Hardware Architectural Support for Caching Partitioned Reconfigurations in Reconfigurable Systems

Juan Antonio Clemente, Rubén Gran, Abel Chocano, Carlos del Prado, Javier Resano

***Abstract*—The efficiency of the reconfiguration process in modern FPGAs can improve drastically if an on-chip configuration memory is included in the system because it can reduce both the reconfiguration latency and its energy consumption. However, FPGA on-chip memory resources are very limited. Thus, it is very important to manage them effectively in order to improve the reconfiguration process as much as possible even when the size of the on-chip configuration memory is small. This paper presents a hardware implementation of an on-chip configuration memory controller that efficiently manages run-time reconfigurations. In order to optimize the use of the on-chip memory, this controller includes support to deal with configurations that have been divided into blocks of customizable size. When a reconfiguration must be carried out, our controller provides the blocks stored on-chip and looks for the remaining blocks by accessing to the off-chip configuration memory. Moreover, it dynamically decides which blocks must be stored on-chip. To this end, the designed controller implements a simple but efficient technique that allows maximizing the benefits of the on-chip memories. Experimental results will demonstrate that its implementation cost is very affordable and that it introduces negligible run-time management overheads.**

Keywords*—FPGA, Configuration Caching, Configuration mapping.

I. Introduction

One of the main drawbacks of reconfigurable devices in general, and FPGAs in particular, is that the tasks' configuration process involves not only important delays in the task execution, but also a significant additional energy consumption that does not exist in Application Specific Integrated Circuits (ASICs) [1], [2].

Typically, this configuration process consists in copying the configuration data of the tasks from an off-chip memory to the configuration memory of the device, usually using a dedicated reconfiguration circuitry. However, an important drawback of this scheme is that loading a configuration from the off-chip memory involves a high overhead in terms of performance (typically in the order of hundreds of milliseconds [3]), and in terms of energy consumption [4]. This trend has been observed especially during the last decade. Thus, over the last few years, the gap between off-chip memory bandwidth and the system performance has been increasing significantly. And at the same time, an on-chip memory access with the current technology costs approximately 250 times less energy per bit on average than an off-chip one [5].

In order to overcome this problem, a number of techniques based on configuration caching have been proposed in the literature [6], [7], [8], [9]. They consist in adding one or several intermediate on-chip memories acting as caches, between the off-chip memory where the configurations are stored and the reconfigurable hardware. They aim at accelerating the run-time reconfigurations, as well as reducing the energy consumption due to the reconfiguration process, and they may have different performance and energy consumption trade-offs [10].

However, a typical limitation of these techniques is the reduced capacity of the on-chip RAM memories available in modern reconfigurable devices. For instance, the Virtex-5 FPGA existing in the Xilinx^{TM}XUPV5-LX110T development board contains just 6,448 Kbits of on-chip memory [11], compared to the 256 MBytes of the external DDR2 memory that can be attached to it. High-end FPGAs have about 50 Mbits on-chip memory, but this is still not enough for large reconfigurable regions. Moreover, the designs implemented on the FPGA may use part of the memory on-chip resources. Therefore, only a percentage of them will be available. This percentage is application dependent.

Hence, in many cases there are not enough memory resources to store the needed configurations on-chip. However, if some memory resources are available, it is still a good idea to store them partially. For this reason this article proposes to manage the configurations at a smaller granularity. With this approach, the benefits of a configuration caching technique depend on how much on-chip memory is available and which is the size of the configurations. If there are enough on-chip memory resources to store a significant part of the needed configurations, the proposed configuration caching approach will be useful. Here, it is important to remark that the tendency in computer architecture is to include more and more on-chip memory resources. Hence, the presented techniques can currently be applied in many situations, and in the future it is likely that there will be even more resources to apply them

Manuscript received September 04, 2014; revised 07 January, 2015 and March 10, 2015; accepted March 18, 2015. This work was supported by the Spanish Ministry of Economy and Competitivity under grants AYA2009-13300, TIN2013- 40968-P, TIN2013-46957-C2-1-P, the "José Castillejo" grant and Consolider NoE TIN2014-52608-REDC; by gaZ: T48 research group (Aragón Government and European ESF); and by HiPEAC-3 NoE (European FET FP7/ICT 287759).

J. A. Clemente, A. Chocano and C. del Prado are with the Computer Architecture Department, Universidad Complutense de Madrid, Madrid 28040, Spain (e-mail: ja.clemente@fdi.ucm.es).

R. Gran and J. Resano are with the Computer Engineering Department, Universidad de Zaragoza, Zaragoza 50015, Spain (e-mails: rgran@unizar.es, jresano@unizar.es).

more efficiently.

This paper proposes to extend traditional configuration caching approaches to divide the configurations into blocks, thereby further reducing the granularity of the configuration management. This approach allows mapping each one of these blocks in different on-chip memories, or storing only some of the blocks on-chip, thereby leaving the remining ones in the off-chip memory.

However, this scheme leads to a considerable amount of management services that have to be carried out at run-time (i.e., as the configurations are fetched) and it is desirable to do it transparently to the user. This paper also focuses on this problem, and presents a hardware implementation of a controller that transparently deals with the configurations in an efficient way. This controller also implements an adaptive mechanism that adapts very well to dynamic situations and that aims at maximizing the hit rate on the on-chip memories. It has been implemented using a Xilinx^{TM}XUPV5-LX110T development board, which features a Virtex-5 XC5VLX110T FPGA. Experimental results demonstrate that the proposed hardware controller has a very affordable resource consumption and it adds an additional time overhead of just a few hundreds of clock cycles, which is negligible compared with the reconfiguration latency. In addition, the adaptive mechanism not only improves the hit rate, but also reduces the number of write operations in the on-chip memories.

The remainder of the paper is organized as follows: Section II overviews other relevant works in the literature on reconfiguration overhead reduction. Next, Section III describes the target architecture. Section IV describes a motivational example for the presented work and Section V presents the hardware implementation of the developed controller. Finally, Section VI describes the experimental results obtained and Section VII presents the conclusions of this work.

II. Related Work

Four interesting recent articles [12], [13], [14], [15] have analyzed the impact of loading the configurations at run-time from a non-volatile external memory. The experimental results presented in [12] conclude that the reconfiguration speed is three orders of magnitude worse than the peak reconfiguration speed of the device if a flash memory is used to store the configurations. In [13] the authors analyze the impact of the reconfigurations in the performance of the High-Performance Reconfigurable Computer Cray XD1, including one or several FPGAs and a conventional multiprocessor system. According to their measurements, loading the configurations from an external memory is again three times slower than the theoretical reconfiguration speed. In these two articles, the reason for the additional reconfiguration delays is the access to the off-chip memory. Finally, [14], [15] demonstrate that, by using an on-chip configuration memory directly connected to the reconfiguration port, it is possible to carry out the reconfigurations at full speed. Hence, including an on-chip intermediate level that works as a configuration cache is a straightforward solution to improve the reconfiguration process.

In fact, this technique, so-called configuration caching in the literature, has been elaborated in the literature from long ago [6], [7]. [6] proposes a set of algorithms well suited for single-context, multi-context and partially run-time reconfigurable devices. Their results show that a factor of about 5x overhead reduction can be achieved over traditional methodologies targeting off-chip memories exclusively. [7] extends these ideas for variable-sized tasks, and presents a couple of techniques, so-called *Penalty-based* and *History-based*. Other interesting approaches are [16], [8], [9]. On the one hand, [16] presents a heterogeneous reconfigurable system that includes several reconfigurable processors. It also proposes to include a configuration memory cache for each processor, as well as a configuration prefetch technique. On the other hand, [8], [9] propose hybrid techniques that combine reconfiguration and data caching for tasks in FPGAs.

In addition to the memory access time, many researchers have pointed out that, in embedded systems, the energy consumption due to the configuration memory hierarchy stands for a very important percentage (around 30%) of the overall energy consumption [17], [18]. And this is true for fine-grain and coarse-grain architectures, as long as frequent reconfigurations are demanded. In this sense, our research group has also proposed to extend traditional caching techniques taking into account this point [10]. In that article we present a memory hierarchy composed of two on-chip memory blocks with different performance/energy consumption trade-offs. This scheme provides fast reconfigurations when they are especially critical for the system performance, and at the same time, it allows reducing the energy consumption generated by the dynamic reconfigurations. We also present a technique to decide the task mapping for acyclic task-graphs in this memory architecture.

However, all these configuration caching methodologies share an important limitation: they assume that the task configurations are indivisible. This limits their applicability in modern embedded systems, where the on-chip memory capacities are usually very restricted. Moreover they do not provide hardware support to manage the configuration caching process efficiently.

III. Target Architectural Model and Assumptions

The hardware controller presented in this paper aims at overcoming the previous limitation. It has been designed to target the hardware architectural model depicted in Figure 1. This model comprises one or several on-chip memories placed between the off-chip configuration memory and the reconfigurable hardware. These memories are sometimes called configuration caches, although normally they are not actual caches, but SRAMs controlled by software (i.e., a scratchpads). If used properly, they may drastically improve the performance of the memory hierarchy, as well as the energy consumption, since they prevent the system from accessing a high-capacitance off-chip bus [19]. These on-chip memories feature lower energy consumption and memory access time than off-chip ones.

