experienced by the HW task caller is about 6.6–8.9 μs higher. The reason for the increased latency has been already explained when describing the *HW task Management API Calls*.

If we compare the overhead created by the calls managed by the Rainbow extension with the overhead created by the base OS kernel, which manages the SW tasks, we will see that the processing time of *HW Task Management API Calls* which do not result in activation of the *HW Task Dispatcher* is comparable to that of the *SW Task Management API Calls* which do not result in context switch. Furthermore, if we look at the results for *HW Task Management API Calls* which result in starting of the *HW Task Dispatcher* task, we will see that the measured times are low when compared to allocation times presented in Section 9.4 and the DCM reconfiguration times given in this section.

In general, it is difficult to directly compare the obtained results with other works due to different communication interconnects being used, different CPUs and their speeds, different memory configurations, different base OSes, and lack of clock scaling (requiring clock domain crossing synchronization) in other works. The work which mostly resembles this one in terms of CPU, interconnect, and OS configuration is ReconOS presented in [20]. In that work, experiments are conducted on a system built on top of a



(a) HWT Dispatcher task started

(b) HWT Dispatcher task not started

Figure 20: HWT Management API Call execution times.

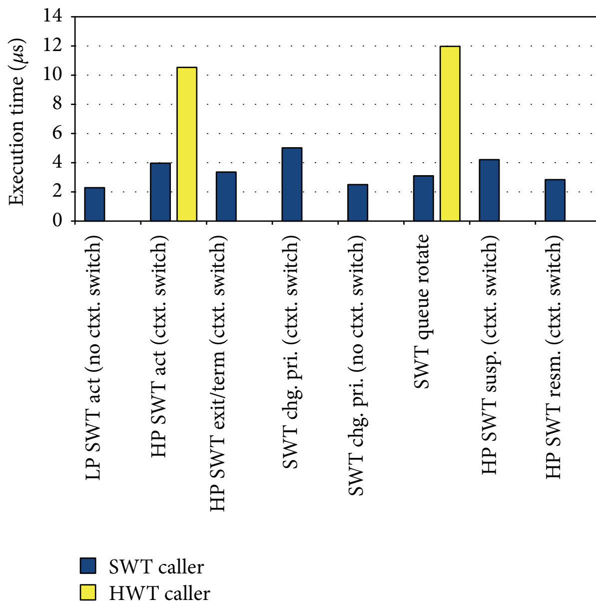


Figure 21: SWT Management API Call execution times.

Xilinx Virtex-II FPGA. That system also contains PPC405 CPU; however, it runs at 300 MHz, that is, three times faster than the CPU used in our tests. Similarly to our system, that work uses a PLB bus for data communication and a DCR bus to control the HW task's OS interface. The buses and the HW tasks in both systems run at 100 MHz. One of the base OS kernels used by ReconOS is eCoS RTOS [10] which similarly to our work is a lightweight RTOS providing only basic OS services. This makes the comparison more feasible than with other works based on Linux kernels.

In order to compare the performance of CPU, memory, and base OS configurations of ReconOS and our system, we tested semaphore turn-around time for SW tasks. This test was performed on the base OS kernel, which in our case is Toppers ASP kernel. When compared to 3.05 μs given in [20], the same test run on our platform resulted in 4.68 μs.

In order to test the efficiency of ReconOS with our OS4RS, we compared the difference in time to make a call by a SW task caller and a HW task caller. Although our system does not use semaphores and mutexes for inter-task synchronization, the call to acquire and release semaphore presented in ReconOS [20] is performed in a similar way as *SW Task Management API Calls* in our OS4RS. For this very case, the difference in processing time between the SW task caller and the HW task caller includes processing time of the HW task OS interface, interrupt response time, and further OS processing overhead. For ReconOS this difference in time varies from 5.08 to 8.76 μs for releasing and acquiring mutex, respectively. In our case it is 6.56–8.87 μs for SW task activation and rotation of the SW task ready queue, respectively. Although our OS4RS supports the HW task clock scalability, which requires additional signal synchronization between clock domains, and the CPU runs at lower frequency, results are similar. This could be explained

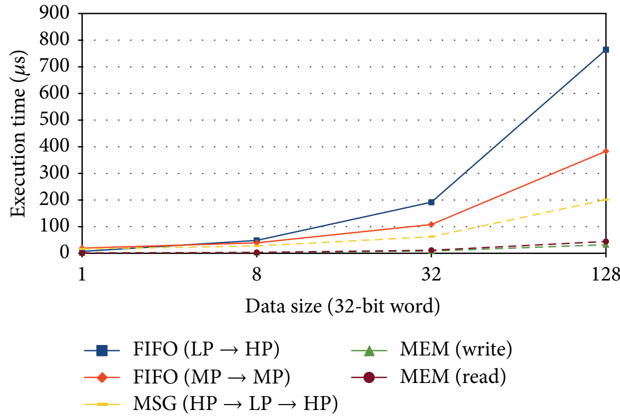


Figure 22: SW-SW communication times for different sizes of data.

by the fact that in our OS4RS the calls made by the HW task caller are handled by the software side of the created extension in more efficient way. In the developed OS4RS, it is the interrupt handler which makes the call on behalf of the HW task without switching to a delegate SW task, as it is the case in ReconOS.

9.6. SW-SW Communication: Performance Results. Figure 22 presents results of SW-SW inter-task communication which is managed by the base OS. *FIFO (LP → HP)* denotes FIFO channel-based communication between a Low Priority (LP) sender and a High Priority (HP) receiver. The timer was started when the sender made the call and was stopped when the receiver's call returned. *FIFO (MP → MP)* denotes FIFO channel-based communication between task of the same priority. In this case, the sender and the receiver made two send/receive calls. While the sender completed its first call without blocking, it blocked on the second one, due to FIFO full condition. The timer was started before the first send and stopped after the first receive call. *MSG (HP → LP → HP)* denotes MSG channel-based communication between an HP sender and an LP receiver. The timer was started when the sender made the call and was stopped when the sender's call returned. *MEM (write)* and *MEM (read)* denote *SHARED MEM* channel-based inter-task communication. While taking the measurements, the tick-timer and UART interrupts were disabled.

In case of *FIFO (LP → HP)*, the communication over the channel with depths bigger than one resulted in multiple context switches between the tasks. It is not the case with *FIFO (MP → MP)* and *MSG (HP → LP → HP)*, where the SW task sender can transfer the whole data before switching to the receiver. The presented results for *SHARED MEM* channel-based communication include read/write transfer times and overhead of processing the API call. It should be noted though that the actual communication over the *SHARED MEM* channel would additionally require single-word FIFO or MSG channel-based synchronization.

