R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs

Xabier Iturbe, Khaled Benkrid, *Senior Member*, *IEEE*, Chuan Hong, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan, *Member*, *IEEE*, and Jon Perez, *Member*, *IEEE*

Abstract—Despite the clear potential of FPGAs to push the current power wall beyond what is possible with general-purpose processors, as well as to meet ever more exigent reliability requirements, the lack of standard tools and interfaces to develop reconfigurable applications limits FPGAs' user base and makes their programming not productive. R3TOS is our contribution to tackle this problem. It provides systematic OS support for FPGAs, allowing the exploitation of some of the most advanced capabilities of FPGA technology by inexperienced users. What makes R3TOS special is its nonconventional way of exploiting on-chip resources: These are used indistinguishably for carrying out either computation or communication tasks at different times. Indeed, R3TOS does not rely on any static infrastructure apart from its own core circuitry, which is constrained to a specific region within the FPGA where it is implemented. Thus, the rest of the device is kept free of obstacles, with the spare resources ready to be used as and whenever needed. At runtime, the hardware tasks are scheduled and allocated with the dual objective of improving computation density and circumventing damaged resources on the FPGA.

Index Terms—Dynamic partial reconfiguration, adaptivity, hardware virtualization, reliability, reconfigurable computing

1 Introduction

Nowadays, with the amount of on-chip resources that can be exploited at any particular time limited by the so-called frequency, or voltage or power wall, the *online specialization* offered by partially reconfigurable FPGAs emerges as a promising way to combine computation in space and time to obtain the best performance per transistor count and unit of consumed energy. Indeed, the same on-chip resources can be reused to efficiently implement different functionalities over time [1]. Furthermore, the same flexibility that makes online specialization possible permits to solve or at least mitigate the reliability concerns that appear when pushing semiconductor manufacturing to its physical limits. Reconfigurable hardware allow for building systems capable of keeping their architecture fault free at all times by reconfiguring around damaged portions of the chip [2].

However, despite the numerous advantages FPGAs are to bring, their success is highly conditioned by the way they reach application developers. After decades of prevalence, software programming style has spread through all application domains, including even those which were traditionally hardware-centered, and thus, it seems impossible to radically change this situation without a major collapse in productivity and increase in cost. Faced with this, the concept of a reconfigurable operating system (ROS) to give to reconfigurable hardware a "software look and feel" has been gaining momentum since the mid-1990s [3], [4], [5], [6].

Following this trend, this paper presents a novel ROS, named as reliable reconfigurable real-time operating system (R3TOS), which is aimed at satisfying the often conflicting requirements of high-performance, real-time, fault-tolerance, and high-level programming.

The remainder of this paper is organized as follows: After an overview of related work in Section 2, Section 3 is dedicated to describing the foundations of R3TOS. Section 4 details the R3TOS computing model, and Section 5 then presents the R3TOS overall architecture. Next, Section 6 covers R3TOS application programming interface (API), after which Section 7 describes a proof-of-concept prototype using Xilinx Virtex-4 FPGAs. Section 8 presents a case study design and implementation based on software-defined radio (SDR) application, and finally, concluding remarks are discussed in Section 9.

- *X. Iturbe is with the System Level Integration Research Group, School of Engineering and Electronics, The University of Edinburgh, King's Buildings, Edinburgh EH9 3JL, Scotland, United Kingdom, and the Embedded System-on-Chip Group, IK4-Ikerlan Research Center, Arrasate-Mondragón 20500, Basque Country, Gipuzkoa, Spain. E-mail: x.iturbe@ed.ac.uk, xiturbe@ikerlan.es.*
- *K. Benkrid, C. Hong, A. Ebrahim, and T. Arslan are with the System Level Integration Research Group, School of Engineering and Electronics, The University of Edinburgh, King's Buildings, Edinburgh EH9 3JL, Scotland, United Kingdom. E-mail: {k.benkrid, c.hong, a.ebrahim, t.arslan}@ed.ac.uk.*
- *R. Torrego, I. Martinez, and J. Perez are with the Embedded System-on-Chip Group, IK4-Ikerlan Research Center, Arrasate-Mondragón 20500, Basque Country, Gipuzkoa, Spain. E-mail: {rtorrego, imartinez, jmperez}@ikerlan.es.*

Manuscript received 1 Sept. 2012; revised 15 Jan. 2013; accepted 24 Feb. 2013; published online 4 Apr. 2013.
Recommended for acceptance by K. Benkrid, D. Keymeulen, U.D. Patel, and D. Merodio-Codinachs.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2012-09-0619.
Digital Object Identifier no. 10.1109/TC.2013.79.

2 Related Work

The term ROS was coined by Brebner [3], and it is essentially referred to a software OS augmented with functions to manage reconfigurable hardware, for example, FPGAs, and execute hardware applications on it. The rational of the ROS is to hide complexity by offering a set of useful services to the application developer. These services should be accessible through an API and should provide runtime support for both task management and FPGA resource management. In [7], the fundamental services to be implemented in an ROS are identified: task loading, memory management, scheduling and allocation, communications (both hardware hardware and hardware software), and input/output. Various attempts to build an ROS for FPGAs can be found in the technical literature. The most significant ones are summarized in the following paragraphs.

OS4RS was a very early ROS prototype developed by IMEC with the main focus of giving runtime support for multimedia applications [4]. Unfortunately, very little information is provided about OS4RS implementation and functioning. Most of the information is related to the major innovation proposed: the possibility to interrupt a hardware task and restart it in software, or vice versa. Notably, this idea has inspired later work, for example, [8], [9].

In [10], Xilinx Inc. provided the XPART API, which was intended to ease the management of FPGA resources. Unfortunately, XPART was rapidly discontinued. In [11], the authors created a Linux driver for FPGA's internal configuration access port (ICAP) and used it in an embedded Linux distribution, i.e., uClinux, running on a Xilinx MicroBlaze processor. This can be considered the first successful attempt to make reconfigurable hardware easily accessible by a software-centric programmer. Later, the same authors completed their work with a Linux driver that allowed first-in, first-out (FIFO)-based data communications with reconfigurable hardware modules [12]. However, we note that the software implementation of these ICAP drivers may potentially lead to significant time overheads when the system workload is high.

HybridThreads (HThreads) has been developed by the University of Kansas [14]. HThreads allows programmers to run software and hardware threads, simultaneously on a CPU and on an FPGA, i.e., Virtex-II Pro. Notably, scheduling, communication, and synchronization services are implemented in hardware, bringing significant performance benefits. However, in HThreads, the hardware threads remain allocated on the FPGA even when they are idle, i.e., reconfiguration is not used. Consequently, HThreads cannot be considered a complete ROS as it fails to manage FPGA resources, i.e., FPGA resources are not shared among the threads.

BORPH was developed by the University of California, Berkeley [6]. It is distributed among five Virtex-II Pro FPGAs: One of them acts as master (control FPGA) and the remaining four implement some control logic (called uK) and allocate the hardware tasks. Because of this, these FPGAs are named as user FPGAs. The control FPGA is connected to the SelectMAP pins of the user FPGAs through a point-to-point, bidirectional, 8-bit bus running at 50 MHz. This bus serves the dual role of configuring the hardware tasks in the user FPGAs, and communicating with the uKs after the functions are configured. BORPH offers an UNIX-like API. Its software kernel is an extended version of Linux 2.4.30 [15], which runs in a PowerPC 405 core in the control FPGA. Communication between hardware and software tasks is implemented by FIFOs and mapped to file descriptors. In BORPH, hardware tasks are assigned to user FPGAs in one-to-one fashion, leading to a very inefficient exploitation of hardware resources. Furthermore, the amount of concurrent tasks running on the system is limited by the number of user FPGAs. In this context, BORPH does not require any specific scheduling or allocation algorithms.

ReconOS was developed by the University of Paderborn and can be seen as a porting of BORPH to a single FPGA, i.e., Virtex-II or Virtex-4, making special emphasis on real-time performance [16]. The user FPGAs of BORPH are assigned separate reconfigurable slots in the same FPGA in ReconOS. These slots are coupled with a control logic (called OSIF), which implements the same function as uK does in BORPH. Being contained in a single FPGA, the user functions are configured through ICAP, and communications are performed through an on-chip bus running at 100 MHz. In light of achieving real-time performance, ReconOS offers an eCOS-based API, which is extended with specific system calls to manage hardware tasks. Allocation and scheduling do not deserve special attention in ReconOS: Scheduling decisions are made by the eCOS kernel and allocation decisions are trivial as there are only a few slots where to map the hardware tasks.

A recent approach that is conceptually close to ReconOS is FUSE, developed by the Simon Fraser University [17]. FUSE relies on a slotted reconfigurable system that is implemented on a Virtex-5 FPGA and provides an embedded Linux-based API with POSIX threads running on a MicroBlaze core. Two features of FUSE are especially interesting. First, shared memories are used to exchange data between the software and hardware tasks, thus reducing data communication overheads. Second, each hardware task is associated a loadable kernel module (LKM) that implements miscellaneous device driver functionality, allowing to treat hardware tasks as memory-mapped I/O device peripherals.

Finally, CAP-OS is being developed by the Karlsruhe Institute of Technology, and it is intended to handle a variety of processors and accelerators under real-time constraints, using Virtex-4 FPGAs [18]. The API offered by CAP-OS is based on message passing interface (MPI). As proposed in OS4RS, the computations in CAP-OS are to be performed either in software, i.e., by any of the processors, or in hardware, i.e., as a coprocessor. However, the currently presented prototype only supports executing software tasks, which can be loaded into the processor's program memory either through a network-on-chip (NoC) or through the ICAP. Notably, CAP-OS uses a priority-based scheduling algorithm that considers both ICAP exclusiveness and intertask dependencies. The latter scheduling algorithm as well as other CAP-OS processes run as separate threads in an embedded processor, for example, MicroBlaze or PowerPC 405, which is equipped with Xilinx Xilkernel multithreading solution. In the future, the authors expect to include the capability to configure hardware tasks upon request by the processors as well as to modify the number of processors in the system. Toward this end, bitstream relocation is pointed as necessary, leading us to understand that the current CAP-OS prototype relies on a slotted architecture.