In addition, each one of them may have different properties. Typically these memories are composed of embedded memory blocks (named Block RAMs or BRAMs in Xilinx^{TM}FPGAs). However, if not enough BRAMs are available, it is also

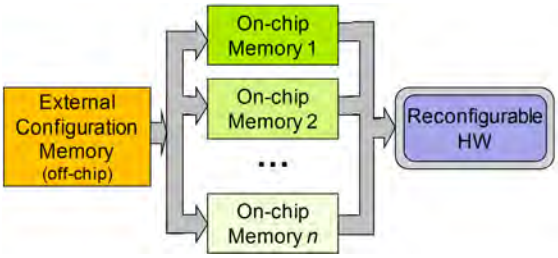


Fig. 1: The target configuration memory hierarchy. The reconfigurable HW is also connected to the off-chip memory by means of a dedicated connection, which is not shown in the figure for simplicity

possible to implement a memory module using the distributed RAM memory resources available in the FPGA slices. This can lead to an heterogeneous scheme where not all the memory components feature the same memory access latency.

In this architecture, it is assumed that the different configurations' blocks are assigned to the different memory/memories that is/are available in the system. These assignments (*mappings* in the remainder of the article) are made depending on the tasks' constraints. For instance, if one task is especially critical for the system, all its blocks can be mapped to the on-chip memory; whereas if another task is executed few times and its execution latency is not critical for the system, its blocks can be mapped to the off-chip memory. Moreover, intermediate options are also supported with some blocks mapped to the on-chip memories and the remaining ones mapped to the off-chip one. It is assumed that an initial mapping is selected for each configuration at design-time, which is used to initialize the hardware controller. This mapping specifies how many blocks must be stored in each one of the on-chip configuration memories. However, the presented controller provides support to dynamically update these mappings at run-time, depending on the system's workload and the pressure exerted on the on-chip memories (this is explained in Section V-B2).

When the system is powered up, *it is assumed that all the configuration blocks are initially stored in the off-chip memory*. When a task is reconfigured, the controller checks which blocks are stored on-chip and fetches the remaining ones from the external memory. Moreover, it checks the mapping of the configuration to identify if some of the blocks fetched from the off-chip memory must be stored on-chip. If the target on-chip memory is full, the controller makes the proper replacement decisions in order to make room for the incoming block. This is explained in greater detail in Section V.

IV. Motivational Example

Figure 2 illustrates the potential benefits of the utilization of the configuration memory hierarchy depicted in Figure 1 in combination with the blocks-based approach hinted in the previous section. In this case, it is assumed that the system executes a loop with two tasks (A and B in the figure) in one reconfigurable unit. The size of each configuration is 100 KB. In order to reduce the reconfiguration latency, the system

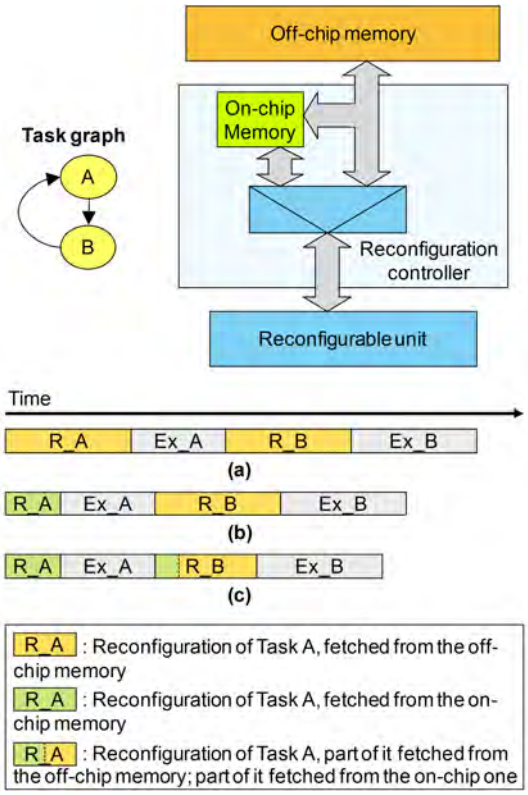


Fig. 2: Motivational example showing the benefits of the blocks-based configuration caching approach

includes an on-chip memory of 160 KB and a configuration controller that can load the configurations both from the on-chip and the off-chip memory (these numbers are reasonable for the family of Xilinx™Virtex-5 FPGAs [3]). The system can also store a configuration on-chip at the same time that it is loading it in the reconfigurable unit.

Since this system does not feature enough memory resources, both configurations A and B cannot be stored in the on-chip memory simultaneously. In fact, if the configurations are indivisible, only one of them can be stored in that memory. Figure 2.a depicts a thrashing problem that exists in this case, when the on-chip memory is used as a conventional cache. In this example, Task A is reconfigured and written in the on-chip memory simultaneously, before its execution. Then, the system has to do the same with Task B. However, in order to store the configuration of Task B in the on-chip memory, Task A has to be replaced. Hence in the following iteration, Task A will have to be written again in the on-chip memory. At the end, the system is not taking advantage of the on-chip memory, but only introducing additional energy overheads.

This thrashing problem can be solved by assigning, for instance, the configuration of Task A to the on-chip memory and by always loading Task B from the off-chip memory. This is shown in Figure 2.b. This allows speeding up the reconfiguration of Task A. Thus, with this approach, half of the configurations are fetched from the on-chip memory and

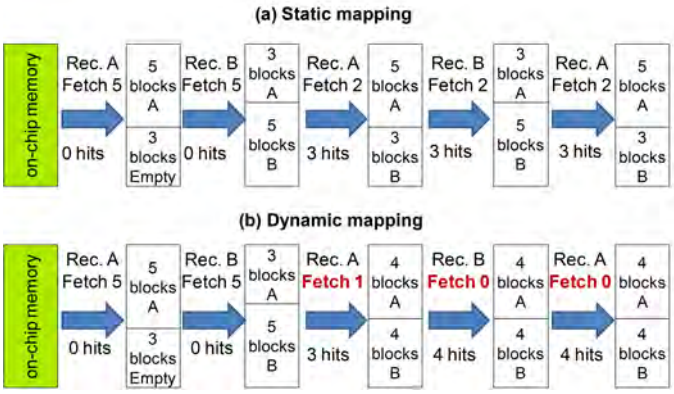


Fig. 3: Motivational example for static vs. dynamic configuration mapping approaches. Configurations are divided into 5 blocks

the remaining ones, from the off-chip one.

However, if the configurations are not indivisible, these results can be further improved. This is depicted in Figure 2.c, where not only the configuration of Task A is stored in the on-chip memory, but also part of that of Task B. In this case, the latency of the reconfiguration of Task B has been reduced since only part of its configuration must be loaded from the off-chip memory. Moreover, the system can also make decisions regarding how many blocks of each configuration are stored in the on-chip memory. In the figure, the on-chip memory stores all the blocks of Task A, plus three blocks of Task B. However, this can also be done the other way round, or by storing 4 blocks of each task. This flexibility can be very useful since some specific tasks can be more critical for the system and storing more blocks in the on-chip memory can help to reduce the reconfiguration latency.

However, in dynamic systems it may not be possible to select an optimal mapping for the tasks at design time since the designer may not know which tasks are going to be executed concurrently. Hence, this paper also presents a set of algorithms that adjust the initial mappings at run time. Figure 3 illustrates the idea continuing with the previous example. This example assumes that the initial mappings for Tasks A and B assign all their blocks on-chip. In Figure 3.a (*static mapping*), only this initial mapping is used. Thus, the on-chip hit rate is 60%, which is suboptimal since there are enough memory resources to achieve an 80% hit rate (eight out of the ten blocks can be stored on-chip). However, the dynamic mapping used in Figure 3.b makes possible to reduce the number of blocks mapped to the on-chip memory when any conflict is detected (i.e. when some blocks existing in the on-chip memory must be replaced in order to load the new ones). In this example, this adjustment has involved reducing the number of blocks mapped on-chip for each task from 5 to 4, thus leading to an 80% hit rate. Subsection V-B2 will explain in detail how this dynamic adjustment is carried out.

In order to achieve the results hinted in Figures 2 and 3, the configuration controller proposed in this article can deal with configurations partitioned in blocks of a given size, and at the same time, it can support one or several on-chip memory modules.

It is also interesting to note that dealing with blocks instead of with large configurations reduces the impact of memory fragmentation. If an on-chip memory stores and replaces configurations of different sizes, it may end up in a situation with several free small-medium memory segments that are not big enough to store a configuration. Hence, in this case, the system is not efficiently taking advantage of the available on-chip resources. Using a scheme based on fix-sized blocks also alleviates this problem. The reason is that all the memory transfers deal with a size that is a multiple of the block size. Hence the size of the available memory segments will be also a multiple of the block size and, with the proper management, all these segments can be used to load new blocks.