In case of *FIFO (LP → HP)*, transfer time varies from 7.24 μs to 764.34 μs for one word data and 128-word data, respectively. In case of *FIFO (MP → MP)*, transfer time varies from 19.55 μs to 383.22 μs for one word data and 128-word data, respectively. In case of *MSG (HP → LP → HP)*, transfer time varies from 16.5 μs to 201.28 μs for one word data and 128-word data, respectively. *MEM (write)* takes from 1.26 μs to 32.41 μs, whereas MEM (read) from 1.29 μs to 44.46 μs for one-word data and 128-word data, respectively.

9.7. SW-HW Communication: Performance Results. Figure 23 shows results for the SW-HW communication for the FIFO, MSG, and REG channels. The results were measured using HW Tasks with their *HW Task Cores* running at 100 MHz and the SR-RR boundary registered on both sides. While taking the measurements, the tick-timer and UART interrupts were disabled. The results are for the case where the HW task containing the channel interface is allocated and holds the RR lock to prevent its deallocation. Furthermore, the channel was locked, so that only the *HW Task Management and Communication Layer* was involved in communication, that is, the layers beneath it were not called when the HW task waited for data. These conditions allowed for measuring the communication overhead induced by the physical transfer and the processing conducted only by the *HW Task Management and Communication Layer*.

The results for the *SHARED MEM* channel are not included here, as these were presented in Section 9.6 for the SW tasks and will be presented in Section 9.8 for the HW tasks. In case of *FIFO* channel, the receiver was activated first and blocked on empty FIFO. Then, the measurement was started before the sender executed the API call and stopped when the unblocked receiver completed execution of the API call. In case of *MSG* channel, the sender was activated first. The measurement was started before the sender executed API call. The measurement was then suspended when the sender blocked waiting for the receiver to start. The measurement value was restored just before the receiver unblocked the sender and, finally, stopped when the sender's API call returned. This was done to make it possible to compare the obtained results with these for SW-SW communication.

The calculated results is composed of two components. The first component are results taken at run-time, whereas the second one is the results taken from the logic simulator. At run-time, the transfer time between the buffer used by the SW task (local variable) and the buffer used by the HW task (channel interface module's FIFO buffer) was measured. This gave us the time of the physical transfer started and completed by the SW task. Additionally, the processing time of the SW task's API call and processing time of the event interrupt handler was measured at run-time. In the simulator, the latencies between the start of access to the channel by the HW tasks and generation of event interrupts were measured. These results were combined.

For *FIFO (SWT → HWT)* (case where SW Task is a sender and HW task is a receiver), communication takes from 2.78 μs to 30.78 μs for one-word data and 128-word data, respectively. It gives speedup of up to 24.83x with respect to corresponding *FIFO (LP → HP)* SW-SW communication.

For *FIFO (HWT → SWT)* (case where HW Task is a sender and SW task is a receiver), communication takes from 12.2 μs to 78.27 μs for one-word data and 128-word data,

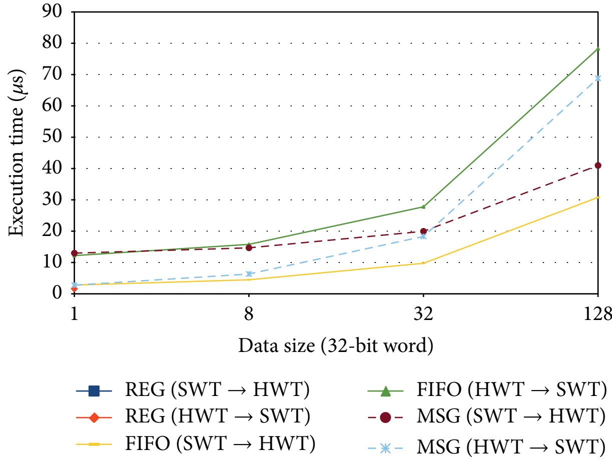


Figure 23: SW-HW communication times for different sizes of data (HWT @ 100 MHz).

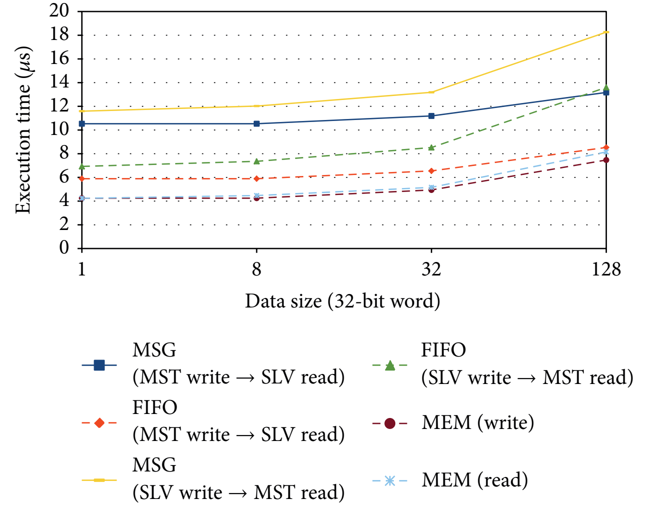


Figure 24: HW-HW communication times for different sizes of data (HWT @ 100 MHz).

respectively. It gives speedup of up to 9.77x with respect to corresponding *FIFO (LP → HP)* SW-SW communication.

For *MSG (SWT → HWT)*, communication takes from 12.98 μs to 40.98 μs for one word-data and 128-word data, respectively. It gives speedup of up to 4.91x with respect to corresponding *MSG (HP → LP → HP)* SW-SW communication.

For *MSG (HWT → SWT)*, communication takes from 2.75 μs to 68.81 μs for one-word data and 128-word data, respectively. It gives speedup of up to 6x with respect to corresponding *MSG (HP → LP → HP)* SW-SW communication.

Finally, *REG (SWT → HWT)* and *REG (HWT → SWT)* for one word of data take 1.35 μs and 1.76 μs, respectively.