We conclude that all of the proposed ROS up to date are based on slotted reconfigurable systems, where the size and shape of the slots are set by the user at design time and cannot be modified at runtime. This limits reconfigurability opportunities in the system, provoking some restrictions to apply. First, FPGA resources are inefficiently exploited as all of the tasks to be allocated on a given slot must be enlarged to its size. Second, the number of tasks that can run concurrently is limited by the amount of reconfigurable slots available in the system. Finally, a single fault affecting a slot might well make that slot useless.

3 R3TOS Foundations

R3TOS is founded on the basis of *resource reusability* and *computation ephemerality*. It makes intensive use of reconfiguration at the finest granularity of Xilinx FPGAs, keeping the resources available to be used by any incoming task at any time. Hence, computing tasks and associated supporting circuitry (e.g., intertask communication channels, clock distribution wires, etc.) are configured when required and removed when they are no longer needed. Therefore, FPGA resources can be used either for computation or for communication purposes at different times. Indeed, R3TOS does not rely on any prerouted communication infrastructure, instead it creates on-demand communication channels among the tasks on the fly.

In R3TOS, the control logic to drive the tasks is attached to their own circuitry, making them self-contained and closed structures that are fully relocatable within the FPGA [19]. This is a completely different approach when compared to related state of the art, where the (enlarged) hardware tasks are executed in predefined reconfigurable slots coupled with fixed control logic and connected to a static communication infrastructure to exchange data among them.

The task control logic (TCL) includes an input data buffer (IDB), an output data buffer (ODB), and a hardware semaphore (HWS) to enable/disable computation. TCLs provide a means to virtually lock physical data and control inputs/outputs of the hardware tasks to logical positions in the configuration memory of the FPGA. Since the TCLs are accessible through the configuration interface whichever memory positions they are mapped to, the allocatability of the tasks is not constrained by the position of the communication interfaces decided at design time anymore. Furthermore, this scheme improves multitasking capabilities as the number of tasks that can be concurrently executed on FPGA is only limited by the amount of on-chip FPGA resources. Finally, note that the fact of replacing "physical" bus macros (BMs) with "logical" TCLs is in consonance with the current trend to hide hardware related aspects to the designer. For example, the latest release of Xilinx tools to develop reconfigurable systems are able to manage the BMs on behalf of the designer, who has not to worry about where and how many BMs are instantiated.

There are several immediate advantages resulting from the innovative use of FPGA resources in R3TOS.

The allocatability of the tasks is improved as the FPGA area is kept free of non-necessary obstacles at all times, for example, static routes. This results in higher flexibility to allocate the tasks around the damaged resources (improved fault tolerance) and to increase the computation density by compacting the tasks in the chip (improved efficiency). Furthermore, the complexity of the allocation algorithms is simplified as they do not need to be aware of any underlying implementation-related irregularities in the reconfigurable area. Note that a traditional reconfigurable system must preserve the static routes, resulting in additional difficulties that penalize the performance.

Tasks can be deallocated very quickly using multiple frame write (MFWR) configuration commands as there is no need to preserve any element within the region occupied by them, i.e., task deallocation simply consists in blanking the whole content of the corresponding configuration frames.

The fact that we have separately and individually fed each task with its highest allowed clock frequency clearly outperforms the easy and more usually chosen option of clocking the entire system at the slowest rate [20].

Since R3TOS relocates the circuitry along the entire device, switching activity naturally tends to distribute among all the resources. As a result, the device ages uniformly, i.e., wear is leveraged, delaying the occurrence of damage [21], [22], [23]. This does not happen in traditional systems where some of the resources are prone to fail earlier due to intensive use, for example, the resources used to implement the static communication infrastructure.

On the other hand, the limitations associated with R3TOS mainly come from the reconfiguration bottleneck provoked by the ICAP. First, the configuration of a hardware task delays its execution by a non-negligible amount of time. Second, the configuration of on-demand communication channels among the tasks incurs an overhead that is significantly greater when compared to the time needed for establishing a virtual connection trough an NoC or to the nearly zero communication delay in an on-chip bus.

4 The R3TOS Computing Model

R3TOS envisions a real-time multitasking scenario, which supports the benefits brought about by reconfigurable hardware, for example, true hardware multitasking and computation specialization, without significantly modifying the traditional programming style. Indeed, task definitions and their interactions are described using parallel software programming syntax (e.g., POSIX Pthreads), but the body of some of the tasks (hardware tasks) is implemented in hardware.

The term "hardware task" is used to reflect the fact the task relies on specific purpose custom circuitry to perform computation, but this does not mean it does not include any software component inside. Indeed, the custom circuitry could be used as a hardware accelerator by a processor running a software program inside the task.

R3TOS addresses two main types of hardware tasks: *Data-stream processing tasks*, to be used in data-intensive applications with regular dependencies, and *hardware-acceleration tasks*, which speed up the execution of portions of computationally intensive software code.

At design time, the user can rely on the R3TOS API to programm his/her reconfigurable application. The API includes a set of system calls to give seamless support for both software and hardware task invocation (e.g., similar to POSIX `pthread_create()` or `fork()`), synchronization (e.g., similar to POSIX `pthread_join()` or `sem_wait()`), and intertask communications (e.g., equivalent to POSIX `msgsnd()` and `msgrcv()`). Therefore, the user only needs to describe his/her reconfigurable application, for example, define the triggering conditions for each task, specify the real-time requirements, missed deadlines handling, and so on.

At runtime, the execution of the tasks is controlled by a main CPU based on the user specifications. Basically, the latter executes a program that is conceptually similar to a traditional RTOS. Indeed, it is based on a software real-time microkernel (SWuK), which is extended with extra functionality to interact with the hardware microkernel (HWuK); i.e., it schedules-executes software tasks and forwards hardware tasks to HWuK.

HWuK gives thus support to the main CPU to deal with the hardware tasks, serving as the substrate upon which the hardware-related services offered by the R3TOS API are built. Namely, R3TOS HWuK includes: 1) a *scheduler* server, expressly designed for scheduling hardware tasks, 2) an *allocator* server to manage FPGA resources, and 3) a *configuration manager* to translate the high-level operations dictated by scheduler and allocator servers into reconfiguration commands for the FPGA. Note that the latter implements most of the hardware abstraction layer (HAL) of R3TOS.

The configuration information to build the circuitry required by the hardware tasks, including the binary code of the software components inside, if any, is stored in a bitstream memory. At runtime, the hardware tasks are allocated to different positions within the FPGA by appropriately changing the frame addresses when their bitstreams are transferred from the bitstream memory to the configuration memory of the FPGA. To achieve the highest reconfiguration bandwidth, the task's bitstream relocation process is done by a dedicated hardware, operating at the highest allowed clock frequency by the ICAP. Consequently, the limited physical FPGA resources are used to implement a large virtual computing resource that is time shared to serve multiple hardware tasks upon request. Each of the tasks uses its own private piece of FPGA resources.

To improve performance, the configuration memory of the FPGA is used as a cache for both tasks and data. Hardware tasks are deallocated from the FPGA only when their resources are required by other coming tasks, and the partial results computed by them are uploaded to the main memory only when they are required by a software task.

Both software and hardware tasks are assigned a space in the main memory attached to the main CPU to hold local data and, in the case of software tasks, to hold the code as well. Since the main memory is shared between both hardware and software tasks, here is where the interactions between both types of tasks occur, as defined by the user at design time and, as implemented by the main CPU at runtime. An advantage of this scheme is its compatibility with most software compilation systems and most task memory mappings of software OS.

To simplify hardware-software communications, a fixed region within the main memory is shared between the main CPU and the HWuK to exchange data through. This region is organized in the form of an ODB and an IDB. The data written by the CPU in the ODB is finally delivered by HWuK to the corresponding hardware task running on the FPGA. Likewise, the data written by HWuK in the IDB is finally relocated by the CPU to the specific data segment assigned to the corresponding software task in the main memory. Therefore, HWuK cannot directly access the data segments of the tasks in the main memory, and the main CPU cannot access the hardware tasks in the FPGA, guaranteeing no interference between them.

4.1 Application Design Phase

The design phase starts with the conceptualization and refinement of specifications to produce a behavioral model of the reconfigurable application. The decision about which functionality should be implemented in hardware and which in software is made based on estimations about cost and performance (e.g., required silicon area, hardware response time, occupied program memory, and software execution time). This codesign process is a challenge in its own right [24], but it is out of the scope of this paper. We assume that the reconfigurable application is successfully decomposed into a set of software functions (SFs), to be executed in the main CPU, and a collection of processing elements (PEs), to be implemented using FPGA resources. There is no exact definition for PEs; they are assumed to be components that transform a set of input data into a set of output results using some specific-circuitry to speed up this process. Therefore, PEs turn the fine-grained and generic FPGA resources into coarse-grained and specific computing machines.

When partitioning the reconfigurable application, related PEs are grouped together into hardware tasks to reduce the longest path. The latter is a key issue toward high performance as R3TOS is able to execute each task at its maximum allowed clock frequency at runtime.

In a second phase, loosely coupled PEs in time or content must be extracted from the tasks until the desired granularity is achieved. Content-coupled PEs assigned to different hardware tasks result in intertask communications, where the amount of data to exchange depends on the coupling level. On the other hand, the PEs mapped to the same tasks include inherent RTL communications that benefit from the huge capability to move data among registers and memory elements delivered by massively parallel FPGAs. Hence, there is a tradeoff between intertask communication bandwidth requirement and task granularity.

