Towards Real-Time Operating Systems for Heterogeneous Reconfigurable Platforms

Marco Pagani, Mauro Marinoni, Alessandro Biondi, Alessio Balsini, Giorgio Buttazzo
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {name.surname}@sssup.it

Abstract—**Heterogeneous platforms equipped with processors and field programmable gate arrays (FPGA) can be exploited to accelerate specific functions triggered by software activities. Thanks to dynamic partial reconfiguration (DPR) capabilities of modern FPGAs, such functions can be programmed at run-time, thus opening a new dimension in the resource management problems for such platforms. To properly exploit the DPR feature, novel operating system supports are needed. With the aim of investigating this direction, we developed a prototype implementation of a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time recurrent tasks.**

This work reports some preliminary experimental studies conducted to evaluate the feasibility of the proposed approach, profile the temporal parameters involved in such systems (e.g., reconfiguration and execution times) and identify possible bottlenecks. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to fully static approaches.

I. Introduction

Modern computing architectures integrate heterogeneous components, like different types of processors and field programmable gate array (FPGA) modules that can be exploited to accelerate specific functions to improve the application performance. FPGAs with dynamic partial reconfiguration (DPR) capabilities allow the user to reconfigure a portion of the FPGA at runtime, while the rest of the device continues to operate [1]. This is especially valuable in mission-critical systems that cannot be disrupted while some subsystems are being redefined [2].

Such a DPR feature opens a new scheduling dimension for systems running on such heterogeneous platforms, giving the possibility of virtualizing the FPGA, using timesharing techniques, so that it can be used to accelerate a number of hardware functions that is higher than that allowed by static partitioning, thus further improving the application performance.

Today, however, reconfiguration times are about three orders of magnitude higher than context switch times in multitasking, therefore FPGA virtualization can only be used for a limited set of applications. As shown in the next section, reconfiguration times significantly reduced in the recent years and are expected to further decrease in the near future. This enables the development of a new generation of operating systems that can manage the FPGA module, handling both software tasks (SW-tasks) and hardware tasks (HW-tasks) in a uniform fashion.

To investigate this issue, this paper presents a prototype implementation of a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time periodic tasks. The results achieved on such a prototype are encouraging and clearly show that, in spite of the relatively high reconfiguration times, a timesharing mechanism on the FPGA can significantly improve the performance of real-time applications with respect to a fully static approach.

A. Trend of Partial Reconfiguration Performance

During a partial reconfiguration process, different hardware modules are involved, such as the memory, the bus, and the FPGA reconfiguration port. As a reconfiguration bitstream traverses such series of modules, the performance of the reconfiguration processes is limited by the slowest element, which represents the DPR bottleneck. Since the DPR feature was introduced in FPGAs, all such elements were improved during the years. In the early 2001, Xilinx developed the Virtex-II FPGA device, which was able to store data on 64x8 bits DDR memory at 294 MHz and write the configuration to the logic elements with a peripheral (denoted as Slave SelectMAP) running at 50 MHz with a data size of 8 bits. The DPR throughput of this device was measured as 60 Mbps.

Nowadays, one of the top gamma products is represented by the Xilinx Zynq Ultrascale+, compatible with DDR4 memory and able to reach a maximum transfer rate of 2400 Mbps. It is connected with the ARM AMBA AXI4 and its logic elements are configured by an evolution of the SelectMAP reconfiguration port, called ICAP, running at a maximum frequency of 200 MHz with a data size of 32 bits.

In addition to the improvements achieved on the memory and the communication bus, a performance boost from the memory storage side has also been obtained through a bitstreams compression [3], moving the actual bottleneck to the reconfiguration interface.

Estimating the throughput of the reconfiguration process is not trivial, as it requires a precise ad-hoc orchestration of each hardware module involved in the process and also requires the availability of all the hardware devices that are intended to be compared. Figure 1 shows the evolution of the FPGA reconfiguration performance during the last years, obtained by comparing the theoretical maximum throughput estimations calculated from the device's datasheets.

Since a higher throughput corresponds to smaller reconfiguration times (for a given bitstream size), the positive trend shown in Figure 1 enables a more dynamic management of the FPGA, allowing the implementation of virtualization mechanisms that can provide great advantages to real-time applications, with respect to fully static approaches.