As mentioned in Section 9.5, the work which is the most similar to the presented work in terms of system configuration is ReconOS [20, 34]. We compared the SW-HW communication based on FIFO semantics, implemented by mailboxes in case of ReconOS and FIFO channels in case of our OS4RS. For this type of communication, ReconOS achieves throughput of 0.13 MB/s, whereas our OS4RS 16.6 MB/s for the case where the SW task is a sender and 6.54 MB/s for the case where the HW task is a sender. This translates into 127.7x and 50.3x speedups, respectively. These results are for the case of communication over a 128-word deep channel. As the ReconOS' results presented in [34] have been conducted on 8 kB data, deeper FIFO channels are preferable.

9.8. HW-HW Communication: Performance Results. Figure 24 shows results for the HW-HW communication for the FIFO, MSG, and MEM channels. The results were measured using HW Tasks with their *HW Task Cores* running at 100 MHz. Similar to SW-HW communication tests, the tick-timer and UART interrupts were off, and HW tasks, which held the RR locks, communicated over the locked channels.

The calculated results are composed of two components. The first component is results taken at run-time, whereas the second one is the results taken from the logic simulator. At run-time, the time between the generation of the *Start Access* event and *End Access* event by the master side were measured. This gave us the time of the physical transfer started and completed by the master side. Additionally, the processing time of the corresponding event interrupt handlers was measured at run-time. In the simulator, the latencies between the start of access and the channel by the HW tasks and generation of event interrupts were measured, for both sides of the channel. These results were combined.

For *FIFO (MST write → SLV read)*, where a master HW task sends data to a slave HW task, communication takes from 5.89 μs to 8.53 μs for one-word data and 128-word data, respectively. It gives speedup of up to 89.61x with respect to corresponding *FIFO (LP → HP)* SW-SW communication.

For *FIFO (SLV write → MST read)*, where a master HW task receives data from a slave HW task, communication takes from 6.93 μs to 13.6 μs for one-word data and 128-word data, respectively. It gives speedup of up to 56.2x with respect to corresponding *FIFO (LP → HP)* SW-SW communication.

For *MSG (MST write → SLV read)*, communication takes from 10.53 μs to 13.17 μs for one-word data and 128-word data, respectively. It gives speedup of up to 15.28x with respect to corresponding *MSG (HP → LP → HP)* SW-SW communication.

For *MSG (SLV write → MST read)*, communication takes from 11.59 μs to 18.26 μs for one-word data and 128-word data, respectively. It gives speedup of up to 11.02x with respect to corresponding *MSG (HP → LP → HP)* SW-SW communication.

MEM (write) takes from 4.25 μs to 7.47 μs, whereas MEM (read) takes from 4.24 μs to 8.15 us for one-word data and 128-word data, respectively. Lower read time for one word, when compared to write time, can be explained by an error due to measurement precision.

When compared to results of memory read and write times for SW tasks, read and write times for HW tasks are 3.4x

higher when transferring single words. On the other hand, when it comes to writing 128-word data, HW tasks are 4.4x faster and, when it comes to reading it, 5.5x faster. The lower performance for small data quantities is mainly caused by the additional latency of the synchronization logic located at the *FCLK-VCLK* clock domain boundary and additional logic located at the SR-RR boundary. These latencies are compensated in case of bigger data quantities. It is because the interface at the aforementioned boundaries is optimized for streaming communication, and data over the bus is transfered in bursts. The SW tasks executing on the CPU transfer the data in single words.

As mentioned previously, the work which is the most similar to the presented work in terms of system configuration is ReconOS [20, 34]. We compared HW-HW communication based on FIFO semantics. In case of ReconOS, it is implemented by FIFO buffers accessed by HW tasks via an additional communication API call decoder and FIFO controller modules. In case of our OS4RS, it is implemented by the FIFO channel whose data transfer is conducted over the PLB bus. For the case of 128-word deep channel, we achieved throughput of 60.02 MB/s (case of FIFO MST write → SLV read) versus 127.20 MB/s possible for ReconOS. As the ReconOS' results presented in [34] were conducted on 8 kB data, deeper FIFO channels are preferable. If we extrapolated obtained results for a 512-word deep FIFO channel, we could get about 124 MB/s which is a similar result as for ReconOS. This is because the communication overhead over the FIFO channel is caused by the data transfer itself and event channel processing overhead of the software side of the OS4RS. The latter one can be assumed constant for all channel depths. Additionally, this is the first transferred word which undergoes the highest latency due to clock domain and SR-RR boundary crossing. Further improvements are possible by moving the processing related to HW-HW inter-task communication to hardware.

While the results of HW-HW communication for Rainbow are similar as for ReconOS, we should take into account additional advantage of dynamic frequency scaling and interconnect scalability offered by our approach. In ReconOS, FIFOs are connected externally to the RRs and each FIFO connects two RRs. Such an approach leads to quadratic increase in the logic area used by the interconnect with the number of RRs. While our approach has been evaluated with a bus, it is suitable for different and scalable types of interconnects such as NoC.

9.9. SW- versus HW-Based Processing Speedups. Figures 25 and 26 show the execution times of IDCT and SHA application, implemented in software (SW-SW), codesigned (SW-HW), and run fully in hardware. The column charts also show the breakdown of the execution time into bare software execution, bare hardware execution, and reconfiguration overhead and nonoverlapped OS overhead. The nonoverlapped OS overhead is this overhead created by the developed OS4RS which does not have impact on the total execution time of a given application. The charts also show the speedup of execution performed in SW-HW and HW-HW configurations versus SW-SW execution.

