Configuration Mapping Algorithms to Reduce Energy and Time Reconfiguration Overheads in Reconfigurable Systems

Juan Antonio Clemente, *Member, IEEE,* Elena Pérez Ramo, Javier Resano, Daniel Mozos, *Member, IEEE,* and Francky Catthoor, *Fellow, IEEE*

***Abstract*—In spite of the increasing success of reconfigurable hardware, the dynamic reconfiguration can introduce important overheads, both in terms of energy consumption and time, especially when configurations are fetched from an external memory. In order to address this problem, this article presents a configuration memory hierarchy including two on-chip memory modules with different access time and energy consumption features. In addition, we have developed two configuration mapping algorithms that take advantage of these memories to reduce the system energy consumption, while increasing its performance. The first algorithm has been optimized for systems with reduced dynamic behavior, hence it optimizes the system for each given set of tasks. The second algorithm targets dynamic systems where the active tasks change unpredictably. Thus, its objective is also to decrease the pressure on the on-chip memories to reduce capacity conflicts. The presented results will demonstrate that, with the proper management, our configuration memory hierarchy leads to an energy consumption reduction up to 81% with respect to fetching the configurations from the external memory, while keeping the system performance very close to the ideal upper-bound one.**

***Index Terms*—FPGA, Configuration Energy Consumption, Configuration Time Overheads, Configuration mapping.**

I. Introduction

RECONFIGURABLE hardware has emerged as a promising technology that offers a very interesting trade-off between flexibility, reusability, performance and low power for embedded systems. Thus, reconfigurable devices (especially FPGAs) have gained popularity for use in various fields, such as digital signal processing, aerospace, cryptography, and computer vision [1], [2], [3]. However, one of its main drawbacks is that the configuration process that needs to be carried out prior to a task execution in the device 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) [4], [5].

Typically, this configuration process consists of 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 [6]), but also in terms of energy consumption [7]. This trend has been observed especially during the last decade [8]. Thus, at the end of the 20^{th} century, off-chip memory bandwidth almost doubled every two years, but over the past few years, this trend has slowed down significantly, which has led to an increasing gap between off-chip memory bandwidth and the system computing capabilities. 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.

Hence, reducing the energy consumption associated to the configuration process becomes an issue of great importance in order to make reconfigurable systems an interesting alternative to ASICs. A straightforward way to achieve this objective is to store the configurations in on-chip memories optimized for low-energy consumption. Most design libraries for embedded systems offer this kind of memories. For instance, Micron Technology™provides memory families designed not only for low-latency [9], but also for low-power [10]. However, this lower energy features come at the price of a higher configuration memory access time, which may further degrade the performance of the system.

Thus, in order to reduce the energy consumption generated by the dynamic reconfigurations *while still optimizing the system performance*, in this article we propose a configuration memory hierarchy that combines high speed (HS) and low energy (LE) modules. This scheme provides fast reconfigurations when they are especially critical for the system performance, and at the same time, it greatly reduces the average energy consumption generated by the dynamic reconfigurations.

In addition, we have developed two configuration mapping algorithms that take advantage of the special features of this hierarchy, which adapt very well to different task execution

Manuscript received August 23, 2012; revised 05 January, 2013 and May 28, 2013; accepted June 09, 2013. This work was supported by the Spanish Ministry of Education, Culture and Sports under grants TIN2009-09806, AYA2009-13300 and Consolider CSD2007-00050; by the Spanish Government and European ERDF under grant TIN2010-21291-C02-01; by the Autonomous Aragon Government and European ESF under grant gaZ: T48 research group; and by HiPEAC under grant HiPEAC-3, FP7/ICT 287759.

J. A. Clemente and D. Mozos are with the Computer Architecture Department, Universidad Complutense de Madrid, Madrid 28040, Spain (e-mails: ja.clemente@fdi.ucm.es, mozos@fis.ucm.es).

E. Pérez Ramo is with the Business Research Department, Telefónica España, Madrid 28013, Spain (e-mail: elena.perezramo@gmail.com).

J. Resano is with the Computer Engineering Department, Universidad de Zaragoza, Zaragoza 50015, Spain (e-mail: jresano@unizar.es).

