Analysis of the reconfiguration latency and energy overheads for a Xilinx Virtex-5 FPGA

JAVIER OLIVITO, University of Zaragoza, 0034876555081, jolivito@unizar.es
FELIPE SERRANO, Complutense University of Madrid
JUAN ANTONIO CLEMENTE, Complutense University of Madrid
HORTENSIA MECHA, Complutense University of Madrid
JAVIER RESANO, University of Zaragoza

In this paper we have evaluated the overhead and the tradeoffs of a set of components usually included in a system with run-time partial reconfiguration implemented on a Xilinx Virtex-5. Our analysis shows the benefits of including a scratchpad memory inside the reconfiguration controller in order to improve the efficiency of the reconfiguration process. We have designed a simple controller for this scratchpad that includes support for prefetching and caching in order to further reduce both the energy and latency overhead.

1. INTRODUCTION AND RELATED WORK

Dynamic partial reconfiguration (DPR) is one of the most interesting features of FPGAs. Reconfiguration enables the reuse of the FPGA hardware resources for different tasks that can be loaded at run-time according to the system needs. Hence, FPGAs can be used to develop flexible platforms that can adapt themselves to the execution of different applications. DPR has been thoroughly explored as a way to reduce area and power in some systems [1-3]. The idea is to reuse the reconfigurable area for different tasks that are not executed simultaneously. However, the reconfiguration process itself introduces overheads both in the execution time and in the energy consumption. The reason is that it involves not only using the reconfiguration circuitry to update the device configuration, but also moving large data sets from the memory where the configurations are stored to the reconfiguration port.

The analysis of the reconfiguration overheads has been the target of many research groups, and they have focused on execution-time overheads. Several works have proposed alternative reconfigurable architectures. For instance, multi-context FPGAs [4] enable the load of a new configuration while another one is being executed. Another example is coarse-grain architectures [5], which offer less flexibility but require smaller configuration bitstreams that can be easily stored on-chip in order to further increase the reconfiguration speed [6-7]. We believe that the techniques that we expose in this article can be also applied on these architectures, but we have focused on single-context FPGAs because they dominate the reconfiguration landscape.

Another approach to reduce the reconfiguration overheads is to apply scheduling techniques that attempt to hide the reconfiguration latency by fetching the configurations in advance and storing them in idle reconfigurable regions. This is a powerful technique for embedded systems where the task execution order is known because these techniques use the task-graph information to prefetch the configurations that will be needed in the near future. Some relevant works that propose reconfiguration scheduling techniques are [8-13]. It is important to mention that the controller described in this article is compatible with any of these techniques

and, in fact, it can improve their results since even when all the reconfigurable regions are busy, the configurations can be still prefetched to the internal memory included in our controller. This is explained in detail in Section 3.3.

Another powerful way to reduce the reconfiguration time overhead is to reduce its size by applying compression techniques. With this approach, configurations are fetched faster but they have to be decompressed before writing them in the device, and this may involve additional execution time and energy penalties. However, these overheads can be smaller if the system includes hardware support to decompress the configurations. Some relevant works on compression are [14] and [15]. Again, these techniques are fully compatible with our work since our controller could easily incorporate a decompression module.

Another option to reduce the reconfiguration overhead is to customize the memory resources used to store the configurations. For instance, [16] proposes to include heterogeneous on-chip memory modules, ones optimized for performance and others optimized for low power. With this approach the designer can explore different power/performance reconfiguration tradeoffs. This idea is orthogonal with our approach and could be included in our controller.

Another interesting technique is configuration caching. The idea is to store different configurations in different reconfigurable regions in the device and design specific replacement techniques to maximize the configuration reuse. Some interesting configuration caching techniques are [17-19]. Our controller has been designed to support configuration caching, and, in fact, its internal memories can be used to apply these techniques in an additional level. This is further explained in Section 3.3.

Regarding the reconfiguration process, many works assume that the reconfiguration latency is a fixed time that can be calculated by dividing the size of the configuration by the peak bandwidth of the reconfiguration port. In fact, the peak reconfiguration bandwidth is usually the only value provided by FPGA vendors. However, is that value relevant? Can we really achieve that bandwidth? How can we do it? In [20], Papadimitriou *et al.* elaborated a theoretical cost model to estimate the reconfiguration time at an early stage of the development. The authors estimated the reconfiguration time when fetching the bitstreams from external memories. They verified the model with measures over a real system implemented on a Virtex-II Pro FPGA and reported a model difference of ~25%, which is due to different clocks of processors, i.e. 300 MHz for the calculated one and 100 MHz for the measured one. Several works have pointed out that the actual reconfiguration latencies obtained in representative case studies are one order of magnitude, or even two, worse than the peak reconfiguration latency [21-22]. The reason for these poor results is that configurations are usually stored in off-chip non-volatile memories, and the actual bandwidth of these memories is much smaller than that of the reconfiguration port. Hence, the bottleneck is not the reconfiguration port, but the bandwidth of the external memories. Some previous works have demonstrated that it is possible to achieve almost peak-performance when using some specific external memories with additional support. For instance, in [23] the authors claim to achieve almost peak performance in a Virtex-6 when using DDR3 memory, a Direct Memory Access (DMA) controller, and some additional FIFOs to hide the latency, and in [24], they retrieve configurations from an external SRAM through a customized DMA. In [23], they also state that it is possible to achieve a speedup of 2 by overclocking the Internal Configuration Access Port (ICAP). However, although each new generation

of external memories is providing more bandwidth, it is likely that the speed of the reconfiguration port will also scale accordingly. In fact, Virtex UltraScale+™ FPGAs can be reconfigured at 200MHz (previous generations were limited to 100 MHz). Moreover, in many systems the energy overhead is even more relevant than performance, and the use of external memories strongly penalizes in power and energy.

Some previous works propose the inclusion of memory resources inside the reconfiguration controller in order to preload configurations. In [25], the authors analyze the reconfiguration latency of a system implemented in a Virtex-4 FPGA that includes a system bus that is used for all the data transfers, including those needed to carry out a reconfiguration, i.e. reading the configuration and sending it to the reconfiguration port. They measured the reconfiguration latencies taking into account the different access schemes provided by the bus, and including a DMA controller. The results demonstrate the impact of the communication scheme in the reconfiguration latency. In their analysis, they claim that the only way to achieve the peak reconfiguration bandwidth is to avoid accesses to the system bus. Even when the system bus is available and transfers are performed through a DMA, these accesses introduce significant delays.

Other interesting works are [26] and [27]. In these articles, the authors propose partial reconfiguration controllers that reach a throughput of 1.433GB/s and 2.2GB/s respectively. These controllers allocate on-chip memory resources to store de configurations, and the ICAP is overclocked up to 550 MHz. Although the manufacturer does not guarantee proper operation at frequencies higher than 100 MHz, these works are a proof of concept of the use of internal memory for future and faster versions of the ICAP.

After analyzing the aforementioned works, it is clear that the reconfiguration latency drastically depends on where the configurations are stored and the communication scheme used to read them. The first contribution of this work is the analysis of how these two parameters affect the reconfiguration latency in a Virtex-5 FPGA. We made the evaluation on the XUPV5-LX110T Development System included in the Xilinx University Program. This evaluation board includes many different memories, and we selected two external memories, Flash and DDR2 DRAM, and the internal on-chip SRAM.

Other key metrics, especially in embedded systems, are power and energy. Hence, as a second contribution, we measured the energy demanded in a reconfiguration, depending on the kind of memory where the configurations are stored. We also measured the power overhead of some components providing support for the process, such as the controllers for the memories, the reconfiguration, or the DMA. This analysis of the energy overhead in the reconfiguration may be helpful for the developers in order to decide where to store the configurations.

Finally, as a third contribution, we have developed a simple and efficient reconfiguration controller that, as in [26] and [27], includes an embedded memory that can be used to store bitstreams. Using that memory the reconfiguration process can be carried out at the maximum speed supported by the ICAP while minimizing the energy consumption. As a novelty, this controller includes two additional working modes in order to provide support for configuration prefetching and caching using the memory embedded in the controller. These modes can be used to reduce the reconfiguration overheads when only some of the configurations can be stored on-chip.

2. TARGET ARCHITECTURE

Our target architecture consists of three different memories that can be used to store configuration bitstreams, a reconfiguration controller that provides an interface with the ICAP [28], a processor, and at least one Reconfigurable Region (RR). Figure 1 depicts the elements of this architecture. In our experiments, the processor is a Xilinx MicroBlaze, the system bus is a PLB 4.6, and the memories are a 1GB Compact Flash, a 64-bit wide 256MB DDR2 SODIMM, and the FPGA internal BlockRAMs (BRAMs). Additionally, DMA and interrupt controllers supplied by Xilinx were added in order to evaluate their performance-energy tradeoffs.

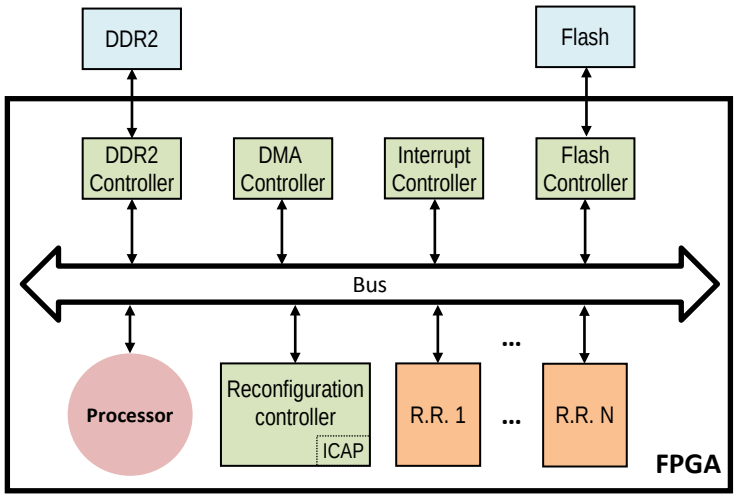

Fig. 1. Target architecture

The selection of the memories where configurations will be stored, and how to move these configurations among these memories and the reconfiguration controller is not straightforward, as it depends on multiple factors such as the power budget, the performance requirements, and the needs of other system components (for example, if the DDR or the Flash memories are demanded by other tasks or components, they would be already present in the system and then the static power of these controllers will not introduce an additional penalty related to DPR). In the experimental results presented in Section 4, we analyze the reconfiguration energy-performance tradeoff in order to help designers to make their decisions.

3. PARTIAL RECONFIGURATION CONTROLLER

3.1 Xilinx IP

Xilinx provides the XPS_HWICAP IP core for the Virtex-5 FPGA to manage partial reconfigurations in processor-based systems [29]. It is composed of several control registers, two small FIFOs, a finite-state machine (FSM), and a PLB bus interface [30]. Xilinx also provides a driver to use this controller from the processor-side. This driver enables to read configurations from any memory in the system and to send the data to the XPS_HWICAP controller through the PLB bus. This controller is easy to use and it should be the starting point for anybody who wants to carry out reconfigurations. However, this driver has not been designed to optimize the data transfers among the memories and the XPS_HWICAP. As a result, as it will be

explained in section 4, it only achieves a reconfiguration throughput of 12 MB/s, far from the peak 400 MB/s supported by the ICAP.

3.2 Multi-Mode ICAP Controller

In order to reach the maximum reconfiguration throughput, the ICAP should receive one 32-bit word per cycle at a 100MHz rate. As explained in the previous section, it is hard to achieve this throughput when configurations are stored in off-chip memories, or even when they are stored on-chip but they are accessed through a shared system bus. A solution is to store them inside the configuration controller as it was previously proposed in [25-27]. With this approach the reconfigurations can be carried out at full speed. In current FPGA architectures, this can be achieved by reserving part of the on-chip RAM for the controller. We refer to this on-chip RAM as bitstream memory.