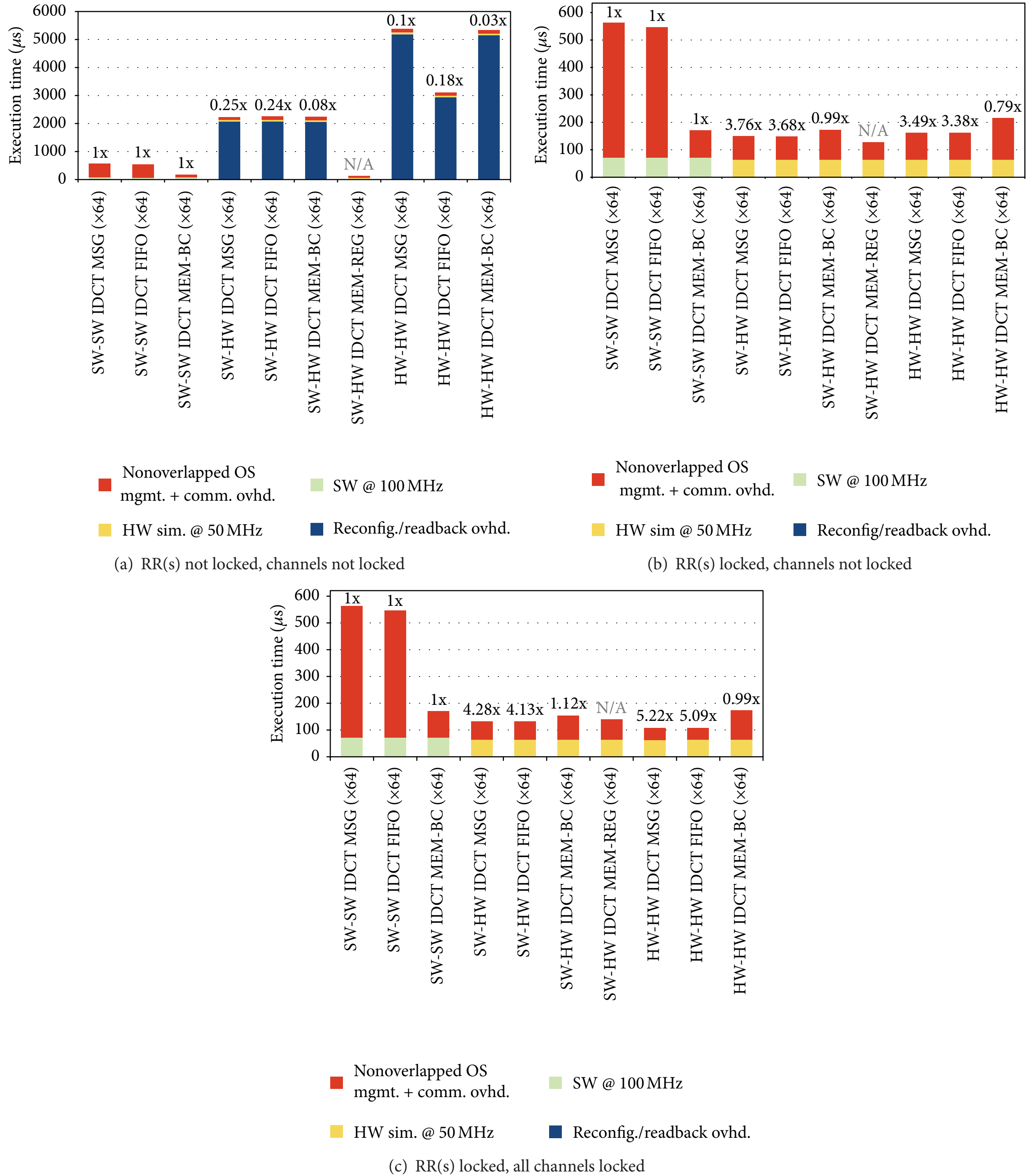
While analyzing the presented results, we should keep in mind the bare HW versus SW speedups achieved by the HW tasks synthesized with the HLS tool used in tests, that is, YXI eXCite HLS tool [57]. These were calculated by comparing the execution times of the algorithms in software and hardware. In software, the times were measured for the configuration where input and output data for the algorithm are contained in the local variables, and the instruction and data caches in the PPC405 are on. The execution times in hardware were measured in the logic simulator. The obtained speedup times are 1.12x for the IDCT cores and 1.49x–1.52x for the SHA cores in different FIFO depth configurations.

While taking the measurements, the tick-timer interrupts were enabled, whereas the UART interrupts were disabled. The test involved testbench tasks which provided the test vectors for the SW and HW tasks implementing the algorithms. The measurement for the SW-SW case was started once both the testbench, and the main SW task was activated, before the testbench task initiated the communication. The measurement was completed when the testbench task received the result.

In the SW-HW case, the testbench was located in software, and time measurement was done in the same way as in case of the SW-SW case. The HW task was activated before the tests. In the HW-HW case, additional channels were used in the HW testbench task which made it possible to send *Start* signal and receive *Done* signal when the communication was completed.

As we can see from the charts, the IDCT application benefits from moving some or all of its processing to hardware only if we restrict reconfigurations or even better prevent the *HW Task Dispatcher* from being called frequently, as it is done when the channel is locked. On the other hand, the SW-HW codesigned SHA application benefits from moving it to hardware even if we allow for reconfigurations. In the performed tests only a few reconfigurations were needed. While preemption can be used to increase FPGA utilization and response time [48], we should only use it at the times when it brings these benefits, while at other times execute the application in a locked manner, as allowed by the developed OS4RS. It should be noted that the SHA cores update their state during the whole execution. While the tests were performed for 8 kB data, the actual size of data streams could be much bigger. It could result in other tasks starving for resources, if the preemption was not allowed at all.

The obtained results could be further improved if the parallelizing optimizations were used during synthesis of the HW tasks with the HLS tool. Due to capacity limitations of the FPGA used in our experiments (Virtex-4 FX60) and large size of the static part of the system, we were compelled to restrict FPGA's area available for Reconfigurable Regions (RRs) and so the resources that can be used by HW Tasks. This, in turn, forced us to switch off any parallelizing optimizations in the HLS tool which could lead to increased logic area of the synthesized circuit. As a result, the HW tasks were synthesized as mostly sequential circuits with lower HW versus SW performance speedups. For this reason, the