V. The Proposed Hardware Controller

The general scheme of the designed hardware controller is depicted in Figure 4. It comprises a number of on-chip memories, directly and exclusively managed by an instance of a *Local Memory Controller* (depicted in detail below). The *Global Memory Controller* controls the operation of the local controllers and it also communicates with the operating system (OS) of the system or a *middleware*. All these modules can access to the off-chip memory, which, in this case, is the 256 MB DDR2 memory block available in the XUPV5-LX110T development board used for implementation of this design. For this reason, all their communication interfaces are connected to a the *MUX/DEMUX* module, which comprises a set of multiplexers (in case of the input signals to the memory) and demultiplexers (for the output ones). They operate through the *local controller selection* signal.

We have included two levels in our controller to improve the scalability and the modularity of the design. With this approach, it can deal with a variable number of independent memories that may have different access latencies. In addition, a distributed approach with several local controllers can operate faster than an equivalent centralized approach with only one controller.

The global memory controller receives two kinds of orders:

- *Store/Update configuration mapping*: An initial mapping is specified for each given configuration. The global controller stores/updates this information in a table (*Conf. Mappings Table* in the figure) accordingly. Each entry in this table contains the information about the configuration tag (a number that identifies each configuration in the system), and its initial mapping. This table is implemented using an embedded BRAM of the FPGA.
- *Reconfigure*: A given configuration must be loaded onto the reconfigurable hardware. In this case, the global controller reads the mapping from the *Rec. Mappings Table* and it sequentially demands the blocks assigned to each on-chip memory, as Figure 5 indicates (Steps 1-4). For this purpose, it invokes the corresponding instance of the Local Memory Controller module (Step 3). Each controller then sends the blocks assigned to it. If the local controller does not have those blocks

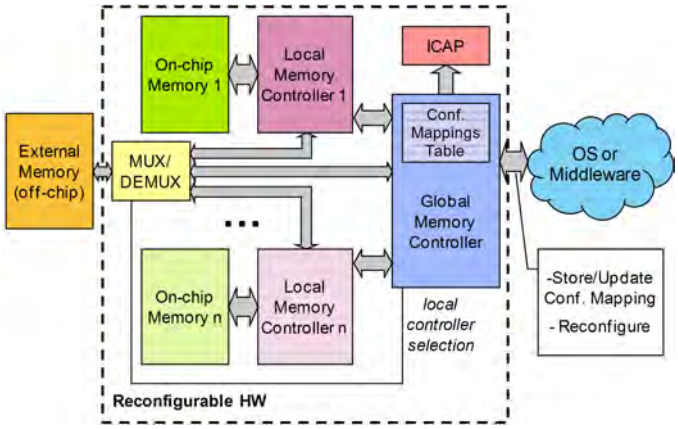


Fig. 4: General scheme of the proposed hardware module

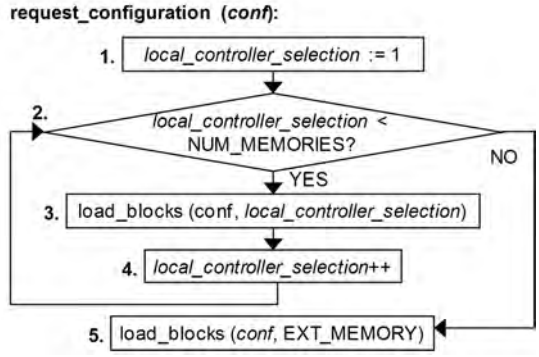


Fig. 5: Flowchart for the operation of the global memory controller

stored in its associated memory, it requests them to the external memory. Each time that a local controller sends a configuration word to the global one, it forwards it to the reconfiguration port (called Internal Configuration Access Port (ICAP) in Xilinx^{TM}FPGAs). A 32-bit bus is used since is the maximum size currently supported by the ICAP [3]. Steps 3 and 4 are executed for each on-chip memory in the system. Finally, if some blocks have been mapped to the external memory, the global controller fetches them and sends them to the ICAP (Step 5). The global controller carries out all these operations by means of a finite-state machine, not appearing in Figure 4 for the sake of simplicity.

Figure 6 depicts the local memory controllers that are instantiated in Figure 4 in greater detail. It contains an *Associative Table* that keeps record of the information currently stored in the associated on-chip memory. Each entry of this table contains the information about which blocks of a given configuration exist in that memory at any given time. Note that this information is complementary with the one that is stored in the *Conf. Mappings Table* of the global controller (explained above in this section). The latter contains information of how the configuration blocks should be mapped in the memories of the system. This information is obtained at design-time and

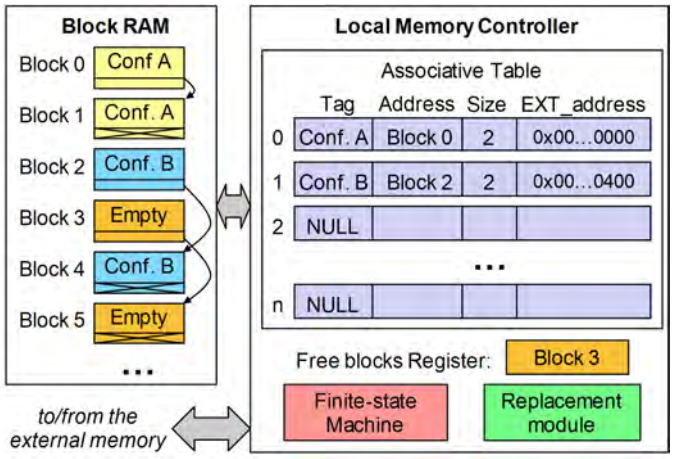


Fig. 6: General scheme of the local memory controller

may (or may not) be adapted at run-time depending on the run-time state of the system (this point is actually studied in Subsection VI-B). On the other hand, the information stored in the associative tables refer to the run-time state of the on-chip memories. Thus, both kinds of tables contain different (and complementary) information.

The information stored in the *Associative Table* is divided into five fields:

- *Tag*, which is the name of the involved configuration.
- *Address*, indicating the initial address in the on-chip memory where the configuration blocks are stored.
- *Size*, which indicates the number of blocks stored in this on-chip memory. It is important to note that this table always manages consecutive blocks (no matter how many times the system replaces and loads the blocks of a configuration, the controller does guarantee that the blocks are stored in order; i.e., if the first one is block n, the next one will always be block $n+1$).
- *EXT_addr*, which is the initial address where this set of blocks are stored in the external memory.
- *Ini_block*, which indicates the relative position of the first stored block in the sequence of blocks assigned to this memory. Thus, is this field is 0, then *Address* points to the first block in that sequence; and if *Ini_block* is 2, that means that the two first blocks assigned to this memory are not currently stored on-chip. Hence, if the global controller requests the configuration of this task, it must manage two misses and fetch these two blocks from the off-chip memory.

In the on-chip memory, the blocks that belong to the same configuration are linked by means of pointers. The pointer associated to the last block in each sequence is equivalent to *null* (for instance, Blocks 1, 4 and 5 in Figure 6). This approach allows the local controller to easily fetch the information from the memory, by simply following the pointers as many times as indicated in *Size*. This controller also includes a register that stores the address of the first free block in the on-chip memory (*Free Blocks Register* in the figure). Again, these free

blocks are also linked among them by means of pointers.

The on-chip memories can be implemented either using the distributed RAM of the FPGA, or by instantiating BRAMs. The developed design uses *generic* parameters in the VHDL code to adapt itself to different memory and block sizes.

Also, it is important to note that the controller associates a parameter to each on-chip memory indicating its memory access time (in clock cycles). For instance, a value of "1" indicates that a new word can be read from the memory at each clock cycle.

A. The Local Memory Controller

The objective of the local memory controller is to dispatch the load requests from the global one. To this end, it delivers the blocks that were assigned to the corresponding on-chip memory. In addition, the associative table, the free blocks register and the data stored in the on-chip configuration memory must be correctly updated. It follows the flowchart of Figure 7. It firstly checks if the involved configuration exists in the local memory by consulting the associative table (Step 1). Depending on this, the register n is updated with the number of blocks that have to be fetched from the external memory. If this configuration has no blocks stored in this on-chip memory, all the assigned blocks must be fetched from the external one (Step 2; n_blocks_conf is the number of blocks assigned to this on-chip memory). Otherwise, the controller only fetches from the external memory those blocks that are not already stored on-chip (Step 3, $Size$ is the number of blocks from the current configuration that are already stored on-chip).

Then, the controller deals with the missing blocks by accessing to the external memory (Step 4), it writes them to the local memory (Step 5) and it updates the associative table and the free blocks register accordingly (Step 6). Note that, if n is equal to 0, then Steps 4-6 do not carry out any computations. Next, n is updated again with the number of blocks that were not fetched from the external memory and that hence, must be loaded from the local memory (Step 7). Thus, if n is greater than 0 (Step 8), these blocks are fetched from the local memory following the pointers-based approach explained above (Step 9) and the algorithm finishes (Step 10).

Note that this approach assumes that *the blocks stored in any memory are a sequence of consecutive blocks, always finishing in the last one in that sequence*. Our local memory controller have been designed in order to ensure that this is always true.