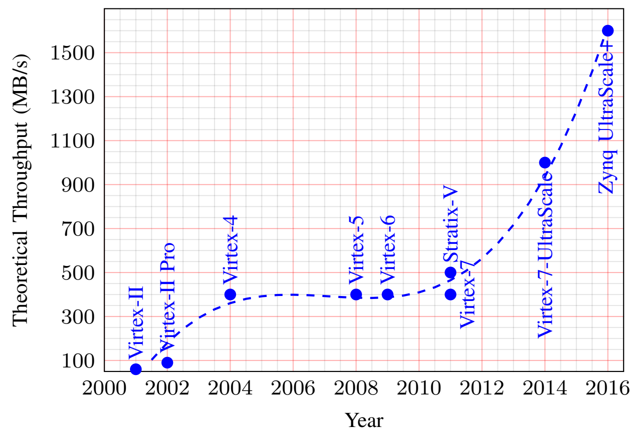

Figure 1: Reconfiguration interface throughput evolution.

II. Related Work

The reduction of reconfiguration times resulting from the FPGA technology evolution allowed exploiting the advantages of DPR for handling applications with a dynamic behavior. For example, a HW-task that could only be statically allocated in the earlier platforms, can now be reconfigured at runtime to implement mode changes in the application. More recently, some authors proposed methods for supporting a reconfiguration that can be periodically requested by SW-task at every job execution. This approach is referred to as *job-level reconfiguration*.

A few approaches have been proposed to provide an operating system support for DPR in platforms including an FPGA. The common adopted solution for exchanging data between SW-task and HW-tasks is through proper software stubs interacting with the kernel scheduler and handling the HW-tasks using a dedicated library.

For instance, Lübbers and Platzner [4] proposed the ReconOS operating system, which extends the classic multi-threading programming model to hardware activities executed on an FPGA. HW-tasks interact with SW-tasks threads trough a custom developed POSIX-style API, using the same operating system mechanisms, like semaphores, condition variables, and message queues. Originally designed for fully-reconfigurable FPGAs, this solution has then been extended by the same authors to support partial reconfiguration [5], with a cooperative multitasking approach dealing with the contentions on a set of predefined reconfiguration slots. More recently, Happe et. al. [6] extended the ReconOS execution environment to provide HW-tasks preemptability. However, the focus of this work is on hardware enabling technologies, rather than kernel support mechanisms.

Iturbe et al. [7] presented the R3TOS operating system to support dynamic task allocation on an FPGA without relying on predefined slot partitioning and static communication channels. In their solution, scheduling and allocation of HW-tasks are performed by a module, called HWuK, which is also in charge of controlling the programming interface in an exclusive manner. The authors proposed a HW-task model, as well as algorithms for scheduling and allocation. However, a worst-case analysis is not provided and nothing is said on the schedulability of SW-tasks. Such a dynamic slot partitioning increases flexibility in the FPGA allocation at the cost of a higher complexity of the reconfiguration algorithms, reflecting in higher worst-case reconfiguration times.

The major problem in such kernel extensions is that they have been designed to improve the *average* system performance, without providing tight worst-case response times bounds. As a consequence, a model of the FPGA runtime behavior based on these methods leads to huge pessimism if used for a real-time scheduling analysis.

In the context of real-time systems, Di Natale and Bini [8] proposed an optimization method to partition the FPGA area between slots allocated to HW-tasks and softcores in charge of executing the remaining tasks. Pellizzoni and Caccamo [9] considered a more dynamic scenario proposing an allocation scheme coupled with an admission test to provide real-time guarantees of applications supporting mode changes. Other authors [10], [11] presented scheduling algorithms to manage job-level reconfiguration of the FPGA, but assuming reconfiguration times negligible or fixed. Dittmann and Frank [12] addressed the issue of scheduling reconfiguration requests as a uniprocessor scheduling problem. However, their model can manage only HW-tasks and it is not suitable for platforms that also integrate softcores or processors. Although these works were aimed at providing real-time bounds, the models used for the reconfiguration infrastructure are too simplistic to describe the complexity of real platforms, hence the corresponding approaches cannot be used for analyzing real implementations with DPR features.

