Hybrid Hardware-Software Architecture for Reconfigurable Real-Time Systems*

Rodolfo Pellizzoni and Marco Caccamo

Department of Computer Science, University of Illinois at Urbana-Champaign
{rpelliz2, mcaccamo}@uiuc.edu

Abstract

Recent developments in the field of reconfigurable SoC devices (FPGAs) will enable the development of embedded systems where software tasks, running on a CPU, can coexist with hardware tasks. We devised a real-time computing architecture that can integrate hardware and software executions in a transparent manner, and can support real-time QoS adaptation by means of partial reconfiguration of modern FPGA devices. Tasks are allowed to migrate seamlessly from CPU to FPGA and vice versa to support dynamic QoS adaptation and cope with dynamic workloads. In this paper, we discuss the design and implementation of an on-chip infrastructure, OS extensions and task design methodology that enable hardware-software transparency in the presence of relocation. The overall architecture is suitable to schedule real-time workloads and we derive bounds on relocation overhead. Finally, we show the applicability of our design methodology on a concrete task design case.

1. Introduction

Recently emerging SoC devices enable the development of embedded systems where software tasks, running on a CPU, can coexist with hardware tasks running on a reconfigurable device (FPGA). Such hybrid platforms are especially suitable for the development of real-time embedded systems because they combine the flexibility of software execution on a CPU with timing predictability and high performance of hardware execution on a Partially Reconfigurable Device (PRD). Even if the theory of real-time reconfigurable computing is still at an early stage [9, 10, 12, 11, 26], progresses have been made recently in the development of operating systems for reconfigurable devices (OSRD) [22, 30, 31]. OSRDs will enable a highly dynamic use of partially reconfigurable FPGAs, running multiple concurrent circuits (hardware tasks) with full multi-tasking capabilities.

In this work, we focus our attention on embedded systems comprised by a general purpose CPU and a Partially Reconfigurable Device (PRD), together with main memory and I/O peripherals. Modern devices, like the Xilinx Virtex family of FPGA [33], implement all of the above on a single configurable SoC. Compared to a classic mono or multi-processor platform, hardware execution of real-time logic functions (tasks) on FPGA fabric has two main advantages: 1) it can greatly improve QoS and 2) it natively supports both temporal isolation (due to parallel execution on distinct reconfigurable regions) and temporal predictability (due to the precise knowledge of execution times). At the same time, implementing all system functionalities in hardware is not desirable for several reasons. First of all, it precludes reuse of already available software libraries and tasks. Second, some control heavy tasks are hard to parallelize and their hardware implementations yield low speedup. Third, some system tasks require little computation and can be executed on a CPU with low utilization while they would consume precious FPGA area if implemented in hardware. The main objective of our research is to devise a real-time computing infrastructure that can integrate hardware and software execution in a transparent manner, and can support real-time QoS adaptation by means of partial reconfiguration of modern FPGA devices. In particular, flexibility of execution is achieved by supporting the seamless migration of tasks from software to hardware and vice versa at run-time. Task migration enables the system to prioritize different types of computation based on environmental factors and to cope with different operational modes at run-time. While in this paper the proposed hybrid real-time computing infrastructure has been mainly tested (see test case of Section 5) for achieving dynamic QoS adaptation and hardware predictability, we are also in the progress of exploiting it for real-time fault-tolerant computing.

A lot of work has been done in the design of SoC communication infrastructure and in the development of working prototypes for OSRD, but much more remains to be done to obtain a feasible and usable platform when real-time execution and on-line reconfiguration are considered. In a previous work [26], we have addressed the problem...
of defining a suitable real-time resource allocation scheme and admission control test that is able to maximize the number of admitted real-time tasks while preserving all timing constraints (deadlines). However, an even more important challenge lies in defining suitable computation and communication abstractions to enable fast and efficient development and deployment of applications on a hybrid hardware-software architecture.

The key contribution of this work is the design and implementation of an on-chip infrastructure, suitable OS interfaces, and a task design methodology that enable hardware-software transparency, run-time task migration, real-time guarantees, and dynamic QoS adaptation on SoC devices comprised by a general purpose CPU and a Partially Reconfigurable Device (PRD). Essential aspects of our design are:

Communication model: since communication must be independent from the task execution mode, both hardware and software tasks must be provided with an equivalent set of communication primitives, possibly with 1-to-1 semantic conversion. We decided to base our system on a shared memory communication model, as it fits well our hybrid multitasking model and allows for an efficient implementation.

Execution model: while from an architectural point of view hardware execution is extremely different from software execution, from an application point of view hardware and software tasks should behave according to a unified model. In particular, as we are mainly interested in applications with real-time requirements, we assume that all tasks behave according to the well-known real-time periodic task model [15].

Real-time reconfiguration: current reconfigurable devices are subject to a variety of physical constraints, including task area consumption, device reconfiguration time, and degree of device homogeneity. Since the system must enforce timing guarantees even during partial reconfiguration, we need to be able to predict any overhead due to the task relocation mechanism.

The rest of the paper is organized as follows. Section 2 presents our SoC platform. Section 3 describes our architecture for hybrid hardware-software execution, detailing task execution and communication model and relocation mechanisms. Section 4 presents our reconfiguration management and QoS adaptation scheme and further shows how scheduling and reconfiguration constraints can be accounted for using run-time analysis. Section 5 details our prototype FPGA and OS implementations and shows how the discussed techniques can be applied using a test case for the design of a concrete task (a real-time cipher [21, 20]). Finally, in Section 6 we discuss related works and in Section 7 we provide concluding remarks and future work.