B. The Replacement Module

On-chip memories are expensive; hence, in the developed system it is very likely that an on-chip memory will not have enough resources to store all the active configurations simultaneously. Thus, when executing Steps 5 and 6 in Figure 7, if not enough free blocks are available in the involved on-chip memory, the local controller must decide which block(s) must be replaced in order to make room for the new incoming one(s). To this end, it invokes the *Replacement Module*. Note that, in order to guarantee the consistency with the algorithm depicted in Figure 7, *a replaced block is always the first one in its sequence*. For instance, if the on-chip memory stores Blocks 1, 2, 3, 4 and 5 of Configuration A, and two of them must be replaced, the controller always replaces Blocks 1 and 2. And after that, it updates the *Size* field (from 5 to 3), the *Ini_block* field (from 1 to 3) and the *Address* field, which now points to the address where block 3 is stored.

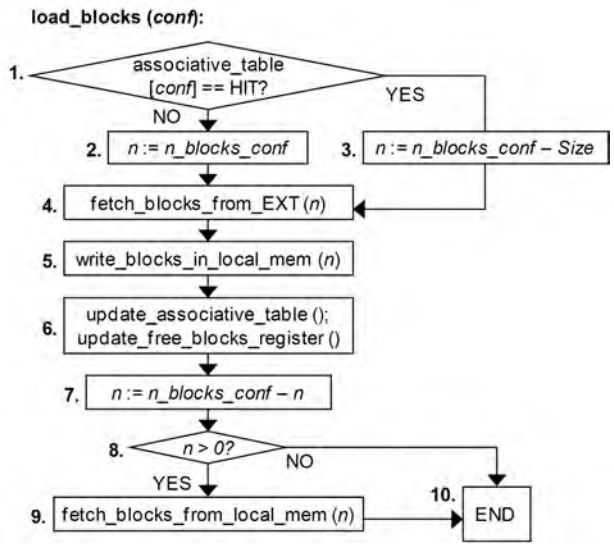


Fig. 7: Flowchart for the operation of the local memory controller

Of course, if the replacement module decides that the only existing block in a sequence must be deleted from the memory (because it is the only block left in that sequence), the corresponding entry in the associative table is deleted too.

Typically, a conventional cache would take advantage of the spatial locality. Indeed, if a given memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. Thus, when the access to a certain memory location generates a miss, the cache fetches a block containing not only the requested memory location, but also some adjacent locations. As a consequence, when the processor accesses these other locations it will obtain hits in the cache. Unfortunately, this behaviour cannot be expected in the configuration on-chip memories because on the one hand, configurations are very large; hence it will not be a good idea to fetch N adjacent configurations in memory when only one of them has been requested, and on the other hand, it is not clear that adjacent configurations will be executed in the near future since they can belong to independent tasks.

Hence, temporal locality has to be exploited instead. If a particular configuration is requested, it is likely that it will be requested again in the near future. Thus, if a configuration is going to be executed only once, it is not a good idea to store it on-chip. Instead of that, only those configurations that are going to be executed several times should be stored on-chip. However, as previously illustrated in the motivational example of Figure 2, when the size of the active configurations exceeded the capacity of the on-chip configuration memory, it is very

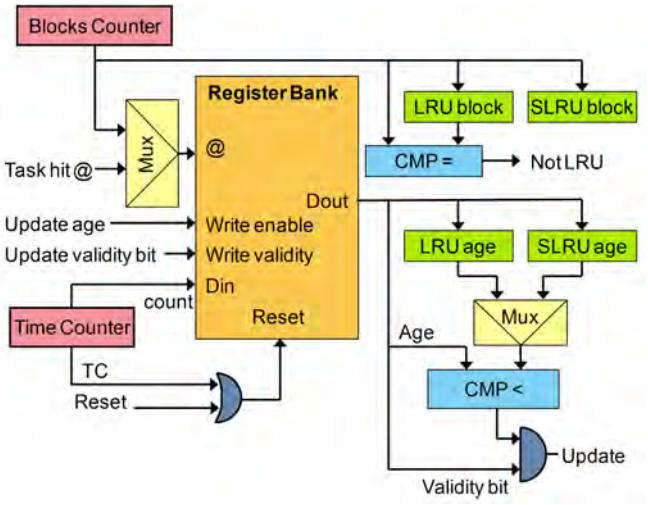


Fig. 8: Datapath of the LRU implementation

easy to fall into a thrashing problem. However, the replacement module satisfactory deals with these situations. It includes mechanisms that optimize the decisions made at run-time in order to maximize the utility of the on-chip memory. On the one hand, it implements different replacement policies and on the other hand, in includes a custom adaptive mechanism. Both features are explained in greater detail in the next two subsections.

1) Replacement Policies: The replacement module implements either of the following three well-known replacement policies:

- *Random*: It replaces a random block. The implementation cost is very affordable: it just involves a counter ranging from 1 to the number of configuration blocks in the memory. When the system makes a replacement, it stops the counter and it replaces the first available block belonging to the victim configuration. This is consistent with the ideas explained above, at the beginning of Subsection V-B.
- *LRU (Least Recently Used)*: It replaces the first available block from the least recently used configuration. If a configuration has not been used for a long time, maybe is not going to be used anymore. Hence it is a good candidate for replacement. This is considered as a very good replacement policy but it is also very expensive. Its hardware implementation involves storing when each configuration was used for the last time, and including some hardware to identify which configuration must be selected as the replacement victim.
- *LFU (Least Frequently Used)*: It replaces the first available block from the least frequently used configuration. Again, the idea is simple: those configurations that are executed more frequently should not be replaced. However, it is also an expensive policy.

Figures 8 and 9 depict the datapath of the LRU and LFU implementation, which include some important features especifically designed for this problem. Both designs include a *Register Bank* that stores the age of the configuration blocks (i.e., indicating the instant of the last use of that block) for the LRU and the frequency for the LFU. It also includes a validity bit for each block. When all the blocks of a configuration have been replaced, the local controller sets this bit to *invalid*, and this configuration will not be selected again. The LRU design also includes a counter that is increased after each reconfiguration and it is used to update the ages (*Time Counter* in Figure 8). Finally, it includes hardware support to read the ages of all the blocks and to select two of them: the LRU and the SLRU (Second Least recently Used). This is done using the *Blocks Counter* to read all the registers, and a comparator to identify if the age of the current block is smaller than the current LRU. When the LRU has been identified, a second iteration is carried out in order to find the SLRU. This has been included to prevent a configuration block from replacing to itself. For instance, it may happen that the on-chip memory contains one block of a configuration and needs to fetch an additional one. In this case, it is not a good idea to replace the block that is already on-chip. If that block is selected as LRU, the SLRU is selected instead. It is important to note that *the SLRU block is selected in such a way that it belongs to a different configuration from that of the LRU block*. This makes possible to always avoid replacing blocks from the configuration that is currently being requested. Of course, this approach involves duplicating the number of cycles needed to select the LRU. But this is not a problem because these cycles do not generate any delay in the configuration management. Indeed, in order to know which configuration should be replaced, it is enough to read the LRU and the SLRU registers. Afterwards, while the replacement is being carried out, both registers are updated for the next time this module is invoked. Since a replacement involves moving large blocks from the external memory to the on-chip one, the latency of the replacement module is completely hidden.

The LFU design (Figure 9) is very similar. The main difference is that, when a configuration is executed, the frequency is incremented instead. To this end, it is read from the register bank, updated, and stored again.

Both designs also have to deal with possible count overflows: when a counter finishes its count, it will return to zero, and then the values will not be consistent. When that happens, the signal *TC* (Terminal Count) is activated and all the values of the register bank are set to 0. This is also true for the LFU design, and in that case these periodic resets are important since they prevent the on-chip memory from keeping configurations that were used many times in the past but are not being used anymore.

2) Adaptive Mechanism: A good replacement policy is a key point to obtain good results, but if the pressure exerted on the on-chip memory is too high, it will not be enough since the configurations will be replaced before they can be reused. Indeed, in that situation, the on-chip memory does not improve the performance, and in fact it increases the energy consumption due to the additional written data. A simple solution is to dynamically adjust the number of blocks assigned to the on-chip memory in order reduce the pressure. This is as simple as adjusting the initial mappings of the configurations.