(a) RR(s) not locked, channels not locked

(b) RR(s) locked, channels not locked

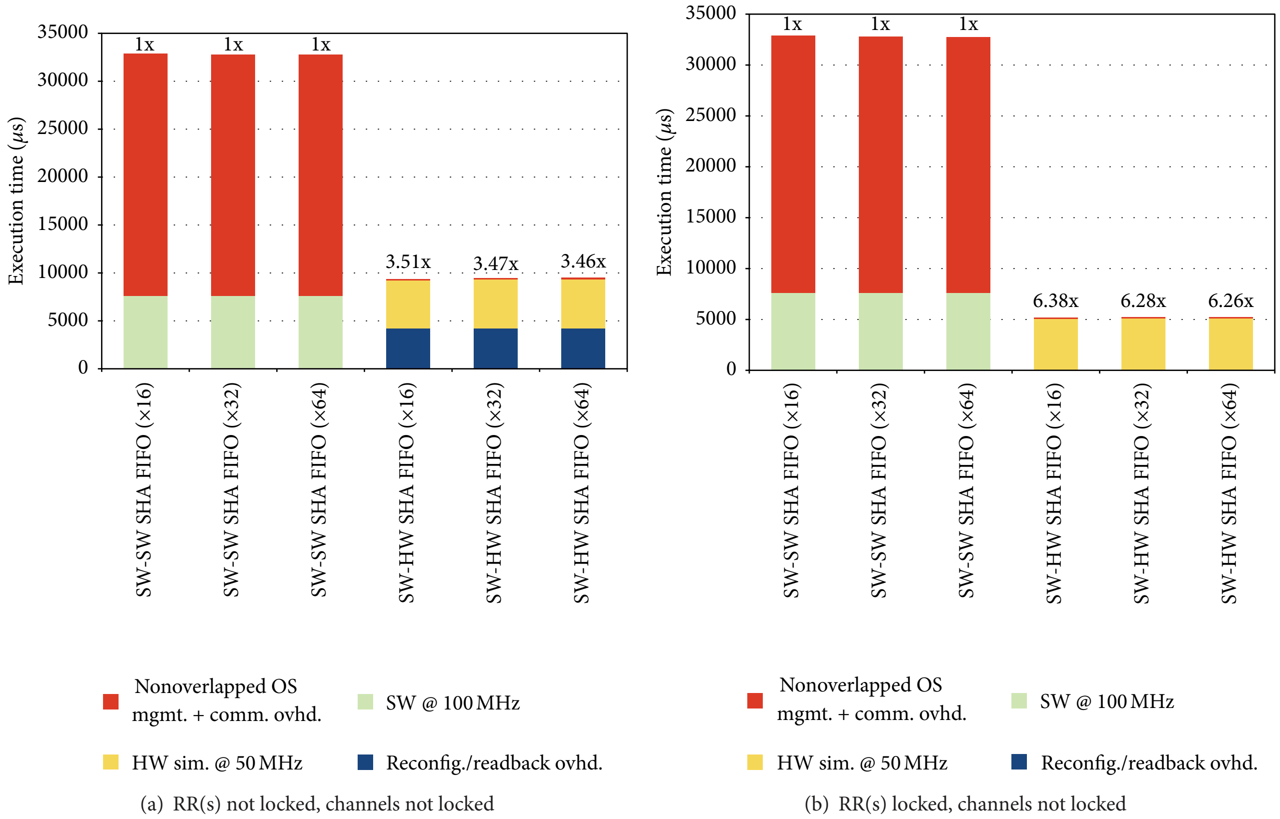
(c) RR(s) locked, all channels locked

Figure 25: HW versus SW processing speedups for IDCT.

obtained results can be further improved if newer and bigger FPGAs are used. As there are more efficient HLS solutions available on the market [61, 62], changing the HLS tool itself may also improve the results.

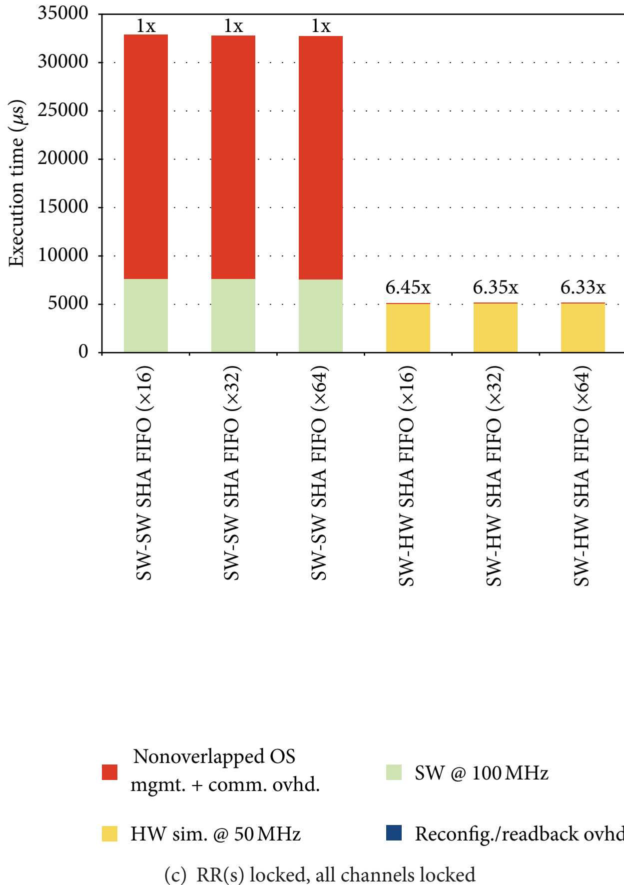
In case of the SHA HW task, after the SW task sends to the HW task, via the MSG channel, the size of the data stream to be processed, both sides have to communicate multiple times via the FIFO channel before the calculated data stream checksum can be received by the SW task. For the tested 8 kB size of the data stream, it is 128, 64, and 32 times for FIFO ×16, ×32, and ×64, respectively.

Although, the OS4RS overhead due to event processing may potentially limit the achieved speedups, here it is not the case. It comes from the way the SHA HW task does its



(a) RR(s) not locked, channels not locked

(b) RR(s) locked, channels not locked



(c) RR(s) locked, all channels locked

Figure 26: HW versus SW processing speedups for SHA.

processing. After receiving the size of the data stream to be processed, it starts receiving the consecutive chunks of data and processes them, till the complete bitstream is received. When the HW task reads the whole chunk of data from the channel, it makes the FIFO on its side empty. This triggers the *Data Notify Event* releasing SW task from waiting on FIFO and letting it sends the next chunk.

While the HW task does processing, the SW task is able to send the next part of the data stream. Since the processing of one chunk of data by the HW task takes only a bit less than processing of *Data Notify Event*, releasing the SW task from waiting and sending the data, the *Nonoverlapped OS Mgmt + Comm Overhead* shown in Figure 26 is very small. The REG channel is used only for SW-HW communication, thus the speedup with respect to SW-SW implementation could not be calculated. For this reason, it is shown as *N/A* in Figure 25.

9.10. Evaluation of Scheduling. In embedded systems with networking or these utilizing a virtual memory, a task may be temporarily blocked while waiting for its input data to be fetched from some remote locations or some secondary storages. Figure 27 presents case of the SHA HW task which is processing data being retrieved from the remote location and blocks when the next chunk of data becomes unavailable.

The packet delivery latencies differ depending on the type of network and its infrastructure. In case of systems communicating over the internet, delays in the range of a few to tens of milliseconds are common. Latencies of a few milliseconds can be also experienced by systems with memory virtualization which use HDDs as their secondary data storage.