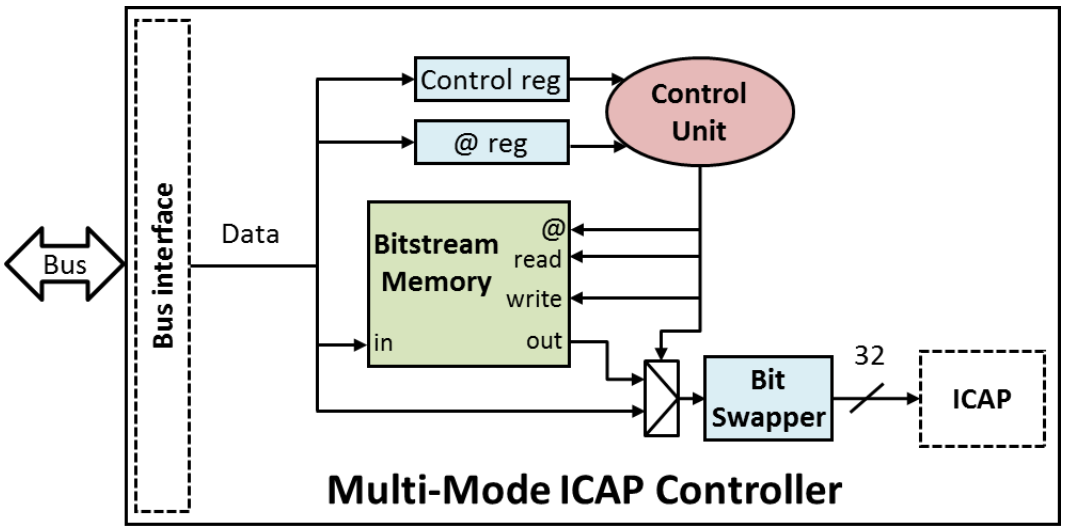

Fig. 2. Multi-Mode ICAP reconfiguration controller

Figure 2 illustrates the architecture of our partial reconfiguration controller, called Multi-Mode ICAP. The bus interface was automatically generated by the Xilinx EDK tool, and the remaining blocks were designed in VHDL and integrated inside the EDK project. The Control and the Address registers are software accessible. Hence the processor can easily provide the information and check if the controller has finished. If the system includes an interrupt controller, the controller can generate an interrupt when the *Done* bit is activated; again this is straightforward by using the EDK tools. This architecture based on an internal memory, a register-based interface, and a control unit that is very similar to those proposed in [25] and [26]. The main difference is that our control unit provides support for several useful working modes.

The Multi-Mode ICAP supports four different working modes:

a) *Mode 0: Configuration Load*. The controller receives a configuration and stores it in the bitstream memory. The controller does not send the configuration to the ICAP. This mode is useful to fetch configurations in advance from the external memories in order to reduce the

reconfiguration latency since, once a configuration is stored, our controller sends it to the ICAP at the maximum supported speed.

b) *Mode 1: External reconfiguration and configuration load*. The controller receives a configuration and forwards it to the ICAP. In parallel, it stores the configuration in the internal bitstream memory. In this mode the reconfiguration speed depends on how fast the configuration data are received from the bus, i.e. our controller does not reduce the reconfiguration latency in this mode. However, if the same configuration is required again, it can be loaded from the internal bitstream memory at the maximum speed.

c) *Mode 2: External reconfiguration*. The controller receives a configuration and forwards it to the ICAP as in the previous case, but in this mode the configuration is not stored in the bitstream memory.

d) *Mode 3: Internal reconfiguration*. The controller sends to the ICAP a configuration previously stored in the bitstream memory. The reconfiguration is carried out at full speed.

In order to support this functionality, only two software accessible registers (Control and Address registers) and little additional control logic are necessary. The control register (Figure 3) is used to configure our controller. Bit 0 reports when an operation has finished, bit 1 triggers the start of an operation, and bits 2 and 3 are used to select the working mode. Finally the remaining 28 bits are used to set the size of the configuration expressed in 32-bits words. The address register specifies the initial address of the bitstream memory to store or load a specific configuration.

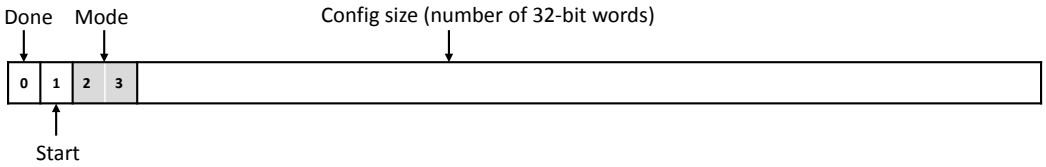

Fig. 3. Control register

The control unit (Figure 4) only requires three states: *Idle*, *Modes 0, 1, 2,* and *Mode 3*. The controller is initially in the *Idle* state. When the start bit is activated it will move forward to state *Modes 0, 1, 2,* or *Mode 3* according to the Mode set on the Control register. In *Modes 0, 1, 2*, every configuration word received from the bus is forwarded to the proper destination: bitstream memory for Modes 0 and 1, and the ICAP for Modes 1 and 2. In Mode 3, the controller reads the configuration stored in the bitstream memory included in the reconfiguration controller, and forwards it to the ICAP. In all the cases a counter is used to know how many words have been processed and to update the next bitstream memory address to be read. When the counter reaches the number of words requested the *Done* bit is activated.

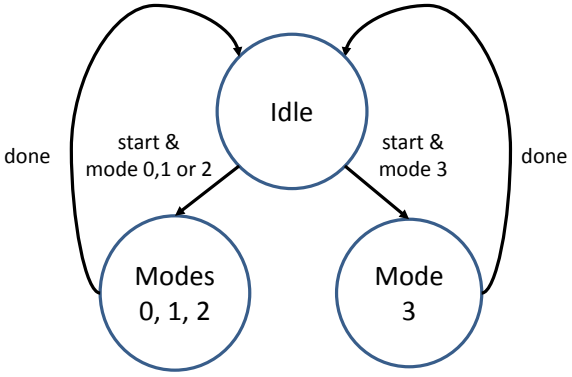

Fig. 4. Multi-Mode ICAP Finite State Machine

The implementation of this control unit requires a 2-bit register to store the current state, a counter that keeps track of how many words have been received, an adder to update the bitstream memory address, and a comparator to know when all the words have been processed. We have also included a bit swapper module. The reason is that Xilinx Plan Ahead, which is the tool that we used to generate the bitstreams for run-time reconfiguration, does not generate the configuration data in the same bit order required by the ICAP [21]. Hence, the bit swapper module reorders each configuration word by swapping the bits within each byte, i.e. from ($b_{31}..b_{24}$, $b_{23}..b_{16}$, $b_{15}..b_8$, $b_7..b_0$) to ($b_{24}..b_{31}$, $b_{16}..b_{23}$, $b_8..b_{15}$, $b_0..b_7$). This step might be instead done in software once a configuration bitstream is generated, and then this module could be removed. We decided to do it in hardware because swapping bits do not introduce any hardware overhead and because it simplifies the use of the controller.

The available on-chip RAM varies a lot depending on the FPGA used. The current trend in computer architecture is to include more and more on-chip memory resources. In fact, Xilinx has recently released a new family of FPGAs, called UltraScale+™, which brings a breakthrough including up to 65,913 Mbits of on-chip RAM [31], making it feasible to store several configurations even for large reconfigurable regions. However, small FPGAs have less than 1 Mbit. Hence, the benefits of including a bitstream memory embedded in our controller depend on the availability of on-chip memory and the size of the reconfigurable regions. If there are enough on-chip memory resources to store a significant part of the needed configurations our controller will help to optimize the reconfiguration process. At this point, other orthogonal techniques such as compression would play an important role as well. Adding compression support to our controller is as simple as including a decompression module before the bit swapper.

In our controller the size of the bitstream memory is a generic parameter that can be instantiated with different values. In our implementation we can use up to 256 KB. Typically, the FPGA includes more RAM resources, but they are used to implement other elements of the system. This size can be used to store the configuration of a reconfigurable region of 8000 LUTs. If this is not enough, the configuration can be partially stored in the bitstream memory, and later it can be carried out by combining modes 2 and 3. This is a key feature to maximize the use of the on-chip memory.

3.3 Configuration Prefetching and Caching support

Our ICAP controller has been designed to provide support both for configuration prefetching and configuration caching. As explained in Section 1, prefetching and caching have been proved to be powerful techniques to reduce the reconfiguration latency. These techniques are usually applied by loading and storing the configurations in different reconfigurable regions.

The idea behind configuration prefetching, as applied in the articles [8-13], is to carry out the reconfigurations in advance. For instance, while a task is executed on a reconfigurable region, we may prefetch the following task by storing it in another reconfigurable region.

Configuration caching in FPGAs consists in storing some selected configurations in idle reconfigurable regions. Hence, when these tasks need to be executed, they are already loaded and no reconfiguration is needed (this is called a configuration cache hit). Some examples are [17-19].

However, the application of these techniques with the reconfigurable regions of an FPGA presents a strong limitation: it is only feasible in systems where some of the reconfigurable regions are idle. Otherwise, it is not possible to prefetch or to cache new configurations.

The on-chip memory included in our controller to store configurations can be leveraged to overcome this limitation since, if all the reconfigurable regions are busy, or if the system includes only one reconfigurable region, it is still possible to apply these techniques using our bitstream memory to store the configurations. Evidently, the benefits are different: if a configuration is stored in a reconfigurable region it can be used when required without carrying out a reconfiguration, whereas if stored in the bitstream memory of our controller, the reconfiguration is still necessary, but it can be carried out at full speed.

Section 4.3 will present a case study that illustrates the benefits of using our bitstream memory for configuration caching and prefetching in a system with only one reconfigurable region.

4. EXPERIMENTAL RESULTS

4.1 Latency

We have measured the reconfiguration latency for both Xilinx XPS_HWICAP controller and our controller retrieving configuration data from three different memories: a non-volatile off-chip memory (Flash), a volatile off-chip memory (DDR2), and a volatile on-chip memory (BRAM). Figure 5 depicts the normalized reconfiguration latencies for all the evaluations. Notice that the results are represented in logarithmic scale. The red line points out the minimum latency according to the ICAP bandwidth.