2. System Model

In our model, each task $\tau_i$ in the system can be provided in multiple configurations $\{\tau^1_i, \tau^2_i, \ldots, \tau^n_i, \ldots\}$, which provide similar functionality at different QoS levels. Configurations can be either software or hardware; a task can provide both but is not required to. A software configuration (or SW task for simplicity) is a portion of code running on the CPU with associated data. A hardware configuration (or HW task for simplicity) is a logical function that has been synthesized for execution on the PRD. Every configuration $\tau^j_i$ is periodically activated with period $p^j_i$ and has deadline equal to its period; note that two SW or two HW configurations can be functionally equivalent and differ only in their period. For convenience $\tau^0_i$ is defined as a special null configuration: it represents the fact that the task is not running in the system. Typically, HW configurations will run at higher frequencies than SW configurations thanks to the speedup achieved by hardware implementation. Therefore, when the system is not under heavy load all tasks with a HW configuration will be run in hardware. If the load on the system increases and the reconfigurable area is filled, some tasks will have to be moved to SW (see Section 4 for details on reconfiguration management).

Software configurations are implemented as executable files in ELF file format. We require all software configurations for the same task $\tau_i$ to be implemented using a single executable file to simplify memory allocation in the OS. However, the software execution path can change between configurations, based on the required activation period; for example, imprecise computation [16] could be used to trade accuracy for reduced computation time. Each hardware configuration is implemented by a partial bitstream, a file representation of the reconfigurable interconnections and logic blocks (CLB) used to implement the task functionality in hardware; a PRD is provided with a reconfiguration port through which partial bitstreams can be loaded thus changing the device functionality [6]. A PRD can execute multiple HW tasks simultaneously by assigning different positions to the HW tasks such that their areas do not overlap; it is possible to reconfigure a single HW task on-line leaving all the other tasks running. The OSRD takes care of reconfiguring HW tasks at run time, managing the device area, storing the individual bitstreams in a separate memory and loading partial bitstreams into the device as necessary; these activities are normally referred to as placement and loading. HW/SW relocation is allowed by the system. This means that tasks running on the PRD can be seamlessly transferred to the CPU and vice versa.

An important issue regards the reconfiguration capability of the PRD. Loading a new bitstream to change the device functionality takes a non negligible amount of time on modern, large devices: up to 2.88ms for a single region HW task and 69ms for the entire system [29]. While downloading a bitstream, the area occupied by it can clearly not be used; furthermore, if several HW tasks must be activated at once,
partial bitstreams must be loaded serially since we can not use multiple configuration ports simultaneously. This restricts the ability to preempt HW tasks; in particular, in [25] it is shown that an HW task that executes for only 25% of the time can not achieve a task activation frequency greater than 3.6 Hz, which is too low for most real-time control applications. Therefore, we assume that each hardware configuration executes for the entirety of its period, so that no reconfiguration is needed unless a new task is activated. This typically results in hardware configurations either exhibiting higher frequency/QoS than the software configuration of the same tasks or being rather small by trading off computation time versus occupied area, which is in line with the reason we want to exploit hardware execution in the first place.

A preliminary description of our reconfigurable hardware implementation has been provided in [26]. We refer the interested reader to [26] for implementation details, while in this section we focus on our new reconfigurable architecture. Figure 1 shows an architectural block diagram for a Xilinx Virtex-4 FPGA. A CPU is used to run the OSRD and software configurations. The Internal Configuration Access Port (ICAP) controller provides reconfiguration capabilities [29]. The controller is able to load partial bitstreams from a connected flash memory and reconfigure the device as requested by the OSRD. An interrupt controller receives interrupt requests from the ICAP controller, HW tasks and I/O devices (e.g. RS232, ethernet) and forwards them to the CPU. A timer module is used to generate a periodic timing signal (every 1 ms in our prototype), which is routed to each HW task and the interrupt controller and used as the base clock resolution (jiffy) by the OSRD. HW tasks follow the 1D area model [30]. Each task occupies a rectangular area on the device. One dimension (traditionally the vertical one) is fixed, while the other dimension can vary, being a multiple of a basic area size known as a reconfigurable region; a Virtex-4 LX200 device can support up to 48 regions. Each HW task consists of two modules: the user logic and the interface module. The user logic implements the actual task logic. The interface module provides transparency and relocation service and connects the task to an on-chip system bus. The bus is divided in two segments: the CPU segment connects the CPU and all peripherals to an external RAM chip called the system RAM, used to hold OS and software configurations instructions and data. The task bus segment connects all HW tasks to an external RAM chip called the shared RAM, used to hold shared memory data. A bridge allows communication between the two segments when required, but otherwise they can operate in parallel. Compared to a single segment solution, the proposed architecture increases the memory bandwidth available to HW tasks as accesses by the CPU to the OS RAM are not propagated on the task bus segment. If additional bandwidth is required, multiple task bus segments can be used at the cost of adding additional shared RAM banks (see for example [28]) and constraining the placement of HW tasks. Compared to implementing a more complex network-on-a-chip [17] solution, we believe that the proposed design is a good compromise between usability, area overhead and available throughput. The bus arbitration is based on a fixed priority scheme, where priorities can be configured by the CPU. In our system, we assign priorities to HW tasks according to rate monotonic scheduling while the CPU has lower priority than any HW task. Note that changing the CPU priority at each software scheduling point is not practical due to overhead constraints, and assigning lower priority to the CPU is consistent with HW tasks typically having both smaller periods than their corresponding SW tasks and stricter timing requirements on the task bus segment.

A final note regards the software architecture on the CPU. In general, we assume that SW tasks are scheduled using any real-time scheduler with proven schedulability bounds and suitable isolation mechanisms, like the EDF scheduler [15] in conjunction with the Constant Bandwidth Server [1] (CBS). The CBS provides isolation between hard and soft tasks so that all jobs of hard tasks are proven to complete within their deadlines if the following condition is met (see [26]):

\[
U = \sum_{\tau^j_i \in T_S} \frac{e^j_i}{p^j_i} \leq 1
\]

where \( T_S \) is the set of tasks executed on the CPU and \( e^j_i \) is either the worst case execution time or the server bandwidth for \( \tau^j_i \), depending on \( \tau^j_i \) being either a hard or soft real-time task respectively; \( U \) is known as the system utilization.