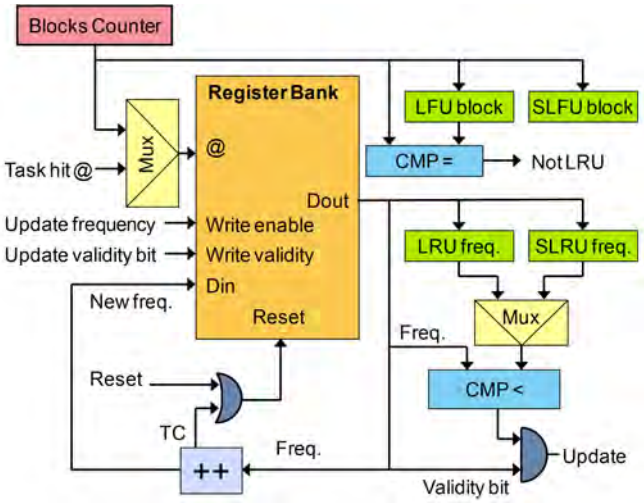


Fig. 9: Datapath of the LFU implementation

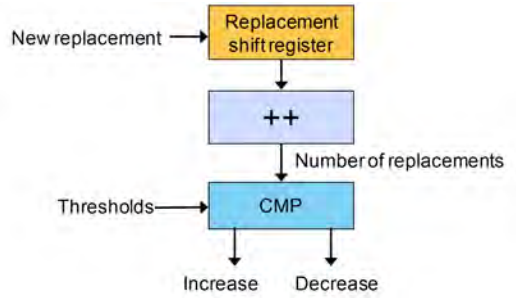


Fig. 10: Scheme of the adaptive mechanism implemented in the replacement module

However, in order to decide the proper mappings for each configuration, it is necessary to know the current pressure on a given on-chip memory, since that pressure may change at run-time depending on dynamic events. For instance, a mapping that works fine when few configurations are active can start being inefficient if more tasks are assigned to the reconfigurable resources. The problem is that it is not always possible to know at design time how many tasks are going to be active simultaneously. In order to deal with this problem, the replacement policy also implements a simple but very effective adaptive mechanism.

The basic idea is to monitor the replacement activity on the on-chip memories in order to identify if the pressure is high or low. If frequent replacements occur on a given on-chip memory, it is likely that that memory is under a big pressure and that the fetched blocks will probably be replaced before they are reused. Therefore, in order to reduce that pressure, in the next miss the developed mechanism modifies the configuration mapping of that configuration, by reducing the number of on-chip blocks from the configuration that triggered the replacement.

This new value is stored in the controller and it will be the reference value for that configuration in the future. In the same way, if the controller detects that no replacements have been carried out on a given on-chip memory during a certain period, maybe the pressure exerted on that memory is low. Hence, in the next iteration the adaptive mechanism increments the number of blocks assigned on-chip.

Figure 10 depicts the implementation of this mechanism. It contains a shift register that stores whether the last N reconfigurations triggered a replacement or not. Then, an adder reports the number of replacements. Finally a comparator compares this number with a lower and an upper threshold. If the number of replacements is over the upper threshold, the *Decrease* signal is set to 1 and if it is smaller than the lower threshold the *Increase* signal is set to 1. This design can be instantiated with different sizes for the shift register and different thresholds. We have experimentally checked that it provides good results even for small registers of just 8 bits. Thus, in our experiments we use a 8-bit shift register; and we set the upper threshold to 3, and the lower threshold to 1.

VI. Experimental Results

This section evaluates the hardware controller presented in this paper, both in terms of resources consumption, runtime overhead and performance. The results are explained in the following subsections.

A. Resource Consumption and Time Overhead

First of all, Table I presents the resources consumption for a hardware module as the one depicted in Figure 4, when it instantiates 2 local memory controllers and the capacity of the on-chip memories are 8 configurations each. The results are shown in terms of reconfigurable resources in a Xilinx^{TM}XUP Virtex-5 XC5VLX110T FPGA. It is interesting to underline that the size of the configurations does not modify the size of the controller since the same information is needed for any configuration independently of its size.

For a local memory controller (Row 5), these results are also broken down into its submodules: the associative table (Row 2), the finite state machine (Row 3) and the replacement module (Row 4). The results for the global memory controller are also presented (Row 6), as well as those of the complete system (Row 7). The resources consumption of the complete design is very affordable (up to 2.70% of the available resources in the FPGA). In this case, its maximum operating frequency is 153.7 MHz, which allows to deliver configuration data to the ICAP at its maximum theoretical speed (100 MHz) [3]. In any case, we consider that the design presented in this paper is a proof of concept. The objective was not to minimize the hardware cost and maximize the frequency, but to demonstrate that the proposed techniques can be implemented in hardware with an affordable cost.

The scalability of the design has also been studied. In this case, the complexity of the design increases with the number of local controllers instantiated, and with the maximum number of different configurations that the local controllers can manage. The results of this study are depicted in Figure 11.

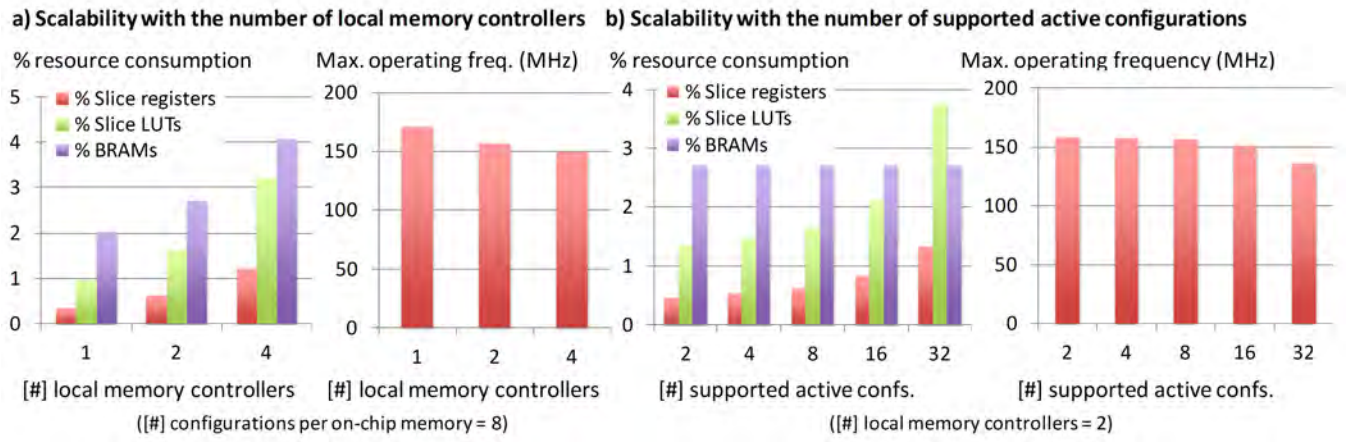


Fig. 11: Scalability of the proposed hardware controller, both in terms of resource consumption and maximum operating frequency (for a Xilinx™XUP Virtex-5 LX110T FPGA)

TABLE I: Resource consumption of the proposed hardware controller, featuring 2 local memory controllers with LRU replacement policy and assuming local memories being able to manage up to 8 different configurations each (for a Xilinx™XUP Virtex-5 LX110T FPGA)

	Slice Registers	*Slice LUTs*	*Block RAMs*
Assoc. Table	8 (0.01%)	43 (0.06%)	1 (0.68%)
FSM	143 (0.21%)	344 (0.50%)	0 (0%)
Replacement Module	49 (0.07%)	141 (0.20%)	0 (0%)
Local Memory Controller	200 (0.29%)	528 (0.76%)	1 (0.68%)
Global Memory Controller	16 (0.02%)	77 (0.11%)	1 (0.68%)
TOTAL	432 (0.63%)	1,130 (1.63%)	4 (2.70%)

On the one hand, Figure 11.a shows that the proposed design scales linearly with the number of local memory controllers (keeping the number of configurations per on-chip memory to 8). Note that the numbers in the x axis grow exponentially. However, in spite of that, the resource consumption keeps reasonable values. In addition, the maximum operating frequency remains very close to 150 MHz.

On the other hand, an important parameter of the proposed controller is the number of active configurations supported. This parameter is used to set the size of the associative table and the structures used for the replacement modules. Figure 11.b shows how the system scales for different values of this parameter (in the figure: *supported active configurations*). In this case, for an implementation with two local memory controllers, where each one of them supports 32 active conguRations, the resource consumption never exceeds 4% of the available resources in the FPGA, while the maximum operating frequency remains always above 100 MHz. The increase that can be observed in the resource consumption and in the performance degradation is due to the increasing complexity of the replacement module and the associative table. In fact, the hardware cost of the proposed LRU and LFU implementations increases linearly with the size of their register bank (see Figures 8 and 9). Thus, for instance, a design that supports 32 configurations per on-chip memory and that uses registers of 9 bits (8 bits for the age or the frequency plus the validity bit) needs 316 slice registers (0.46%) and 160 slice LUTs (0.23%). This is a very affordable implementation cost. Finally, it has also been checked that the adaptive mechanism barely has any impact on the implementation cost. With an 8-bit shift register (see Figure 10), and setting the upper and lower thresholds to 3 and 1 respectively, the hardware cost of the adaptive mechanism is negligible.

The time overhead that the proposed controller introduces at run-time has also been studied (i.e. the number of management cycles introduced by our controller in order to identify where each block is stored, carrying out the block replacements, and updating all the information accordingly). Table II shows these results for different granularities of the configurations. Thus, three situations have been considered: when the configurations are indivisible and when they comprise 8 and 64 blocks each. This experiment involved a single configuration whose blocks were initially mapped on the external memory. Thus, in this example the total reconfiguration latency of a configuration that has been split into 64 blocks will be the transfer time (that will be different each time depending on whether the blocks are stored on-chip or off-chip), plus the management time that is the data presented on this table. In this case, as the table shows, it will be at most 778 clock cycles.