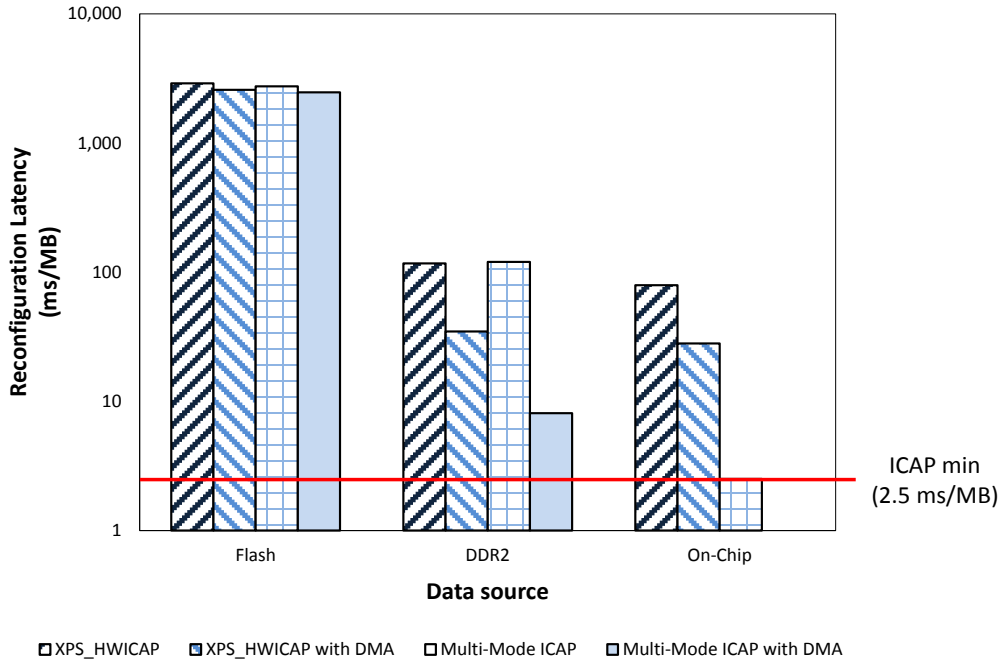

Fig. 5. Reconfiguration latencies

With the XPS_HWICAP controller, reconfigurations from Flash take 2,900 ms/MB. In the case of data transferences from the Flash memory, the inclusion of a DMA controller does not reduce significantly the latency because the Flash controller limits transferences to only 2 bytes [32]. If data are read from DDR2, the latency decreases to 117 ms/MB without DMA and 34.7 ms/MB with DMA, still far from taking fully advantage of the reconfiguration port bandwidth. Finally, retrieving the configurations from on-chip memory is the fastest choice, with a latency of 79 ms/MB without DMA, and 28 ms/MB with DMA.

With our controller, both the Flash and the DDR2 alternatives got similar latencies than the XPS_HWICAP controller since in these cases the controller is not the bottleneck. However, the on-chip reconfiguration is clearly faster. The reason is that our on-chip memory is embedded in the controller. Hence, no accesses to the system bus are needed. For this reason, our controller achieves the minimum reconfiguration latency (one write operation per cycle at 100MHz, i.e. 400 MB/s), which is 31 times faster than the XPS_HWICAP. It is important to remark that, as explained in the introduction, other on-chip controllers [26-27] can achieve this performance. In fact they report even better performance by overclocking the ICAP. Our controller could also take advantage of overclocking, since it can be clocked at 500 MHz, but, as explained before, Xilinx does not guarantee proper operation for frequencies above 100 MHz in this FPGA [28].

We have also evaluated the benefits of including a DMA controller to manage the transactions from the different memories without processor intervention. As it can be seen in the Figure 5, the DMA controller reduces the latencies significantly, but even in the best case is still eight times slower than using the embedded bitstream memory. Moreover the original Xilinx functions for the XPS_HWICAP do not support DMA transactions. Hence they have to be modified by the developer.

4.2 Resources and Power Overheads

Some previous works have evaluated the power consumption during reconfiguration. In [33-34] the authors propose models to estimate the power consumption during the reconfiguration. In addition, in [33] they carry out measurements of the FPGA power consumption using a high-speed digital oscilloscope and the shunt resistor method. They use these measures to evaluate the accuracy of their model, reporting a small error of 14%. These models are interesting but only focus on the FPGA power consumption. In this work we also want to take into account the consumption of the memory elements involved in the reconfiguration process. [35] is a recent work that carries out FPGA power measurements during reconfiguration using a high-speed digital oscilloscope on a Virtex 5 FPGA. Their objective is to compare the power consumption during partial and total reconfiguration. They report a dynamic consumption of 160mW during on-chip partial reconfiguration and 220mW during total reconfiguration. Again they do not take into account the impact of the memory elements. The results presented for the on-chip reconfiguration are very similar to our measures for on-chip reconfiguration: we have measured 180mW for the dynamic power in our board whereas they have measured 160mW.

Fig. 6. Power measurement setup

Figure 6 presents the setup used to measure the power consumption. We have used a Yokogawa WT210 digital power meter, which is an accepted device by the Standard Performance Evaluation Corporation (SPEC) for power efficiency benchmarking [36]. As explained before, our objective is to measure the contribution of all the components involved in the reconfiguration process, and not only the FPGA. The power supply of the evaluation board is connected to the power meter, therefore the power measures include the consumption of the evaluation board and the power

adapter. The power meter samples both voltage and current at a frequency of 100 KHz.

To measure the static power consumption of each component, we first measured the power consumption of the system with all the components in an idle state, and then we removed them one by one and repeated the power measures. The difference between these sequent measures corresponds to the static power of each component removed. Ambient temperature was constant in all the measures. In all the cases we measured the power consumption for two minutes, and then computed the average value.

Regarding the dynamic power consumption, we implemented a loop that reconfigures thousands of times for a total time in the order of minutes. Finally, we take the average power consumption for each reconfiguration scheme.

Table I summarizes the FPGA resources and the static power overheads of each component. Although these data may change from one FPGA to another, we believe that it provides an interesting hint for any designer in order to decide whether to include or not any component based on the overheads, the resources available, the power budget, and the system requirements. For instance, in systems with a very constrained power budget, avoiding the use of the DDR2 memory, if possible, will significantly help to meet this power budget.

Table I. Resources used and Static Power

Component	LUTs	FFs	BRAMs	Static power (W)
DDR2 controller	2531	3693	5	3.54
Flash controller	102	213	0	0.32
Interrupt controller	85	121	0	0.02
DMA controller	695	562	0	0.64
Multi-Mode ICAP (128KB)	439	355	32	0.28
Multi-Mode ICAP (256KB)	446	357	64	0.45
XPS_HWICAP	717	702	2	0.16

LUTs and FFs required by the Multi-Mode ICAP controller keep almost constant and very low regardless its storage capacity (the small variations are due to the additional addressing bits required for higher memory capacities), and the static power consumption linearly increases with the memory size by a factor of 1.6.

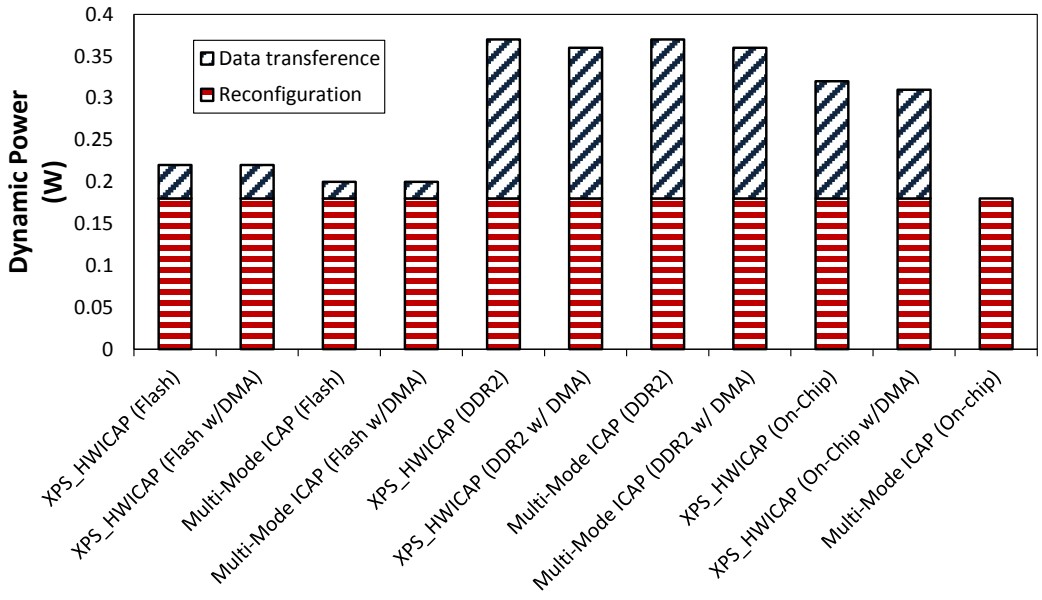

Fig. 7. Dynamic Power

Figure 7 depicts the dynamic power consumption due to the reconfigurations, both for our controller and for the XPS_HWICAP. Each bar is divided into two terms: 'Data Transference', which includes the dynamic power consumption due to data movement from the external memories to the reconfiguration controller, and 'Reconfiguration', which includes the power consumption of the reconfiguration controller and the FPGA reconfiguration circuitry.

We took two different measures in order to split the power consumption between reconfiguration and data transference. In the first one, we carried out the reconfiguration, which involves reading the configuration from a given memory and loading it onto the FPGA. In the other one, we used the mode 0 of our reconfiguration controller. In this mode, the configuration bitstream is transferred from an external memory to the internal memory of our controller, but it is not loaded onto the FPGA. Therefore, the difference between these two measures is due to the power term corresponding to the reconfiguration. In both cases, we repeated the process inside a loop for several minutes in order to minimize noise.

As it can be seen in the figure, the term 'Reconfiguration' keeps almost constant about 180mW in all the setups. On the contrary, the 'Data Transference' term greatly varies depending on the case. Dynamic power is much lower in the setups retrieving data from the Flash memory than in setups accessing to the DDR2. In the case of the on-chip storage, this term is null when using our controller because all the process is carried out inside the controller; whereas if the XPS_HWICAP controller is used instead, despite being also on-chip, data have to go through the bus.

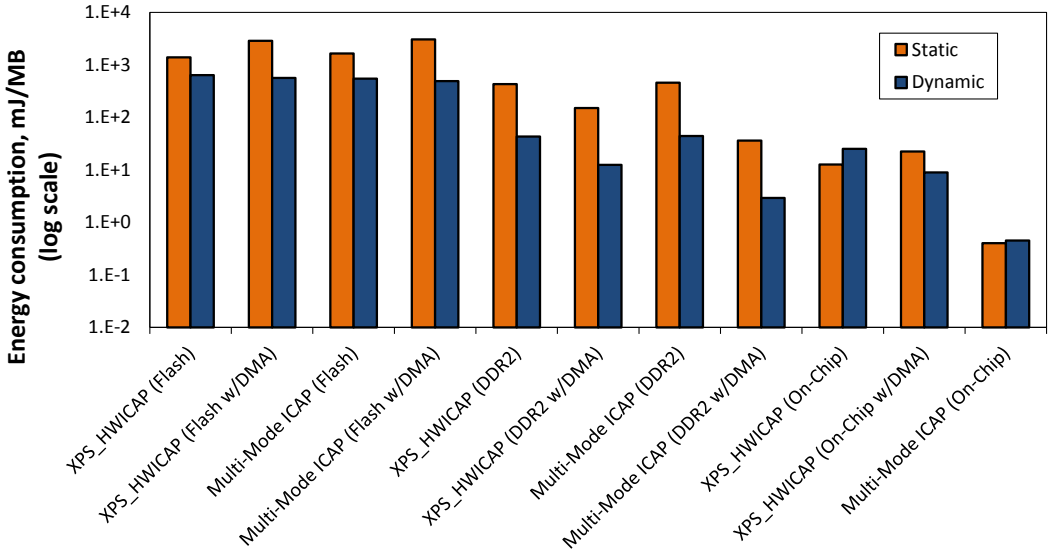

Fig. 8. Energy Consumption

Figure 8 shows the energy consumed, in mJ per MB reconfigured, in the setups evaluated in Figure 7. When configurations are retrieved from the Flash memory, the results (taking into account both static and dynamic energy) are worse. Despite being less power-hungry than the DDR2-based setups, the large latency of this memory (see Figure 5) causes a strong energy penalty. On the contrary, setups that use an on-chip memory are the most energy efficient ones. It is remarkable the fact that our Multi-Mode ICAP controller requires two orders of magnitude less energy than the XPS_HWICAP (on-chip). In fact, the energy required by this setup is, at least, one order or magnitude lower than any other setup. Hence, a bitstream memory embedded in the reconfiguration controller drastically reduces the reconfiguration energy overheads.