We distinguish between two types of tasks based on their communication requirements. *High-bandwidth communication (HBC) tasks* refer to tasks that process a high amount of data within a relatively short amount of time, i.e., communication dominates computation. On the other hand, *low-bandwidth communication (LBC) tasks* refer to tasks that process a reduced amount of data within a relatively long amount of time, i.e., computation dominates communication. Note that task communication is a commonly used feature to classify the tasks. For instance, in [25], the "number of operations per transferred data item" is used as main feature to develop a machine-learning-based compiler that is proved to find efficient task partitioning.

After having partitioned the reconfigurable application into tasks, related PEs are synthesized together with a wrapping TCL to obtain a single-merged relocatable partial bitstream for each of the hardware tasks. Note that the tasks are separately synthesized and constrained to specific closed regions within the FPGA.

On their part, the SFs are compiled together with the application description and the R3TOS API to generate the application executable, including the code of the software tasks.

Then, the partial bitstreams of the hardware tasks are uploaded to the bitstream memory, and the application executable is uploaded to the main memory. The design phase concludes with the merging of application-dependent information (e.g., task dependencies, real-time deadlines, etc.) and target FPGA-dependent information (e.g., chip size, layout, FPGA Device ID, etc.) into R3TOS to create the appropriate execution environment for the reconfigurable application. This information is to be collected and formatted by a feature extractor script. Additionally, R3TOS is provided with specific implementation-dependent information (e.g., HWuK placement within the FPGA).

4.2 Intertask Communications and Synchronization

The reconfigurable application can be modeled as a directed acyclic graph (DAG) where *vertices* represent tasks and *edges* represent intertask communication channels. Vertices produce and consume data from edges, which in turn buffer the data in an FIFO fashion. Note that buffering capability is mandatory in R3TOS as the tasks can be executed at different slots of time. Besides this, synchronization is mandatory to coordinate data producers and consumers when accessing the communication channels.

TCLs attached to the hardware tasks provide support for communications, synchronization, and data buffering.

Indeed, the TCL delivers the data to be processed by the associated task from its internal IDB to the PEs and stores the subsequent results computed by the latter in its internal ODB. Hence, the data buffers of the TCL are functionally equivalent to the FIFO queues commonly inserted in-between data processing pipeline stages or to the local caches used in traditional processors. Therefore, TCLs provide hardware tasks with dedicated access to their data buffer. The data buffers of HBC tasks are implemented using high-density storage resources (namely, BRAMs), while the buffers of LBC tasks are implemented using low-density storage resources (namely LUT-RAMs).

After a task completes its computation, the computed results remain stored in its former ODB until they are required as input by another task. Temporarily buffering the partial results along the entire chip, which is a result of the flexible allocation scheme used in R3TOS, is the natural way to exploit the distributed nature of FPGA fabrics. Hence, hardware tasks leave a *data trace* in the user data

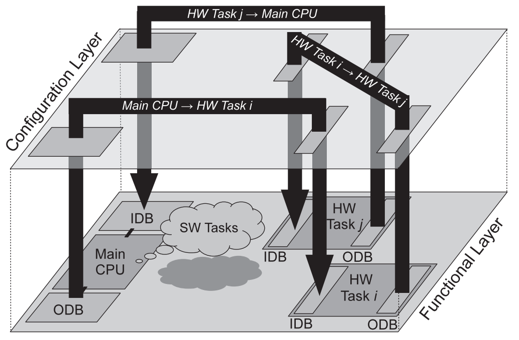


Fig. 1. Hardware-software intertask communications.

plane of the FPGA when they finish. Note that data traces do not significantly constrain the allocatability of the tasks within the configuration plane; i.e., the memory blocks that store data traces can be bypassed when allocating new incoming tasks in overlapping locations as long as they are not specifically used by them. This is shown in [26].

In this context, data are transmitted from a producer task P to a consumer task C by copying the content of P's ODB to C's IDB. In the case of software tasks, the data buffers are mapped into the main memory accessible from the main CPU. Hence, data traces are relocated along the chip as required by the tasks. If possible, the ODB of the producer task P is configured to be the IDB of the consumer task C so that there is no need to relocate any data; i.e., data are in the position the consumer task expects to be. Otherwise, R3TOS harnesses the ICAP interface to establish on-demand "virtual" channels among the hardware tasks through the configuration layer (see Fig. 1). Data buffer sharing between producer and consumer tasks is limited by two factors. First, the occupation of the FPGA, as other running tasks may prevent the allocation of the consumer task next to the producer. Second, the FPGA layout, in case there are not the necessary FPGA resources required by the consumer task in the position where the producer task is placed.

To speed up the relocation of high amounts of data in HBC tasks, R3TOS configures physical routes to connect BRAMs when there are no obstacles between them, i.e., other tasks. Indeed, the physical routes offer a potentially higher bandwidth than ICAP-based "virtual" channels, not only because the data are read and written at once, but also because the usable clock frequency can be made higher than that of the ICAP. The physical routes and the logic to drive the BRAMs are grouped together to form data relocating tasks (DRTs), which are managed from R3TOS as standard computing tasks.

Therefore, as shown in Fig. 1, the tasks perform computation in the functional layer of the FPGA and intertask communications are carried out, or at least initiated, through the configuration layer. The synchronization needed to coordinate access to data buffers from both layers is provided by the HWS included in the TCL. The HWS acts as the internal reset signal for the task; i.e., the task starts computing only when the HWS is enabled (by HWuK through the ICAP), and once it completes the computation, the task itself disables its HWS. Therefore, the HWS is active only while the task is performing active computation, and hence, it is also used to implement the exclusive access to FPGA resources.

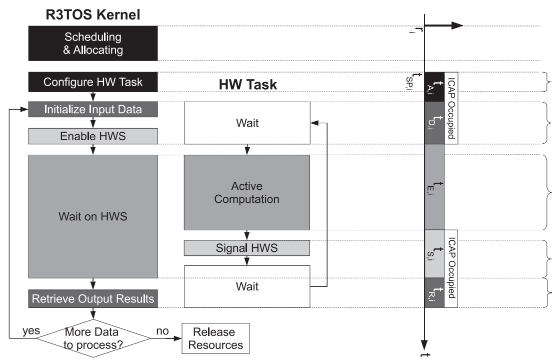


Fig. 2. Execution phases of a hardware task in R3TOS.

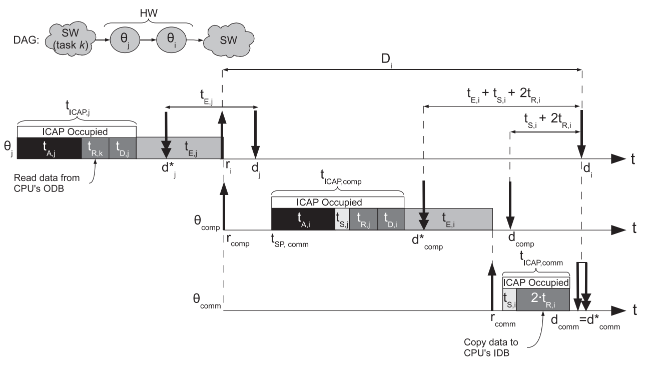


Fig. 3. Consecutive execution of hardware tasks in R3TOS.

R3TOS targets an event-triggered data-dependent computation model where data exchanges among tasks are carried out only prior to task execution, with the computation thereafter performed atomically. The tasks are triggered when all their input operands are ready to be processed. This functioning enables *temporal isolation* among hardware tasks execution, avoiding most of communication related problems in reconfigurable computing, such as deadlocks or race conditions. As a result, the system is predictable enough to approach real-time performance. Last but not least, we note that this functioning perfectly matches the way a traditional software OS works, i.e., the context of the task is loaded when granted with CPU/FPGA access.

4.3 Real-Time Hardware Task Model

From a computational point of view, the merger of a set of PEs and a TCL gives rise to a task with "software look and feel" which is unequivocally identified by means of a TaskID. Indeed, the main CPU can access the advanced computation capabilities delivered by the hardware tasks, without having to know anything about their implementation details, and regardless of their placement within the FPGA. The CPU simply writes the data to be processed to its ODB, which is mapped into the main memory, and after some time, it retrieves the computed results from its IDB.

In the area domain, a task θ_i is considered to occupy an arbitrarily sized rectangular region on the FPGA, which is defined by its width and height, $h_{x,i}$ and $h_{y,i}$, respectively. The internal architecture of the task, which depends on the location where it was originally synthesized in the FPGA, is described as the succession of the resources it uses column by column, from the leftmost to the rightmost column.

In the time domain, a task θ_i requires five different phases to complete a computation (see Fig. 2).

During the *setup phase*, the task is configured in the FPGA. Previously existing tasks in overlapping positions are deallocated, if any, and a suitable clock signal is routed, i.e., a clock signal of the required frequency [20]. In general, the duration of the setup phase of a task, $t_{A,i}$, is proportional to its size. However, as the amount of time needed to deallocate overlapping tasks cannot be estimated a priori, a worst-case penalty must be added. In any case, we note that the latter time is minimal as a result of using MFWR commands.

During the *input data delivery phase*, the IDB of the task is filled by HWuK with actual data to be processed. In general, the time required to do so, $t_{D,i}$, is proportional to the amount of data to be loaded. When all input data are copied into the IDB, HWuK enables the HWS of the task.