This paper. In summary, none of the presented papers addressed the problem of modelling the timing behavior of the reconfiguration interface and the interaction between SW-tasks and HW-tasks in such a way that they can be used for a tight real-time analysis. To address this issue, a prototype implementation of a job-level FPGA management has been developed to **(i)** profile the timing behavior of the reconfiguration port with the purpose of deriving such a model, **(ii)** investigate the practical feasibility of the job-level approach for real-time applications, and **(iii)** identify possible bottlenecks. Section V reports the results of some experimental studies conducted on such a prototype implementation.

III. System description

This work considers a heterogeneous computing system consisting of *one* processor and a DPR-enabled FPGA fabric, both sharing a common DRAM memory. A representative block diagram of the considered system is illustrated in Figure 2.

Possible representative platforms compatible with the considered system include the Zynq-7000 family by Xilinx, which provides ARM Cortex A9 processors and a FPGA fabric ranging from 28K up to 444K logic cells. Two types of computational activities can run on such a system:

- *software tasks* (SW-tasks): they are computational activities running on the processor; and
- *hardware tasks* (HW-tasks): they are functions implemented in programmable logic and executed on the FPGA fabric.

SW-tasks can speedup parts of their computation by requesting the execution of HW-tasks, which can be considered as *hardware accelerated functions*.

The area of the FPGA fabric is divided into a reconfigurable region and a static region. The reconfigurable region hosts the HW-tasks while the static region includes support modules for the HW-tasks, such as communication devices. The reconfigurable region is partitioned into *slots*, each including the same number of logic blocks. A HW-task can execute only if it has been programmed into a slot. Each slot can be reconfigured at run-time by means of a *FPGA reconfiguration interface* (FRI) and can accommodate at most one HW-task.

As typical for most real-world platforms (e.g., [13], [14]), the FRI

(i) can reconfigure a slot without affecting the execution of the HW-tasks currently programmed in other slots;
(ii) is a peripheral device external to the processor (e.g., like a DMA [15]) and hence does not consume processor cycles to reconfigure slots; and
(iii) can program at most one slot at a time.

To program a given HW-task into a slot, the FRI has to program all the logic blocks of the slot. This is because unused logic blocks have to be disabled to "clean" possible previous configurations. The FRI is characterized by a *throughput* ρ, meaning that a time $r = b^S/\rho$ is needed to reconfigure a slot, where b^S is the number of logic blocks in each slot.

Each SW-task uses a set of HW-tasks by alternating execution phases with *suspension* phases where the SW-task is descheduled to wait for the completion of the requested HW-task. The same HW-task cannot be used by more than one SW-task. Each SW-task is periodically (or sporadically) released, thus generating an infinite sequence of execution instances (denoted as jobs). SW-tasks are also subject to timing constraints, meaning that each of its jobs must complete its execution within a *deadline* relative to its activation. Figure 3 reports the pseudo-code defining the implementation skeleton of a SW-task that calls a single HW-task.

The HW-task is initialized at line 7, where the label `sample_hw_task` is used to refer its implementation stored in memory. At line 15, the SW-task configures the HW-task by specifying two memory locations: (i) `input_ptr`, that contains the *input data* for the HW-task and (ii) `output_ptr`, prepared to contain the *output data* produced by the HW-task. Finally, at line 18, the SW-task executes a *blocking* call that triggers the reconfiguration and executes the HW-task. The SW-task correspondingly suspends its execution until the completion of the HW-task. The inter-task communication mechanism is discussed in the following section.

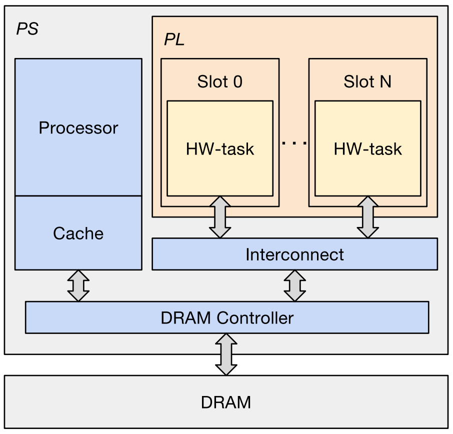

Figure 2: Block diagram of the considered system.

```
1  void sample_software_task()
2  {
3   // Task initialization (executed only once)
4   << Initialization part >>
5
6   // Define an instance of an HW-task
7   Hw_Task hw_task = hw_task_init(sample_hw_task);
8
9   // Task body
10  while (1)
11  {
12    << Software elaborations chunk >>
13
14    // Configure input and output data for the HW-task
15    hw_task_set_args(hw_task, input_ptr, output_ptr);
16
17    // Reconfigure and execute the HW-task
18    rcfg_manager_execute_hw_task(hw_task);
19
20    << Software elaborations chunk >>
21
22    // Wait for the next job
23    suspend_until(period);
24  }
25 }
```

Figure 3: Pseudocode of a SW-task calling a HW-task.