4.3 Configuration Prefetching and Caching

Our Multi-Mode controller has been designed to provide an additional level for configuration caching and prefetching. Using the bitstream memory, we can apply these techniques even when all the reconfigurable regions are busy. In this section, we present a case study to illustrate the benefits of this approach.

We have selected a 3D rendering application based on the open source Pocket-GL library (Pocket GL). We do not have the code of the application but a dynamic trace obtained at IMEC R&D [37]. They characterized the dynamic events of a Pocket-GL application that were adapting its execution to its input data. For each possible run-time scenario, they obtained the sequence of configurations that must be executed. This is a very interesting test since it includes 10 different dynamic tasks executed in 20 different sequences (task-graphs). We analyzed the same application in a previous article that presented a run-time scheduler for reconfigurable systems [13]. This scheduler included support both for configuration prefetching and caching taking advantage of idle reconfigurable regions to reduce the reconfiguration overhead. However, since these are sequential graphs, the system designer may decide that only one reconfigurable region is needed to execute them. In that case there would be no idle regions available. Hence, we have extended that previous

approach to apply prefetching and caching by using the bitstream memory of our controller.

The results are depicted in Figure 9. The leftmost column shows the initial overhead when the reconfigurations are carried out on demand and without applying neither prefetching nor caching. We assume that the configurations are stored in the DDR2 memory, and that the system includes a DMA controller. In this case the application needs 70% more time due to the reconfigurations. If we use the bitstream memory to apply a prefetch approach we can reduce that overhead to 38%. The approach implemented is very simple: while executing one task in the reconfigurable region the following one is stored in the bitstream memory, totally or partially, depending on the available time. When the first task finishes, we have to carry out a reconfiguration to load the following task. During this reconfiguration the portion stored in the bitstream memory can be loaded 3.2 times faster than the remaining part stored in the external memory.

These results can be further improved by caching some critical tasks. The idea is to store in the bitstream memory the configuration of those tasks that generate the largest reconfiguration overheads. In the figure we can see that when the configuration of the most critical task is cached the overhead is reduced to 27%, and if the two most critical tasks are cached, it is 22%. In the rightmost column we can see that caching the remaining 8 tasks provides no further reductions. The reason is that the prefetch technique is already reducing the reconfiguration latency of those tasks. In fact when the two most critical tasks are cached the reconfiguration overhead is reduced by a factor of 3.2, which is the best result that can be achieved in this scenario. Hence, if the bitstream memory provides enough space to store three configurations (the two cached ones plus the one that is prefetched each time) the system will provide the same performance with a system that stores all the configurations on-chip.

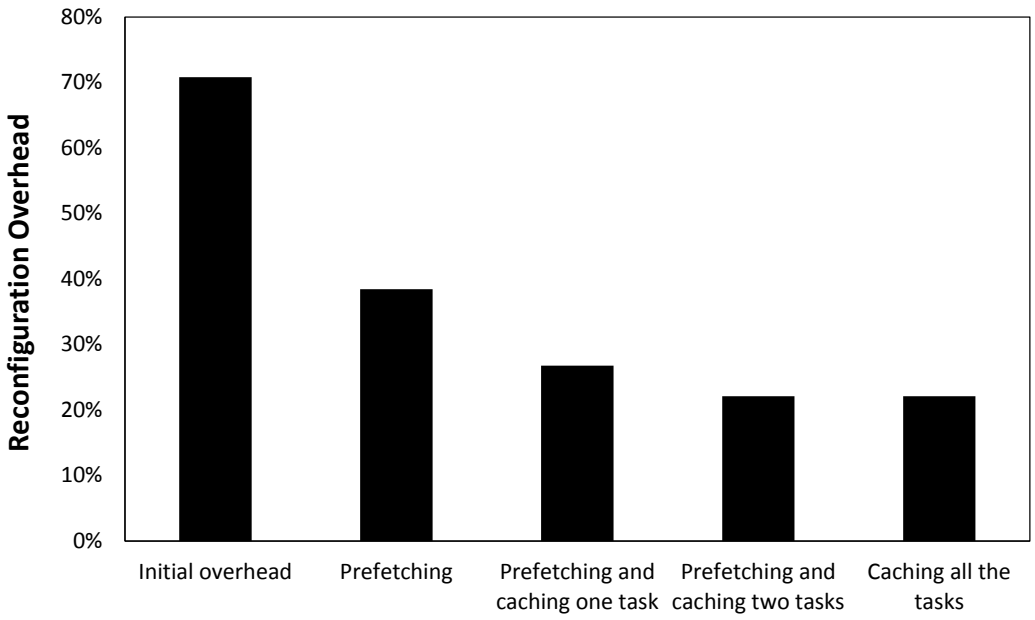

Fig. 9. Reconfiguration overheads for the Pocket GL application when using the bitstream memory to apply configuration prefetching and caching

5. CONCLUSIONS

Configurations in reconfigurable devices can be stored in very heterogeneous memories with different latencies and power consumption. In this paper we have analyzed the performance, energy and power tradeoffs when carrying out reconfigurations from several representative memories in a Xilinx XUPV5LX110T FPGA. This can be an interesting reference for any designer attempting to optimize the reconfigurations of a given system. In addition we have developed a simple reconfiguration controller and described its implementation in detail. This controller includes an internal memory to store, totally or partially, the bitstreams. With this approach it carries out the reconfigurations at the maximum speed supported by the ICAP and reduces both the reconfiguration latency and energy consumption. In addition, it provides support for prefetching and caching techniques that can further reduce the reconfiguration overheads.

ACKNOWLEDGEMENTS

This work was supported in part by grants, TIN2013-46957-C2-1-P (Spanish Gov. and European ERDF), Consolider NoE TIN2014-52608-REDC (Spanish Gov.), gaZ: T48 research group (Aragón Gov. and European ESF).

REFERENCES

[1] T. Becker, W. Luk, P.Y.K. Cheung, "Energy-Aware Optimization for Run-time Reconfiguration", Field-Programmable Custom Computing Machines, pp. 55-62, 2010
[2] S. Liu, R. N. Pittman, A. Forin, J. Gaudiot, "Achieving Energy Efficiency Through Runtime Partial Reconfiguration on Reconfigurable Systems", ACM Trans. on Embedded Computing Systems, vol. 12, no. 3, Mar 2013
[3] K. Paulsson, M. Hübner, and J. Becker, "Dynamic power optimization by exploiting self-reconfiguration in Xilinx Spartan 3-based systems", Microprocessors and Microsystems, vol. 33, no. 1, pp. 46-52, Feb 2009
[4] D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas, and M. Jones, "Evaluation of rapid context switching on a CSRC device," in in Proceedings of the International Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA), 2002, pp. 209–215
[5] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in Proceedings of the conference on Design, automation and test in Europe (DATE), 2001, pp. 642–649.
[6] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE transactions on computers 49 (5), 465-481
[7] P. Beeck , F. Barat , M. Jayapala , and R. Lauwereins, "Crisp: A template for reconfigurable instruction set processors", International conference on Field Programmable Logic (FPL 2002), pp. 296-305
[8] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA), 2002, pp. 187–195.
[9] J. Noguera, and R. M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling," ACM Transactions on Embedded Computing Systems, vol. 3, no. 2, pp. 385–406, May 2004.
[10] Y. Qu, J. Pekka Soininen, and J. Nurmi, "A parallel configuration model for reducing the run-time reconfiguration overhead," in IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE), 2006, pp. 965–970.
[11] J. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich, "Interprocedural placement-aware configuration prefetching for FPGA-based systems," in IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), may 2010, pp. 179–182.
[12] W. Fu and K. Compton, "Scheduling intervals for reconfigurable computing," in Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008, pp. 87–96
[13] J. Clemente, J. Resano, C. Gonzalez, and D. Mozos, "A hardware implementation of a run-time scheduler for reconfigurable systems," IEEE Transactions on Very Large Scale Integration (VLSI)

Systems, vol. 19, no. 7, pp. 1263–1276, July 2011.
[14]Z. Li and S. Hauck, “Configuration compression for virtex FPGAs,” in Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, pp. 147–159.
[15]A. Dandalis, and V. K. Prasanna, “Configuration compression for FPGA-based embedded systems,” in Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA), New York, NY, USA, 2001, pp. 173–182
[16]J.A., Clemente, E. Perez Ramo, J. Resano, D. Mozos, and F. Catthoor, "Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.22, no.6, pp.1248,1261, June 2014.
[17]Z. Li, K. Compton, and S. Hauck, “Configuration caching management techniques for reconfigurable computing,” in Proceedings of the annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 22–36
[18]R. Kalra, and R. Lysecky, “Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 4, pp. 671–674, Apr. 2010.
[19]J. A. Clemente, J. Resano, and D. Mozos, “An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems”, ACM Transactions on Embedded Computing Systems, vol. 13 Issue 4, article 90, November 2014.
[20]K. Papadimitriou, A. Dollas, and S. Hauck, “Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," ACM Transactions on Reconfigurable Technology Systems, vol. 4, no. 4, pp. 36:1-36:24, December 2011
[21]K. Papadimitriou, A. Anyfantis, and A. Dollas, “An effective framework to evaluate dynamic partial reconfiguration in FPGA systems,” IEEE Transactions on Instrumentation and Measurement, vol. 59, no. 6, pp.1642–1651, Jun. 2010
[22]E. El-Araby, I. Gonzalez, and T. El-Ghazawi, “Exploiting partial runtime reconfiguration for high-performance reconfigurable computing,” ACM Transactions on Reconfigurable Technology Systems, vol. 1, no. 4, pp.21:1–21:23, Jan. 2009.
[23]K. Vipin, S. A. Fahmy, “A High Speed Open Source Controller for FPGA Partial Reconfiguration”, in Proceedings of the International Conference on Field Programmable Technology (FPT), Seoul, Korea, December 2012, pp. 61–66
[24]S. Liu, N. Pittman, and A. Forin, “Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller”, Microsoft Research 2009
[25]L. Ming, W. Kuehn, L. Zhonghai, and A. Jantsch, "Run-time Partial Reconfiguration speed investigation and architectural design space exploration," Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on , vol., no., pp.498-502, Aug. 31 2009-Sept. 2 2009
[26]R. Bonamy, P. Hung-Manh, S. Pillement, D. Chillet, "UPaRC—Ultra-fast power-aware reconfiguration controller," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012 , vol., no., pp.1373-1378, 12-16 March 2012
[27]S. Gimle, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," IEEE International Parallel & Distributed Processing Symposium, 2011, pp.174-180, 16-20 May 2011
[28]*Virtex-5* FPGA *Configuration Guide* UG191 (v3.11) October 19, 2012. www.xilinx.com/support/documentation/user_guides/ug191.pdf
[29]Xilinx DS86 LogiCORE IP XPS HWICAP. June, 2011. https://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap/v5_01_a/xps_hwicap.pdf
[30]Xilinx DS531 Processor Local Bus (PLB) v4.6 (v1.05a). Sept 2010. http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf
[31]Virtex UltraScale+ Product Table
https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable
[32]XPS SYSACE (System ACE) Interface Controller (v1.01a) December 2, 2009.
www.xilinx.com/support/documentation/ip_documentation/xps_sysace.pdf
[33] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys, “Power consumption model for partial and dynamic reconfiguration”, International Conference on Reconfigurable Computing and FPGAs, 2012
[34] P. Stenström, “Transactions on High-Performance Embedded Architectures and Compilers IV”, Lecture Notes in Computer Science, vol. 6760, 2011
[35]A. Nafkha, and Y. Louet, “Accurate Measurement of Power Consumption Overhead During FPGA Dynamic Partial Reconfiguration”, International Symposium on Wireless Communication Systems (ISWCS), 2016
[36]Yokogawa WT210/WT230 Digital Power Meters.
http://tmi.yokogawa.com/discontinued-products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-power-meters/
[37] Interuniversitair Micro-Electronica Centrum.
https://www.imec-int.com/

Analysis of the reconfiguration latency and energy overheads for a Xilinx Virtex-5 FPGA

JAVIER OLIVITO, University of Zaragoza, 0034876555081, jolivito@unizar.es
FELIPE SERRANO, Complutense University of Madrid
JUAN ANTONIO CLEMENTE, Complutense University of Madrid
HORTENSIA MECHA, Complutense University of Madrid
JAVIER RESANO, University of Zaragoza

In this paper we have evaluated the overhead and the tradeoffs of a set of components usually included in a system with run-time partial reconfiguration implemented on a Xilinx Virtex-5. Our analysis shows the benefits of including a scratchpad memory inside the reconfiguration controller in order to improve the efficiency of the reconfiguration process. We have designed a simple controller for this scratchpad that includes support for prefetching and caching in order to further reduce both the energy and latency overhead.