3. HW-SW Architecture

In this section we describe the architectural components of the reconfigurable platform detailed in Section 2 that enable efficient task relocation and HW-SW transparency. To simplify the exposition, we break up our description in four parts: in Section 3.1 we introduce the basic mechanism ex-

Figure 1: Architecture block diagram.
plaining our shared memory layout. In Section 3.2 we detail the architecture and functionality of the interface module used in each HW task, while in Section 3.3 we describe required OS extensions. Finally, Section 3.4 wraps up the discussion describing our intended design flow.

### 3.1. Memory Layout

As introduced in Section 1, we base our transparency mechanism on a shared memory architecture. An advantage of this choice is that the software compilation model and memory layout of the OS require minimal changes. We make no assumption on software task loading, i.e. tasks can be either preloaded with the OS image or the OS can provide an executable loader/relocator with dynamic loading capabilities; note that even if tasks are preloaded, they can still be activated dynamically at run-time. We believe that our description is general enough to be applicable to most embedded OSs using a Unix-like task memory mapping and employing the ELF executable file format. Furthermore, note that our description is largely independent from the choice of a specific CPU, as our only major requirement is support for both external and software interrupts. Both cached and non cached CPUs can be supported. Note that the CPU used in our system, a MicroBlaze soft-core [32], does not include a memory management unit. The system can be extended to support memory virtualization by imposing suitable constraints on the memory mapping and page replacement policy for shared memory areas; however, for simplicity we omit such mechanisms from our description.

Figure 2 shows our proposed memory layout for the system. There are three types of memory in the system: the system RAM, the local memory for each HW task, and the shared RAM. The system RAM is used to hold the OS image and data, a heap memory pool and all SW tasks. Each SW task memory region is divided in four main areas. The text area contains the task code, and corresponds to the .text section of the software executable ELF file. The RO-data contains read-only data (typically constant variables); it corresponds to the .rodata section of the ELF file. The SW data (state) area contains all task global data, both initialized and non initialized; it corresponds to the .data and .bss sections of the task ELF executable. Finally, the stack area is used for the task stack, which is allocated per task and assigned a fixed dimension when the task is loaded. There is no per-task heap; the OS dynamically allocates heap memory on-demand from a memory pool.

Each HW task includes a variable amount of local memory, in the form of both FPGA embedded block RAM (BRAM) and hardware registers. The local memory of each HW task is divided in two areas: the HW data (state) area, and the working memory area. The HW data (state) area is an exact mirror of the SW data (state) area found in system memory: the same global variables are allocated at the same relative address in both the SW and HW data (state) areas. The HW data (state) area is always implemented as BRAM memory inside the interface module of the HW task. The working memory area can consist of both BRAM and hardware registers and it is used by the user logic to implement the task logic; it does not need to mirror any memory area used by the software implementation.

Finally, the shared RAM holds a shared memory pool. The OS can reserve shared memory blocks in the shared pool which are used by both software and hardware tasks for inter-task communication; shared memory blocks are allocated at an address known to all communicating tasks.

By virtue of the mirrored data (state) areas, relocation is possible by using the following key idea: we require that both the software and hardware configuration of each task save the entire task state between successive periodic instances (jobs) in the SW data (state) and HW data (state) areas respectively. Therefore, if we constrain the system to relocate a task at the end of its period, the only memory area that needs to be copied is the data (state) area. In particular, the following steps are used to migrate a task from software to hardware: first, we load the HW task bitstream in a free reconfigurable region. Then, at the end of the next task period, we stop the software configuration, we transfer the data (state) area from system memory to HW local memory, and we start the HW task immediately after (a similar, mirrored series of steps are used to migrate the task from HW to SW). If the CPU uses a cache, then either the data (state) memory must not be cached or the CPU must flush it from cache before copying it to HW local memory. To avoid adding unnecessary overhead on the CPU, the memory transfer is controlled by the HW task’s interface module which has bus master capabilities (see Section 3.2).

Note that when not performing relocation a HW task accesses the bus only to retrieve shared data; a solution where each HW task directly accesses the data (state) area in system memory would easily suffer from a severe bottleneck in the bus communication speed. Furthermore, the schedul-
ing overhead caused by our relocation mechanism is limited and predictable as shown in Section 4. Finally, note that for simplicity we do not cover the case of migration from HW to HW (i.e. changing the position of a HW task) since the reconfiguration algorithm described in Section 4 can produce any desired system configuration by migrating tasks from HW to SW and from SW to HW only.

We argue that the proposed scheme closely matches the traditional coding structure of a periodic real-time task. At first activation, the task typically executes initialization code where it initializes global variables and it sets up a periodic activation timer. The task then enters a while loop where it first performs a `wait_period()` system call to block until the next period and then it calls the periodic job code. In our model, the job code can freely use stack and heap resources, but it must save its state in the data (state) area before calling `wait_period()`. Since the heap memory and the stack used by the initialization code are left unchanged while the task is executing in HW configuration, the SW configuration does not need to free them at the end of each period; however, any variable allocated on the heap or the stack that can be modified during job execution must be invalidated if the task migrates in HW. Similarly, the user logic portion of the HW configuration is equivalent to the SW job code: it can freely use the local working memory but it must save the task state between periodic executions in the HW data (state) memory.

A final note regards data sharing: both hardware and software tasks must use suitable synchronization mechanism. In [8], the authors describe how different types of semaphores can be implemented as hardware modules accessible by both the CPU and HW tasks. Alternatively, at the cost of additional buffer memory a non-blocking technique such as non-blocking queues can be implemented.