Figure 27(a) shows a case where the latency of the communication link is high; thus it is advantageous to deallocate the blocked task and allocate another one (shown as X-task in the figure), thereby improving FPGA utilization and reducing application's overall execution time. As our results presented in Section 9.4.2 indicated, the reconfiguration and state restore time for the SHA HW task is about 2 ms, whereas its state saves time about 120 μs (Table 8). Also, the time required for clock scaling (frequency synthesis) is, for the FPGA used in our tests, in the range of 49.2–64.1 μs, as indicated in Section 9.5. Thus, the total overhead of switching HW tasks will be lower than the data delivery latencies. For the high-latency case, the HW task communicating over the link is activated with RR lock acquired before its execution and expiring immediately after it blocks. The RR lock is acquired to speedup the HW task's execution, as presented in our speedup results in the previous section. While in the Figure 27(a), the RR lock's timeout is set to zero, some small nonzero value could be used to filter out cases where the blocking time is very short.

Figure 27(b) shows a case where the latency of the communication link is low, thus justifying the decision to keep the task allocated on the FPGA, while it is blocked. For this low-latency case, the HW task communicating over the link is activated with RR lock acquired before its execution, and its timeout is set to some large values, greater than the mean blocking time. Timeout is needed here as a recovery measure when the HW tasks becomes nonresponsive for a long time, making the other tasks starving for the reconfigurable resources.

As shown in Figures 27(a) and 27(b), the Rainbow OS4RS allows the HW task scheduling to be adapted to match the latency characteristics of a given system. Moreover, by allowing dynamic frequency scaling, HW tasks may run at the clock frequencies allowing their maximum execution speedups.

ReconOS [32] implements cooperative scheduling which allows for deallocations only at predefined time points, that is, when a HW task blocks. Its scheduling result shown in Figure 27(c) is similar to the one presented for Rainbow in Figure 27(a). As ReconOS does not support clock scaling, clock frequency of all HW tasks executing in the assigned RR would be restricted by the slowest task in a set. For this reason, the total execution time of SHA and X tasks would be higher than for Rainbow.

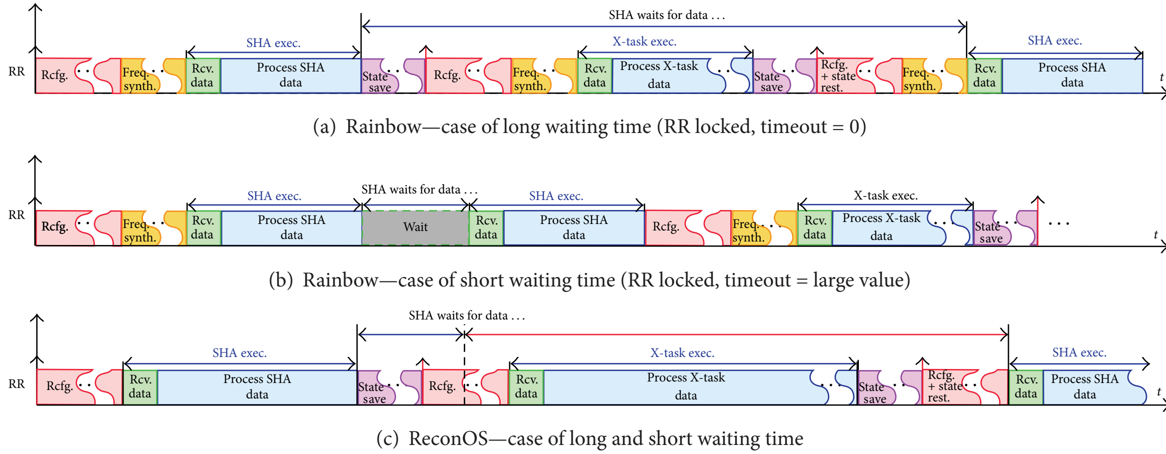
The clock scaling is restricted by the parameters of the frequency synthesizing logic (Digital Clock Manager in our case) and post-PAR clock frequency results for HW tasks. The performance improvement of an executing application which results from the higher clock frequency will be restricted by an additional time needed to perform the clock scaling. As the higher clock frequency will mostly affect the HW task execution phase only, the speed-up improvement for the whole application will be also restricted by the efficiency of the communication link and overhead of the OS4RS which is involved in the inter-task communication.

Our results in Section 9.9 showed that SHA task operating at 50 MHz, holding the RR lock and communicating over unlocked FIFO channel, requires about 5.1 ms to complete its execution. As at this clock frequency, the OS overhead related to communication is overlapped with the data processing conducted by the HW task, and further decrease in the clock frequency would only result in increase of its total execution time. The HW task executing on ReconOS could be restricted to run at frequencies far below 50 MHz increasing its total execution time by milliseconds. In this context, even additional 64.1 μs required for clock scaling would be worth spending. Also the same scenario could be possible for the X task.

Another point worth mentioning is that ReconOS would schedule execution of the SHA and X tasks irrespective of the blocking time as shown in Figure 27(c). For the low-latency case, this would further increase total execution time of the application by conducting time consuming and unnecessary switching of HW tasks.

Also, response time for any critical task that is activated while the X task, which performs very long processing, is running can be higher for ReconOS which does not allow for task preemption at arbitrary point in time, as Rainbow does.

Figures 28(a) and 28(b) compare scheduling results of Rainbow OS4RS and ReconOS [32] for the periodically executed IDCT HW task and SW task feeding it with data. *OS overhead* shown in the figure indicates inter-task communication overhead incurred by the OS4RS. In case of Rainbow, this includes channel event processing and transfer of data. In case of ReconOS, this mainly includes execution time of the delegate task which performs communication

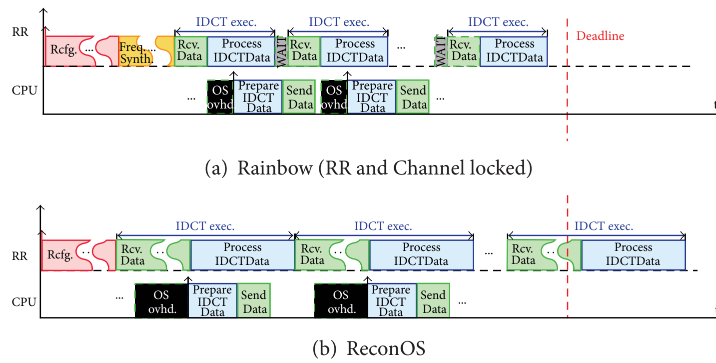


(a) Rainbow—case of long waiting time (RR locked, timeout = 0)

(b) Rainbow—case of short waiting time (RR locked, timeout = large value)

(c) ReconOS—case of long and short waiting time

Figure 27: Rainbow versus ReconOS (blocked HW task case).