F. Catthoor is with the Katholieke Universiteit Leuven, Department ESAC - IMEC, Leuven 3000, Flanders, Belgium (e-mail: catthoor@imec.be).

contexts. On the one hand, the first one has been optimized for systems with only one active task graph, or systems with several active task graphs, but where the switch in the execution from a task graph to another is infrequent. In the remainder of the paper we will refer to these systems as *static systems*. On the other hand, the second algorithm has been optimized for systems with several active task graph competing for the resources, following an unpredictable pattern (referred in the remainder of the article to as *dynamic systems*).

We have validated our techniques both for fine-grain and coarse-grain reconfigurable platforms. The presented experimental results will demonstrate that, with the proper management, the proposed algorithms achieve very significant reductions in the configuration energy consumption and time penalty generated by the task reconfigurations.

The remainder of this article is structured as follows: Section II briefly describes the proposed configuration memory hierarchy. Section III provides an overview of recent works involving energy consumption minimization on reconfigurable systems. Section IV describes the architectural model that has been assumed during the development of the presented techniques, whereas Sections V and VI describe the proposed techniques. Section VII compares them by means of a practical example and Section VIII presents the experimental results. Finally, Section IX concludes the article.

II. The Proposed Configuration Memory Hierarchy

Figure 1.a shows the typical configuration memory hierarchy for reconfigurable hardware, which comprises a reconfigurable fabric that stores the configurations of the tasks that are ready to be executed, plus an off-chip memory where the remaining configurations are stored.

An interesting enhancement to this scheme that allows to reduce the energy consumption and that has been introduced for reconfigurable devices [11] is shown in Figure 1.b. It consists on adding a smaller on-chip *Internal Configuration Memory* between the off-chip memory and the reconfigurable fabric. This on-chip memory is sometimes called configuration cache, although normally it is not a real cache, but a SRAM controlled by software (i.e., a scratchpad). If used properly, it may drastically improve the performance of the memory hierarchy, as well as the energy consumption, since it prevents the system from accessing a high-capacitance off-chip bus [12]. These on-chip memories are frequently high-speed (HS) SRAMs. However, unfortunately these memories still generate an important percentage of the total energy consumption of the embedded system.

For this reason, most memory vendors have developed memories with low energy (LE) consumption features for embedded systems. Nevertheless, the energy savings achieved always come at a cost of additional delays, which make the LE memories slower than the HS ones, but still much faster than an off-chip memory.

Frequently in an embedded system, a few tasks are especially critical for the performance of the system, and including additional delays to their execution is not acceptable. Hence

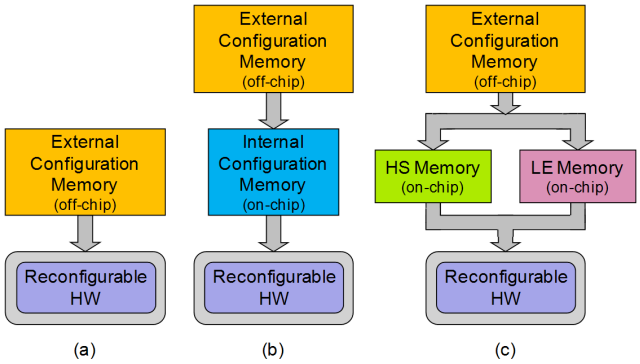


Fig. 1. (a) Typical configuration memory hierarchy for fine-grain devices. (b) Enhanced memory hierarchy introduced for coarse-grain devices. (c) The proposed configuration memory hierarchy. In (b) and (c) the *Reconfigurable HW* is also connected to the off-chip memory by means of a dedicated connection, which is not shown in the figure for simplicity

these tasks must be loaded from a HS configuration memory. In contrast, other tasks are less critical, and a small additional delay in their execution is completely acceptable. Hence, it is interesting to take advantage of this flexibility in order to reduce the energy consumption of the system. This can be achieved by loading those tasks from a LE memory.

Since designers need both HS and LE features, in this article we propose to include two different memory modules in the configuration memory hierarchy: one optimized for HS and other optimized for LE. Thus, we are potentially providing both features in the same configuration memory hierarchy. The goal of this scheme is to reduce the energy consumption of the system, while maintaining high performance. Figure 1.c depicts this configuration hierarchy memory scheme.