### 3.2. Interface Module

Figure 3 shows the block diagram for the interface module. Since each interface module acts both as a slave and a master on the system bus, it connects to the bus macros using two components: a slave module and a master module. The slave module drives two write registers, `reg_cntr` and `reg_data`, and one read register, `reg_state`. The registers are used to command the controller module and to obtain state information respectively. The master module provides DMA functionality to the HW task, including burst write/reads using a 32x32-bit word buffer. The master module can be driven by both the controller module and the user logic; the user logic uses it to perform read/write accesses to shared memory blocks in shared RAM and possibly to other peripherals.

The controller module implements the state machine that drives the execution of the user logic and interfaces with the OS to provide support for relocation. The counter module receives as input the interrupt line from the system timer and implements a simple periodic counter which counts down from an initial value and generates a signal to the controller module upon reaching zero. While the HW task is running, the initial value is the length of the HW task period in jiffies. The controller module uses the counter information to signal the start of the execution period to the user logic using the `sig_start` wire; the user logic uses a similar `sig_stop` wire when it finishes execution.

The embedded block RAM (BRAM) is used to hold the HW data (state) memory, and it supports exclusive access by either the master module or the user logic. The user logic is provided with a single memory interface to both access the BRAM and drive the master module.

Relocation support is provided by the controller module responding to writes performed by the OS to the `reg_cntr` register. In particular, the OS can send three main commands to the controller module by writing constants CMD_START, CMDDOWNLOAD, and CMDSTOPUPLOAD to `reg_cntr`.

- **CMD_START**: this command is used to start execution of the user logic after a specified interval of time. The controller first sets the initial value of the counter to the value contained in `reg_data`. When the counter reaches zero, the controller uses the `sig_start` wire to activate the user logic and sets the initial value of the counter to the length of the HW task period.

- **CMDDOWNLOAD**: the controller drives the master module to download the content of the data (state) area from system memory into the local BRAM module. The `reg_data` register contains the starting address of the data (state) area in system memory.

- **CMDSTOPUPLOAD**: next time the counter reaches zero, the controller first disables it, then it drives the master module to upload the content of the data (state) area from the local BRAM module to system memory. Again, the starting address of the SW data (state) area is contained in the `reg_data` register. When the transfer is completed, the controller signals the CPU using its interrupt line.

---

1 As an added optimization, both the HW and SW configurations can be coded to only save the task state in the data (state) memory before a migration is performed; in this case, the added overhead from the state save operation must be included in the $T^{cold}_m$ time used in Section 4.
When read, the reg\_cnt register returns controller state information. In particular, reg\_cnt is set to an error code if a download operation is not finished before the user logic is first activated or if the user logic fails to signal the end of execution before the beginning of the next period. Finally, note that according to the above discussion, if the user logic behaves correctly (meaning that it does not access the memory interface when it is not active according to the described sig\_start/sig\_stop handshake), then it is never the case that the master module and the user logic access the BRAM simultaneously. Hence, there is no need for arbitrating access or implementing a dual port RAM.

### 3.3. OS Extensions

We can extend the OS to enable relocation by making use of the facilities offered by the interface module detailed in Section 3.2 and implementing two new kernel functions: migrateSWtoHW(taskid, pos, config) migrates task \( \tau_i \) with id taskid currently running on the CPU from software to its hardware configuration indexed by config using a free reconfigurable region indexed by pos. migrateHWtoSW(taskid, config) migrates task \( \tau_i \) with id taskid back from HW to its SW configuration indexed by config; note that according to our previous migration scheme [26] we require each task to be first activated in software before being able to migrate to hardware. Before we describe how the two functions work, we need to point out a couple of assumptions. First, each task must provide to the OS its list of configurations and their periods. Second, we assume that either each process descriptor is augmented or new data structures are defined in order to keep track of added task information, including the index of the current configuration, the position of the reconfigurable region used, and additional possible execution states. Third, to simplify the description we ignore error detection and recovery (in practice, to ensure safety the OS must check the correctness of each step).

Under these assumptions, the migrateSWtoHW(taskid, pos, config) function can be implemented as follows:

1. Upon entering the OS kernel, the ICAP controller is programmed to load the bitstream of task \( \tau_i \) in the specified position. Execution is then returned to the caller.
2. When the ICAP finishes reconfiguration, it raises an interrupt to the CPU. In the corresponding interrupt handler, the CPU marks \( \tau_i \) for migration.
3. When the next interrupt from the system timer is processed, the kernel examines the marked task and computes the time \( \Delta \) in jiffies until the next periodic task activation. If the current SW job of \( \tau_i \) has not yet finished execution, the kernel then sends a CMD\_START command to the HW interface controller using value \( \tau_i \).

\[ \Delta \text{ for reg\_data; otherwise, it sends a CMD\_START using } \Delta + T \text{ for reg\_data, where } T \text{ is the SW task period (this command has the effect of delaying the start of the HW configuration by one SW task period). Figure 4(a) shows a timeline for the entire mechanism assuming the latter case; straight upward arrows represent activation events while undulated arrows are used for communication events.} \]

4. When the next job of \( \tau_i \) finishes its execution, the kernel sends a CMD\_DOWNLOAD command to the interface controller and suspends the SW configuration of \( \tau_i \). However, the task periodic activation timer is kept running using the period of the HW configuration; this keeps the CPU synchronized with the HW configuration and is required to avoid activation jitter when the task is migrated back to the CPU.

As detailed in Section 3.2, the interface controller must finish to load the data (state) memory area before the HW configuration user logic can first be activated. Referring to Figure 4(a), this implies that the last software job must terminate at least \( T_i^{\text{Load}} \) time units before the next period, where \( T_i^{\text{Load}} \) is a bound on the time required to copy the data (state) area of \( \tau_i \). In Section 4 we show how this constraint can be taken into account in the system schedulability analysis. Also note that since both the CPU and all HW tasks are synchronized on the same hardware timer, the migration mechanism experiences no jitter, in the sense that the HW configuration user logic is first activated exactly at the end of the last SW configuration period.