During the *execution phase*, the input data are transformed into results by the task's PEs. The combination of temporarily isolated tasks and hardware-based deterministic computation leads to predictable timing behavior. Indeed, a task uninterruptedly completes its computation $t_{E,i}$ units of time after it started, regardless of system workload. However, $t_{E,i}$ is not always fixed and known, for example, iterative calculations with variable number of iterations. To deal with these situations, the tasks automatically signal their HWS to acknowledge HWuK when the results are ready in their ODB.

The *synchronization phase* refers to the polling process from HWuK to the tasks' HWS to detect a computation completion, and spans $t_{S,i}$ units of time.

During the *output result retrieval phase*, which spans $t_{R,i}$ units of time, the results computed by the tasks are finally read from the ODB. Note that this phase may be delayed until the results are required by another task.

When two hardware tasks communicate each other, data must be read from the producer's ODB prior to being copied to the consumer's IDB. That is, the *input data delivery phase* of the consumer task θ_i is immediately followed by the *output data retrieval phase* of the producer task θ_j. Furthermore, the latter two phases are to be preceded by the *setup phase* of the data consumer task. When merging these three phases with the *synchronization phase* of the data producer task, a single *ICAP access period* is formed for each task θ_i that spans $t_{ICAP,i} = t_{A,i} + t_{D,j} + t_{S,j} + t_{R,i}$ consecutive units of time (see Fig. 3). During this time, the task is effectively set up in the FPGA. The fact of grouping together the task phases that need to access the ICAP is beneficial to schedule its access in a more predictable way.

As previously introduced, R3TOS uses several ways to reduce $t_{ICAP,i}$ toward a higher performance. First, the direct access from a data consumer task to producer task's ODB results in $t_{D,i}$ and $t_{R,j}$ circumvention. Second, the use of DRTs to quickly relocate data between data buffers results in reduced $t_{D,i}$ and $t_{R,j}$. Finally, the reuse of previously configured tasks results in $t_{A,i}$ circumvention. In addition, $t_{E,i}$ can also be reduced by feeding the task with the highest clock rate.

The real-time constraint of R3TOS involves the existence of a relative *execution deadline* for each task, D_i, which is defined by the application programmer and represents the maximum acceptable delay for that task to finish its execution. The absolute execution deadline for each new task instance, d_i, is computed by adding the task release time, r_i, to its relative execution deadline, $d_i = D_i + r_i$. Even more important is the absolute *setup deadline*, d_i^*, which represents the maximum acceptable delay for a task to start the computation to meet its execution deadline; i.e., $d_i^* = d_i - t_{E,i}$. A task is considered to be ready to start its computation when it is completely configured in the device and the data to be processed is already loaded in its IDB. To achieve the predictability required by real-time behavior, it is always considered the worst-case that any of the aforementioned performance enhancements cannot be exploited.

As shown in Fig. 3, the setup deadlines are different for hardware tasks that communicate with other hardware tasks and for those which exchange data with software tasks. This is because the data retrieval phase is included in the model of the data consumer hardware tasks used by HWuK, but it is not in the model of data consumer software tasks used by SWuK. Indeed, the data retrieval operation of a data consumer software task must be invoked from the data producer hardware task itself. Unfortunately, this functioning could interfere with real-time behavior, and thus, a specific model is derived for it. This model is depicted in Fig. 3. The absolute setup deadline of a "standard task" that communicates with other hardware tasks, or which receives data from a software task, is equal to $d_i^* = d_i - t_{E,i}$. This is the case of θ_j. On the other hand, hardware tasks that deliver data to the main CPU, such as θ_i, are modeled as two separate "standard tasks" to harmonize their management:

- An exclusively computing task θ_{comp} with $h_{x,comp} = h_{x,i}$, $h_{y,comp} = h_{y,i}$, $r_{comp} = r_i$, $t_{E,comp} = t_{E,i}$, $t_{ICAP,comp} = t_{A,i} + t_{R,j} + t_{S,j} + t_{D,i}$, where θ_j is the data producer for the task θ_i, $d_{comp} = d_i - t_{S,i} - 2 \cdot t_{R,i}$ and $d^*_{comp} = d_i^* - t_{E,i} - t_{S,i} - 2 \cdot t_{R,i}$. Note that the multiplication by 2 is because both to read data from task's ODB and to copy it to CPU's IDB are needed.
- An exclusively communicating task θ_{comm} with $h_{x,comm} = h_{y,comm} = 0$, $t_{ICAP,comm} = t_{S,i} + 2 \cdot t_{R,i}$, $t_{E,comm} = 0$, $r_{comm} = t_{SP,comp} + t_{ICAP,comp} + t_{E,i}$, and $d_{comm} = d^*_{comm} = d_i$.

To achieve the predictability required by real-time behavior, it is always considered the worst-case execution time $t_{E,i}$ for the tasks. As a result, the HWS must be accessed only once, and thus, $t_{S,i} = t_S \; \forall \; \theta_i$, where t_S is equal to the time needed to read back a single frame from the FPGA device. Note that in this situation, HWSs are checked only to confirm the correct ending of tasks' execution.

As shown in Fig. 4, a hardware task goes through several states during its life cycle. The task is in *waiting state* until it is triggered. When this occurs, the task switches to *ready state*, waiting to be granted with ICAP access and assigned a set of resources on FPGA to start computation. If the task waits for a long time, it may miss its deadline, in which case it is discarded, switching back to *waiting state*. When the task is scheduled on time, it switches to the *setting-up state*. The task remains in this state while it is configured in the FPGA and also while it is provided with a set of data to process. While the task performs active computation it is in *executing state*, and it is in *allocated state* when it remains configured in the FPGA after having completed its computation. Note that allocated tasks do not consume dynamic power as they remain in reset state, i.e., their HWS is disabled.

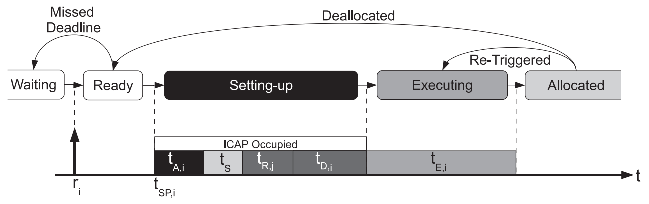


Fig. 4. Life cycle of a hardware task in R3TOS.

R3TOS keeps track of the state of the tasks at runtime, by grouping them into different task queues: *ready*, *executing*, and *allocated*. There is no need for a *setting-up* queue as only one task can be at this state at any time.

4.4 Scheduling and Allocating Hardware Tasks

At runtime, HWuK needs to decide *when* to schedule and *where* to allocate the tasks buffered in the ready queue. In few words, R3TOS selects at every kernel-tick t_{KT} the most suitable ready task to be executed according to both time and area criteria. Note that t_{KT} is configurable by the user depending on the timing constraints of the application. The fact that scheduling and allocation decisions are made online enables data-dependant computation and permits to cope with the unpredictable degradation provoked by spontaneous faults as well as with high workload peaks.

R3TOS uses a nonpreemptive EDF porting to schedule access to the ICAP, named as *finishing-aware EDF (FAEDF)*. To compensate for the lack of preemption, which on the other hand is not well suited to be used with FPGAs, FAEDF includes the capability to "look ahead" to find future releases of adjacent pieces of area when executing tasks finish. The time left until then is used to schedule other ready tasks that can be completely set up. If there are no ready tasks meeting this, other R3TOS services that need to access the ICAP are executed. Note that FAEDF is capable of selecting the next task to be executed in a very short time if the task queues are ordered as needed.

To cope with fragmentation, which is the enemy to beat in such a flexible scenario as defined by R3TOS, a novel allocation heuristic, called *empty area compaction (EAC)* [27], and a novel task allocation strategy, called *snake* [28], are used. EAC assigns a score to each position in the FPGA based on what measure that position contributes to form adjacent pieces of empty area. These calculations are carried out in parallel with task configuration through the ICAP, so that the subsequent task allocation decisions can be quickly made by consulting the precomputed EAC scores, which collectively form the so-called empty area descriptor (EAD). On its part, *snake* is aimed at linking HBC tasks together to promote data exchange among them through the functional layer, minimizing the use of ICAP and at the expense of increasing fragmentation on the device. To deal with this fragmentation, LBC tasks (which need short time to exchange data through the ICAP) are efficiently allocated

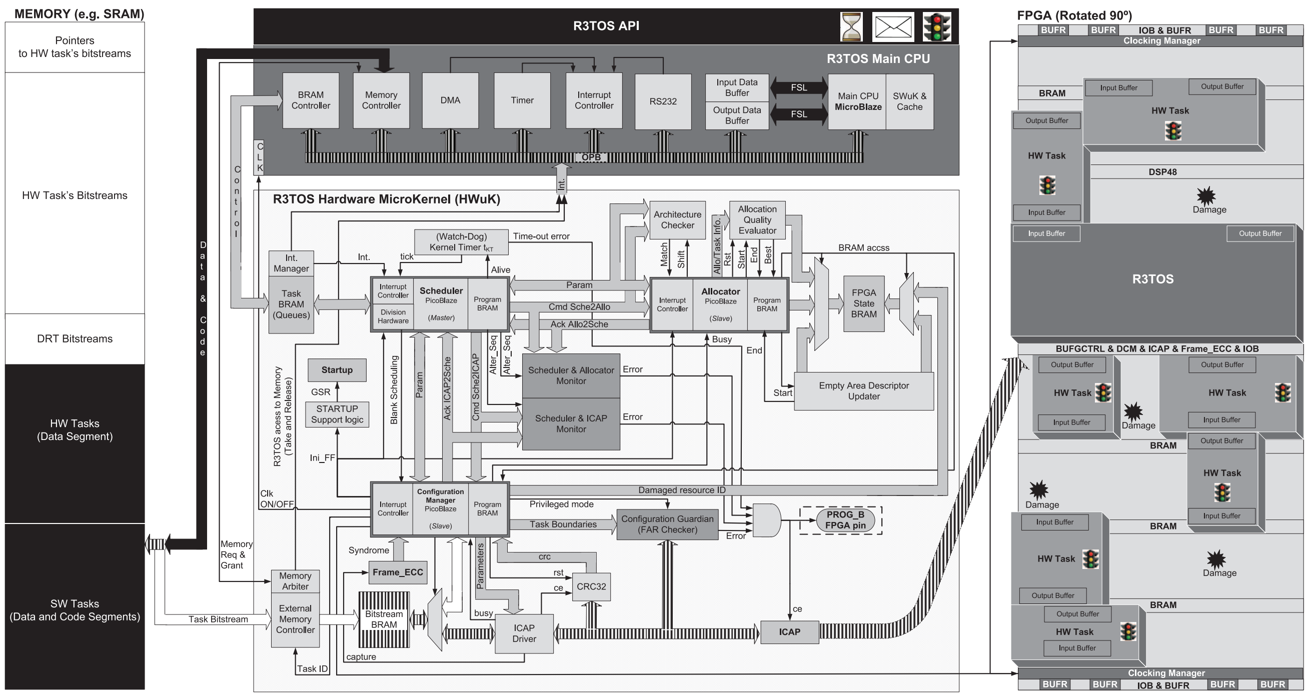


Fig. 5. R3TOS block diagram.

using the EAC heuristic on the resulting FPGA area fragments in between the HBC chains.