(a) Rainbow (RR and Channel locked)

(b) ReconOS

Figure 28: Rainbow versus ReconOS (periodic execution case).

on behalf of the HW task by means of the communication primitives located in the software side of the OS4RS. As indicated in our results presented in Section 9.7, the latter one is significantly bigger. In Figure 28(a), *WAIT* indicates period of time where the HW task waits for the next chunk of data to be received.

For ReconOS, the clock frequency of the IDCT task could be restricted in the same way as in case of the SHA task, described before. This would in turn increase the total execution time of the IDCT task.

The longer communication and execution times described above would cause ReconOS to miss deadline for the IDCT task executed periodically. This is shown in Figure 28.

10. Conclusions and Future Works

In this paper we presented a complete model and implementation of the lightweight and portable OS4RS supporting preemptable and clock-scalable HW tasks. While DFS was discussed in the context of FPGAs by the previous works, none of them proposed a complete model and implementation of the OS4RS architecture supporting this concept. The DFS allows improveing performance of the SW-HW codesigned applications and avoid some of the restrictions imposed by the underlying DPR technology.

We showed a novel scheduling mechanism based on timely and priority-based reservation of reconfigurable resources allowing for use of preemption only at the time it brings benefits to the total performance of the system. The architecture of the scheduler and the way it schedules allocations and deallocations of the HW tasks on the FPGA array results, as presented in our evaluation, in shorter latency of API calls, thereby reducing the overall OS overhead.

Last but not the least, we presented a novel model and implementation of a channel-based inter-task communication and synchronization suitable for SW-HW multitasking with preemptable and clock-scalable HW tasks. When compared to previous approaches, the model and its implementation allow for optimizations of the communication on per task basis and takes advantage of more efficient point-to-point message passing rather than shared-memory communication, whenever it is possible. While the model has been implemented on top of the bus interconnect, it is suitable for more efficient and scalable interconnects, such as NoC, where its benefits would be even more visible. Our evaluation results proved the efficiency of the proposed model and its implementation.

The overall performance results could be further improved if better HLS tools were used. A prospective area of future work includes, but is not limited to, moving OS4RS

processing related to HW-HW communication to FPGA and more complex application tests.

References

[1] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *IEE Proceedings*, vol. 153, no. 3, pp. 157–164, 2006.

[2] W. Fornaciari and V. Piuri, "Virtual FPGAs: some steps behind the physical barriers," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '98)*, 1998.

[3] G. Brebner, "A virtual hardware operating system for the Xilinx XC 6200," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL '96)*, 1996.

[4] G. Brebner, "Automatic identification of swappable logic units in XC6200 circuitry," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL '97)*, 1997.

[5] G. Wigley and D. Karney, "The first real operating system for reconfigurable computers," in *Proceedings of the Computer Systems Architecture Conference*, 2001.

[6] P. A. Hsiung, M. D. Santambrogio, C. H. Huang et al., *Reconfigurable System Design and Verification*, CRC Press, 2008.

[7] B. Walder and M. Platzner, "A runtime environment for reconfigurable hardware operating systems," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL '04)*, 2004.

[8] 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.

[9] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.

[10] Free Software Foundation. eCos Project, http://ecos.sourceware.org/.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: from prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[12] 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.

[13] A. Ali, M. Jomaa, B. Romanous et al., "An operating system for a reconfigurable active SSD processing node," in *Proceedings of the 19th International Conference on Telecommunications (ICT '12)*, 2012.

[14] L. Ye, J.-P. Diguet, and G. Gogniat, "Rapid application development on multi-processor reconfigurable systems," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 285–290, September 2010.

[15] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "An FPGA run-time system for dynamical on-demand reconfiguration," in *Proceedings of the Reconfigurable Architectures Workshop (RAW '04)*, pp. 1841–1848, April 2004.

[16] M. D. Santambrogio, V. Rana, and D. Sciuto, "Operating system support for online partial dynamic reconfiguration management," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 455–458, September 2008.

[17] I. Beretta, V. Rana, M. D. Santambrogio, and D. Sciuto, "On-line task management for a reconfigurable cryptographic architecture," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, May 2009.

[18] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *Transactions on Embedded Computing Systems*, vol. 7, no. 2, article 14, 2008.

[19] 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, August 2007.

[20] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 17–22, September 2008.

[21] D. Andrews, R. Sass, E. Anderson et al., "Achieving programming model abstractions for reconfigurable computing," *IEEE Transactions on Very Large Scale Integration*, vol. 16, no. 1, pp. 34–43, 2008.

[22] MathWorks Simulink, http://www.mathworks.com/products/simulink.

[23] V. Nollet, P. Coene, D. Verkest et al., "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '03)*, 2003.

[24] TOPPERS project. Official website, http://www.toppers.jp/.

[25] 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.

[26] J. A. Clemente, J. Resano, C. González, and D. Mozos, "A Hardware implementation of a run-time scheduler for reconfigurable systems," *IEEE Transactions on Very Large Scale Integration*, vol. 19, no. 7, pp. 1263–1276, 2011.

[27] F. Ghaffari, B. Miramond, and F. Verdier, "Run-time HW/SW scheduling of data flow applications on reconfigurable architectures," *Eurasip Journal on Embedded Systems*, vol. 2009, Article ID 976296, 2009.

[28] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*, pp. 123–128, fra, April 2007.

[29] F. Dittmann and S. Frank, "Caching in real-time reconfiguration port scheduling," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 740–744, August 2007.

[30] Y. Qu, J.-P. Soininen, and J. Nurmi, "Improving the efficiency of run time reconfigurable devices by configuration locking," in *Proceedings of the IEEE Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 264–267, March 2008.

[31] 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)*, April 2010.

[32] E. Lübbers and M. Platzner, "Cooperative multithreading in dynamically reconfigurable systems," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 551–554, September 2009.

[33] Xilinx, Embedded Development Kit (Xilkernel), http://www.xilinx.com/tools/platform.htm.

[34] E. Lübbers and M. Platzner, "Communication and synchronization in multithreaded reconfigurable computing systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '08)*, pp. 83–89, July 2008.

[35] M. Vuletić, L. Pozzi, and P. Ienne, "Seamless hardware-software integration in reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 102–113, 2005.