The migrateHWtoSW(taskid, config) function can be similarly implemented as follows:
1. The kernel issues a **CMD_STOP UPLOAD** to the HW task interface controller, then returns from the system call.

2. At the beginning of the next task period, the periodic activation timer for the task is changed to use the SW configuration period. The software configuration is then activated again after receiving an interrupt from the HW task controller.

Figure 4(b) shows a timeline for HW to SW migration. Note that the relocation mechanism raises a schedulability constraint: the first job activation of the SW configuration is delayed by at most $T_i^{\text{Load}}$. Again, Section 4 shows how to account for it.

Finally, the OS supports changing the SW configuration of a task through an additional `changeConfig(taskid, config)` function, which simply updates the activation timer of the task at the beginning of the next period. A new system call `getConfig(taskid)` is exported by the kernel to obtain the current configuration of a running task.

### 3.4. Design Flow

We conclude the discussion on our designed transparency and relocation mechanism by presenting a brief discussion on the overall task design flow. Currently, each software configuration is compiled from C code into object files and then linked with system libraries to produce an ELF executable. The hardware configuration user logic, on the other hand, is specified in either VHDL or Verilog Register Transfer Level (RTL), synthesized with the interface module and finally placed and routed to produce the HW task bitstream. The interface module itself is implemented as a VHDL component that can be reused for every HW task: two generic parameters are used to specify the size of the data (state) memory, and therefore of the BRAM, and the HW task period. To simplify the implementation of the user logic, we developed a script that parses the symbol table of the ELF file and generates a `data_state vhdl` file containing information on the data (state) memory; the file is then included in the user logic. In particular, for each variable `<VAR>` in the `.data` and `.bss` sections of the ELF file, the script determines the relative memory location of the variable and associates it to a `<VAR>_LOC` constant in `data_state vhdl`.

A final note regards the overall implementation effort. Specifying the hardware configuration in RTL clearly requires knowledge of hardware design methodology and tools, which prevents a majority of system programmers from using it. A lot of effort is being spent by the research community to devise solutions to this barrier [14, 5, 7]. Ideally, in the future the designer should be able to provide a single high-level specification for each task (for example using languages such as SystemC and Handel-C) and then automatically generate both a software executable and a RTL description using a behavioral synthesis tool. The tool would need to be instrumented to properly handle accesses to the data (state) memory and to convert OS system calls (in particular for the synchronization mechanism); furthermore, the designer must be allowed enough control over the tool to efficiently explore the design space for computation time, functionality and area trade-offs. We will continue to explore ways to ease the design flow in our future work.

### 4. Reconfiguration Management

Our OS design includes a reconfiguration management layer that keeps track of task configurations and enforces timing constraints by performing on-line schedulability analysis and admission control. Tasks are logically grouped into applications, that are collections of data-dependent tasks. At any time, each application uses one of possibly several different execution **modes**. A mode specifies a single configuration for each task comprising the application. For example, a video encoding application could specify two modes with different encoding frequency. In this case, configurations with different frequencies will be selected for each task in the application, and some tasks could be required to be executed in hardware in the fastest frequency mode. Note that if an application is not required to run in the system at all times, it can specify so with a mode comprised of null task configurations.

The input to the reconfiguration management consists of application mode change requests. Upon receiving a request, the reconfiguration layer determines if it can be accepted while preserving all timing guarantees. If this check succeeds, then it uses the relocation support described in Section 3.3 to perform the mode change. Otherwise, it reports back that the change is not acceptable, together with the detected timing violation. The way the requests are generated is system dependent; they could be specified by the system user, or be generated based on changes in the environment, or an optimization algorithm could be run using feedback from the reconfiguration layer to maximize some system-wide QoS metric (for an example, see [26] where the system utility was defined as the total number of tasks running in the system).

Performing schedulability analysis requires some intermediate steps. First of all, a **swapping algorithm** must be run to determine the order in which tasks must be relocated; the algorithm creates pairs of groups of migrating tasks from HW to SW and vice versa (a so called group "swap"). Since reconfiguring the FPGA takes time, to perform a swap we first migrate a group from HW to SW, then we reconfigure the device and finally we migrate the corresponding group from SW to HW. Hence, the algorithm pays a price in term of CPU overhead, in the sense that we must leave some free computational power on the CPU to feasibly schedule an
additional group during the swap. In [25, 26], we showed that the maximum CPU overhead is equal to the smallest utilization of any swapping group, and furthermore, our algorithm is able to minimize the utilization of such group. We therefore derived a schedulability bound by modifying Equation 1 with the inclusion of an additional group utilization. However, as shown in Section 3.3 the CPU schedule can be further modified due to data (state) transfer:

- When a task $\tau_i^j$ is migrated from SW to HW, the last software job must complete execution $T_i^{Load}$ time units before the next period. Equivalently, we schedule the last software job of $\tau_i^j$ with a reduced relative deadline $p_i^j - T_i^{Load}$.
- When a task $\tau_i^j$ is migrated from HW to SW, the first software job is activated $T_i^{Load}$ after the beginning of the period. This is equivalent to $\tau_i^j$ suffering a blocking time $T_i^{Load}$.

Given this insight, we can modify Equation 1 as shown in the following theorem.