The obtained results when simulating our scheduling and allocation algorithms using a wide range of different task parameters can be found in [29].

4.5 Executing Hardware Tasks

As previously mentioned, HBC tasks are connected together to speed up the transmission of intermediate partial results among them, i.e., if possible, the consumer task accesses its input data directly from the ODB of the producer task. While this scheme improves the performance, by minimizing ICAP time for carrying out intertask communications, it also constrains multitasking capabilities. Indeed, only half of the HBC tasks in a computation chain can perform active computation simultaneously as only one task can access the shared data buffers at each time. This restriction does not apply in the case of LBC tasks, which do not share the data buffers.

4.6 Fault-Handling Strategy

The key benefit of R3TOS is its ability to exploit the underlying reconfigurability of the FPGA chip. Specifically, continuous task reconfiguration prevents the accumulation of soft-errors and the permanent damage in the chip is conveniently circumvented when allocating tasks as long as it does not affect the HWuK itself. We note that the fine-grained management of FPGA resources implemented by R3TOS allows to discard only the damaged resources, contrasting with traditional slotted systems where an entire slot must be discarded in the event of a fault affecting it.

The resources assigned to a task that has computed an erroneous result are kept in quarantine while an exhaustive diagnostic test is carried out on them. Namely, this test consists in loading all-ones and all-zeros frames to detect stuck-at configuration bits. The ultimate objective is to detect and prevent the future use of damaged resources, if any. Due to the criticality of the HWuK, its configuration state is periodically checked, using Frame_ECCs, to correct any errors before they lead to system failure. If the configuration error is the result of a damaged resource, the system initiates a fail-safe shutdown.

When it is detected a malfunctioning of the HWuK, a full FPGA reconfiguration is forced by toggling the PROG_B pin after having disabled the ICAP interface. As soon as R3TOS is recovered, the configuration state of the HWuK is checked to determine whether the error was provoked by a damaged resource. If this is the case, the system initiates a fail-safe shutdown.

5 R3TOS General Architecture

Fig. 5 shows the general block diagram of R3TOS, which basically comprises three main parts: HWuK, main CPU, and memory.

5.1 R3TOS Hardware Microkernel

HWuK includes the configuration manager and two servers that run upon it: scheduler and allocator. Each component is separately implemented to enable parallelism in the execution of HWuK processes. The parallel cooperation of simple components does not only result in low runtime overhead but also in acceptable area overhead; i.e., the main core of all HWuK components is a tiny Xilinx PicoBlaze, which requires only 96 FPGA slices. The cooperation among the HWuK components is mastered by the scheduler, with the allocator and the configuration manager acting as slaves. The communication between the components follows a very strict set of rules that are supervised by two monitors. Each monitor is thus capable of detecting any malfunctioning in each pair of communicating components, i.e., scheduler-allocator and scheduler-configuration manager.

The internal architecture of the HWuK components is structured around the PicoBlaze core. The PicoBlaze executes an optimized assembly program that is based on interruptions to reduce the response time, relying on an interrupt controller to handle the interruptions. Furthermore, each PicoBlaze uses a dedicated data BRAM to store the information associated with the corresponding HWuK process(es) it executes. Hence, the scheduler manages the task queues in the task BRAM, the allocator keeps track of the available resources on an FPGA BRAM (state BRAM), and the configuration manager executes predefined sequences of configuration commands from a bitstream BRAM. The fact that these memories are dual-ported is conveniently exploited. For instance, the configuration manager can mark any detected damaged resource as nonusable directly in the FPGA state BRAM, and the main CPU can set in ready state any triggered task directly in the task BRAM.

Specific for the scheduler is a real-time timer to generate the kernel ticks. Additionally, the kernel timer supervises the correct functioning of the scheduler, i.e., it acts as watchdog timer. The scheduler's PicoBlaze must generate at least one *alive* pulse within a maximum number of kernel ticks. Indeed, it is crucial for the reliability of the system to monitor the state of the scheduler as it is the master in HWuK. Specific for the allocator is an architecture checker to speed up the search of feasible allocations where the FPGA resources are arranged as required by the tasks, and an empty area descriptor updater to accelerate the intermediate computations required by the allocation algorithm, i.e., EAC scores. The configuration controller is equipped with an FSM to drive the ICAP at the highest allowed clock frequency, i.e., 100 MHz, and with a CRC32 module to compute the 32-bit CRC over a set of data read from the ICAP at the data rate, i.e., when accessing the ODBs of data producer tasks. Our ICAP controller achieves an effective reconfiguration bandwidth as high as 380 MB/s when working with relatively small partial bitstreams (e.g., less than 10 KBs) and reaches 390 MB/s when working with larger partial bitstreams (e.g., some tens of KBs). Notably, the configuration manager also includes a configuration guardian (CG) to ensure safe accesses to the configuration memory. The latter exclusively allows access to the resources assigned to a given task to configure it or to communicate with it, i.e., hardware tasks are isolated in the configuration domain. Besides, the CG prohibits access to the resources assigned to HWuK except when the *privileged mode* is enabled. To detect and localize upsets in the configuration frames, the configuration manager interacts with the Frame_ECC logic as well, which automatically checks the correctness of every read-back configuration frame. Finally, the configuration manager has access to the STARTUP primitive with the objective of initializing the flip-flops of the hardware tasks with a predefined value, i.e., INIT values. Since this is a very critical primitive that allows accessing internal signals to the configuration logic of the FPGA, i.e., global set/reset (GSR), specific support logic is added to guarantee its safe functioning.

Finally, HWuK includes some extra functionality distributed along the device. This includes TCLs, attached to the hardware tasks, and the circuitry to manage and diagnose the clocking resources, which is implemented next to the rightmost and leftmost IOB/BUFR columns.

5.2 R3TOS Main CPU

In the current R3TOS implementation, a Xilinx on-chip processor, namely a 32-bit MicroBlaze soft-core, is used as the main CPU. The MicroBlaze is coupled with a set of peripherals to provide additional functionality (e.g., timer or interrupt controller), connection to the external world (e.g., RS232 serial line or ethernet), or to increase the performance (e.g., DMA). The peripherals are interconnected by means of an on-chip peripheral bus (OPB). The interface with HWuK is based on interruptions and shared memory (i.e., task BRAM and IDB/ODB).

The program executed by the main CPU is held in a directly accessible program memory. It is important to note that as this memory is implemented using dual-ported BRAMs in MicroBlaze, it must be disabled and its clock must be stopped by HWuK prior to accessing the content of any BRAM within the FPGA. Otherwise, the program code could get corrupted. Despite this undesirable effect does not appear when using other processors that do not use dual-ported BRAMs to store their program, the clock stopping capability is still required when using the STARTUP primitive. Indeed, while the GSR signal is active, the BRAMs cannot be correctly accessed, and thus, the processor must be stopped to prevent executing undesired instructions.

FIFO-based high-speed communications between the IDB, ODB, and MicroBlaze are achieved by connecting them using fast serial links (FSLs). In addition, the IDB and ODB are also accessible through the aforementioned OPB bus to allow individual access to specific positions.

5.3 Memory

To reduce the amount of components in the R3TOS system, both the bitstream memory and the main memory are implemented using a single external memory chip, despite the fact they are conceptually independent components. Therefore, the external memory chip stores: 1) the data and code segments of the software tasks, 2) the data segments and bitstreams of the hardware tasks, and 3) the bitstreams of the DRTs. There is a pointer table located in the lowest part of the memory, which is used by HWuK to know the exact location of each task bitstream. The bitstream of a task, or DRT, with identifier TaskID starts in the position specified by the pointer located at position TaskID, and it ends in the position specified by the pointer located at position TaskID+1. In Fig. 5, note that the white parts of the memory are accessed exclusively by HWuK, while the black parts are accessed only by the main CPU. The most typical case of a single port memory is assumed in the current R3TOS implementation, and hence, HWuK includes an arbiter to coordinate access from HWuK itself and from the main CPU.