IV. System prototype

This section presents the implementation of a system prototype to handle HW-tasks under DPR on a real platform. The prototype has been used to conduct some preliminary experiments to evaluate the feasibility and the performance of the proposed approach.

A. Reference platform

The Zynq-7000 SoC family has been chosen as a reference platform for developing a working prototype of the system. It includes a dual-core ARM Cortex-A9 processor and a DPR-enabled FPGA fabric integrated on the same die.

The internal structure of a Zynq SoC comprises two main functional blocks referred to as processing system (PS) and programmable logic (PL) [15]. The PS block includes the ARM Cortex-A9 MPCore, the memory interfaces and the I/O peripherals, while the PL block includes the FPGA fabric. The subsystems in the PS are interconnected among themselves, and to the PL side, through an *ARM AMBA AXI Interconnect*.

The Interconnect can be accessed by custom logic modules (configured on the PL side) through a set of *master* and *slave* AXI interfaces exported by the PS to the PL side. In particular, the slave interfaces allow hardware modules hosted on the PL to access the global memory space where the physical RAM memory is mapped. This is achieved by implementing an AXI master interface inside the module logic. Such a master interface can be connected to the corresponding slave interfaces offered by the PS. In this way it is possible to implement a *shared-memory* infrastructure between the processor and the custom modules deployed on the PL.

The SoCs of the Zynq family supports dynamic partial reconfiguration under the control of the software running on the PS. The FPGA fabric included in the PL can be fully or partially reconfigured via the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams (i.e., images of custom modules to be configured onto the FPGA) from the main memory to the PL. This is achieved by means of the the processor configuration access port (PCAP).

B. Prototype architecture

In the system prototype, the area of the FPGA fabric included in the PL is divided into a static region and a reconfigurable region. The static region contains the static portion of the communication infrastructure (consisting in interconnection blocks similar to switches) and other support modules, while the reconfigurable region hosts the hardware modules that implement the HW-tasks and a common communication interface.

Such a common interface is similar to the one adopted by Sadri et al. [16] and includes (i) an *AXI master interface* for accessing the system memory, (ii) an *AXI slave interface* through which the HW-task can be controlled by the PS, and (iii) an *interrupt signal* to notify the PS when the computation has been completed. In the current setup, the AXI master interfaces included in the HW-tasks are attached to high-performance (HP) ports exported by the PS, while the AXI slave control interfaces are attached to the PS AXI master general purpose ports.

The reconfigurable region is partitioned into a fixed number of slots, each containing an equal number of logic resources. Each slot can accommodate a single HW-task. Since bitstreams relocation is not supported by the Xilinx's standard tools [13] [14] (i.e., the same bitstream cannot be used for multiple slots), each HW-task is synthesized as a *set of bitstreams*, one for each slot defined in the PL.

C. Software support

The software part of the system prototype has been developed as a user-level library for the FreeRTOS [17] operating system. The library facilitates the reconfiguration and the execution of HW-tasks by providing a simple API that enables the client programmer to exploit hardware acceleration.

From the client programmer perspective, the library models the concept of hardware acceleration with a set of HW-task objects and a software module named reconfiguration service. The interface of the reconfiguration service offers a single function to request the execution of a HW-task (as shown in Figure 3, line 18). Each HW-task object includes the following information: (i) a set of bistreams, one for each slot; (ii) the input parameters (memory pointers or data); (iii) two optional callbacks (linked to the start and the completion of the HW-task) that can be used to ensure memory coherence. The library has been build on top of the Xilinx software support library [18].

Before executing a HW-task, our implementation flushes the portion of cache containing the input data prepared by the SW-task, thus ensuring that the HW-task can access coherent data from the RAM memory.