**Theorem 1** Let $T^S$ be the set of tasks running on the CPU (including the additional swapping group as detailed in Theorem 4 of [26]), and furthermore assume that in any period of any task there is at most one active job subject to migration. Then the following set of equations is a sufficient schedulability condition:

$$\forall \tau_i^j \in T^S : U + \frac{T_i^{Load}}{p_i^j} \leq 1$$

(2)

**Proof.**

Following the assumption, during any period of any task in $T^S$ at most one job has a reduced deadline or one job suffers a blocking time. In [4, 3] it is shown that in the former case the following schedulability condition applies: for $\tau_i^j \in T^S$:

$$\sum_{\tau_i^j \in T^S, q \neq p_i^j} \frac{e_i^j}{p_i^j} + \frac{e_i^j}{p_i^j - T_i^{Load}} \leq 1,$$

while in the latter case Equation 2 is a sufficient condition. But since for any feasible value of $e_i^j, T_i^{Load}$ it holds: $\frac{e_i^j}{p_i^j} + \frac{T_i^{Load}}{p_i^j} \geq \frac{e_i^j}{p_i^j - T_i^{Load}}$.

Equation 2 is also a sufficient condition for the first case. Hence, the theorem holds. □

It is important to note that although the swapping algorithm has low complexity, it still takes a non negligible time to execute. Instead of running it in the kernel, it can be executed as a software task scheduled by a CBS server. The bandwidth assigned to the CBS server represents a trade-off between limiting the overhead of the reconfiguration management and the frequency at which mode changes can be processed.

A second issue is relative to the scheduling on the communication infrastructure. In this section, we have implicitly assumed that the worst-case execution time $e_i^j$ for a software configuration includes communication delay and any OS imposed delay (context switches). Providing bounds on $e_i^j$ is outside the scope of this paper: the computation of worst-case execution time for software tasks when communication is taken into account is still an open question. This is particularly related to difficulties in predicting access patterns to main memory in cached CPU architectures (see for example [27], which describes an experimental evaluation of the schedulability effect of DMA activities on the front side bus in a standard Intel PC architecture), and testing is still widely used to assess computation times. However, an analysis can be formulated for the task bus segment since the CPU uses it only to access shared memory and send commands to HW task controller module.

However, note that knowledge about the access pattern of each HW task is clearly necessary to formulate an analysis, while our architecture does not mandate any specific communication model. Hence, in this section we show how an analysis can be formulated based on a simple double-buffered communication model. In this model, during each job $k$ of HW task $\tau_i^j$ the user logic processes inputs from an input buffer and outputs results to an output buffer. In parallel, at the beginning of job $k$ the master module of the HW task is instructed to first write into shared memory the output produced during job $k - 1$ and then to read from shared memory the input for the next job $k + 1$. Since both buffers have size two, the master module must finish executing within a period. Therefore, if $R_i^j$ is the worst case finish time of I/O transfer for HW task $\tau_i^j$, the schedulability condition becomes: $R_i^j \leq p_i^j$. If the condition is not satisfied, then the user logic can be delayed by the I/O transfer and the period must be changed to be at least equal to $R_i^j$. In what follows, let $b_i^j$ be the time that the master module of $\tau_i^j$ takes to transfer I/O data on the bus and $data_q$ be the time required to transfer the data (state) memory of the task assuming no bus contention. Finally, let $L$ be the maximum duration of a non preemptive data transfer on the bus. Then $R_i^j$ can be computed using the following iterative formula of response time analysis[4]:

$$R_i^j(k + 1) = L + \sum_{\tau_i^j \in T^H, p_i^j \leq p_i^j} \left[ \frac{R_i^j(k)}{p_i^j} \right] b_i^j + \left( \frac{\max_{\tau_i^j, p_i^j \leq p_i^j} data_q + L}{} \right)$$

(3)

where it is assumed again that in any period of any task there is at most one active job subject to migration. In the equation, the first term $L$ represents the blocking time due to lower priority data transfers on the task bus segment. The second term $\sum_{\tau_i^j \in T^H, p_i^j \leq p_i^j} \left[ \frac{R_i^j(k)}{p_i^j} \right] b_i^j$ represents the maximum interference due to higher priority tasks, and the third term $\max_{\tau_i^j, p_i^j \leq p_i^j} data_q + L$ is the additional interference by the data (state) memory transfer of a higher priority task (the additional term $L$ models the blocking term suffered by
the data (state) transfer on the CPU bus segment). Note that access to the task bus segment by the CPU do not contribute to the interference since the CPU is assigned lower priority that any HW task. Finally, \( T_{\text{reloc}} \) can be computed by simply substituting \( b_T \) with \( \text{data}_i \) in the equation and ignoring the \( \max_{r, p} \) term.

### Table 1: Task implementation results.

<table>
<thead>
<tr>
<th>Key (bits)</th>
<th>DES</th>
<th>TDES</th>
<th>AES</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>168</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Block (bits)</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Macroblock (bytes)</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Cipher Period (cycles)</td>
<td>68</td>
<td>204</td>
<td>32</td>
</tr>
<tr>
<td>I/O Time (cycles)</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>HW Period (ms)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Macroblocks/Period</td>
<td>1469</td>
<td>487</td>
<td>3124</td>
</tr>
<tr>
<td>HW Throughput (Kbytes/s)</td>
<td>47008</td>
<td>15584</td>
<td>99968</td>
</tr>
<tr>
<td>Data (state) (words)</td>
<td>22</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>SW Execution Time (ms)</td>
<td>60</td>
<td>60</td>
<td>115</td>
</tr>
<tr>
<td>SW Throughput (Kbytes/s)</td>
<td>783</td>
<td>260</td>
<td>869</td>
</tr>
</tbody>
</table>

### 5. Implementation and Test Case

We implemented the described architecture on a Xilinx ML401 development board. The board employs a small Virtex-4 LX25 FPGA which can accommodate up to 8 reconfigurable regions. The MicroBlaze CPU, bus segments, peripherals and all HW tasks are clocked at 100Mhz; the system RAM uses an 8 Mbytes synchronous SRAM chip while the shared RAM uses a 64Mbytes, 266Mhz DRAM chip. To obtain maximum software performance, we changed the standard MicroBlaze CPU configuration to use a 8Kbytes instruction and 16Kbytes data cache together with a barrel shifter; we did not implement a FPU since our test case does not use floating point operations. Note that to preserve coherency, the data (state) area in system RAM and the whole shared RAM are not cached.