TABLE 1
Basic Functions Implemented by the Configuration Manager

Function	Description
RBF	Read-back a subset of frames
WSF2S	Write a single frame to a single location
WSF2M	Write a single frame to multiple locations
WMF2S	Write multiple frames to a single location
BlF	Blank a subset of consecutive frames
PBR	Relocate a partial bitstream

6 R3TOS API

Despite the fact that a R3TOS standard user is likely to interact with the FPGA hardware through a high-level software POSIX-like API, it might happen that in some specific applications the developer would need to take a low-level hands-on approach to meet strict real-time requirements, or simply to save FPGA resources if there is no need to use a main CPU. In this case, when the application is to be based on bare hardware, the low-level R3TOS HWuk services provide a direct and more agile interface to the FPGA resources, enabling improved performance and higher reliability levels. These services make up the HAL and are mainly based on the basic functionality implemented by the configuration manager (see Table 1). R3TOS services provide support for task allocation, deallocation, communication, and synchronization.

In light of a higher productivity, high-level programming is a must. This is achieved by wrapping the R3TOS HWuK with a software OS layer that is executed in the main CPU. The latter software layer is known as software microkernel (SWuK). The OS-CPU provides the basic platform to execute application software routines, i.e., the serial portions of the application that are not amenable to be accelerated neither by parallel processing nor by computation specialization. On the whole, the combination of R3TOS HWuK and SWuK result in a good framework to develop hardware-software hybrid applications.

The R3TOS SWuK is currently based on FreeRTOS, which is an easy-to-use and open-source real-time microkernel designed specifically to have a small memory footprint. Indeed, our FreeRTOS porting to R3TOS requires 29.8 KB, thus fitting in only 16 BRAMs. Two features of FreeRTOS are especially attractive to us. First, its high dependability, which is supported by the fact that a microkernel derived from it, i.e., SafeRTOS, has been certified for safety-critical applications. Second, its popularity, which is confirmed by the 2011 EETimes embedded systems market study, where FreeRTOS comes in top in two categories: the kernel currently being used, and the kernel being considered for the next project to develop. This is indeed our objective: provide R3TOS with the most attractive software skin for application developers.

While the core part of FreeRTOS has been kept intact in our porting to R3TOS, new interrupt service routines (ISRs) have been programmed to enable communication with HWuK. Likewise, the scheduler included in the commercial distribution of FreeRTOS has been modified to provide the necessary support for dealing with hardware tasks. Finally, new functionality has been developed to ease common operations in R3TOS, for example, DMA transfers between the IDB, ODB, and main memory.

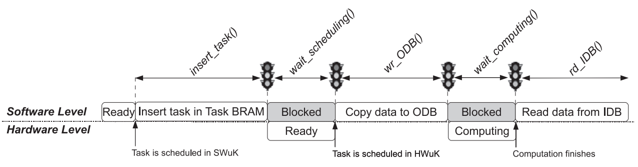


Fig. 6. Management of a hardware task in SWuK.

The hardware tasks are provided with a "ghost software body" that includes HWuK-related system calls with the objective of making them manageable in SWuK. For instance, wr_ODB and rd_IDB permit to deliver/retrieve data to/from the hardware tasks by accessing the ODB and IDB, respectively. The generic software body of a pure hardware task is shown in Listing 1, and the interaction between the software and hardware levels during the life cycle of a task, which is based on FreeRTOS semaphores, is depicted in Fig. 6.

Listing 1. Representation of a hardware task in the SWuK
```
static void vTask_ID (void *pvParameters)
{
  insert_task(task_ID,task_params);
  wait_scheduling(task_ID);
  wr_ODB(& input_data_base_addr, length);
  wait_computing(task_ID);
  rd_IDB(& output_results_base_addr, length);
}
```

7 R3TOS Proof-of-Concept Prototype

A proof-of-concept R3TOS prototype has been implemented on a Xilinx Virtex-4 XC4VLX160 FPGA. This device includes up to 12 clock regions, 88 CLB columns, 7 BRAM columns, and 1 DSP48 column. The layout of this chip, shown in Fig. 7, is very interesting as it includes a 28 CLB column wide homogeneous region in the central part of the chip (sandbox) with the heterogeneous resources located in the edges; in the rightmost edge there are three BRAM columns and one DSP48 column and in the leftmost edge there are four BRAM columns. Based on this layout, the vertical granularity is set to be a clock region, i.e., $Hy = 12$, while the horizontal granularity is either four CLB columns in the sandbox or a single heterogeneous resource column in the edges, i.e., $Hx = 15$.

As shown in Fig. 7, the R3TOS core circuitry is located in the upper right quadrant of the chip, leaving 3/4 of the FPGA free to allocate the hardware tasks. Overall, R3TOS logic consumes 4,268 Slices and 30 BRAMs. Note that only the clock distribution lines span across the partially reconfigurable region (PRR) to reach the regional clocking resources, i.e., BUFRs, located in the leftmost and rightmost IOB columns. Next to the BUFRs, and occupying only one CLB column, are the associated diagnostic circuits, which permit to detect any malfunction of the former clocking resources.

The R3TOS circuitry and the clocking resources together make up the R3TOS static infrastructure, which provides support for the execution of the hardware tasks. Indeed, unlike in related work, the BUFRs are not dedicated to any specific hardware task, but shared among all of the tasks

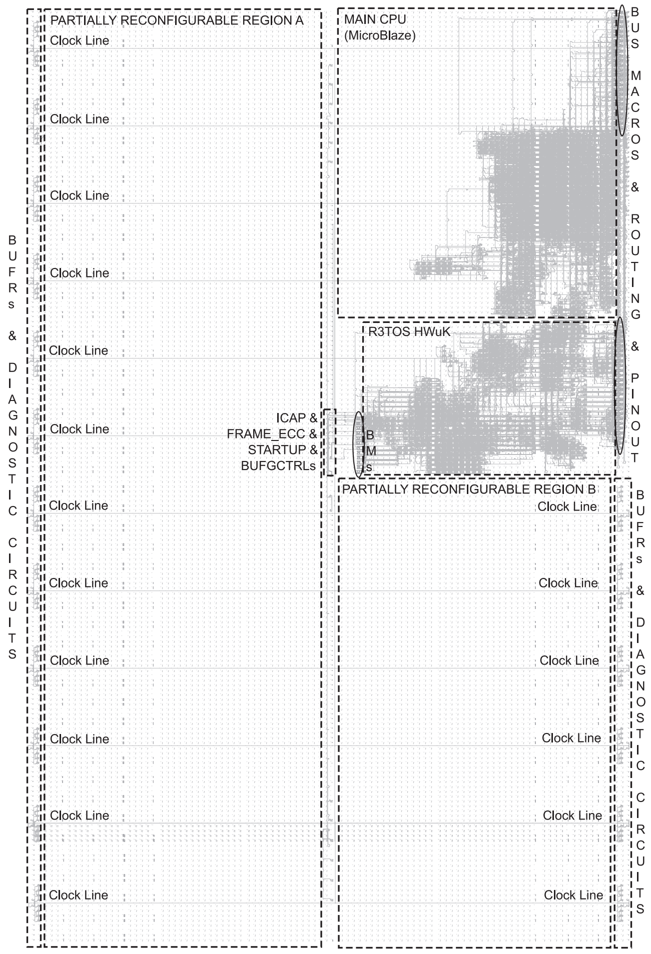

PARTIALLY RECONFIGURABLE REGION A
MAIN CPU (MicroBlaze)
BUS MACROS & ROUTING & PINOUT
Clock Line
BUFRs & DIAGNOSTIC CIRCUITS
R3TOS HWuK
ICAP & FRAME_ECC & STARTUP & BUFGCTRLs
BMs
PARTIALLY RECONFIGURABLE REGION B

Fig. 7. FPGA implementation of R3TOS prototype.

placed in the same clock region of the FPGA. By doing so, tasks can be clocked using multiple BUFRs located in different clock regions, and thus, their maximum height is not constrained. Moreover, not including the BUFRs in the architecture of the tasks enhances their (horizontal) allocatability, i.e., tasks can be arbitrarily horizontally shifted within a clock region.

For simplicity and clarity purposes, the implementation shown in Fig. 7 does not consider board-dependent IOB location issues. Indeed, in the real implementation using the Virtex-4 LX development kit, the pins to access the external memory are located in the bottom-left corner, and therefore, other two clock regions are occupied with static routes.

The R3TOS core circuitry comprises two different parts: 1) HWuK, which spans two FPGA rows in height, and 2) the main CPU (i.e., MicroBlaze), which spans four FPGA rows. Both R3TOS parts communicate with each other as well as with I/O device pins through a set of BMs located in the rightmost four CLB columns of the chip. Indeed, we note that all of the used IOBs (i.e., 69) could be mapped to the upper right quadrant. This modular implementation of R3TOS favors the adaptability, as the main CPU can be replaced without interrupting the normal HWuK functioning. In addition, HWuK includes a set of BMs in the leftmost side to access the FPGA's hard primitives that are located in the central part of the chip, namely, Frame_ECC, STARTUP, and ICAP.

TABLE 2
Measured Times in the R3TOS Prototype

	Duration (μs)	
	Min.	*Max.*
Write/Read to/from data buffers	<1	165
Scheduling algorithm execution	<1	<100
Queues and task state updating	<1	<300
Allocation algorithm execution	<1	<100
Empty Area Descriptor updating	10	<200
Transfer data buffer using ICAP	60.18	60.18
Transfer data buffer using DRTs	36.18	39.9
Switching BRAMs between neighbor tasks	10.03	10.03
Activation of a Hardware Semaphore (HWS)	3.7	3.7
Polling of a Hardware Semaphore (HWS)	1.6	1.6

Table 2 shows the most significant performance figures measured in the R3TOS prototype, when clocking it at 100 MHz. Notably, the amount of time needed by the scheduler and allocator to sort the task queues and update the EAD is in the same range of that needed to configure a typical hardware task using the ICAP, usually in the range of milliseconds [30], thus reducing the time overheads introduced by R3TOS as the three processes can be concurrently carried out.