The following two experiments have been carried out:

1) When the benchmarks are executed for the first time on the system. The objective is to observe how the controller works in case of block misses (initially, all the on-chip memories are empty). In this case, it has to fetch the configuration blocks from the external

TABLE II: Time overhead in terms of clock cycles of the proposed controller, for different block sizes

	Indivisible configurations	*8 blocks per configuration*	*64 blocks per configuration*
1st execution	281	292	309
2nd execution	473	494	778

memory and to copy them on the on-chip memories.

2) For a second (and subsequent) executions of the same benchmarks in the system. The objective is to observe how the controller works in case of block hits in the on-chip memories.

As the table shows, in all the cases, all the involved operations introduce delays in the order of several hundreds of clock cycles. Assuming that the controller works at the maximum theoretical speed of the ICAP port (100 MHz), these delays are negligible with respect to the reconfiguration latency of the applications. Besides, note that the second (and subsequent) execution(s) of the same tasks on the reconfigurable system generate more overhead than the first execution of the same task. The reason is that in that case, the controller must individually check if the involved blocks are still stored on-chip or not. Otherwise, this is not necessary since the associative tables reported that the entire configuration is off-chip.

Finally, it is important to note that the proposed design scales very well with the number of blocks per configuration (again, see Table II). Thus, even with 64 blocks per configuration, the proposed controller works very well.

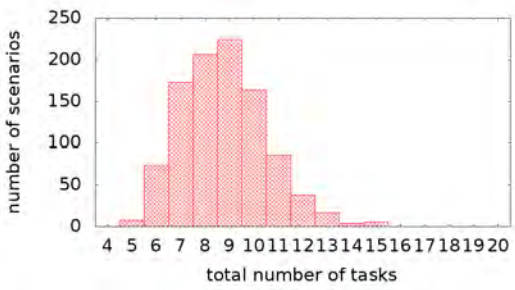
B. Performance Evaluation

This subsection presents some experiments to show performance results of the developed controller. Hit rate on the on-chip memory has been used as the metric to measure the performance since other metrics (such as the global execution time) heavily depend on additional factors such as the external memory controller, the technology of the off-chip memory, the hierarchy and congestion of the bus, or whether the system includes a DMAC (Direct Memory Access Controller) or not. On the contrary, for a given execution sequence, the hit rate only depends on the configuration management.

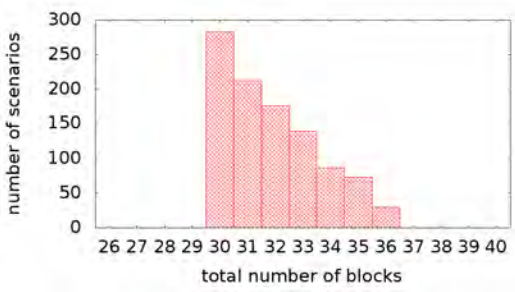
A trace-based functional simulator was implemented for these experiments. This simulator models an organization as shown in Figure 4, with a single local memory controller. Both non-adaptive (NA) and adaptive (A) mechanisms have been modeled with 3 replacement policies: Random, LRU, LFU. Non-adaptive mechanisms use an initial mapping that assigns all the blocks to the memories available in the system, whereas the adaptive mechanisms start from the same mapping but they can update it at run time.

In order to test the capabilities of both mechanisms, the simulator is fed with a sequence of tasks to be reconfigured. 5 different reconfiguration sequences have been modeled:

1) *Cyclic*: This sequence fetches the reconfiguration of all the tasks cyclically. The order that each task occupies in a reconfiguration cycle is always the same.



(a) Histogram of # of tasks



(b) Histogram of # of blocks

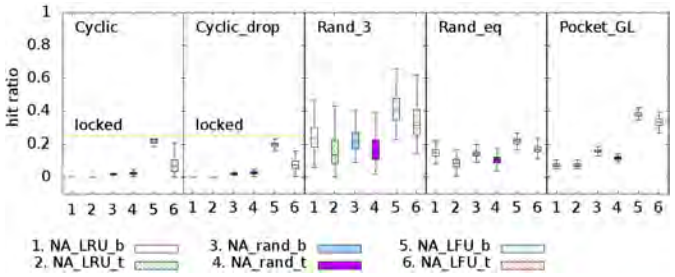
Fig. 12: Characterization of the 1000 scenarios

2) *Cyclic_drop*: A cyclic sequence in which tasks are removed from the sequence with a probability of 5%.
3) *Rand_eq*: A randomly generated sequence of tasks in which all the tasks have the same probability to appear.
4) *Rand_3*: A randomly generated sequence that prioritizes 3 tasks above the others. A 70% of the tasks in the traces is one of these 3 tasks.
5) *Pocket_GL*: A trace that corresponds to the real-world application Pocket-GL [20]. It imitates the actual reconfiguration pattern observed for this application.

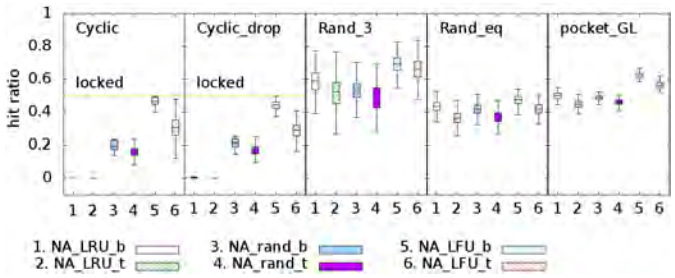
For the first four sequence types, 1000 different scenarios have been generated. Each one of them had a random number of tasks and blocks per task. All these scenarios had at least 30 blocks in total, and no task had more than 8 blocks. Figures 12a and 12b show two histograms that characterize these scenarios. The first one shows the total number of blocks in the scenario, whereas the second one shows the number of tasks in it.

For each one of these 1000 different scenarios, the *Cyclic*, *Cyclic_drop*, *Rand_eq* and *Rand_3* reconfiguration sequences included 1024 different tasks. In case of *Pocket_GL*, it includes 1000 reconfiguration sequences randomly selected among the reconfiguration patterns observed in the real application.

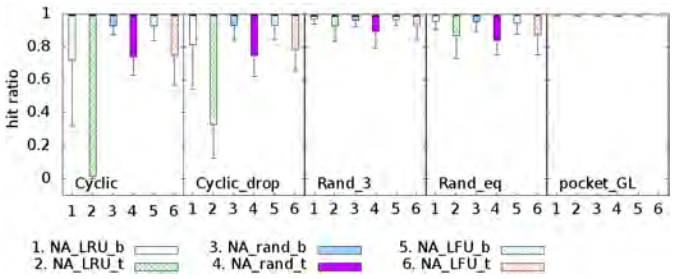
Figure 13 shows the hit ratios obtained for cache sizes of 8, 16 and 32 blocks, when using the non-adaptive (NA) mechanism in combination with the discussed replacement policies: LRU, Random and LFU. Additionally, for each replacement policy there are two hit-ratio boxplots: one of them labeled with *_t* and the other with *_b*. In the former case, the replacement mechanism victimizes not only the victim block,



(a) Cache size = 8 blocks

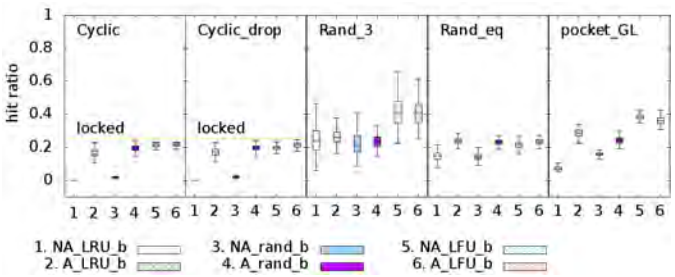


(b) Cache size = 16 blocks

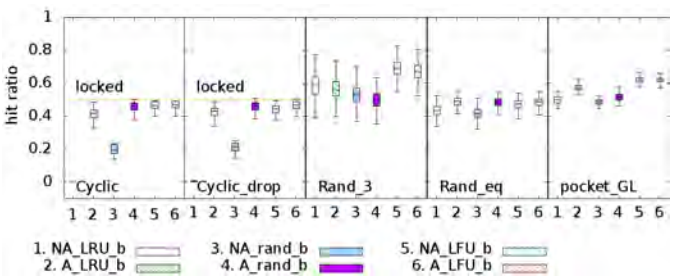


(c) Cache size = 32 blocks

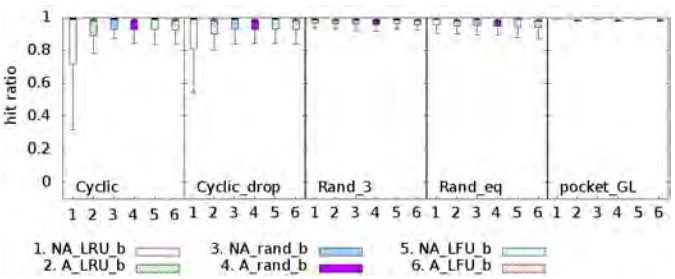
Fig. 13: Hit ratios in the 5 reconfiguration sequences for cache sizes: 8, 16, 32 blocks of non-adaptive mechanism with replacement policies: LRU, Random and LFU. The suffix *_b* means that the replacement is done at block granularity, whereas *_t* means that the replacement is done at task granularity. For each model and reconfiguration sequence, we show a box-plot from the 1000 samples. The central rectangle spans the first quartile to the third quartile (interquartile range). Whiskers indicate the minimum and maximum values (first and fourth quartiles)