[36] X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez, "Methods and mechanisms for hardware multitasking: executing and synchronizing fully relocatable hardware tasks in xilinx FPGAs," in *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL '11)*, pp. 295–300, September 2011.

[37] K. Kosciuszkiewicz, F. Morgan, K. Kepa et al., "Run-time management of reconfigurable hardware tasks using embedded linux," in *Proceedings of the IEEE Computer Symposium on Emerging VLSI Technologies and Architectures*, 2006.

[38] S. Mahadevan, V. S. Gopinath, R. Lysecky, J. Sprinkle, J. Rozenblit, and M. W. Marcellin, "Hardware/software communication middleware for data adaptable embedded systems," in *Proceedings of the 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS '11)*, pp. 34–43, April 2011.

[39] S. Narayanan, D. Chillet, S. Pillement, and I. Sourdis, "Hardware OS communication service and dynamic memory management for RSoCs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, pp. 117–122, December 2011.

[40] J. Congfeng, "System level power characterization of multi-core computers with dynamic frequency scaling support," in *Proceedings of the International Conference on Cluster Computing Workshops*, 2012.

[41] Intel, *Enhanced Intel SpeedStep Technology For the Intel Pentium M Processor*, White Paper, 2004.

[42] A. Tiwari, "A partial reconfiguration based approach for frequency synthesis using FPGA," in *Proceedings of the International Conference on Communication Technology and System Design (ICCTSD '11)*, 2011.

[43] C. Schuck, B. Haetzer, and J. Becker, "Reconfiguration techniques for self-X power and performance management on xilinx virtex-II/virtex-II-pro FPGAs," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 671546, 2011.

[44] 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, June 2010.

[45] X. Iturbe, K. Benkrid, R. Torrego et al., "Online clock routing in Xilinx FPGAs for high-performance and reliability," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2012.

[46] Xilinx, "UG702 (v12. 3): partial reconfiguration user guide," Tech. Rep., 2010.

[47] Altera, "Quartus II Handbook," Tech. Rep., Design and Synthesis, 2012.

[48] K. Jozwik et al., "Comparison of preemption schemes for partially reconfigurable FPGAs," *IEEE Embedded Systems Letters*, vol. 4, no. 2, 2012.

[49] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[50] H. Kalte and M. Porrmann, "REPLICA2Pro: task relocation by bitstream manipulation in virtex-II/Pro FPGAs," in *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*, pp. 403–412, May 2006.

[51] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pp. 35–44, April 2007.

[52] J. Carver, "Relocation of FPGA partial configuration bit-streams for soft-core microprocessors," in *Proceedings of the Workshop on Soft Processor Systems*, 2008.

[53] S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration*, vol. 17, no. 11, pp. 1650–1654, 2009.

[54] Xilinx, "UG070 (v2. 5): virtex-4 FPGA user guide," Tech. Rep., 2008.

[55] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 352–355, September 2010.

[56] P. A. Laplante, *Real-Time Systems Design and Analysis*, Wiley-Interscience, IEEE Press, 3rd edition, 2004.

[57] Y Explorations eXCite, http://www.yxi.com/.

[58] C. E. Cummings, "Simulation and synthesis techniques for asynchronous FIFO design," in *Proceedings of the Synopsys Users Group Conference (SNUG '01)*, 2001.

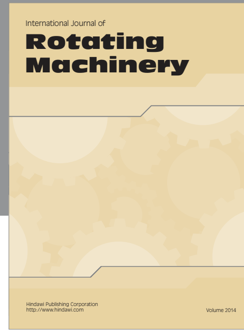
[59] Xilinx, "DS302 (v3. 7): virtex-4 FPGA data sheet: DC and switching characteristics," Tech. Rep., 2009.

[60] Y. Hara, H. Tomiyama, S. Honda et al., "Proposal and quantitative analysis of the chstone benchmark program suite for practical C-based high-level synthesis," *IPSJ Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

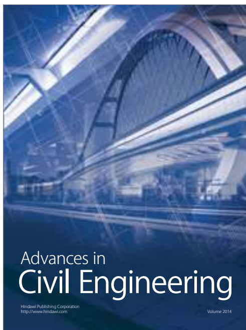
[61] NEC CyberWorkBench, http://www.nec.com/cyberworkbench.

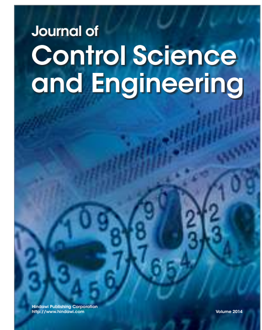
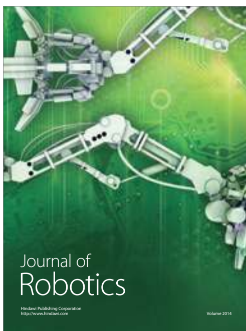
[62] Xilinx, "Vivado Design Suite," http://www.xilinx.com/vivado.

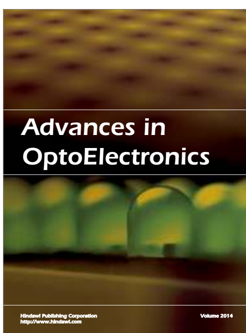
Journal of
Engineering

The Scientific World Journal

International Journal of
Rotating Machinery

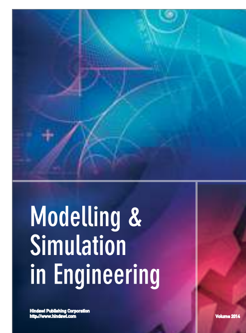
Journal of
Sensors

International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering

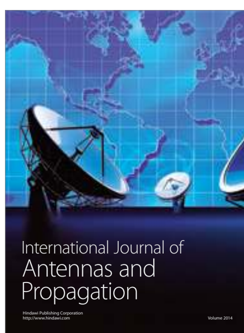
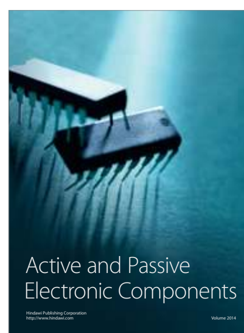
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science and Engineering

Journal of
Robotics

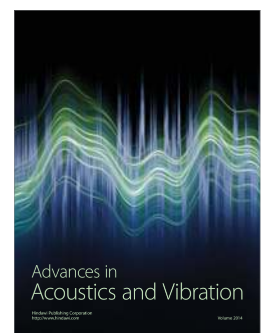
Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration