On the FPGA Dynamic Partial Reconfiguration Interference on Real-Time Systems

João Gabriel Reis*, Antônio Augusto Fröhlich*, Arliones Hoeller Jr.*†
*Software/Hardware Integration Lab
Federal University of Santa Catarina
{jgreis,guto}@lisha.ufsc.br
†Telecommunications Department
Federal Institute of Santa Catarina
arliones.hoeller@ifsc.edu.br

***Abstract*—This work proposes a deterministic hardware and software reconfiguration scheme capable of mitigating interference on reconfiguration execution time generated by system components performing I/O operations. The scheme decomposes the reconfiguration process into small steps such that it is preemptable, transparent, dynamic and compliant with real-time requirements. Moreover, the impact of the interference on system reconfiguration time was modeled and analyzed. Results show that using the Xilinx Zynq-7000 platform employing an ARM Cortex-A9 processor the reconfiguration time can grow up to 92% when real-time threads are performing I/O operations during hardware reconfiguration.**

***Keywords*—Dynamic partial reconfiguration, Real-Time, Field-programmable gate arrays (FPGAs), System-level design, HW/SW co-design, High-level synthesis.**

I. Introduction

Modern embedded applications are programmed arguably not through a series of hardware instructions, but through the specification of models and their interaction with language and API (Application Program Interfaces). Behind each API call, nevertheless, there is a series of software and hardware operations that may take different forms, for example, according to the availability of multiple implementations, hardware accelerators, network interfaces, or processor cores. Hardware/-Software co-design techniques have expanded the implementation space for APIs even further: models and APIs can be partitioned and synthesized into different cuts of software and hardware components. Nevertheless, the quality and timeliness of each API call might vary according to its implementation and the substrate (hardware, software) in which it is being deployed. The API call cost (number of processor cycles, energy consumed, or hardware area) for different implementations and substrates may also change according to system load as well as environmental and manufacturing-related variations (e.g. elevated power dissipation at higher temperatures). In this scenario, Dynamic Partial Reconfiguration (DPR) can help systems cope with dynamic non-functional requirements (such as performance and power), hardware defects (e.g. due to NBTI or PVT variations), or application requirements unforeseen at design time.

We have identified three main requirements for enabling effective and efficient DPR: 1) hardware reconfiguration support, 2) API for reconfiguration, and 3) non-intrusive reconfiguration policies. Device manufacturers handle the first requirement: the Xilinx Vivado suite, for example, allows users to change the configuration of part of an FPGA while keeping the rest of the device operational. Reconfigurable Operating Systems (ROSs) handle the second requirement by providing APIs for DPR. A typical ROS may provide, minimally and in addition to the standard hardware drivers and task scheduling functions, APIs for migrating a component between hardware and software implementations, and for managing hybrid software/hardware inter-process and inter-component communication. The final requirement is often left to application programmers, who must monitor any important parameters and decide when and under what circumstances to reconfigure the system.

The DPR process is carried by transferring a bitstream of the new components from a storage device to the FPGA reconfiguration interface. In most systems, this data transfer shares the I/O buses with the part of the system that still operational during reconfiguration, increasing contention for bus access that can interfere with the ordinary system execution. Care must be taken to prevent such interference from delaying the execution of hard real-time tasks.

In this paper, we model and analyze the interference of other systems components on partial reconfiguration time. Moreover, we also propose a deterministic reconfiguration scheme that can be used in real-time systems to mitigate the interference phenomenon. We quantitatively analyzed the interference that other peripherals impose to the time of hardware reconfiguration. Our results under the Xilinx Zynq-7000 platform using an ARM Cortex-A9 processor running the EPOS [1] operating system showed that reconfiguration time can grow up to 92 % when system peripherals are performing I/O operations during hardware reconfiguration.

The remaining of this paper is structured as follows. Section II presents related work and Section III models the interference issues faced by hardware reconfiguration. Section IV describes our approach to handling DPR interference, and finally Section V presents a quantitative analysis of the interference phenomenon.

II. Related Work

Several works were proposed to leverage FPGAs potential for reconfigurable computing applications by providing a concise hardware infrastructure. Nevertheless, none of them

consider the reconfiguration interference on real-time tasks and vice versa. The Erlangen Slot Machine (ESM) [2] computing platform proposes a uniform reconfigurable partition allocation allowing multimedia applications to benefit from an interchangeable processing blocks. In ESM, the reconfigurable modules interface with the external world is not constrained to a fixed location on the FPGA due to ESM I/O multiplexing system. Peripherals such as cameras and displays are accessible to the reconfigurable modules deployed in different FPGA regions. In HybridThreads (HThreads) [3], Andrews et al. extend software engineering concurrency patterns to reconfigurable computing platforms. Threads running on CPU and threads implemented on FPGAs coexist and interact through a POSIX threads compatible layer while common operating system synchronisation mechanisms are implemented on reconfigurable hardware. Configuration Access Port Operating System (CAP-OS) was conceived to exploit reconfiguration in heterogeneous Multiprocessor Systems-on-Chip (MPSoCs) implemented in FPGAs [4]. Computations are performed either by software threads in a processor or by hardware threads in a processor coupled with a hardware accelerator handling compute-intensive tasks, and communication is based on Message Passing Interface (MPI). CAP-OS uses a preemptive real-time priority-based scheduling algorithm divided into a static list scheduling and a dynamic scheduling step capable of scheduling the task reconfiguration during runtime. The scheduling algorithm considers resource constraints such as Internal Configuration Access Port (ICAP) exclusiveness and intertask dependencies.

Works such as BORPH [5], FUSE [6] and ReconOS [7] use standard operating systems as a backend for reconfigurable computing. Their approach lowers the learning curve for newcomers in reconfigurable computing and eases porting software-only applications to reconfigurable platforms. BORPH and FUSE extend the Linux kernel providing native support for FPGAs, treating them as computational resources instead of coprocessors. Unlike BORPH, FUSE partitions the FPGA in slots, thus one FPGA can contain multiple hardware threads. ReconOS provides a common abstraction layer for software and hardware threads as well as a set of communication and synchronisation primitives for them. It focuses on real-time applications and multithreading providing to the user an API based on the eCos operating system as well as on Linux.

Other works explore FPGA's reconfigurable resources in niche applications such as streaming or even high-reliability applications. The SPREAD programming model [8] is a proposal for reconfigurable computing that focuses on high throughput point-to-point streaming applications. It presents a common software/hardware thread interface and unlike the other solutions, in SPREAD a thread can be set as reconfigurable and thus switch domain during runtime. A stub thread creates a wrapper around the thread's software and hardware implementations to guarantee a common reconfigurable thread interface. R3TOS [9] is an operating system that focuses on reliable computing and provides real-time support for hardware threads. It differs from previous approaches in its capability of allocating hardware tasks in any FPGA region during runtime. R3TOS provides hardware tasks with virtual channels; the FPGA's ICAP is used to transfer data between hardware threads. R3TOS manages the reconfiguration of stateless hardware threads as a single operation carried prior to thread execution.

On a wider perspective, most initiatives extend off the shelf operating systems to offload applications running on CPUs by using hardware accelerators accessed through standard APIs. We focus on analyzing three aspects in those initiatives:

1) Do they perform DPR?
2) Does the reconfiguration process respect real-time constraints?
3) Is the reconfiguration process aware of interference sources that might delay the reconfiguration?

Dynamic partial reconfiguration seems to be supported only on SPREAD, R3TOS, and ReconOS. Despite their rich set of features, in the three projects, the application programmer is charged with deciding what thread or component to reconfigure and when to reconfigure. As BORPH and FUSE rely on Linux as their backend, they are not suited for real-time applications due to the inherent non-real-time behavior of the kernel. ReconOS might be configured to use eCos real-time scheduler. In this case, hardware and software threads present real-time capabilities, but it is not clear if thread migration can cope with real-time aspects. None of the analysed works explore and try to mitigate the interference sources that might arise during partial reconfiguration of the FPGA such as bus contention.

III. Dynamic Partial Reconfiguration Interference Sources

DPR is the process of modifying reconfigurable hardware circuitry by downloading partial bitstreams while the remaining system continues to operate without interruption. It allows designers to implement complex applications in smaller FPGAs by reconfiguring its resources during execution time, reducing power and improving system flexibility. With DPR, designers can make efficient use of silicon by only loading into the FPGA the necessary functionality at any point in time.

The DPR process is carried by transferring the bitstream from a storage device to the FPGA reconfiguration interface as presented in Figure 1. FPGAs normally employ two methods for bitstream loading [10], [11]:

- The CPU reads the bitstream from the storage device and write it, word by word, to the memory mapped register that controls the reconfiguration interface.
- The reconfiguration interface has a DMA engine that fetches data directly from the storage device; the CPU must only issue a start command to initiate the process.

The granularity and reconfiguration method of each FPGA depends on the architecture dictated by its vendor. Xilinx 7 Series and Zynq devices all support dynamic partial reconfiguration. The 7 Series devices are equipped with ICAP reconfiguration interface [12]. The ICAP relies on a user-designed DPR controller to load partial bitstreams through its interface for partial or full reconfiguration. The DPR controller can be implemented in the static region the FPGA, and thus operate without interruption throughout reconfiguration, or even from outside the reconfigurable fabric. Bitstreams are stored in non-volatile storage devices that can be accessed by the DPR controller thus the reconfiguration time of an N word bitstream

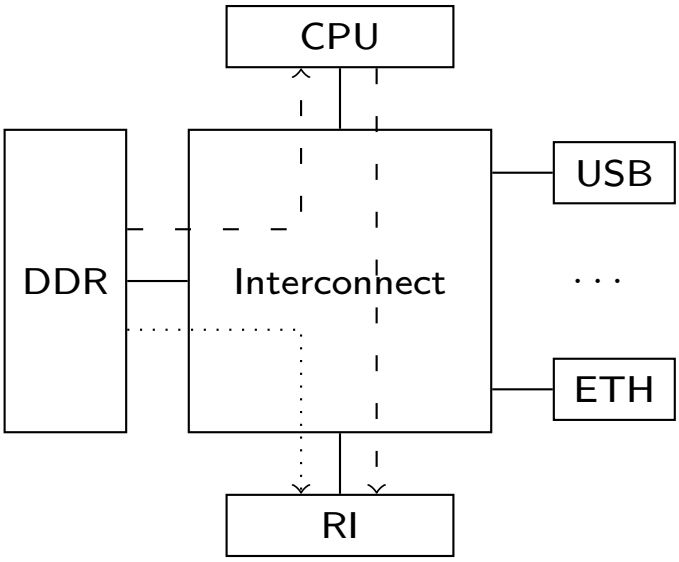

Figure 1: Bitstream data path during partial reconfiguration in a SoC with a central interconnect. Dotted line represents a DMA based datapath and dashed line a datapath performing programmed I/O, using the CPU to transfer data.

is given by

$$t_{icap} = N(t_{load} + \alpha_{load} + t_{store} + \alpha_{store})$$

where t_{load} and t_{store} are the execution times of a load from the storage device into the DPR controller and a store of the loaded value into the reconfiguration interface respectively. α_{load} and α_{store} account for the time spent due to interconnect contention in each operation.

Given the overhead implied in having one load and one store operation for each word moved to the reconfiguration interface, newer devices propose alternative reconfiguration schemes to relieve the DPR controller. Processor Configuration Access Port (PCAP) is a reconfiguration interface present on Xilinx Zynq SoCs addressing this issue. Instead of relying on a DPR controller to move data into it, PCAP has its DMA controller used to fetch bitstreams from the storage device. PCAP is not completely autonomous; the DPR controller still needs to set the bitstream length and its initial address in the storage device before issuing a DMA start command. As PCAP is a memory mapped device, the DPR controller set all parameters necessary for starting a partial reconfiguration with load operations. This way, the reconfiguration time is expressed by

$$t_{pcap} = M(t_{store} + \alpha_{store}) + N(t_{load} + \alpha_{load})$$

in which M is the number of registers that need to be configured to start the reconfiguration process. As in most cases $M \ll N$, PCAP is usually much faster than ICAP [1].

In all DPR schemes, the bitstream data path crosses the system interconnect shared by system peripheral and CPUs. If the interconnect is based on a bus scheme, parallel access to storage devices by the peripherals will imply in bus contention, lowering data transfer throughput and augmenting the reconfiguration time. On a broader perspective, a thread performing I/O can interfere with the reconfiguration process, a fact that must be considered when designing a real-time system. Not only the system threads interfere with partial reconfiguration but the opposite also occurs. If the reconfiguration is triggered while a real-time thread is performing I/O through the central interconnect, it might consume a bandwidth previously allocated to the real-time thread. We consider that is the application programmer responsibility to assure enough bandwidth for multiple real-time threads performing I/O, however, it is not his responsibility to take the reconfiguration bandwidth into account. The operating system must assure there is enough bandwidth for the partial reconfiguration process and that it will not interfere with high priority threads or be interfered by threads with lower priority.

IV. Dealing with Interference

The operating system itself is a major source of interference for the hardware reconfiguration process. Context saving and restoring, if not managed properly, can threaten the reconfiguration determinism. A context switch occurring at the wrong instant can disrupt partial reconfiguration and cause several drawbacks to applications depending on it. Moreover, most of the times the operating system I/O mechanisms, as previously stated, are not taken into account when dealing with interference. In EPOS framework, partial reconfiguration is split into small steps executed while the operating system is idle [13]. Hence, even with little available idle time, reconfiguration can be carried out transparently, comply with real-time requirements and be aware of possible interference sources in the system.

The *idle* thread is, therefore, the entity in charge of keeping the system configuration in tune with the reconfiguration policy specified by the user. The speculative reconfiguration triggered by the *idle* thread is a key element in EPOS reconfiguration approach. We analyzed which operations in the reconfiguration process are more susceptible to external interference. As expected, the most susceptible operations are those that move significant amounts of data from one subsystem to another: `restore_state()` and `load_comp()`. The former moves binary blobs containing the component's state from the DDR to the component's hardware implementation in the FPGA, and the latter performs the bitstream loading.

EPOS deploys two mechanisms, or steps, to keep reconfiguration time deterministic even when sources of interference on the interconnect are operating. These mechanisms are described in sections IV-A and IV-B.

A. Power Down of Low-Priority Components

The first mechanism consists of powering down peripherals in use by threads with priorities bellow the reconfiguration threshold priority P. The reconfiguration threshold P is the highest priority among the threads benefiting from the reconfiguration. This way, the reconfiguration process functions without interference from lower priority threads. Turning off the peripherals is carried by `Component_Manager power_down()` method right before starting to load the new

[1] Altera FPGAs reconfiguration interface, Fast Passive Parallel (FPP) [11], is deployed in schemes similar to Xilinx reconfiguration interfaces despite being developed by a different vendor. Unlike ICAP and PCAP, FPP is not available from within the FPGA, the DPR controller performs partial reconfiguration from outside the FPGA.

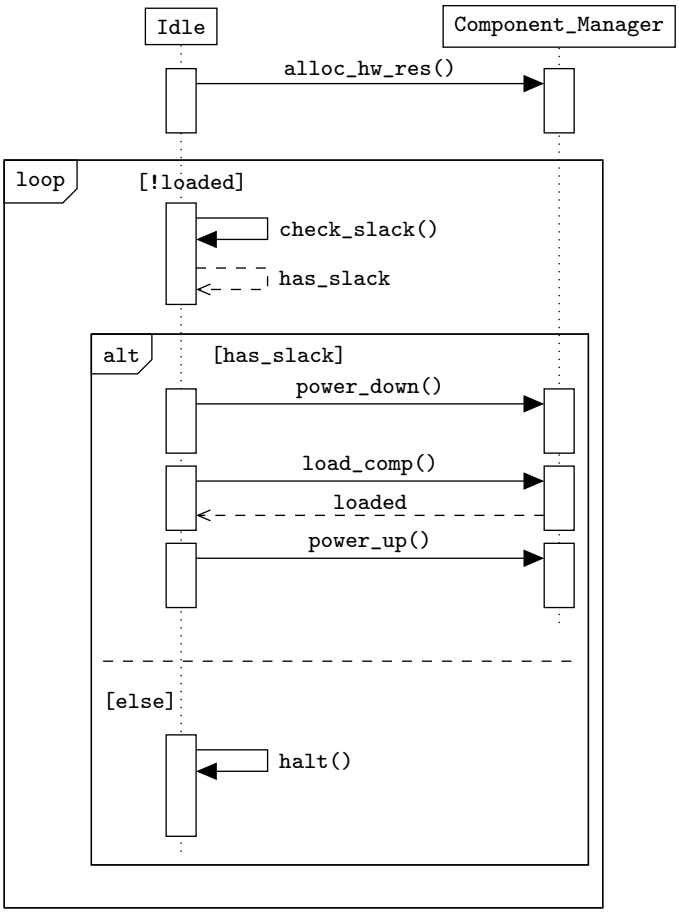

Figure 2: Reconfiguration triggering and component loading.

component in the *idle* thread as showed in Figure 2. After loading the new component, the peripherals are turned on by the `power_up()` method.

Both `power_up()` and `power_down()` methods are wrappers to EPOS Power Manager interface. EPOS uniform power management interface allows changing operating modes of individual components, including the ability to turn them on and off [14]. The power manager also keeps track of the relation between system components, ensuring consistency of operating mode transitions. This behavior is achieved by navigating the hierarchical organization of EPOS components.

The power manager also controls concurrent access to operating mode transitions of system devices. When threads instantiate components, the power manager is notified so it can identify the client threads of each device. If more than one thread uses the same device, an operating mode transition only takes place if all client threads agree on the target operating mode. If two or more threads issue different operating mode transition commands, the targeted device stays at the requested operating mode that delivers the larger set of services or the higher performance.

In EPOS, components deliver their power management capabilities through a Power Manager API consisting of a getter and a setter for the component operating mode: `power()` and `power(op_mode)`. The system also provides a set of predefined operating modes for portability purposes: `OFF`, `STANDBY`, `LIGHT`, and `FULL`. Also, all EPOS components can have their power managed, including operating system abstractions such as threads or sockets, not only peripherals.

The `Component_Manager` uses the power management interface of EPOS `Thread` component to power down peripherals whose use are exclusive to low priority threads. The `Thread` component implements two operating modes: `FULL` and `OFF`. When a call to the `power(OFF)` of a thread takes place, the Power Manager navigates all components to which the target thread is a client and forwards the same command to them. Each component can in turn use other components or devices, so the Power Manager propagates the command until it reaches the bottom of the component hierarchy. Each device reached is shut down if the target thread is its sole client. The device stays in the higher level of operation requested by its client threads otherwise.

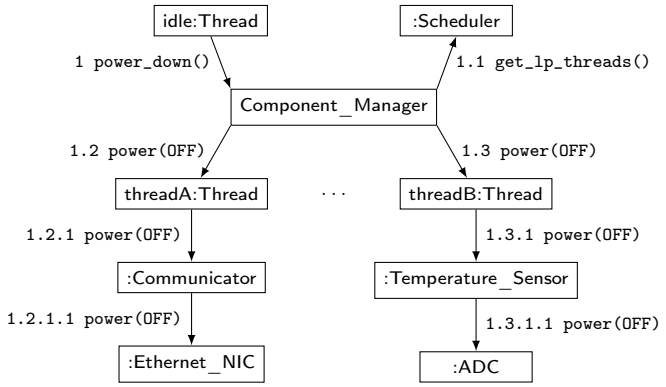

Figure 3: Example power down of low prioirty components.

Figure 3 shows an example power down of low priority components. After the *idle* thread calls the `Component_Manager` `power_down()` method, the component manager accesses the system scheduler interface to identify which threads have priority bellow the priority of the reconfiguration process (P). After retrieving the list of threads, the `Component_Manager` proceeds to issue `power(OFF)` commands to all low priority threads. The example shuts down two low priority threads. The first one, `threadA`, uses a `Communicator` abstraction[2] to which the `power(OFF)` command is forwarded. The `Communicator` accesses the Power Manager to check on its clients. If `threadA` is the unique active client of `Communicator`, the command is forwarded to the `Ethernet_NIC` mediator, which handles the device shut down. The second thread, `threadB`, uses the `Temperature_Sensor` abstraction. Similarly to what happens to `threadA`, the `power(OFF)` command is forwarded to the underlying components, eventualy reaching the `ADC` mediator and shutting it down if it has no active clients.

B. Interference-Aware Reconfiguration

The second mechanism monitors interference sources in the *idle* thread by profiling each `load_comp()` execution time, t_{lc}. The canonical `load_comp()` execution time without any

[2] `Communicator` is a configurable communication component assembled to abstract the protocol stack of a communication subsystem

source of interference, T_{lc}, is measured on system startup before launching the application. If $t_{lc} > T_{lc}$ for any `load_comp()` execution, EPOS considers that threads with priority greater than or equal to P issued I/O operations that use the interconnect and are sleeping, waiting their completion. If high priority threads are executing and interfering with `load_comp()` operation, the *idle* thread can only abort the recofiguration. The reconfiguration abort procedure will invoke the `power_up()` method to wake all peripherals being used by threads with priority lower than P.

Extra attention is also taken when saving and restoring the state of the component being reconfigured [13]. As it is an atomic operation, we cannot interrupt it as with the component loading step. We can, however, account for its worst interference case, thus deferring its execution until it becomes possible. To do that, the impact each peripheral has on the interconnect when performing I/O is annotated with an entry in its EPOS hardware mediator traits. This entry is called interconnect deterioration factor and is a fractional number bigger than 1. When checking if there is enough slack to perform the atomic operation, the slack value is multiplied by the sum of the deterioration factors of all system peripherals abstracted by hardware mediators. For example, let the Ethernet have a deterioration factor (d) of 1.5 and an execution time of the atomic operation (t) of 0.8 ms. Also suppose that there is an available slack time (s) of 1 ms and the Ethernet is the only peripheral performing DMA. In this case, $d \cdot t = 1.2\,\text{ms} > s = 1\,\text{ms}$, and the atomic operation will not take place. During execution, the `Component_Manager` retrives a list of all deployed hardware mediators and their traits from EPOS `System` component.

V. RESULTS

For the experimental analysis and evaluation of the interference phenomenon described in Section III we performed an experiment on a SoC containing an FPGA and a dual-core ARM processor, the Xilinx Zynq. The software was compiled with GCC 4.4.4 targeting the ARMv7 ISA, and the hardware platform was prototyped in a Xilinx's XC7Z020 SoC using Xilinx's Vivado 13.4 for RTL hardware synthesis. Synthesis constraints were adjusted on Vivado to minimize circuit area.

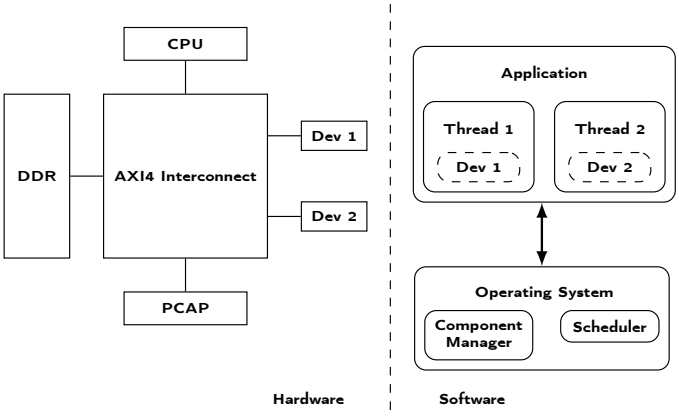

Figure 4: Experiment overview. Application and operating system run on the CPU and each thread on the application instantiates a peripheral connected to the AXI4 interconnect.

The experiment consists of periodic real-time threads starting a DMA transfer on each activation. Figure 4 presents an overview of the experiment architecture. We consider that these threads performing I/O through DMA are the only threads running in the systems besides the *idle* thread where the reconfiguration will be carried. Also, the DMA execution time is bigger than the reconfiguration time executed by the *idle* thread when there is no interference in the interconnect. Each thread runs in Zynq's CPU and utilizes a different hardware accelerator instantiated on Zynq's FPGA. The accelerator is connected to the rest of the system through an AXI4 master port. The communication between the threads and the accelerators is carried by sharing a DDR memory region accessed by the accelerator's DMA controller. Even tough Zynq has multiple CPUs, we decided to use only one of them in this experiment and leave further investigation on the subject for future works. Zynq's DMA-based reconfiguration interface, PCAP, fetches the bitstream from the DDR memory.

In the experiment, we varied the DMA controller operating frequency and the write burst length of each write transaction. After performing the handshake for starting a write transaction, the accelerator will gain permission to transfer a number of words equal to its write burst length. Depending on the bus arbitration scheme, bigger burst lengths might allow bus access to the peripheral for a bigger share of time and interfere more in the reconfiguration process. The same argument is valid for augmenting the operating frequency of the DMA controller: the peripheral will issue write transactions more frequently causing more interference. We also varied the normalized size of the bitstream being loaded for each configuration of frequency and burst length. The PCAP reconfiguration time with zero, one, and two threads using hardware accelerators interfering in the reconfiguration process for all combinations of operating frequency and burst length are presented in Figure 5. We choose two burst length values: 256 words, the maximum burst length value defined by the AXI4 specification and 32 words, a smaller value representing smaller burst lengths. For the operating frequencies, the chosen values were 200 MHz, which is a fair operating frequency for IPs implemented on mid-range FPGAs, and 10 MHz to represent slower devices. The partial bitstream used as the normalization base has 151048 bytes, while a bitstream for fully reconfiguring the FPGA has 4045564 bytes. In a rough estimate, the partial bitstream reconfigures 4% of all FPGA resources.

Examining the gathered results, it is clear that for all combinations of burst length and operating frequency, the number of peripherals performing DMA and using the interconnect is inversely proportional to the reconfiguration time. The peripherals operating frequency has a clear influence on the reconfiguration time: the lower their frequencies more time the PCAP DMA has to wait to receive ownership of the bus. Smaller burst lengths, unlike we predicted, do not generate more interference than bigger ones. It happens probably due to the bus arbitration scheme that defines a fair bandwidth distribution for all bus connected peripherals, including PCAP. Moreover, the reconfiguration time is directly proportional do the bitstream size in all cases, even with interference. The reconfiguration time rises 92% compared to the reconfiguration time without interference when two real-time threads perform I/O operations. The increase happens when both peripherals execute at 10 MHz with their DMAs issuing 256 words burst