The proposed scheme presents a new challenge, because it is necessary to decide where to load each involved configuration from, the HS memory, the LE one or to fetch it directly from an external memory. In this work we present an extension of the approach initially presented in [13] by our research group with two novel configuration mapping algorithms that tackle this new challenge by making these decisions automatically and that adapt to different task execution contexts. In our previous work [13] we already proposed to include an on-chip configuration memory layer in order to reduce both the reconfiguration delays and the energy overheads, presenting a similar figure as Figure 1. In that article we attempted to demonstrate the potential benefit of including this memory. To this end, we assumed that the on-chip configuration memory was big enough to store all the requested configurations and we analyzed the benefits both in terms of performance and energy consumption.

However, this was a simplistic assumption because, in embedded systems on-chip memory resources are frequently very constrained. Hence, in this article we assume that the configurations are stored initially on an off-chip memory, and that the system includes an intermediate on-chip memory layer that can store a few configurations. In addition, we propose two novel techniques to efficiently manage this intermediate level taking into account its size and the dynamism of the

applications. These two approaches are described in greater detail in Sections V and VI.

III. State-Of-The-Art

Many research groups have developed techniques that can significantly reduce the reconfiguration time overhead [14]. A straightforward way to attain this objective is to use partial reconfiguration, which is supported in many FPGAs. The idea is to reconfigure only a certain region of the device whereas the remaining area remains unaltered. This is especially interesting for multi-tasking systems where several tasks can be executed simultaneously in the reconfigurable device [15].

Another way to reduce the reconfiguration time overhead is to develop new architectures. Good examples are multi-context FPGAs [16], which permit loading a new configuration while another one is being executed, and especially coarse-grain ones [17], which feature a reduced configuration memory access time, but at the cost of reducing their programming flexibility. The techniques presented in this article can be used both for FPGAs and coarse-grain devices, and the experimental results demonstrate that they are useful in both cases.

In embedded systems, applications are frequently represented as task graphs. Hence another good way to reduce the reconfiguration time overhead is to overlap the computation of one or several nodes of the graph with the reconfiguration of their successors. The idea of prefetching the configurations, i.e. loading them in advance, was initially proposed in [18]. After that, several schedulers developed for reconfigurable systems have included their own prefetch technique. Some relevant examples are [19], where the authors propose a list-based approach; [20], where the authors propose to add more reconfiguration controllers to carry out several reconfigurations in parallel; [21] and [22], where the authors present specific scheduling techniques to take full advantage of the data-parallelism of a given application; [23], where the authors propose a novel prefetch technique based on probabilities; and [24], where the authors present several algorithms to select which tasks are assigned to hardware at run-time taking into account the needs of each application, the available resources, and the reconfiguration time overheads. Our techniques are fully compatible with a prefetch technique, and in fact, our simulation environment includes a scheduler that prefetches the configurations (this scheduler is described in [25]).

Another popular way to reduce the reconfigurations time overheads is to compress the configurations in order to reduce its size. Some relevant techniques are [26] and [27]. With this approach configurations are fetched faster. However, they have to be decompressed before writing them in the device, and this may involve additional execution-time and energy penalties. To reduce these penalties some authors have proposed to include specific hardware support to efficiently decompress the configurations. Again, these techniques are fully compatible with and complementary to our work.

Some works have focused on optimizing the management of the memories that store the configurations. Most of them assume that the memory hierarchy for configurations includes two levels: an external memory and the reconfigurable device. The focus of these works is to manage the latter trying to reduce the number of reconfigurations needed by applying replacement techniques that maximize the configuration reuse. The first work that proposed a replacement technique for FPGAs was [28]; [25] is another interesting approach that proposes a replacement technique that interacts with a task scheduler in order to reduce the reconfiguration time overhead; [29] is an interesting article that analyzes the relationship between several scheduling algorithms and their impact on the number of reconfigurations required to execute a set of hardware tasks. In addition, it proposes three new hardware scheduling algorithms with different replacement policies. Once more, our work is compatible with these techniques, since we are not addressing the management of the configurations stored in the device, but proposing to add an additional intermediate level. Hence the management of the configurations stored in the device is fully complementary to our work related to this additional level. Compared to our own earlier work [13] we also present two novel techniques which are specifically developed to optimize the use of that level.