1. INTRODUCTION AND RELATED WORK

Dynamic partial reconfiguration (DPR) is one of the most interesting features of FPGAs. Reconfiguration enables the reuse of the FPGA hardware resources for different tasks that can be loaded at run-time according to the system needs. Hence, FPGAs can be used to develop flexible platforms that can adapt themselves to the execution of different applications. DPR has been thoroughly explored as a way to reduce area and power in some systems [1-3]. The idea is to reuse the reconfigurable area for different tasks that are not executed simultaneously. However, the reconfiguration process itself introduces overheads both in the execution time and in the energy consumption. The reason is that it involves not only using the reconfiguration circuitry to update the device configuration, but also moving large data sets from the memory where the configurations are stored to the reconfiguration port.

The analysis of the reconfiguration overheads has been the target of many research groups, and they have focused on execution-time overheads. Several works have proposed alternative reconfigurable architectures. For instance, multi-context FPGAs [4] enable the load of a new configuration while another one is being executed. Another example is coarse-grain architectures [5], which offer less flexibility but require smaller configuration bitstreams that can be easily stored on-chip in order to further increase the reconfiguration speed [6-7]. We believe that the techniques that we expose in this article can be also applied on these architectures, but we have focused on single-context FPGAs because they dominate the reconfiguration landscape.

Another approach to reduce the reconfiguration overheads is to apply scheduling techniques that attempt to hide the reconfiguration latency by fetching the configurations in advance and storing them in idle reconfigurable regions. This is a powerful technique for embedded systems where the task execution order is known because these techniques use the task-graph information to prefetch the configurations that will be needed in the near future. Some relevant works that propose reconfiguration scheduling techniques are [8-13]. It is important to mention that the controller described in this article is compatible with any of these techniques

and, in fact, it can improve their results since even when all the reconfigurable regions are busy, the configurations can be still prefetched to the internal memory included in our controller. This is explained in detail in Section 3.3.

Another powerful way to reduce the reconfiguration time overhead is to reduce its size by applying compression techniques. With this approach, configurations are fetched faster but they have to be decompressed before writing them in the device, and this may involve additional execution time and energy penalties. However, these overheads can be smaller if the system includes hardware support to decompress the configurations. Some relevant works on compression are [14] and [15]. Again, these techniques are fully compatible with our work since our controller could easily incorporate a decompression module.

Another option to reduce the reconfiguration overhead is to customize the memory resources used to store the configurations. For instance, [16] proposes to include heterogeneous on-chip memory modules, ones optimized for performance and others optimized for low power. With this approach the designer can explore different power/performance reconfiguration tradeoffs. This idea is orthogonal with our approach and could be included in our controller.

Another interesting technique is configuration caching. The idea is to store different configurations in different reconfigurable regions in the device and design specific replacement techniques to maximize the configuration reuse. Some interesting configuration caching techniques are [17-19]. Our controller has been designed to support configuration caching, and, in fact, its internal memories can be used to apply these techniques in an additional level. This is further explained in Section 3.3.

Regarding the reconfiguration process, many works assume that the reconfiguration latency is a fixed time that can be calculated by dividing the size of the configuration by the peak bandwidth of the reconfiguration port. In fact, the peak reconfiguration bandwidth is usually the only value provided by FPGA vendors. However, is that value relevant? Can we really achieve that bandwidth? How can we do it? In [20], Papadimitriou *et al.* elaborated a theoretical cost model to estimate the reconfiguration time at an early stage of the development. The authors estimated the reconfiguration time when fetching the bitstreams from external memories. They verified the model with measures over a real system implemented on a Virtex-II Pro FPGA and reported a model difference of ~25%, which is due to different clocks of processors, i.e. 300 MHz for the calculated one and 100 MHz for the measured one. Several works have pointed out that the actual reconfiguration latencies obtained in representative case studies are one order of magnitude, or even two, worse than the peak reconfiguration latency [21-22]. The reason for these poor results is that configurations are usually stored in off-chip non-volatile memories, and the actual bandwidth of these memories is much smaller than that of the reconfiguration port. Hence, the bottleneck is not the reconfiguration port, but the bandwidth of the external memories. Some previous works have demonstrated that it is possible to achieve almost peak-performance when using some specific external memories with additional support. For instance, in [23] the authors claim to achieve almost peak performance in a Virtex-6 when using DDR3 memory, a Direct Memory Access (DMA) controller, and some additional FIFOs to hide the latency, and in [24], they retrieve configurations from an external SRAM through a customized DMA. In [23], they also state that it is possible to achieve a speedup of 2 by overclocking the

Internal Configuration Access Port (ICAP). However, although each new generation of external memories is providing more bandwidth, it is likely that the speed of the reconfiguration port will also scale accordingly. In fact, Virtex UltraScale+™ FPGAs can be reconfigured at 200MHz (previous generations were limited to 100 MHz). Moreover, in many systems the energy overhead is even more relevant than performance, and the use of external memories strongly penalizes in power and energy.

Some previous works propose the inclusion of memory resources inside the reconfiguration controller in order to preload configurations. In [25], the authors analyze the reconfiguration latency of a system implemented in a Virtex-4 FPGA that includes a system bus that is used for all the data transfers, including those needed to carry out a reconfiguration, i.e. reading the configuration and sending it to the reconfiguration port. They measured the reconfiguration latencies taking into account the different access schemes provided by the bus, and including a DMA controller. The results demonstrate the impact of the communication scheme in the reconfiguration latency. In their analysis, they claim that the only way to achieve the peak reconfiguration bandwidth is to avoid accesses to the system bus. Even when the system bus is available and transfers are performed through a DMA, these accesses introduce significant delays.

Other interesting works are [26] and [27]. In these articles, the authors propose partial reconfiguration controllers that reach a throughput of 1.433GB/s and 2.2GB/s respectively. These controllers allocate on-chip memory resources to store de configurations, and the ICAP is overclocked up to 550 MHz. Although the manufacturer does not guarantee proper operation at frequencies higher than 100 MHz, these works are a proof of concept of the use of internal memory for future and faster versions of the ICAP.

After analyzing the aforementioned works, it is clear that the reconfiguration latency drastically depends on where the configurations are stored and the communication scheme used to read them. The first contribution of this work is the analysis of how these two parameters affect the reconfiguration latency in a Virtex-5 FPGA. We made the evaluation on the XUPV5-LX110T Development System included in the Xilinx University Program. This evaluation board includes many different memories, and we selected two external memories, Flash and DDR2 DRAM, and the internal on-chip SRAM.

Other key metrics, especially in embedded systems, are power and energy. Hence, as a second contribution, we measured the energy demanded in a reconfiguration, depending on the kind of memory where the configurations are stored. We also measured the power overhead of some components providing support for the process, such as the controllers for the memories, the reconfiguration, or the DMA. This analysis of the energy overhead in the reconfiguration may be helpful for the developers in order to decide where to store the configurations.

Finally, as a third contribution, we have developed a simple and efficient reconfiguration controller that, as in [26] and [27], includes an embedded memory that can be used to store bitstreams. Using that memory the reconfiguration process can be carried out at the maximum speed supported by the ICAP while minimizing the energy consumption. As a novelty, this controller includes two additional working modes in order to provide support for configuration prefetching and caching using the memory embedded in the controller. These modes can be used to reduce the reconfiguration overheads when only some of the configurations can be stored on-chip.

2. TARGET ARCHITECTURE

Our target architecture consists of three different memories that can be used to store configuration bitstreams, a reconfiguration controller that provides an interface with the ICAP [28], a processor, and at least one Reconfigurable Region (RR). Figure 1 depicts the elements of this architecture. In our experiments, the processor is a Xilinx MicroBlaze, the system bus is a PLB 4.6, and the memories are a 1GB Compact Flash, a 64-bit wide 256MB DDR2 SODIMM, and the FPGA internal BlockRAMs (BRAMs). Additionally, DMA and interrupt controllers supplied by Xilinx were added in order to evaluate their performance-energy tradeoffs.

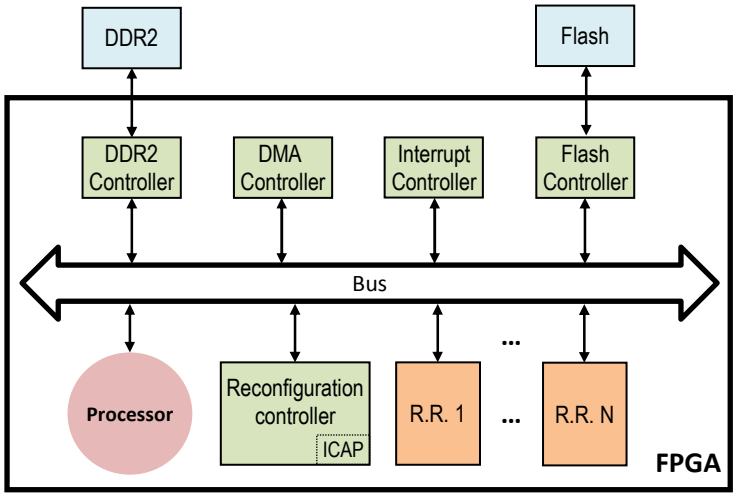

Fig. 1. Target architecture

The selection of the memories where configurations will be stored, and how to move these configurations among these memories and the reconfiguration controller is not straightforward, as it depends on multiple factors such as the power budget, the performance requirements, and the needs of other system components (for example, if the DDR or the Flash memories are demanded by other tasks or components, they would be already present in the system and then the static power of these controllers will not introduce an additional penalty related to DPR). In the experimental results presented in Section 4, we analyze the reconfiguration energy-performance tradeoff in order to help designers to make their decisions.

3. PARTIAL RECONFIGURATION CONTROLLER

3.1 Xilinx IP

Xilinx provides the XPS_HWICAP IP core for the Virtex-5 FPGA to manage partial reconfigurations in processor-based systems [29]. It is composed of several control registers, two small FIFOs, a finite-state machine (FSM), and a PLB bus interface [30]. Xilinx also provides a driver to use this controller from the processor-side. This driver enables to read configurations from any memory in the system and to send the data to the XPS_HWICAP controller through the PLB bus. This controller is easy to use and it should be the starting point for anybody who wants to carry out reconfigurations. However, this driver has not been designed to optimize the data transfers among the memories and the XPS_HWICAP. As a result, as it will be

explained in section 4, it only achieves a reconfiguration throughput of 12 MB/s, far from the peak 400 MB/s supported by the ICAP.