Once the input data have been prepared, the SW-task checks for a vacant slot performing a *wait* operation on a FreeRTOS counting semaphore (initialized with the number of available slots). If all the slots are busy, the calling task is suspended until one of the slots will be released. When at least one slot is available, the function searches if any of the vacant slots already contains the requested HW-task. If none of the vacant slots contains the required HW-task, one of the vacant slots is reconfigured with the corresponding bistream. The calling task is suspended until the reconfiguration has been completed.

As soon as the requested HW-task is configured, it starts executing. The calling SW-task suspends its execution until the completion of the HW-task. When the HW-task completes, the calling SW-task is resumed and performs a *signal* operation on the slots counting semaphore. The completion is notified to the PS with the interrupt signal predisposed in the common interface described in Section IV-B. Once the SW-task is resumed, our implementation invalidates the cache portion corresponding to the output data produced by the HW-task, thus ensuring that the processor can access coherent data.

D. Experimental setup

To perform a set of experiments, the system prototype has been deployed on a ZYBO board that includes the Z-7010 Zynq SoC and 512 MB of DDR3 memory. The ARM core included in the PS of the Z-7010 runs at 650 MHz, while the clock frequency for the PL is set to 100 MHz.

In the experimental setup, 50% of the logic resources of the PL are allocated to the reconfigurable partition, while the remaining 50% are allocated to the static part. The reconfigurable partition is divided into two slots of equal size. Each slot contains half of the resources available in the reconfigurable partition. Since both slots contain an equal number of resource, the corresponding bitstreams (resulting from the logic synthesis of HW-task in each slot) have the same size, equal to 338 KB. Considering the size of the RAM memory available on the platform (512 MB), a large number of partial bitstreams can be stored without any relevant impact on the available memory.

V. Experimental results

This section reports the results of a set of experiments that have been conducted to evaluate the proposed approach on a case study application.

To test the system, four standard algorithms have been implemented as both HW-tasks and equivalent software procedures. The test set includes tree simple implementations of image convolution filters (*Sobel*, *Sharp* and *Blur*) and an integer matrix multiplier (referred to as *Mult*). The HW-tasks have been designed with the Vivado high-level synthesis tool, while the software versions have been implemented in the C language.

The Blur and the Sharp filters have been configured to process images of size 800×600 pixels, while the Sobel filter has been configured to process images of size 640×480 pixels. All the three filters process images with 24-bit color depth. The matrix multiplier processes matrices of size 64×64 elements.

A. Speed-up evaluation

A first experiment has been carried out to measure the speed-up factors achievable by the HW-task implementation of the four algorithms used in the case study. For each of such algorithms, the execution time of the corresponding

HW-task has been compared with the equivalent full software implementation for more 1000 runs. The results of this test are reported in Table I. The minimum speedup has been computed as the ratio between the minimum observed execution time of the software implementation and the maximum observed execution time for the HW-task.

As can be seen from the table, even though the FPGA is running at a lower clock frequency (100 MHz) compared to the processor (650 MHz), HW-tasks provide a consistent speed-up ranging from 2.5 to 15.2. The small differences between average and worst-case execution times can be explained by the fact that the functions are essentially stream processing operations with no branches depending on the input data.

Algorithm		Mult	Sobel	Sharp	Blur
Observed HW execution times	Average [ms]	0.785	12.710	24.631	24.628
	Longest [ms]	0.785	12.712	24.633	24.629
Observed SW execution times	Average [ms]	1.980	115.518	304.975	374.785
	Longest [ms]	2.017	115.521	304.994	374.811
Speedup	Average	2.523	9.089	12.381	15.217
	Minimum	2.515	9.087	12.380	15.216

Table I: Speed-up evaluation.

B. Response-time evaluation

A second experiment has been performed to evaluate the system behavior in a scenario where the number of HW-tasks to be executed exceeds the number of slots available on the FPGA fabric. Please note that such a scenario *is only possible by exploiting DPR*. The task set used for this experiment consists of four periodic SW-tasks with implicit deadline (i.e., deadlines equal to task periods). Each SW-task requests the execution of the HW-task corresponding to the algorithm of the case study (Section V). SW-tasks priorities are assigned according to the Rate-Monotonic algorithm. As mentioned in Section IV-C, each SW-task executes a flush operation (denoted as *cache flush*) before calling the HW-task and invalidates the cache when the HW-task completes (*cache invalidate* operation).