8 Case Study: SDR Application

This section explains the way R3TOS helps developing SDRs with dynamically changeable characteristics, such as communication standard, modulation, speed, or transmission spectrum [31]. SDRs can adapt their functionality depending on the environment with the objective of ensuring safe, uninterrupted and ubiquitous communications. When used together, R3TOS and SDR permit to build highly adaptive systems, with capability to deal not only with "internal" hazards emerging on chip, i.e., spontaneously occurring faults, but also with "external" threats appearing in the environment where the chip is working, for example, spontaneous interferences, or to reconfigure themselves based on network configuration, for example, a different set of networks go down and up as the user moves around, and to serve the computing requests triggered by the user, for example, multimedia or video gaming.

Some of the capabilities described above have been demonstrated with a R3TOS-based SDR multistandard transmitter prototype shown in Fig. 8. The data to be transmitted are digital video that is first compressed using JPEG, i.e., data are preprocessed.

The prototype is based on the proof-of-concept development presented in Section 7, but extended with a superheterodyne radio frequency (RF) front end and a color 352×288 pixels image sensor that provides 8-bit RGB outputs. Consequently, the R3TOS implementation shown in Fig. 7 is coupled with some specific static logic to drive the RF front end, i.e., A/D and D/A controllers, and for receiving the RGB video signal to be transmitted. Central to this logic are ping-pong memories, which are composed of two memories that are alternatively switched between read and write. Namely, while one of the memories in the D/A controller is written by R3TOS, the content of the other memory is transmitted through the RF front end, and similarly, while one of the memories of the A/D controller is read by R3TOS, the other memory is filled with data received from the RF front end.

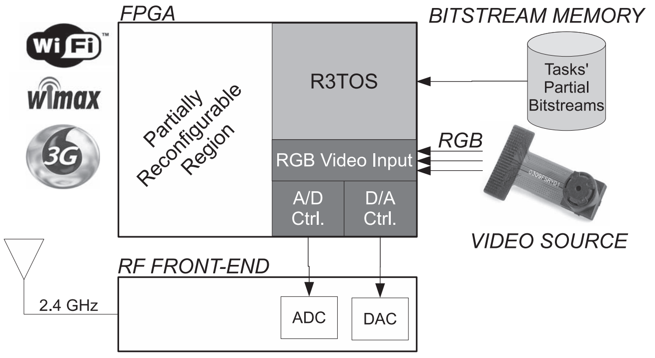


Fig. 8. SDR demonstrator prototype: block diagram.

Our SDR prototype implements a simplified version of three of the currently most used communication standards: IEEE 802.11 WiFi, IEEE 802.16 WiMAX, and IMT-2000 UMTS (also known as 3G), allowing to switch them on the fly to achieve smooth transition of the service as the system moves from one network connection to another. However, as we have not implemented the communication standard detection functionality yet, standard switches are forced by pushing some buttons in the prototype. Additionally, our SDR prototype implements a "home-made" cognitive radio solution that is able to autonomously detect and circumvent any interferences provoked in the transmission channel by using spectral switches.

The SDR functionality was first developed and validated using Xilinx System Generator tool. Based on the system generator models, we decided to implement the task set listed in Table 3. The data buffers of the tasks were implemented using four BRAMs. The three communication standards implemented by our SDR prototype as well as our cognitive radio solution are based on the aforementioned tasks, which are executed in a different order and with some minor changes in each case. For instance, the generation polynomial used in the convolutional encoder is different in WiMAX and UMTS standards, and the interleaving pattern and length changes from one standard to the other. Therefore, the tasks were parameterized to allow small online adjustments to be made to fulfil the requirements of each communication standard. To ease this process, the adjustable parameters were mapped to directly accessible FPGA resources, such as BRAMs and LUTs. Xilinx system generator tool was used to translate the high-level task models into VHDL code, which was in turn adapted to the partial reconfiguration design flow, i.e., nonreconfigurable resources were extracted (DCM, BUFG, etc.). The modified VHDL was then combined with R3TOS specific logic, for example, TCL, and placement constraints, for example, PRRs definition, prior to synthesizing it using XST to obtain the partial bitstreams of the tasks. Finally, the latter bitstreams were loaded to the bitstream memory in the R3TOS prototype.

The fact that our SDR application was programmed in a software environment, i.e., Xilinx software development kit (SDK), demonstrates the feasibility of accessing FPGA resources using R3TOS-based traditional software-like interfaces. Moreover, we note that we enjoyed a nearly complete high-level programming experience as the SDR hardware tasks were implemented using currently existing high-level hardware design tools, i.e., Xilinx system generator.

TABLE 3
Task Set in the SDR Prototype

ID	Task	Slices	BRAM	DSP48
θ_1	JPEG Compressor	2,859	15	2
θ_2	Randomizer	53	10	-
θ_3	Reed-Solomon Encoder	196	10	-
θ_4	Convolutional Encoder	46	10	-
θ_5	Puncturer	39	12	-
θ_6	Inter-leaver	468	12	-
θ_7	Data Symbol Mapper	132	9	-
θ_8	WCDMA Modulator	3,041	82	67
θ_9	OFDM 64-Carrier Builder	749	9	-
θ_{10}	Train Sequence Inserter	1,771	18	-
θ_{11}	64-tap IFFT	1,536	11	26
θ_{12}	16-word Cyclic Prefix Inserter	59	11	-
θ_{13}	OFDM 256-Carrier Builder	3,332	9	-
θ_{14}	256-tap IFFT	2,585	11	42
θ_{15}	64-word Cyclic Prefix Inserter	903	11	-
θ_{16}	OQPSK Modulator	596	11	70
θ_{17}	Power Spectral Density estimator	1,573	11	32

At runtime, the order of execution of the tasks is dictated by the application running on the main CPU. Since most of the times there is only one task ready, the R3TOS scheduler does not perform a significant role in the SDR application herein described. On the other hand, R3TOS allocator does play a key role as it optimizes data transfers among successive SDR tasks in the DAG. Most of the times, data producer and consumer tasks are packaged in pairs within the same clock region in order to promote sharing of the data buffers between them as well as to reuse previously configured circuitry. The reason to package only two tasks together is the layout of the used FPGA, which has only three heterogeneous resource columns (e.g., BRAM and DSP48). Indeed, SDR tasks cannot be placed in the CLB sandbox, as they need to use BRAMs and DSP48s. We note that the only specific consideration assumed at design time to ease the packaging of the tasks is the position of their IDB and ODB. This is a minor aspect which does not constrain the flexibility of relocation: tasks can be allocated to any other position in the FPGA where there are the required resources. The only implication is that the position of the data buffers to be accessed through the ICAP are different for each task.

In our SDR prototype, data consumer tasks directly access producer tasks' IDB 45 percent of the times in the case of WiFi, 50 percent in WiMAX, 57 percent in UMTS, and 25 percent in the case of our cognitive radio solution. Likewise, DRTs are used in the 27 percent of intertask communications carried out in WiFi, 20 percent in WiMAX, 14 percent in UMTS, and never in our cognitive radio solution.

We used the IDB and ODBs of the tasks as test points to check the correctness of the system functioning. The content of these buffers was read back in our prototype and compared against the simulation produced in the system generator simulation environment. Using this method, we verified that the data sent to the RF front end was correct.

While this case-study allowed us to show the feasibility of using R3TOS for the rapid development of dependable reconfigurable applications, it is not well suited to fully show the extent of its possibilities. Indeed, the execution phases of the SDR application are significantly shorter than their allocation phases ($t_{E,i} < t_{A,i}$), resulting in relatively large time overheads and limited multi-tasking. Therefore, the time needed to transmit an image ranges from less than a second (in WiFi and WiMAX) up to some seconds (in UMTS and cognitive radio), which is not sufficient for most of applications, e.g., video streaming. Better results are expected when using R3TOS in applications with a high computation to allocation ratio.

Nevertheless, we would like to note that R3TOS enables new possibilities that could be explored in the scope of SDR as well. For instance, SDR hardware tasks could be based on a small processor (e.g., PicoBlaze) coupled with custom logic instead of exclusively on custom hardware. By doing so, a better trade-off between resource usage (task size would be smaller) and execution time (it would be longer) could be achieved. Hence, allocation times would be likely to be smaller than execution times, thus approaching the situation where R3TOS is expected to achieve better results. Moreover, this scenario is interesting as other functionality could be executed on the saved area, allowing to target more sophisticated applications.

Finally, we would like to point out that R3TOS is currently being also used to develop a K-means clustering application based on the work presented in [32] and a highly reliable traction controller for a railway vehicle. In the scope of our K-means application, preliminary results show a 10-fold performance improvement with regard to a software implementation.

9 Conclusions

This paper has introduced the bases of R3TOS, a novel reliable reconfigurable real-time operating system aimed at putting the versatile resources embedded in modern FPGAs at the service of computation in a reliable way, thus promoting direct translation of electronic advances into system performance improvement. By sharing FPGA resources in time, R3TOS is able to execute very large applications in small devices, harnessing the maximum performance of the available silicon.

While one of the key features of R3TOS is its ability to manage FPGA resources at low-level, it presents the user with a set of high-level OS-like mechanisms and services, for example, task relocation and low-level data exchanges among tasks, which abstract low-level related issues and ease the development of reconfigurable applications. Central to R3TOS is real-time performance and fault tolerance. While both software and hardware tasks are scheduled based on real-time constraints, software tasks are executed in a main CPU, and hardware tasks are dynamically mapped to nondamaged FPGA resources and individually provided with a reliable clock source of the highest usable frequency by each task. Damaged resources are located by means of a Frame_ECC-based self-diagnosis mechanism.