Two interesting recent articles have analyzed the impact of loading the configurations at run-time from an external memory. The experimental results presented in [30] conclude that the reconfiguration speed is three orders of magnitude worse than the peak reconfiguration speed of the platform if a flash memory is used to store the configurations. In [31] the authors analyze the impact of the reconfigurations in the performance of the High-Performance Reconfigurable Computer Cray XD1, which includes one or several FPGAs and a conventional multiprocessor system. According to their measurements, loading the configurations from an external memory is 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. Hence, including an on-chip intermediate level that works as a configuration cache can be a straightforward solution. However few works have analyzed this option. One of them is [32], where the authors present a heterogeneous reconfigurable system that includes several reconfigurable processors. They also propose to include a configuration memory cache for each processor, as well as a configuration prefetch approach.

The main focus of all the works mentioned in this section is to reduce the reconfiguration time overhead. However, 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 [33], [34]. And this is true for fine-grain [35] and coarse-grain [36] architectures, as long as frequent reconfigurations are demanded. Hence, reconfigurable systems with energy-efficient reconfigurations are needed and the reconfiguration energy overhead must be also taken into account [13].

IV. Target Architecture

In this work we have adopted the reference hardware architectural model shown in Figure 2. This architecture is assumed to be implemented using reconfigurable hardware and, as it

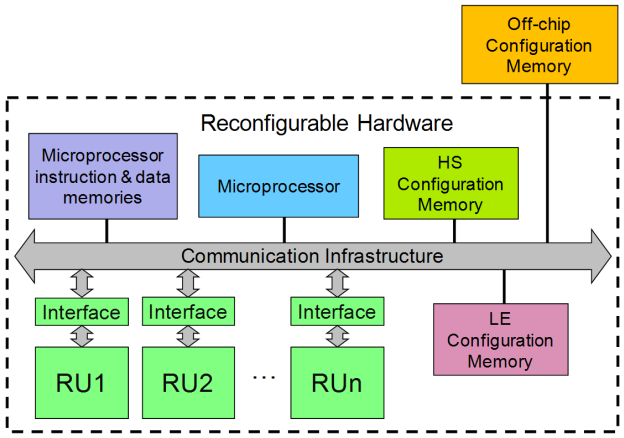


Fig. 2. Target architecture

will be explained in the remainder of this section, it includes the configuration memory hierarchy that was presented in Section II of this article.

This platform is a microprocessor-based system that contains a set of cores that are connected by means of a *Communication infrastructure*, which could be implemented as one or several buses or as a Network-on-a-Chip (NoC) [37].

The *Microprocessor* manages the general operation of the system, and it uses a couple of memories to store the microprocessor instructions and data, respectively. The system also contains a set of elementary reconfigurable regions called *Reconfigurable Units* (*RUs* in the figure), which represent the smallest piece of area that can be dynamically reconfigured at a time. The functionality of these RUs can be programmed at run-time through partial reconfiguration, but only one of them can be reconfigured at a time (this is a realistic assumption for state-of-the-art commercial reconfigurable devices, since all of them currently only feature one configuration circuitry). All the RUs are wrapped with a fixed *Interface* that provides the basic operating system (OS) and communication functionality. With this support, each RU can independently execute a task (in this work, we assume that only one task is assigned to each RU). In addition, since they are connected through the same communication infrastructure, they can communicate with the other processing elements with similar latencies. Finally, the system also includes the configuration memory hierarchy that has been described in Section II. As previously explained, this hierarchy comprises two on-chip memories (with HS and LE features, respectively); plus an off-chip configuration memory, which is external to the reconfigurable hardware.