(a) Cache size = 8 blocks



(b) Cache size = 16 blocks



(c) Cache size = 32 blocks

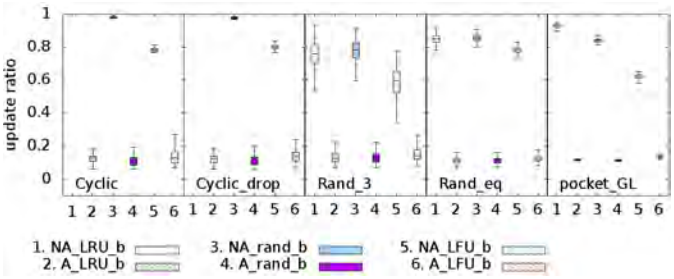
Fig. 14: Hit ratios in the 5 reconfiguration sequences for cache sizes: 8, 16, 32 blocks of non-adaptive and adaptive mechanisms with replacement policies: LRU, Random and LFU

but also all the blocks belonging to the same task (i.e., it carries out a task granularity replacement). In case of *_b*, the replacement policy will evict only the victim block.

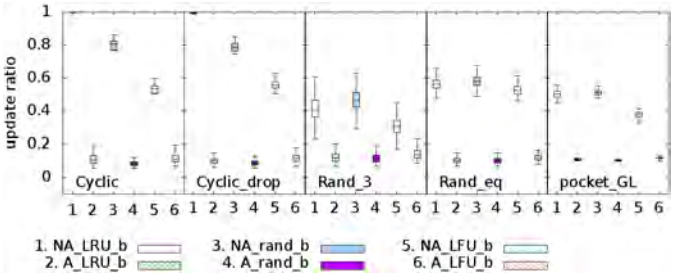
Focusing on the comparison of task and block granularity eviction, all the cases reveal that victimizing at task granularity is an aggressive decision that has a negative impact in the performance. For instance, victimizing a task with a large amount of blocks will free the corresponding blocks of the cache. If they stayed unoccupied until the next reconfiguration of that large task, then the system would incur into additional block misses. This does not not occur in case of block granularity victimization.

In addition, while the cache size increases, it also increases the hit ratio. However, *NA_LFU_b* always performs better than the other replacement policies. On the one hand, for 16 blocks of cache size and reconfiguration scenarios *Rand_3* and *Pocket_GL*, LFU is able to identify which blocks are most frequently accessed and then they stay in cache above other blocks in case of replacement. In *Rand_eq*, all configurations have the same chance to appear, similarly to *Cyclic*, and also there is not a pattern of reconfiguration as in *Rand_3*. In such random reconfiguration sequence, victimizing a block is not neither better nor worse than victimizing any other block. Even more, in *Rand_eq*, all replacement policies perform similarly. The reason is that, in these circumstances, every configuration in cache have the same chance to be reconfigured next, independently of how the replacement policy behaves.

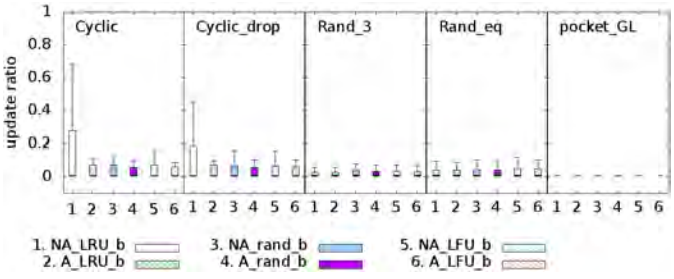
On the other hand, despite the simplicity and predictability of *Cyclic* and *Cyclic_drop*, LRU and Random evict blocks that are accessed with the same probability than the blocks that update the cache. Moreover, because of the cyclic pattern, the evicted blocks are accessed again before than the ones



(a) Cache size = 8 blocks



(b) Cache size = 16 blocks



(c) Cache size = 32 blocks

Fig. 15: Ratios of updated blocks in the 5 reconfiguration sequences for cache sizes: 8, 16, 32 blocks of non-adaptive and adaptive mechanisms with replacement policies: LRU, Random and LFU

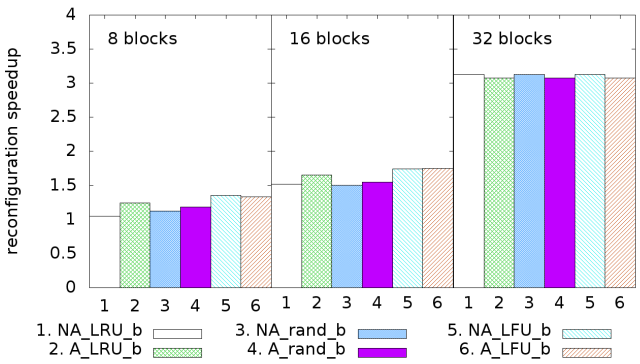
that are being updated, therefore blocks come in/out of the cache but locality is not captured (thrasing). Only if the cache is large enough to hold the blocks of all the tasks, this trashing effect does not occur and then the replacement policy stops penalizing performance (Figure 13c). In such cyclic reconfiguration sequences, if the number of blocks of all the tasks is greater than the size of the cache, then a good approach consists in locking some blocks in cache and disabling the replacement. This strategy assures that, in every cycle of tasks, accesses to locked blocks will hit. Assuming that the average size of the scenarios is 32 blocks (Figure 12b), in case of cache size of 8 blocks, this locking mechanisms assure as 25% of hit ratio (50% when the cache size is 16). This locking model is represented in the figure by means of a horizontal line and it can be understood as an ideal upper-bound of the hit ratio without help of predictive mechanisms such as prefetch. It can be observed that LFU is quite close to this locked cache. The reason is that blocks of the first tasks of the cycle prevail in cache over the blocks at the end of it. In case of replacement, note that a task of the beginning of the *Cyclic* pattern is always as frequent (or even more) as the task at the end of the pattern. Therefore, the cache is filled up with blocks of the first tasks and then a similar effect to a locked cache is achieved.

Next, Figure 14 shows the hit ratio for the adaptive mechanism and compares it with the non-adaptive one. In this figure, all the replacement mechanisms work at block granularity.

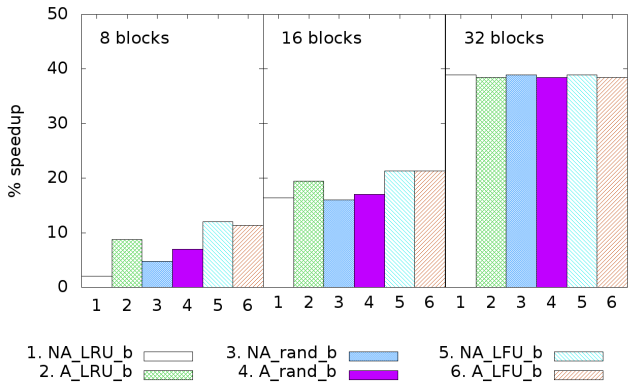
The results show that for the adaptive mechanism, LFU is the replacement policy that performs the best. It is noticeable that, in reconfiguration sequences *cyclic* and *cyclic_drop*, it works well independently of the replacement policy used. In case of Random and LRU, the adaptation reduces the number of blocks to be loaded in cache by each task, and thus, it manages to allocate in cache a set of blocks that will be hit every cycle. The adaptive mechanism slightly improves the results in almost all the cases. Hence, if it is not acceptable to pay for a complex replacement policy, an interesting alternative is to use a random replacement combined with the adaptive mechanism. The implementation cost of this approach is very small and the results are only slightly worse than those obtained when using complex replacement policies, such as LFU. Moreover, there is an additional advantage, which can be observed in Figure 15. This figure shows the number of updated blocks in cache with respect to the total number of blocks accessed. It is clear that the adaptive mechanism requires less update operations than the non-adaptive one. The adaptive mechanism manages to adjust the number of blocks to be loaded in cache by each task. Once that adjustment is done, the mechanism stays in a stationary regime in which the update activity is very low in comparison with the non-adaptive one. In case of change in the features of the reconfiguration sequence, the adaptive mechanism starts an adjusting phase until it stabilizes. This lower ratio of updating activity can be directly translated into energy savings.