A proof-of-concept implementation of R3TOS on Xilinx Virtex-4 FPGAs and a case-study SDR demonstrator were also presented in this paper. Future work will build on the prototype presented in this paper to produce a more mature version of R3TOS on the latest FPGAs. We also expect to revisit the codesign process of the SDR application reported in this paper to improve its performance. Finally, we want to characterize the behavior of R3TOS in the scope of other applications belonging to different fields.

References

[1] C. Dubach, T.M. Jones, E. Bonilla, and M.F.P. O'Boyle, "A Predictive Model for Dynamic Microarchitectural Adaptivity Control," *Proc. Ann. IEEE/ACM Int'l Symp. Microarchitecture,* 2010.

[2] X. Iturbe, K. Benkrid, T. Arslan, I. Martinez, M. Azkarate, and M.D. Santambrogio, "A Roadmap for Autonomous Fault-Tolerant Systems," *Proc. Int'l Conf. Design and Architectures for Signal and Image Processing,* 2010.

[3] G.J. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," *Proc. Int'l Workshop Field-Programmable Logic, Smart Applications, New Paradigms and Compilers,* 1996.

[4] J.Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip," *Proc. Conf. Design, Automation and Test in Europe,* 2003.

[5] H. Walder, "Operating System Design for Partially Reconfigurable Logic Devices," PhD thesis, Swiss Fed. Inst. of Technology, Zurich, Switzerland, 2005.

[6] H.K.-H. So, "BORPH: An Operating System for FPGA-based Reconfigurable Computers," PhD thesis, Univ. of California at Berkeley, 2007.

[7] G. Wigley and D. Kearney, "Research Issues in Operating Systems for Reconfigurable Computing," *Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms,* 2002.

[8] B. Zhou, W. Qiu, and C. Peng, "An Operating System Framework for Reconfigurable Systems," *Proc. Int'l Conf. Computer and Information Technology,* 2005.

[9] R. Pellizzoni and M. Caccamo, "Adaptive Allocation of Software and Hardware Real-Time Tasks for FPGA-Based Embedded Systems," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.,* 2006.

[10] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-Reconfiguring Platform," *Proc. Int'l Conf. Field-Programmable Logic and Application,* 2003.

[11] J.A. Williams and N.W. Bergmann, "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip," *Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms,* 2004.

[12] J.A. Williams, N.W. Bergmann, and X. Xie, "FIFO Communication Models in Operating Systems for Reconfigurable Computing," *Proc. Ann. IEEE Symp. Field-Programmable Custom Computing Machines,* 2005.

[13] A. Donato, F. Ferrandi, M. Santambrogio, and D. Sciuto, "Operating System Support for Dynamically Reconfigurable SoC Architectures," *Proc. IEEE Int'l System-on-Chip Conf.,* 2005.

[14] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "Hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel," *Proc. IEEE Conf. Emerging Technologies and Factory Automation,* 2005.

[15] H.K. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers Using BORPH," *ACM Trans. Embedded Computing Systems,* vol. 7, no. 2, pp. 1-28, 2008.

[16] E. Lubbers, "Multithreaded Programming and Execution Models for Reconfigurable Hardware," PhD thesis, Univ. of Paderborn, Germany, 2010.

[17] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," *Proc. Ann. IEEE Int'l Symp. Field-Programmable Custom Computing Machines,* 2011.

[18] D. Gohringer, M. Hubner, E.N. Zeutebouo, and J. Becker, "Operating System for Runtime Reconfigurable Multiprocessor Systems," *Int'l J. Reconfigurable Computing,* vol. 2011, article 3, 2011.

[19] 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," *Proc. Int'l Conf. Field-Programmable Logic and Applications,* 2011.

[20] X. Iturbe, K. Benkrid, R. Torrego, A. Ebrahim, and T. Arslan, "Online Clock Routing in Xilinx FPGAs for High-Performance and Reliability," *Proc. NASA/ESA Conf. Adaptive Hardware and Systems,* 2012.

[21] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Vijaykrishnan, and K. Sarpatwari, "FLAW: FPGA Lifetime Awareness," *Proc. Ann. Design Automation Conf.,* 2006.

[22] S. Feng, Shuguang, S. Gupta, A. Ansari, and S.A. Mahlke, "Maestro: Orchestrating Lifetime Reliability in Chip Multiprocessors," *Proc. Int'l Conf. High-Performance Embedded Architectures and Compilers,* 2010.

[23] J. Angermeier, D. Ziener, M. Glass, and J. Teich, "Stress-Aware Module Placement on Reconfigurable Devices," *Proc. Int'l Conf. Field-Programmable Logic and Applications,* 2011.

[24] K. Bertels, *Hardware/Software Co-Design for Heterogeneous Multi-Core Platforms: The Hartes Toolchain.* Springer, 2011.

[25] D. Grewe and M.F.P. O'Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," *Proc. Int'l Conf. Compiler Construction,* 2011.

[26] T. Becker, W. Luk, and P.Y.K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," *Proc. Ann. IEEE Symp. Field-Programmable Custom Computing Machines,* 2007.

[27] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, and I. Martinez, "Empty Resource Compaction Algorithms for Real-Time Hardware Tasks Placement on Partially Reconfigurable FPGAs Subject to Fault Occurrence," *Proc. Int'l Conf. Reconfigurable Computing and FPGAs,* 2011.

[28] X. Iturbe, K. Benkrid, A. Ebrahim, C. Hong, T. Arslan, and I. Martinez, "Snake: An Efficient Strategy for the Reuse of Circuitry and Partial Computation Results in High-Performance Reconfigurable Computing," *Proc. Int'l Conf. Reconfigurable Computing and FPGAs,* 2011.

[29] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, T. Arslan, and I. Martinez, "Runtime Scheduling, Allocation and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence," *Int'l J. Reconfigurable Computing,* vol. 2013, article 905057, 2013.

[30] M. Liu, W. Kuehn, L. Zhonghai, and A. Jantsch, "Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration," *Proc. Int'l Conf. Field-Programmable Logic and Applications,* 2009.

[31] J. Mitola, "Software Radios: Survey, Critical Evaluation and Future Directions," *IEEE Aerospace and Electronic Systems Magazine,* vol. 8, no. 4, pp. 25-36, Apr. 1993.

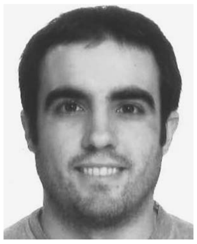
[32] H.M. Hussain, K. Benkrid, A.T. Erdogan, and H. Seker, "Highly Parameterized K-Means Clustering on FPGAs: Comparative Results with GPPs and GPUs," *Proc. Int'l Conf. Reconfigurable Computing and FPGAs,* 2011.

Xabier Iturbe received the MSc degree in electronics and telecommunications from the University of the Basque Country in 2006, the research proficiency degree from the same university in 2009, and the PhD degree in electronics from the University of Edinburgh in 2013. Since 2006, he has been working in the Electronics Department of IK4-Ikerlan research center, where he has been involved in a number of research projects both with the industry and academia. From 2009 to 2013, Dr. Iturbe was also a member of the System Level Integration Research group in the University of Edinburgh, where he was the principal designer of R3TOS. His current research interests include FPGAs with special focus on methods and tools for developing runtime reconfigurable applications and hardware virtualization.

Khaled Benkrid received the PhD degree in computer science, the first Class Ingenieur d'Etat degree in electronic engineering, and the executive MBA degree. He is a senior member of the IEEE, chartered UK engineer, and senior lecturer in electronic engineering at the University of Edinburgh. His research has been focused in high-performance computing using reconfigurable hardware and electronic design automation. More recently, he has been exploring other accelerator technologies such graphics processor units and multicore processors.

Chuan Hong received the BEng degree in automation from Tianjin University in 2009 and the MSc degree in electronics and electrical engineering from the University of Kent in 2010. He joined the System Level Integration Research group, University of Edinburgh in 2010 as a PhD student. His current research interests include development of FPGA-based systems with intelligent fault-tolerant capability and fast reconfiguration methods.

Ali Ebrahim received the BEng and MSc degrees in electronics and electrical engineering from the University of Glasgow in 2007 and 2008, respectively. He was a research assistant with the University of Bahrain until 2010, and then he joined the System Level Integration group, University of Edinburgh as a PhD student. His current research interests include dynamic partial reconfiguration for fault tolerance and intellectual property security in FPGA devices.



Raul Torrego received the MSc degree in automation and industrial electronics from the University of Mondragon in 2008, where he is currently working toward the PhD degree. He is a researcher in the Communications Department of IK4-Ikerlan Research Center. His current research interests include dynamic partial reconfiguration, software-defined radios, rapid prototyping tools, embedded systems, and system on programmable chips.

Imanol Martinez received the MSc and PhD degrees in industrial automation and electronics from the University of Mondragon in 2002 and 2006, respectively. He joined the IK4-Ikerlan Research Center in 2006. His current research interests include system-on-chip architectures for real-time and safety-critical systems as well as model-based design methodologies.

Tughrul Arslan received the BEng and PhD degrees in electronic engineering from the University of Hull in 1989 and 1993, respectively. He currently holds the chair of integrated electronic systems at the School of Engineering, University of Edinburgh. He is a member of the Integrated Micro and Nano Systems Institute and leads the System Level Integration group in this University. His current research focuses on enhancing personal mobility and the design of high-performance embedded wireless systems. He is a member of the IEEE.

Jon Perez received the BEng degree in industrial and robotics from the University of Mondragon in 1999, the MSc degree in electronics and electrical engineering from the University of Glasgow in 2000, and he finished his doctoral studies in computer science at TU Wien in the field of safety-critical embedded systems in 2011. He currently leads the Electronics Department and Embedded Systems research in the IK4-Ikerlan Research Center. His current research interests focus on distributed real-time and safety-critical embedded systems. He is a member of the IEEE.