Such a system can be implemented on last-generation FPGAs, such as Xilinx™Virtex-7 and Zynq-7000 EPP devices [38], [39], or Altera™ones [40], [41], although it is also suitable for implementation on coarse-grain systems [42]. To this end, vendors provide specific design tools to develop custom SoCs. On the one hand, the Xilinx™EDK development tool [43] can be used to develop a processor-based system; for instance, using the Microblaze soft-processor [44] or the ARM hard-processor [39]. In the latest versions of this software, several options for the communication infrastructure are possible: for instance the AMBA Advanced Extensible Interface (AXI) communication infrastructure [45] can be used to attach the computational cores, and the LMB bus [46], for the memories. The RUs can be implemented as peripherals, and their dynamically partial reconfiguration can be easily managed by using the Plan Ahead tool [47]. On the other hand, the Altera™Quartus-II software [41] allows the development of systems based on the Nios®II soft-processor core [48].

In any case, it is very important to underline that the configuration mapping algorithms proposed in this article do not rely on the specific reconfigurable device that is finally used or the final implementation of the system, as long as it is compatible with the architectural model requirements depicted in Figure 2.

V. Static Configuration Mapping Algorithm

This section explains the configuration mapping algorithm proposed for static systems in greater detail. As mentioned above, this algorithm has been optimized for the execution of only one active task graph. It is also possible to deal with more than one task graph when they are executed always following the same schedule. For that case, it is enough to merge the graphs adding some additional edges. Hence, we assume that one task graph is going to be executed several times without sharing any resource with others. The system may have to deal with several task graphs, and switch from one to another when needed, but in this case we assume that these switches are not frequent. Hence, the objective of this algorithm is to optimize the execution of each task graph separately neglecting the context switch overhead.

This algorithm receives as input a task graph, and obtains a mapping of its tasks on the proposed configuration memory hierarchy. *It aims at reducing the energy consumption of the system generated by the dynamic reconfigurations, while keeping the same performance as if all the tasks of the task graph were fetched from the HS memory*.

It is also important to note that *all the computations of this algorithm (as well as the dynamic one, explained in Section VI) are carried out at design time*. The only computations proposed in this article that are carried out at run time belong to the *Configuration Replacement Technique*, which is explained in Section V.C.

In our scheme, we consider that the applications are modelled using a two-level hierarchy: on the upper level, applications are described as a set of tasks graphs interacting among them dynamically and in a non-deterministic way. The lower level describes each task-graph as a set of tasks, where each task is a piece of code with enough entity to be separately assigned to a reconfigurable unit or processor. This model only allows a limited dynamic behaviour of applications inside each task graph. And therefore, the main part of the dynamic behaviour, and above all the most unpredictable behaviour, must be kept out of the task graphs. The aim of this division is to separate the part of the application specification that must be managed and optimized at design time, from the one that can be managed at run time. These task graphs are the inputs of our mapping algorithms.

Algorithm 1 shows the pseudo-code of this approach. As this algorithm indicates, it is divided into three important steps, which are explained in detail in the next subsections.

Algorithm 1 The proposed static configuration mapping algorithm

```
 1: for each task graph do

#Step 1: Task-graph profiling
 2:    compute_criticalities ();
 3:    assign_all_tasks_2_HS;
 4:    schedule_reconfigurations (&reference_sch);
 5:    assign_all_tasks_2_LE;
 6:    schedule_reconfigurations (&current_sch);
 7:    compare (reference_sch, current_sch, &penalty, &task);
 8:    while (penalty > 0) do
 9:       assign_2_HS (&task);
10:       schedule_reconfigurations (&current_sch);
11:       compare (reference_sch, current_sch, &penalty, &task);
12:    end while;

#Step 2: Move tasks from HS to LE
13:    while size (HS_tasks) > size(HS) do
14:       task = select_least_critical_task (HS_tasks);
15:       assign_2_LE(task);
16:    end while;

#Step 3: Move tasks from LE to HS and the external memory
17:    while ((size (LE_tasks) > size (LE) and size (HS_tasks <
       size (HS))) do
18:       task = select_most_critical_task (LE_tasks);
19:       assign_2_HS (task);
20:    end while;
21:    while (size (LE_tasks) > size (LE)) do
22:       task = select_least_critical_task (LE_tasks);
23:       assign_2_EXT (task);
24:    end while;
25: end for;
```

A. Step 1: Task-Graph Profiling

The basic idea of this step (Lines 2-12) is to identify which tasks generate delays due to their reconfiguration since they are the initial candidates for the HS memory. This step also characterizes each task with a metric that measures the impact of its reconfiguration in the execution of the graph. We have called that metric *criticality*. Let *TG* be a task graph, *T* the set of tasks of *TG*, and *t* a task $\in$ *TG*, the criticality of *t* is computed as follows: *The criticality of a task t is the difference between the execution time of TG when all its tasks are fetched from the external configuration memory, and the execution time when t is fetched from HS (keeping the remaining tasks in the external memory).* Thus, the more critical a task is, the more important is to fetch it from an on-chip memory.