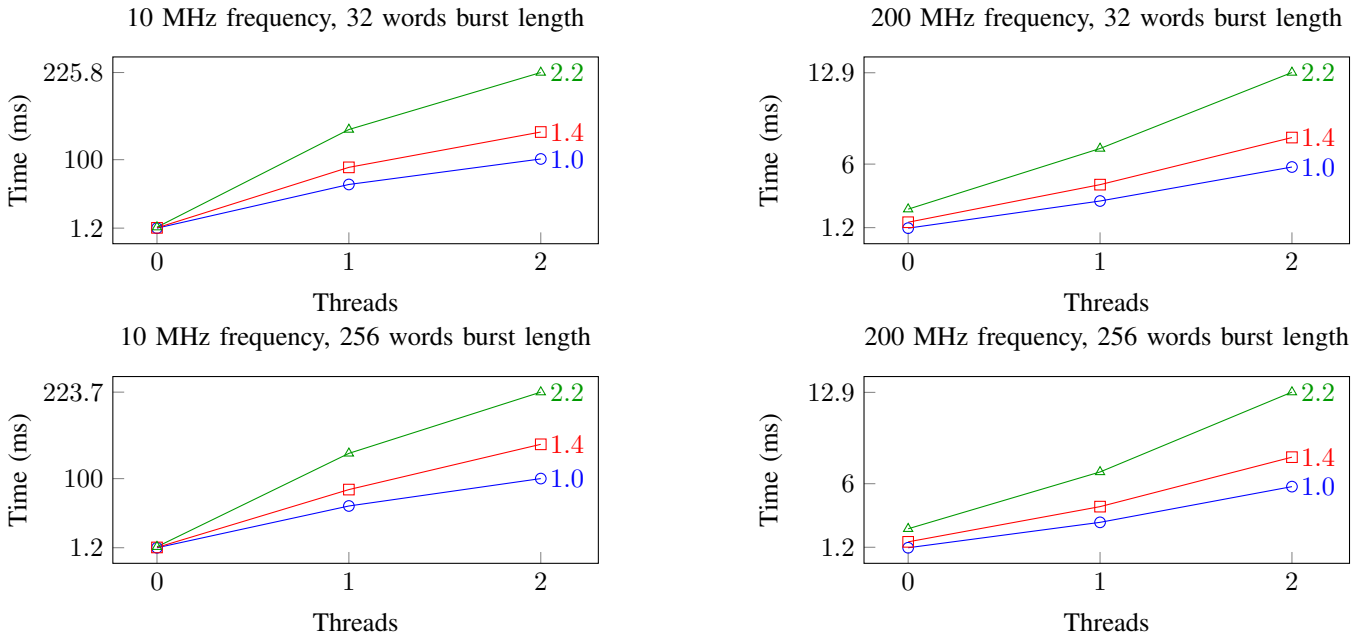

Figure 5: Reconfiguration time for different normalized bitstream sizes with interference inflicted by peripherals performing DMA. Different configurations of peripheral's AXI4 transaction burst length, threads number, and peripheral operating frequency.

length AXI transactions.

VI. Conclusion and Future Work

This paper presented an analysis of the interference on the DPR execution time generated by system peripherals performing I/O operations. Experiments showed that reconfiguration time can rise 92% when other peripherals are performing I/O operations. The analysis considered peripherals synthesized in an FPGA platform with two variable characteristics: DMA operating frequency and AXI4 burst length. While the DMA operating frequency and number of peripherals performing DMA seems to have a significant impact on reconfiguration time, AXI4 burst length sizes does not seem to impact it.

Also, we proposed a preemptable and transparent reconfiguration scheme that takes into consideration the interference issues and maintains its execution time deterministic. Our operating system infrastructure manages the whole dynamic reconfiguration process and ensures that it can be used in real-time systems without interfering in its time constraints. The reconfiguration scheme is performed by the operating system's *idle* thread and will turn off the peripherals deployed in threads with lower priority. High priority threads are not interrupted and thus, the reconfiguration is aborted when they are issuing I/O operations that can interfere on reconfiguration time.

On-going and future work investigate how the proposed framework can cope with multiple design objectives such as dependability, efficiency, and real-time operation by providing a transparent approach to DPR.

References

[1] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, aug 2001, no. 17.

[2] C. Bobda et al., "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," in *Proc. Intl. Conf. on Field-Programmable Technology*, 2005, pp. 37–42.

[3] D. Andrews et al., "Achieving Programming Model Abstractions for Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[4] D. Göhringer et al., "Operating System for Runtime Reconfigurable Multiprocessor Systems," *International Journal of Reconfigurable Computing*, 2011.

[5] R. Brodersen et al., "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Proc. Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 259–264.

[6] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 170–177.

[7] E. Lubbers 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.

[8] Y. Wang et al., "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model," *IEEE T. on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2179–2192, 2013.

[9] X. Iturbe et al., "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1542–1556, August 2013.

[10] *Vivado Design Suite User Guide: Partial Reconfiguration v2015.1*, Xilinx, 4 2015.

[11] *Altera Configuration Handbook*, Altera, 10 2008.

[12] *7 Series FPGAs Configuration: User Guide*, Xilinx, 6 2015.

[13] J. G. Reis, L. F. Wanner, and A. A. Fröhlich, "X-ware: Mutant computing substrates," in *Proceedings of the 2015 IEEE International Symposium on Rapid System Prototyping*, Amsterdam, oct 2015.

[14] A. Hoeller Jr., L. F. Wanner and A. A. Fröhlich, "A Hierarchical Approach For Power Management on Mobile Embedded Systems," in *5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, Braga, Portugal, Oct. 2006, pp. 265–274.