3.2 Multi-Mode ICAP Controller

In order to reach the maximum reconfiguration throughput, the ICAP should receive one 32-bit word per cycle at a 100MHz rate. As explained in the previous section, it is hard to achieve this throughput when configurations are stored in off-chip memories, or even when they are stored on-chip but they are accessed through a shared system bus. A solution is to store them inside the configuration controller as it was previously proposed in [25-27]. With this approach the reconfigurations can be carried out at full speed. In current FPGA architectures, this can be achieved by reserving part of the on-chip RAM for the controller. We refer to this on-chip RAM as bitstream memory.

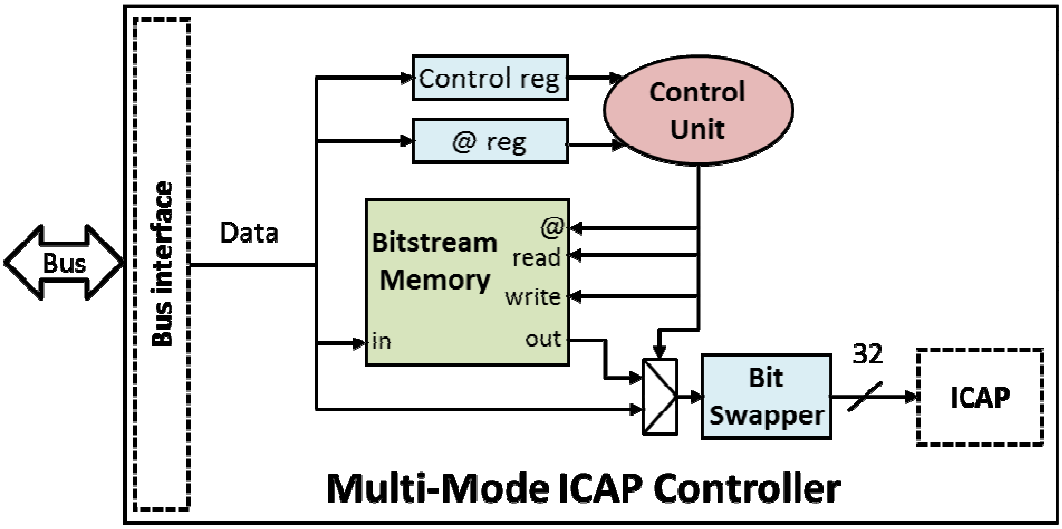

Fig. 2. Multi-Mode ICAP reconfiguration controller

Figure 2 illustrates the architecture of our partial reconfiguration controller, called Multi-Mode ICAP. The bus interface was automatically generated by the Xilinx EDK tool, and the remaining blocks were designed in VHDL and integrated inside the EDK project. The Control and the Address registers are software accessible. Hence the processor can easily provide the information and check if the controller has finished. If the system includes an interrupt controller, the controller can generate an interrupt when the *Done* bit is activated; again this is straightforward by using the EDK tools. This architecture based on an internal memory, a register-based interface, and a control unit that is very similar to those proposed in [25] and [26]. The main difference is that our control unit provides support for several useful working modes.

The Multi-Mode ICAP supports four different working modes:

a) *Mode 0: Configuration Load*. The controller receives a configuration and stores it in the bitstream memory. The controller does not send the configuration to the ICAP. This mode is useful to fetch configurations in advance from the external memories in order to reduce the

reconfiguration latency since, once a configuration is stored, our controller sends it to the ICAP at the maximum supported speed.

b) *Mode 1: External reconfiguration and configuration load*. The controller receives a configuration and forwards it to the ICAP. In parallel, it stores the configuration in the internal bitstream memory. In this mode the reconfiguration speed depends on how fast the configuration data are received from the bus, i.e. our controller does not reduce the reconfiguration latency in this mode. However, if the same configuration is required again, it can be loaded from the internal bitstream memory at the maximum speed.

c) *Mode 2: External reconfiguration*. The controller receives a configuration and forwards it to the ICAP as in the previous case, but in this mode the configuration is not stored in the bitstream memory.

d) *Mode 3: Internal reconfiguration*. The controller sends to the ICAP a configuration previously stored in the bitstream memory. The reconfiguration is carried out at full speed.

In order to support this functionality, only two software accessible registers (Control and Address registers) and little additional control logic are necessary. The control register (Figure 3) is used to configure our controller. Bit 0 reports when an operation has finished, bit 1 triggers the start of an operation, and bits 2 and 3 are used to select the working mode. Finally the remaining 28 bits are used to set the size of the configuration expressed in 32-bits words. The address register specifies the initial address of the bitstream memory to store or load a specific configuration.

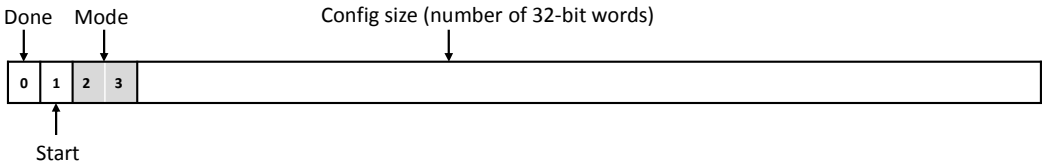

Fig. 3. Control register

The control unit (Figure 4) only requires three states: *Idle*, *Modes 0, 1, 2,* and *Mode 3*. The controller is initially in the *Idle* state. When the start bit is activated it will move forward to state *Modes 0, 1, 2,* or *Mode 3* according to the Mode set on the Control register. In *Modes 0, 1, 2*, every configuration word received from the bus is forwarded to the proper destination: bitstream memory for Modes 0 and 1, and the ICAP for Modes 1 and 2. In Mode 3, the controller reads the configuration stored in the bitstream memory included in the reconfiguration controller, and forwards it to the ICAP. In all the cases a counter is used to know how many words have been processed and to update the next bitstream memory address to be read. When the counter reaches the number of words requested the *Done* bit is activated.

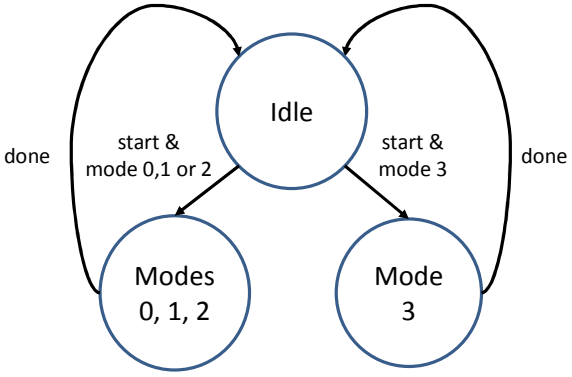

Fig. 4. Multi-Mode ICAP Finite State Machine

The implementation of this control unit requires a 2-bit register to store the current state, a counter that keeps track of how many words have been received, an adder to update the bitstream memory address, and a comparator to know when all the words have been processed. We have also included a bit swapper module. The reason is that Xilinx Plan Ahead, which is the tool that we used to generate the bitstreams for run-time reconfiguration, does not generate the configuration data in the same bit order required by the ICAP [21]. Hence, the bit swapper module reorders each configuration word by swapping the bits within each byte, i.e. from ($b_{31}..b_{24}$, $b_{23}..b_{16}$, $b_{15}..b_{8}$, $b_{7}..b_{0}$) to ($b_{24}..b_{31}$, $b_{16}..b_{23}$, $b_{8}..b_{15}$, $b_{0}..b_{7}$). This step might be instead done in software once a configuration bitstream is generated, and then this module could be removed. We decided to do it in hardware because swapping bits do not introduce any hardware overhead and because it simplifies the use of the controller.

The available on-chip RAM varies a lot depending on the FPGA used. The current trend in computer architecture is to include more and more on-chip memory resources. In fact, Xilinx has recently released a new family of FPGAs, called UltraScale+™, which brings a breakthrough including up to 65,913 Mbits of on-chip RAM [31], making it feasible to store several configurations even for large reconfigurable regions. However, small FPGAs have less than 1 Mbit. Hence, the benefits of including a bitstream memory embedded in our controller depend on the availability of on-chip memory and the size of the reconfigurable regions. If there are enough on-chip memory resources to store a significant part of the needed configurations our controller will help to optimize the reconfiguration process. At this point, other orthogonal techniques such as compression would play an important role as well. Adding compression support to our controller is as simple as including a decompression module before the bit swapper.

In our controller the size of the bitstream memory is a generic parameter that can be instantiated with different values. In our implementation we can use up to 256 KB. Typically, the FPGA includes more RAM resources, but they are used to implement other elements of the system. This size can be used to store the configuration of a reconfigurable region of 8000 LUTs. If this is not enough, the configuration can be partially stored in the bitstream memory, and later it can be carried out by combining modes 2 and 3. This is a key feature to maximize the use of the on-chip memory.

3.3 Configuration Prefetching and Caching support

Our ICAP controller has been designed to provide support both for configuration prefetching and configuration caching. As explained in Section 1, prefetching and caching have been proved to be powerful techniques to reduce the reconfiguration latency. These techniques are usually applied by loading and storing the configurations in different reconfigurable regions.

The idea behind configuration prefetching, as applied in the articles [8-13], is to carry out the reconfigurations in advance. For instance, while a task is executed on a reconfigurable region, we may prefetch the following task by storing it in another reconfigurable region.

Configuration caching in FPGAs consists in storing some selected configurations in idle reconfigurable regions. Hence, when these tasks need to be executed, they are already loaded and no reconfiguration is needed (this is called a configuration cache hit). Some examples are [17-19].

However, the application of these techniques with the reconfigurable regions of an FPGA presents a strong limitation: it is only feasible in systems where some of the reconfigurable regions are idle. Otherwise, it is not possible to prefetch or to cache new configurations.

The on-chip memory included in our controller to store configurations can be leveraged to overcome this limitation since, if all the reconfigurable regions are busy, or if the system includes only one reconfigurable region, it is still possible to apply these techniques using our bitstream memory to store the configurations. Evidently, the benefits are different: if a configuration is stored in a reconfigurable region it can be used when required without carrying out a reconfiguration, whereas if stored in the bitstream memory of our controller, the reconfiguration is still necessary, but it can be carried out at full speed.

Section 4.3 will present a case study that illustrates the benefits of using our bitstream memory for configuration caching and prefetching in a system with only one reconfigurable region.

4. EXPERIMENTAL RESULTS

4.1 Latency

We have measured the reconfiguration latency for both Xilinx XPS_HWICAP controller and our controller retrieving configuration data from three different memories: a non-volatile off-chip memory (Flash), a volatile off-chip memory (DDR2), and a volatile on-chip memory (BRAM). Figure 5 depicts the normalized reconfiguration latencies for all the evaluations. Notice that the results are represented in logarithmic scale. The red line points out the minimum latency according to the ICAP bandwidth.