Table II reports the periods of the SW-tasks, the execution times of the cache flush and cache invalidate operations, and the response-times of the SW-tasks observed in 8 hours of execution.

Based on the collected data, it is worth observing that the considered application *cannot be scheduled without DPR* for the following reasons:

- due to the large execution times (see Table I), the application cannot be scheduled with a full software implementation;
- since the FPGA fabric has only two slots, it is not possible to statically configure all the four HW-tasks of the application;
- if the algorithms that cannot be allocated on the FPGA as HW-tasks are executed on the processor as pure software implementation, *any* possible combination of HW-tasks and software implementations leads to a non schedulable system.

This example shows that virtualizing the FPGA by the proposed timesharing mechanism can effectively improve the schedulability of applications on current heterogenous platforms.

The longest observed response time for the *Mult* SW-task shows that, even if this task has the highest priority in the system, it may experience high delays due to slot contention with other HW-tasks issued by lower-priority SW-tasks.

This happens because of the FIFO ordering of the semaphores used in the implementation. The execution of HW-tasks can hence be delayed by the reconfiguration and the execution of all the HW-tasks requested by other SW-tasks (independently of their priority). The analysis of such a delay is beyond the scope of this paper.

For some applications, the response-times can be improved by adopting different scheduling policies (i.e., different from FIFO) to manage HW-tasks. However, since HW-tasks execute in a non-preemptive manner, the largest execution time of the HW-tasks will always impose a lower-bound for the slot contention delay.

SW-task		Mult	Sobel	Sharp	Blur
Period [ms]		30	50	80	100
Cache flush [ms]		0.030	1.123	1.754	1.754
Cache invalidate [ms]		0.017	1.240	1.939	1.939
Observed Response time	Average [ms]	3.829	17.603	31.416	35.624
	Longest [ms]	24.017	20.418	33.086	43.160

Table II: Hardware accelerated task-set.

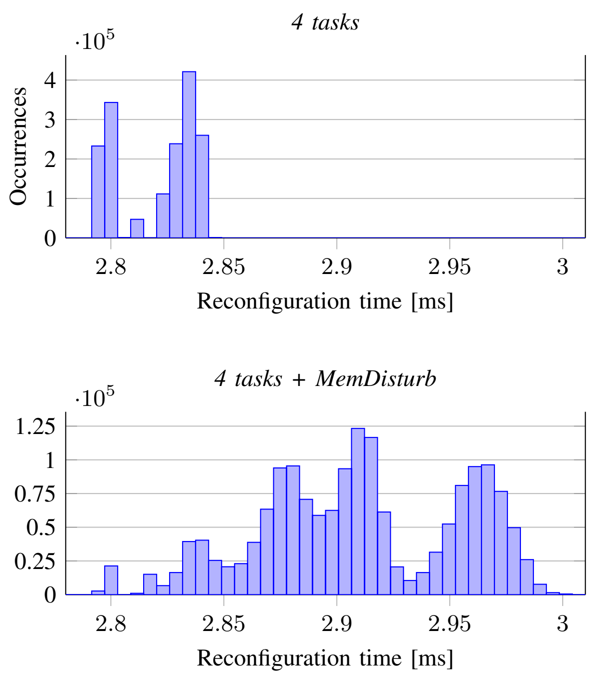

Figure 4: Distribution of reconfiguration times.

C. Reconfiguration times profiling

Finally, a third experiment has been conducted to profile reconfiguration times. The reconfiguration of the FPGA fabric is performed by the DevC subsystem described in Section IV-A. Such a module transfers bitstreams from the main memory to the PL configuration memory trough the PCAP port, which exploits the DevC DMA engine. The DMA accesses the system memory (where bistreams are stored) through an AXI master interface connected to the internal AXI Interconnect. Unlike the processor and the HW-tasks connected to the AXI slave ports, the DevC subsystem is not directly connected to the DRAM controller. In fact, it contends the access to the DRAM controller with other peripherals in the PS side.

In general, the throughput achievable by the DevC DMA depends on the traffic conditions on the AXI Interconnect, and the load on the DRAM controller. Modeling the bus contention on the AXI Interconnect and evaluating its performance goes beyond the scope of this paper. However, a first test was carried out to evaluate how a memory intensive SW-task interferes with the DevC, and hence affects reconfiguration times.