Finally, in order to assess the effective improvement of our techniques in comparison to not caching reconfigurations, we have evaluated the improvement achieved in terms of overall performance (execution time) and in the reconfiguration time, for the Pocket-GL application. Figure 16a shows the speedup of our techniques over the time taken to reconfigure that application. In addition, Figure 16b shows the percentage of performance speedup of caching reconfigurations. Three cache sizes are modeled: 8, 16 and 32 blocks. Also, we have evaluated both the adaptive and non-adaptive mechanisms with the three different replacement policies: LRU, Random and LFU. All the replacement mechanisms work at block granularity. In this experiment, the Xilinx™Virtex-5 XUPV5-LX110T FPGA has been used. In that device, the off-chip DDR2 memory stored the configurations, and a DMA controller optimized its access. With these settings, we have measured that the reconfiguration time of a block from the DDR2 memory using DMA is 3.2 times slower than from the on-chip one.

Figure 16a shows how many times faster is a system with our controller than a bare system without a reconfiguration cache for Pocket-GL application. Our controller provides significant reconfiguration speedups even when using only 8 blocks. Moreover, for larger cache sizes the speedup increases



(a) Reconfiguration time speedup



(b) Percentage of performance speedup

Fig. 16: Reconfiguration time and % performance speedup of our techniques over a bare system without reconfiguration cache for the Pocket-GL application. Three cache sizes have been evaluated: 8, 16 and 32 blocks for adaptive and non-adaptive mechanism with the three different replacement policies: LRU, Random and LFU

accordingly to the hit ratios of Figure 14. Hence, since the hit ratio for cache size of 32 blocks is approximately 99% for all the configurations, the speedup of our techniques is close to 3.2, which is the upper-bound speedup for this system.

Next, Figure 16b depicts the overall speedup obtained in the execution of Pocket-GL thanks to our controller (note that Figure 16b shows the percentage of overall speedup, whereas Figure 16a shows the speedup in global terms). In this application, the execution time of the tasks outweighs (1.4X) their reconfiguration time on average. That means that completely removing the reconfiguration overhead will achieve a 1.7X speedup, and that a 100% on-chip hit ratio will achieve a 1.4X speedup. Since these weights vary from application to application, the performance improvement of the reconfiguration cache is expected to be highly dependent on the application itself.

In Figure 16b, for large cache sizes, the 99% of hit ratio leads to an almost 40% performance improvement of Pocket-GL, which is in fact the upper limit that our controller can provide for this application. Moreover, even for the smallest size (8 blocks), the best techniques are delivering around a 12% of performance improvement for a very constrained budget. We would like to remark that the baseline organization we are comparing against (off-chip DDR + DMA), is a disadvantageous scenario because the penalty for reconfiguring a block from that memory is very low in comparison to other alternatives such as: reconfiguring from an off-chip DDR2 without DMA, or from a FLASH memory. In these situations the latency of the off-chip memories can be tens or hundreds times worse than the latency of a block RAM. Hence, the proposed controller can provide even better results.

VII. Conclusions

This paper presents a hardware implementation of a controller that efficiently manages the reconfiguration process by taking advantage of on-chip memory resources to apply conguration caching. In order to optimize the use of the on-chip resources, it includes support to divide the configurations into blocks of customizable size. This approach increases the flexibility and reduces the fragmentation in the on-chip memories. In addition, it features an adaptive mechanism and several replacement techniques. Experimental results have demonstrated that the presented mechanism, combined with a LFU replacement policy provides very good results, allowing to maximize the benefits of the on-chip configuration memory. In order to measure the runtime delays and the implementation cost generated by this controller, it has been implemented in a Xilinx^{TM}Virtex-5 FPGA. The results show that the hardware cost is affordable and the delays are very small. Moreover, the maximum operation frequency of the developed controller is greater than 100 MHz, hence it is possible to carry out the reconfigurations at their maximum speed.

References

[1] T. Becker, W. Luk, and P. Y. K. Cheung, "Energy-Aware Optimisation for Run-Time Reconfiguration," in *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 55–62.

[2] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 2, pp. 203–215, 2007.

[3] Xilinx, "Virtex-5 FPGA Configuration User Guide, UG191(V3.10)," 2011.

[4] S. Liu, R. N. Pittman, A. Form, and J.-L. Gaudiot, "On Energy Efficiency of Reconfigurable Systems with Run-Time Partial Reconfiguration," in *Proceedings of the IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, 2010, pp. 265–272.

[5] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[6] 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.

[7] S. Sudhir, S. Nath, and S. Goldstein, "Configuration Caching and Swapping," in *Field-Programmable Logic and Applications (FPL)*, ser. Lecture Notes in Computer Science, 2001, vol. 2147, pp. 192–202.

[8] D. Deshpande, A. K. Somani, and A. Tyagi, "Configuration Caching vs Data Caching for Striped FPGAs," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA)*, 1999, pp. 206–214.

[9] D. Deshpande, A. Somani, and A. Tyagi, "Hybrid Data/Configuration Caching for Striped FPGAs," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1999, pp. 294–295.

[10] J. Clemente, E. Ramo, J. Resano, D. Mozos, and F. Catthoor, "Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–1, 2013.

[11] Xilinx, "Virtex-5 Family Overview, DS100(V5.0)," 2009.

[12] K. Papadimitriou, A. Anyfantis, and A. Dollas, "An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems," *IEEE Transactions on Instrumentation and Measurement (TIM)*, vol. 59, no. 6, pp. 1642–1651, 2010.

[13] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing," *ACM Transactions on Reconfigurable Technology Systems (TRETS)*, vol. 1, no. 4, pp. 21:1–21:23, 2009.

[14] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-Time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 498–502.

[15] R. Bonamy, H.-M. Pham, S. Pillement, and D. Chillet, "UPaRC - Ultra-Fast Power-Aware Reconfiguration Controller," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2012, pp. 1373–1378.

[16] S. Chevobbe and S. Guyetant, "Reducing Reconfiguration Overheads in Heterogeneous Multicore RSoCs with Predictive Configuration Management," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 8:4–8:4, 2009.

[17] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, "A Power Modeling and Estimation Framework for VLIW-based Embedded Systems," in *ST Journal of System Research*, 2001, pp. 26–28.

[18] M. Jayapala, F. Barat, T. Vander, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors," *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 672–683, 2005.

[19] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The Energy Efficiency of IRAM Architectures," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, 1997, pp. 327–337.

[20] J. Resano, D. Mozos, D. Verkest, and F. Catthoor, "A Reconfigurable Manager for Dynamically Reconfigurable Hardware," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 452–460, Sept 2005.

Juan Antonio Clemente received his computer science degree from Universidad Complutense de Madrid (UCM), Madrid, Spain, in 2007; and his Ph.D. in 2011. He is Assistant Professor and Researcher with the GHADIR research Group. Since 2012, he collaborates with the TIMA Laboratory, in Université Grenoble-Alpes, Grenoble, France.

His research interests are: dynamically reconfigurable hardware, FPGA design and task scheduling. Also, his research is focused on the study of Single Event Effects (SEE) tolerance of digital circuits implemented on FPGAs and he is being conducting experiments evaluating the robustness of memories face to neutrons and heavy ions with TIMA Labs and the Laboratoire de Physique Subatomique et de Cosmologie (LPSC), at CNRS-IN2P3 research center.

Rubén Gran graduated in Computer Science from the University of Zaragoza (Spain) an hold his Ph.D in 2010 from the Polytechnic University of Catalonia (UPC, Spain). Since then, he is Assistant Professor in the Department of Computer Science and Systems Engineering at the University of Zaragoza. His research interests are hard real-time systems, hardware for reducing worst-case execution time and energy consumption, microarchitecture and effective programming for parallel and heterogeneous systems.

Abel Chocano received his Computer Science Degree from Universidad Complutense de Madrid (UCM), Spain, in 2014. Upon deciding the topic of his Master's Thesis, which lead to this publication, he felt attracted by hardware design thanks to the tuition of several professors during his first years of his Bachelor's Degree at the Computer Science faculty at UCM.

Nowadays, he works as a web developer at www.destinia.com. He is also a passionate of entrepreneurship.

Carlos del Prado received his Computer Science Degree from Universidad Complutense de Madrid (UCM), Spain, in 2014. Passionate of computers since he was a child, he decided to develop a Master's Thesis in hardware design in order to broaden his knowledge in different fields of Computer Science. This publication is the result of several months of work, always motivated by his professors, relatives and friends.

Nowadays he works as a software developer in Telefónica, thanks to a Talentum grant. He is also involved in several enterprise projects.

Javier Resano received his Bachelor Degree in Physics in 1997, his Master Degree in Computer Science in 1999, and the Ph.D. degree in 2005 at the Universidad Complutense of Madrid (UCM), Spain. Currently he is Associate Professor at the Computer Eng. Department of the Universidad of Zaragoza, and he is a member of the GHADIR research group, from UCM, and the GAZ research group, from Universidad de Zaragoza. He is also member of the Engineering Research Institute of Aragón (I3A). His research has been focused on hardware/software codesign, task scheduling techniques, dynamically reconfigurable hardware and FPGA design.

He has designed hardware accelerators for different fields, including remote sensing and artificial intelligence, and his designs have received several international awards including the first prize in the Design Competition of the IEEE International Conference on Field Programmable Technology in 2009 and in 2010 and the second prize in 2012.