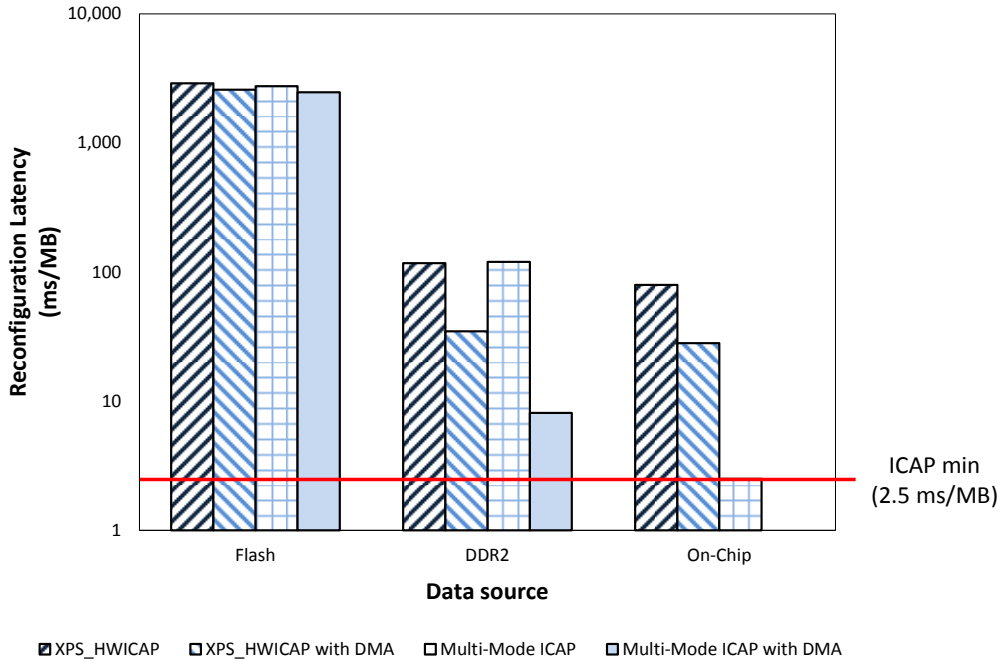

Fig. 5. Reconfiguration latencies

With the XPS_HWICAP controller, reconfigurations from Flash take 2,900 ms/MB. In the case of data transferences from the Flash memory, the inclusion of a DMA controller does not reduce significantly the latency because the Flash controller limits transferences to only 2 bytes [32]. If data are read from DDR2, the latency decreases to 117 ms/MB without DMA and 34.7 ms/MB with DMA, still far from taking fully advantage of the reconfiguration port bandwidth. Finally, retrieving the configurations from on-chip memory is the fastest choice, with a latency of 79 ms/MB without DMA, and 28 ms/MB with DMA.

With our controller, both the Flash and the DDR2 alternatives got similar latencies than the XPS_HWICAP controller since in these cases the controller is not the bottleneck. However, the on-chip reconfiguration is clearly faster. The reason is that our on-chip memory is embedded in the controller. Hence, no accesses to the system bus are needed. For this reason, our controller achieves the minimum reconfiguration latency (one write operation per cycle at 100MHz, i.e. 400 MB/s), which is 31 times faster than the XPS_HWICAP. It is important to remark that, as explained in the introduction, other on-chip controllers [26-27] can achieve this performance. In fact they report even better performance by overclocking the ICAP. Our controller could also take advantage of overclocking, since it can be clocked at 500 MHz, but, as explained before, Xilinx does not guarantee proper operation for frequencies above 100 MHz in this FPGA [28].

We have also evaluated the benefits of including a DMA controller to manage the transactions from the different memories without processor intervention. As it can be seen in the Figure 5, the DMA controller reduces the latencies significantly, but even in the best case is still eight times slower than using the embedded bitstream memory. Moreover the original Xilinx functions for the XPS_HWICAP do not support DMA transactions. Hence they have to be modified by the developer.

4.2 Resources and Power Overheads

Some previous works have evaluated the power consumption during reconfiguration. In [33-34] the authors propose models to estimate the power consumption during the reconfiguration. In addition, in [33] they carry out measurements of the FPGA power consumption using a high-speed digital oscilloscope and the shunt resistor method. They use these measures to evaluate the accuracy of their model, reporting a small error of 14%. These models are interesting but only focus on the FPGA power consumption. In this work we also want to take into account the consumption of the memory elements involved in the reconfiguration process. [35] is a recent work that carries out FPGA power measurements during reconfiguration using a high-speed digital oscilloscope on a Virtex 5 FPGA. Their objective is to compare the power consumption during partial and total reconfiguration. They report a dynamic consumption of 160mW during on-chip partial reconfiguration and 220mW during total reconfiguration. Again they do not take into account the impact of the memory elements. The results presented for the on-chip reconfiguration are very similar to our measures for on-chip reconfiguration: we have measured 180mW for the dynamic power in our board whereas they have measured 160mW.

Fig. 6. Power measurement setup

Figure 6 presents the setup used to measure the power consumption. We have used a Yokogawa WT210 digital power meter, which is an accepted device by the Standard Performance Evaluation Corporation (SPEC) for power efficiency benchmarking [36]. As explained before, our objective is to measure the contribution of all the components involved in the reconfiguration process, and not only the FPGA. The power supply of the evaluation board is connected to the power meter, therefore the power measures include the consumption of the evaluation board and the power

adapter. The power meter samples both voltage and current at a frequency of 100 KHz.

To measure the static power consumption of each component, we first measured the power consumption of the system with all the components in an idle state, and then we removed them one by one and repeated the power measures. The difference between these sequent measures corresponds to the static power of each component removed. Ambient temperature was constant in all the measures. In all the cases we measured the power consumption for two minutes, and then computed the average value.

Regarding the dynamic power consumption, we implemented a loop that reconfigures thousands of times for a total time in the order of minutes. Finally, we take the average power consumption for each reconfiguration scheme.

Table I summarizes the FPGA resources and the static power overheads of each component. Although these data may change from one FPGA to another, we believe that it provides an interesting hint for any designer in order to decide whether to include or not any component based on the overheads, the resources available, the power budget, and the system requirements. For instance, in systems with a very constrained power budget, avoiding the use of the DDR2 memory, if possible, will significantly help to meet this power budget.

Table I. Resources used and Static Power

Component	LUTs	FFs	BRAMs	Static power (W)
DDR2 controller	2531	3693	5	3.54
Flash controller	102	213	0	0.32
Interrupt controller	85	121	0	0.02
DMA controller	695	562	0	0.64
Multi-Mode ICAP (128KB)	439	355	32	0.28
Multi-Mode ICAP (256KB)	446	357	64	0.45
XPS_HWICAP	717	702	2	0.16

LUTs and FFs required by the Multi-Mode ICAP controller keep almost constant and very low regardless its storage capacity (the small variations are due to the additional addressing bits required for higher memory capacities), and the static power consumption linearly increases with the memory size by a factor of 1.6.

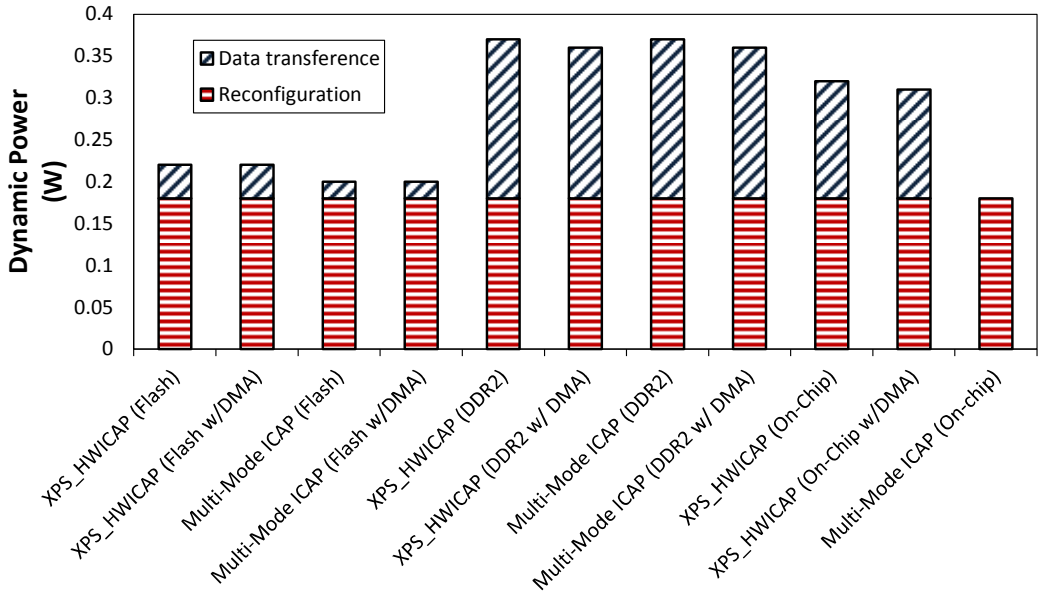

Fig. 7. Dynamic Power

Figure 7 depicts the dynamic power consumption due to the reconfigurations, both for our controller and for the XPS_HWICAP. Each bar is divided into two terms: 'Data Transference', which includes the dynamic power consumption due to data movement from the external memories to the reconfiguration controller, and 'Reconfiguration', which includes the power consumption of the reconfiguration controller and the FPGA reconfiguration circuitry.

We took two different measures in order to split the power consumption between reconfiguration and data transference. In the first one, we carried out the reconfiguration, which involves reading the configuration from a given memory and loading it onto the FPGA. In the other one, we used the mode 0 of our reconfiguration controller. In this mode, the configuration bitstream is transferred from an external memory to the internal memory of our controller, but it is not loaded onto the FPGA. Therefore, the difference between these two measures is due to the power term corresponding to the reconfiguration. In both cases, we repeated the process inside a loop for several minutes in order to minimize noise.

As it can be seen in the figure, the term 'Reconfiguration' keeps almost constant about 180mW in all the setups. On the contrary, the 'Data Transference' term greatly varies depending on the case. Dynamic power is much lower in the setups retrieving data from the Flash memory than in setups accessing to the DDR2. In the case of the on-chip storage, this term is null when using our controller because all the process is carried out inside the controller; whereas if the XPS_HWICAP controller is used instead, despite being also on-chip, data have to go through the bus.

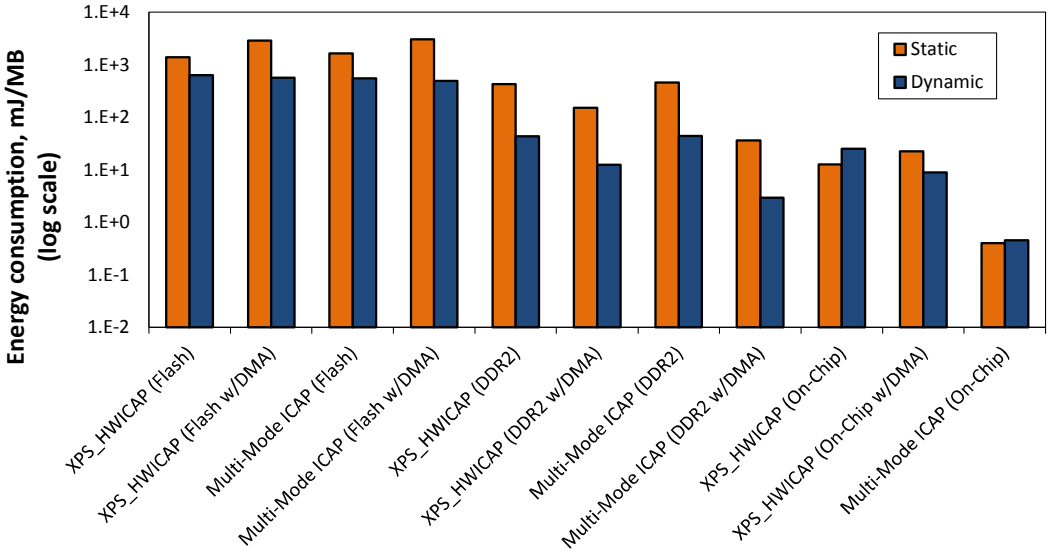

Fig. 8. Energy Consumption

Figure 8 shows the energy consumed, in mJ per MB reconfigured, in the setups evaluated in Figure 7. When configurations are retrieved from the Flash memory, the results (taking into account both static and dynamic energy) are worse. Despite being less power-hungry than the DDR2-based setups, the large latency of this memory (see Figure 5) causes a strong energy penalty. On the contrary, setups that use an on-chip memory are the most energy efficient ones. It is remarkable the fact that our Multi-Mode ICAP controller requires two orders of magnitude less energy than the XPS_HWICAP (on-chip). In fact, the energy required by this setup is, at least, one order or magnitude lower than any other setup. Hence, a bitstream memory embedded in the reconfiguration controller drastically reduces the reconfiguration energy overheads.