The task set used for this test includes the four tasks described in the experiment of Section V-B, and an additional *memory intensive* software activity (referred to as *MemDisturb*) continuously running in background without invoking HW-tasks. The MemDisturb software activity performs memory transfers between two memory buffers of 32 MB. The sizes of the buffers exceed the size of the processor L2 cache. Therefore, such a memory transfers generate a continuous stream of request to the DRAM controller that simulates a memory intensive SW-task.

Table III compares the reconfiguration times with and without the MemDisturb activity. Figure 4 illustrates the reconfiguration times distribution in both cases. The results of this experiment show that, despite a memory intensive software activity can affect reconfiguration times, its impact is very small and in the order of 0.1 ms. We believe that this result, although preliminary and far from being complete, is encouraging for exploiting partial reconfiguration in real-time systems, where bounded reconfiguration delays are essential to guarantee the system predictability. Given the size of the partial bistreams (338 KB), the average observed throughput for the DevC amounts to 117 MB/s without MemDisturb and to 113 MB/s with MemDisturb.

Experiment	Reconfiguration time [ms]		
	Min	Avg	Max
4 tasks (Section V-B)	2.791	2.820	2.846
4 tasks + MemDisturb	2.795	2.910	3.012

Table III: Observed reconfiguration times.

VI. Conclusions

This work presented an experimental study aimed at evaluating the use of dynamic partial reconfiguration for implementing a timesharing mechanism to virtualize the FPGA resource in heterogeneous platforms that also include a processor. Hence, an application consists of both software computational activities (running on the processor) and hardware modules implemented in programmable logic to be dynamically allocated on the FPGA, as requested by the software tasks. The temporal parameters involved in such a system (e.g., reconfiguration and execution times) have been profiled for a case study application. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to a fully static approach.

Besides the encouraging results, the experimental studies highlighted two major bottlenecks of today's platforms. First, all the evaluated FPGA platforms provide only a single reconfiguration interface, which is then contended by all the HW-tasks. Second, when the main memory is used to store both data and bistreams, an additional contention there exists on the Interconnect and the DRAM controller, which introduces further complications in the timing analysis. As a consequence, the presence of memories dedicated to bitstream storage would significantly improve both performance and predictability.

Future challenges include **(i)** the design and the analysis of scheduling algorithms for HW-tasks, **(ii)** the investigation of partitioning approaches for the FPGA area to limit contention on the reconfiguration interface, **(iii)** the implementation of improved inter-task communication mechanisms, and **(iv)** the design of real-time operating system mechanisms to support such a dynamic approach.

References

[1] M. Goosman, N. Dorairaj, and E. Shiflet. (2006) How to take advantage of partial reconfiguration in fpga designs. [Online]. Available: www.eetimes.com/document.asp?doc_id=1274489

[2] S. Altmeyer and G. Gebhard, "WCET analysis for preemptive scheduling," in *Proceedings of the 8th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2008.

[3] R. Stefan and S. D. Cotofana, "Bitstream compression techniques for virtex 4 FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008.

[4] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, October 2009.

[5] ——, "Cooperative multithreading in dynamically reconfigurable systems." in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, August 2009.

[6] M. Happe, A. Traber, and A. Keller, *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC)*. Springer International Publishing, April 2015, ch. in Preemptive Hardware Multitasking in ReconOS, pp. 79–90.

[7] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, and T. Arslan, "Microkernel architecture and hardware abstraction layer of a reliable reconfigurable real-time operating system (r3tos)," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, pp. 5:1–5:35, March 2015.

[8] M. D. Natale and E. Bini, "Optimizing the fpga implementation of hrt systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

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

[10] K. Danne and M. Platzner, "Periodic real-time scheduling for fpga computers," in *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded System*, May 2005.

[11] S. Saha, A. Sarkar, and A. Chakrabarti, "Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, March 2015.

[12] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, April 2007.

[13] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer-Verlag New York, February 2012.

[14] *Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, 2015, v2015.4.

[15] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx, 2015, v1.10.

[16] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using xilinx zynq," in *Proceedings of the 10th FPGAworld Conference*, September 2013.

[17] R. T. E. Ltd. Freertos real-time operating system. [Online]. Available: http://www.freertos.org/

[18] *OS and Libraries Document Collection*, Xilinx, 2015, v2015.3.