We implemented our described OS relocation support by modifying Xilkernel [34]. Xilkernel is a lightweight, modular embedded kernel for the MicroBlaze processor. It supports priority driven preemptive scheduling, POSIX API, and can dynamically load tasks using the ELF executable format. We modified the kernel to add support for periodic tasks and migration and we exported all functionalities described in Section 3.3 through new POSIX-like system calls. For simplicity, we implemented the reconfiguration manager as a middleware layer executed inside a periodic task.

To validate our interface module and task design methodology we developed several real-time symmetric cipher tasks including cipher-block chaining versions of DES, TDES and AES [21, 20]. We decided to focus on cipher algorithms for our evaluation due to three main reasons. First of all, they are standard algorithms well documented in literature. Second, both optimized RTL cores and software libraries are available off-the-shelf. Third, since these algorithms have moderate to high throughput requirements and are efficiently implemented in HW, they offer us a good test case to show how our architecture can be exploited for high performance computation. To test the cipher tasks, we developed two applications. The first application consists of two tasks: the cipher and a I/O task. The I/O task uses a serial I/O controller attached to the CPU bus to perform data input/output. The serial controller communicates with a standard PC over a RS232 connection. The I/O task reads control information together with input plain-text blocks (either 64 or 128 bits) from the PC and transfers them into a buffer in shared memory. It then reads output cipher-text blocks from a second buffer in shared memory and writes them back to the PC using the same serial connection. The cipher task reads plain-text blocks from shared memory, encrypts them, and writes the resulting cipher-text blocks back into shared memory. The cipher task has both a SW and a HW configuration running at different frequencies; the I/O task can only be executed on the CPU, but it also comes in two different configurations using different frequencies. The second application is similar to the first one, but it uses two ciphers, an encrypter and a decrypter with the output of the encrypter being used as input for the decrypter. Again, the application supports two execution modes with different frequencies. In the remainder of this section, we detail a test case for the implementation of the DES task. The test case reveals two key benefits of the proposed architecture: first, we can easily reuse available software tasks and algorithms with minimal modifications. Second, we can adapt existing HW cores to run as HW tasks by paying minimal time overhead since core execution and data transfer can be parallelized. By implementing the cipher in HW we achieved a QoS improvement (measured in term of throughput) of 60 with respect to SW execution. A summary of operational characteristics and implementation results for all cipher tasks is provided in Table 1. Code for our HW architecture, OS modifications and test case can be downloaded at [24]. In the following description, note that in our system each word consists of 4 bytes.

A DES cipher operates by encrypting/decrypting an input 64-bit block into an output 64-bit block using a 56-bit key. We based our HW task implementation on a publicly available VHDL core. The core was able to perform both encryption and decryption in electronic codebook format. To implement cypher-block chaining, we wired the additional required xor operation. The obtained core required 17 clock cycles to process a single block. Note that each access to the bus necessarily incurs in some overhead mainly due to memory latency (up to 4 bus cycles in our implementation), but a 32-bit word can then be transferred every cycle. We therefore decided to process larger macroblocks of 32 bytes each (4 blocks), resulting in a cipher period of \( 17 \cdot 4 = 68 \) cycles and a total I/O time \( b_T \) of \( (4 + 8) \cdot 2 = 24 \).
cycles every cipher period. We parallelized the processing and I/O activity by using the double-buffered scheme described in Section 4. While the HW configuration could use the cipher period as its task period, the system clock granularity is much larger to reduce the overhead on the CPU due to timer interrupts. Therefore, we set a period of 1 ms for the HW task, and process multiple macroblocks in each period.

Figure 5 shows the layout of the data (state) memory for the task. The input_address and output_address variables are pointers in shared RAM where input blocks are read and output blocks are written respectively. The key and block variables hold the DES key and the last processed block respectively, while the decrypt flag determines whenever the task should perform an encryption or decryption operation. Finally, the input_buffer and output_buffer variables contain copies of the task input and output 32 bytes buffers. Note that since data is read from shared memory during the cipher period before being used and it is written to shared memory during the cipher period after being produced, these buffers must be copied in the data (state) memory before the end of each task period and read from the data (state) memory at the beginning of each task period to allow relocation. Managing the buffers adds an overhead of 16 clock cycles to the HW task, which is small compared to the period of 1 ms. The overall size of the data (state) area is 22 words, which accounts for a data value of 26 cycles. We can now perform a schedulability analysis for the task bus segment following Equation 3 using a cipher period of 68 cycles and a blocking term $L = 12$ cycles and compute the maximum possible number of macroblocks in each HW period. Following our second application scenario, we consider two DES tasks running in HW. Note that out of all macroblocks processed in a period, at most one can suffer from reconfiguration overhead in the form of the data + $L$ interference term. For this macroblock, we obtain a value $R_i^2 = 98$ and we must account for a period of 98 cycles. For all other macroblocks we obtain $R_i^1 = 60$ and therefore they can be completed within their original cipher period of 68 cycles. Let $M$ be the number of macroblocks executed in a single HW period of 1 ms. Including the 16 cycles buffer overhead, the following inequality must hold for the HW task to meet its deadline: $16 + (M - 1) \cdot 68 + 98 \leq 100,000$. The maximum number of macroblocks is $M = 1469$, which results in an overall processing throughput of $32 \cdot 1469 = 47008$ Kbytes/s.