4.3 Configuration Prefetching and Caching

Our Multi-Mode controller has been designed to provide an additional level for configuration caching and prefetching. Using the bitstream memory, we can apply these techniques even when all the reconfigurable regions are busy. In this section, we present a case study to illustrate the benefits of this approach.

We have selected a 3D rendering application based on the open source Pocket-GL library (Pocket GL). We do not have the code of the application but a dynamic trace obtained at IMEC R&D [37]. They characterized the dynamic events of a Pocket-GL application that were adapting its execution to its input data. For each possible run-time scenario, they obtained the sequence of configurations that must be executed. This is a very interesting test since it includes 10 different dynamic tasks executed in 20 different sequences (task-graphs). We analyzed the same application in a previous article that presented a run-time scheduler for reconfigurable systems [13]. This scheduler included support both for configuration prefetching and caching taking advantage of idle reconfigurable regions to reduce the reconfiguration overhead. However, since these are sequential graphs, the system designer may decide that only one reconfigurable region is needed to execute them. In that case

there would be no idle regions available. Hence, we have extended that previous approach to apply prefetching and caching by using the bitstream memory of our controller.

The results are depicted in Figure 9. The leftmost column shows the initial overhead when the reconfigurations are carried out on demand and without applying neither prefetching nor caching. We assume that the configurations are stored in the DDR2 memory, and that the system includes a DMA controller. In this case the application needs 70% more time due to the reconfigurations. If we use the bitstream memory to apply a prefetch approach we can reduce that overhead to 38%. The approach implemented is very simple: while executing one task in the reconfigurable region the following one is stored in the bitstream memory, totally or partially, depending on the available time. When the first task finishes, we have to carry out a reconfiguration to load the following task. During this reconfiguration the portion stored in the bitstream memory can be loaded 3.2 times faster than the remaining part stored in the external memory.

These results can be further improved by caching some critical tasks. The idea is to store in the bitstream memory the configuration of those tasks that generate the largest reconfiguration overheads. In the figure we can see that when the configuration of the most critical task is cached the overhead is reduced to 27%, and if the two most critical tasks are cached, it is 22%. In the rightmost column we can see that caching the remaining 8 tasks provides no further reductions. The reason is that the prefetch technique is already reducing the reconfiguration latency of those tasks. In fact when the two most critical tasks are cached the reconfiguration overhead is reduced by a factor of 3.2, which is the best result that can be achieved in this scenario. Hence, if the bitstream memory provides enough space to store three configurations (the two cached ones plus the one that is prefetched each time) the system will provide the same performance with a system that stores all the configurations on-chip.

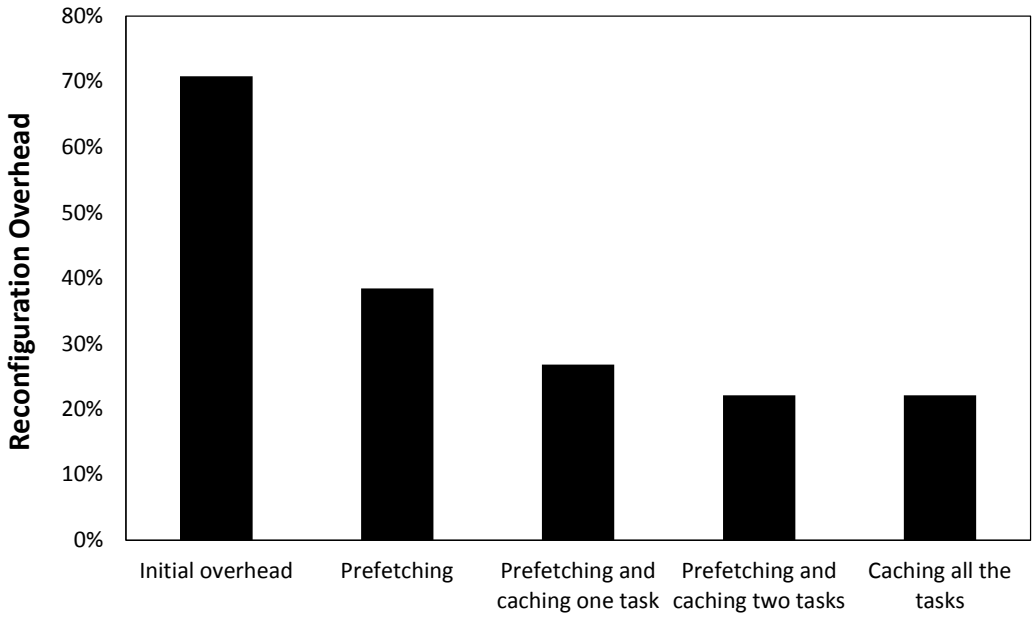

Fig. 9. Reconfiguration overheads for the Pocket GL application when using the bitstream memory to apply configuration prefetching and caching

5. CONCLUSIONS

Configurations in reconfigurable devices can be stored in very heterogeneous memories with different latencies and power consumption. In this paper we have analyzed the performance, energy and power tradeoffs when carrying out reconfigurations from several representative memories in a Xilinx XUPV5LX110T FPGA. This can be an interesting reference for any designer attempting to optimize the reconfigurations of a given system. In addition we have developed a simple reconfiguration controller and described its implementation in detail. This controller includes an internal memory to store, totally or partially, the bitstreams. With this approach it carries out the reconfigurations at the maximum speed supported by the ICAP and reduces both the reconfiguration latency and energy consumption. In addition, it provides support for prefetching and caching techniques that can further reduce the reconfiguration overheads.

ACKNOWLEDGEMENTS

This work was supported in part by grants, TIN2013-46957-C2-1-P (Spanish Gov. and European ERDF), Consolider NoE TIN2014-52608-REDC (Spanish Gov.), gaZ: T48 research group (Aragón Gov. and European ESF).

REFERENCES

[1] T. Becker, W. Luk, P.Y.K. Cheung, “Energy-Aware Optimization for Run-time Reconfiguration”, Field-Programmable Custom Computing Machines, pp. 55-62, 2010
[2] S. Liu, R. N. Pittman, A. Forin, J. Gaudiot, “Achieving Energy Efficiency Through Runtime Partial Reconfiguration on Reconfigurable Systems”, ACM Trans. on Embedded Computing Systems, vol. 12, no. 3, Mar 2013
[3] K. Paulsson, M. Hübner, and J. Becker, “Dynamic power optimization by exploiting self-reconfiguration in Xilinx Spartan 3-based systems”, Microprocessors and Microsystems, vol. 33, no. 1, pp. 46-52, Feb 2009
[4] D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas, and M. Jones, “Evaluation of rapid context switching on a CSRC device,” in in Proceedings of the International Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA), 2002, pp. 209–215
[5] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in Proceedings of the conference on Design, automation and test in Europe (DATE), 2001, pp. 642–649.
[6] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho, “MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications”, IEEE transactions on computers 49 (5), 465-481
[7] P. Beeck , F. Barat , M. Jayapala , and R. Lauwereins, “Crisp: A template for reconfigurable instruction set processors”, International conference on Field Programmable Logic (FPL 2002), pp. 296-305
[8] Z. Li and S. Hauck, “Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation,” in Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA), 2002, pp. 187–195.
[9] J. Noguera, and R. M. Badia, “Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling,” ACM Transactions on Embedded Computing Systems, vol. 3, no. 2, pp. 385–406, May 2004.
[10]Y. Qu, J. Pekka Soininen, and J. Nurmi, “A parallel configuration model for reducing the run-time reconfiguration overhead,” in IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE), 2006, pp. 965–970.
[11]J. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich, “Interprocedural placement-aware configuration prefetching for FPGA-based systems,” in IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), may 2010, pp. 179–182.
[12]W. Fu and K. Compton, “Scheduling intervals for reconfigurable computing,” in Proceedings of the

IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008, pp. 87–96

[13]J. Clemente, J. Resano, C. Gonzalez, and D. Mozos, "A hardware implementation of a run-time scheduler for reconfigurable systems," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 7, pp. 1263–1276, July 2011.

[14]Z. Li and S. Hauck, "Configuration compression for virtex FPGAs," in Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, pp. 147–159.

[15]A. Dandalis, and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," in Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA), New York, NY, USA, 2001, pp. 173–182

[16]J.A., Clemente, E. Perez Ramo, J. Resano, D. Mozos, and F. Catthoor, "Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.22, no.6, pp.1248,1261, June 2014.

[17]Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in Proceedings of the annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 22–36

[18]R. Kalra, and R. Lysecky, "Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 4, pp. 671–674, Apr. 2010.

[19]J. A. Clemente, J. Resano, and D. Mozos, "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems", ACM Transactions on Embedded Computing Systems, vol. 13 Issue 4, article 90, November 2014.

[20]K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," ACM Transactions on Reconfigurable Technology Systems, vol. 4, no. 4, pp. 36:1-36:24, December 2011

[21]K. Papadimitriou, A. Anyfantis, and A. Dollas, "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems," IEEE Transactions on Instrumentation and Measurement, vol. 59, no. 6, pp.1642–1651, Jun. 2010

[22]E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Exploiting partial runtime reconfiguration for high-performance reconfigurable computing," ACM Transactions on Reconfigurable Technology Systems, vol. 1, no. 4, pp.21:1–21:23, Jan. 2009.

[23]K. Vipin, S. A. Fahmy, "A High Speed Open Source Controller for FPGA Partial Reconfiguration", in Proceedings of the International Conference on Field Programmable Technology (FPT), Seoul, Korea, December 2012, pp. 61–66

[24]S. Liu, N. Pittman, and A. Forin, "Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller", Microsoft Research 2009

[25]L. Ming, W. Kuehn, L. Zhonghai, and A. Jantsch, "Run-time Partial Reconfiguration speed investigation and architectural design space exploration," Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on , vol., no., pp.498-502, Aug. 31 2009-Sept. 2 2009

[26]R. Bonamy, P. Hung-Manh, S. Pillement, D. Chillet, "UPaRC—Ultra-fast power-aware reconfiguration controller," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012 , vol., no., pp.1373-1378, 12-16 March 2012

[27]S. Gimle, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," IEEE International Parallel & Distributed Processing Symposium, 2011, pp.174-180, 16-20 May 2011

[28]*Virtex-5* FPGA *Configuration Guide* UG191 (v3.11) October 19, 2012. www.xilinx.com/support/documentation/user_guides/ug191.pdf

[29]Xilinx DS86 LogiCORE IP XPS HWICAP. June, 2011. https://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap/v5_01_a/xps_hwicap.pdf

[30]Xilinx DS531 Processor Local Bus (PLB) v4.6 (v1.05a). Sept 2010. http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf

[31]Virtex UltraScale+ Product Table
https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable

[32]XPS SYSACE (System ACE) Interface Controller (v1.01a) December 2, 2009.
www.xilinx.com/support/documentation/ip_documentation/xps_sysace.pdf

[33] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys, "Power consumption model for partial and dynamic reconfiguration", International Conference on Reconfigurable Computing and FPGAs, 2012

[34] P. Stenström, "Transactions on High-Performance Embedded Architectures and Compilers IV", Lecture Notes in Computer Science, vol. 6760, 2011

[35]A. Nafkha, and Y. Louet, "Accurate Measurement of Power Consumption Overhead During FPGA Dynamic Partial Reconfiguration", International Symposium on Wireless Communication Systems (ISWCS), 2016

[36]Yokogawa WT210/WT230 Digital Power Meters.
http://tmi.yokogawa.com/discontinued-products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-power-meters/
[37] Interuniversitair Micro-Electronica Centrum.
https://www.imec-int.com/