The SW configuration was adapted from a publicly available cipher library. The same semantic of the HW configuration is used, processing 1469 macroblocks every period and writing the buffers in the data (state) area at the end of the period. Encrypting/decrypting a block in SW actually takes two steps. First, the 56-bit key is expanded into a key schedule of 16 48-bit subkeys; then, the same key schedule is reused to encrypt/decrypt each block. We therefore decided to call the function that generates the key schedule in the initialization code and to save the resulting data on the stack. Implementing the whole SW configuration required minimal effort; a total of 18 lines of C code were added to declare and call the required variables and functions. In order to determine the worst case execution time for the SW configuration, we ran our second application in software execution mode. We recorded a worst case execution time of 60 ms, corresponding to a processing throughput of $47008/60 = 783$ Kbytes/s if the task consumes the whole processor utilization.

Finally, in Table 2 we show the computed area for all implemented HW tasks and the MicroBlaze CPU in terms of FPGA slices [33] (a Virtex-4 LX25 device has 10,752 slices). For each HW task, we show both the area of the original cipher core and of the whole user logic to provide an estimate of the area overhead required to implement the buffering and periodic I/O operations. Based on the table, this overhead is around 20% for all implemented tasks, which we believe is an acceptable price to allow relocation. Given the area, we can now compute a performance metric for the tasks. We believe that a key metric is the ratio between maximum throughput and used area. For the DES HW configuration, we obtain a performance of $47008/614 = 77$ Kbytes/slice. For the SW configuration, assuming again that the task consumes the whole processor utilization, we obtain a performance of $783/1769 = 0.443$ Kbytes/slice, which is roughly 170 times less than the HW solution.

<table>
<thead>
<tr>
<th>Module</th>
<th>Area in FPGA Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1769</td>
</tr>
<tr>
<td>DES/TDES Core</td>
<td>504</td>
</tr>
<tr>
<td>DES User Logic</td>
<td>614</td>
</tr>
<tr>
<td>TDES User Logic</td>
<td>618</td>
</tr>
<tr>
<td>AES Core</td>
<td>468</td>
</tr>
<tr>
<td>AES User Logic</td>
<td>582</td>
</tr>
</tbody>
</table>

Table 2: Module Areas in FPGA Slices.
6. Related Work

The problem of providing HW-SW transparency to relocatable tasks has first been addressed by Mignolet, Nollet et al. in [19, 18, 22]. Their system model differs from ours in some important points. First of all, it uses a message passing communication model based on an on-chip packet-switching network. This design choice constraints tasks to a fixed dimension and furthermore imposes a high area overhead to implement the required packet routers. Second, they do not provide any standardized and reusable abstraction to preserve the task state between migration, which severely complicates the application developer’s job. Finally, the system does not minimize the relocation jitter and does not take into account any real-time constraint in the analysis.

HW-SW task relocation, called morphing, is discussed in [13] as part of the ReCoNets project. Similarly to our work, task morphing is only allowed between successive function (task) executions. However, they require tasks to follow an actor-based communication model, and furthermore HW tasks are automatically instrumented to extract HW state. While this approach does not require the designer to identify the state memory, it incurs in significant area and time overheads.

The Hithreads project [23, 2] has developed both hardware and software infrastructures for the deployment of hybrid HW-SW FPGA systems. Their approach is similar to our work in the sense that it is based on a shared memory model using a system bus; both SW and HW tasks execute according to a multitasking model. Furthermore, standardized interfaces for HW tasks and their integration with OS services are thoroughly discussed. However, tasks are not relocatable, which simplifies the transparency design. Finally, there is no discussion on bus scheduling or required task throughput.

Sedcole has proposed Sonic-on-a-Chip [28], a platform that integrates reconfigurable hardware processing elements (PEs) using a custom-designed bus architecture. Apart from discussing bus interfaces for hardware modules, a complete analysis of bus bandwidth requirements and buffer sizes is provided. However, the supported computational model is more restricted than what we allow since all PEs must be implemented as stream processors composed of a processing engine and a communication router isolated by buffering elements. While this programming model is simpler for a specific class of applications (video processing), buffer sizes tend to be larger than what is required in our architecture, making it unsuitable for relocation.

The topic of real-time scheduling of hardware tasks has recently received increased attention [9, 10, 12]. In particular, in [10] a new algorithm for periodic real-time HW tasks is described, which assumes a system architecture based on the 1D model similar to the one we described and takes reconfiguration overhead into account. However, to the best of our knowledge, the only work on combined scheduling of SW/HW tasks that has been published outside of [25, 26] is [11]. In [11], the design of a reconfigurable OS is described where OS services can migrate between SW and HW. Both FPGA reconfiguration time and the time required to migrate the state data are accounted for in the analysis. However, the reconfiguration overhead is partially added to the execution time of the HW services, which limits the utilization of hardware resources. Furthermore, no analysis of the communication infrastructure is provided.

7. Conclusions

Thanks to the development and rapid improvements of SoC devices, hybrid systems that incorporate both software processors and hardware reconfigurable modules on the same chip are becoming increasingly attractive, especially for demanding real-time applications. In this context, we believe that employing a multitasking model for both the SW and HW elements can significantly simplify development by making use of a familiar abstraction at the highest level. Furthermore, relocation is a required feature to obtain a flexibility of use similar to the one provided by general purpose processing units. In this paper, we designed a full SoC architecture, including HW interfaces and OS mechanisms, that supports standardized abstractions for communication transparency and real-time task relocation. As a proof of concept, we implemented the designed architecture on a FPGA device and developed relocatable tasks using the provided abstractions.

We are looking forward to extend our work in three main directions. First, we plan to integrate co-design tools into our design flow to facilitate the implementation of both the SW and HW configuration starting from a common specification. While obtaining efficient automatic synthesis of RTL code is a long term objective, our goal in the short term is to improve our tool support to ease a consistent view of state memory between HW and SW implementations for a variety of applications. Second, we will continue to optimize the memory architecture, synchronization primitives and on-chip communication infrastructure both to improve performance and to be able to formulate better real-time bounds on communication delay. Third, we will extend the system to support multiple CPUs by interconnecting them using an interface module similar to the one employed for HW tasks.

References