In the previous definition, in order to calculate the execution time of *TG* in the system, it is necessary to use a task-graph scheduler that assigns the tasks to the RUs. Any task-graph scheduler can be used to compute the criticality of the tasks of *TG*, as long as it respects two basic rules: (1) only one task can be assigned per RU and (2) the configurations of the tasks must be carried out sequentially. In our experiments, we have used two external schedulers, depending on whether the involved task graph is executed on fine grain [25] or coarse grain [42] reconfigurable devices (Section VIII will provide more details about these two simulation environments).

After computing the criticality of each task (Line 2), an iterative process identifies which tasks must be assigned to HS in order to optimize the system performance. Thus, the tasks are firstly assigned to HS (Line 3) in order to obtain the ideal optimal mapping for performance (in this step the capacity of the HS memory is not taken into account, since this is not the final mapping but a profiling step). Next, the external task-graph scheduler is invoked in order to obtain a *reference schedule* (*reference_sch*, Line 4), which will be used as a reference during the remainder of the algorithm.

Then, the algorithm looks for a mapping with the same performance as *reference_sch*, but with the maximum number of configurations assigned to LE. For this purpose, it firstly assigns all the tasks of the task graph to LE (Line 5) and invokes the external scheduler in order to obtain a *current schedule* (*current_sch*, Line 6).

Next, both schedules are compared (*compare* function, Line 7) to identify if any extra *penalty* has been generated when the tasks are assigned to LE instead of HS. In this context, *the term penalty is the difference between the execution time of current_sch and reference_sch*. This term can only be greater than or equal to zero, since *reference_sch* is used as upper bound regarding performance. Thus, if *penalty* is greater than 0, the function returns in the variable *task* the reconfiguration that generates the greatest penalty; i.e., the reconfiguration that introduces the largest delay in the execution of the task graph.

Finally, the *while* loop (Lines 8-12) iteratively assigns the selected *task* to HS instead of LE (Line 9), schedules again the reconfigurations in order to update *current_sch* (Line 10) and compare both schedules (Line 11). This loop iterates as many times as necessary until all the tasks that generate any penalty when assigned to the LE memory are moved to HS.

This initial step assumes that there is always enough available space in the HS and LE memories to store as many configurations as necessary. However, this assumption is not valid in most of the cases, since on-chip memories have a very limited capacity in current reconfigurable devices. Thus, the next steps of this algorithm refine this initial solution.

B. Step 2: Move Tasks from HS to LE

The objective of this step (Lines 13-16) is to move as many tasks as necessary from HS to LE until the tasks assigned to HS do not exceed its capacity. Thus, the *while* loop iteratively selects the *least critical task* initially assigned to HS (Line 14) and assigns it to LE (Line 15). The idea is to keep in HS the tasks that generate the greatest time overheads unless they are fetched from that memory.

C. Step 3: Move Tasks from LE to HS and the External Memory

Finally, this step (Lines 17-24) moves the exceeding tasks assigned to LE to HS and to the external memory. For this purpose, the first *while* loop checks if the tasks assigned to LE exceed the capacity of this memory, and if the HS memory is

not full (Line 17). If this condition is true, this loop selects *the most critical task* from the ones assigned to LE (Line 18) and assigns it to HS (Line 19). Again, the idea here is to move to HS those tasks that generate the greatest time overheads. This process is repeated as many times as necessary until all the tasks assigned to LE fit in that memory, or no more tasks can be assigned to HS. Note that this loop is executed only in case there was some available space in HS after the execution of the previous step.

Finally, if after the execution of the first *while* loop, the size of the tasks assigned to LE is still greater than the capacity of that memory, the second *while* loop (Lines 21-24) iteratively selects the least critical task from LE (Line 22) and assigns it to the external memory (Line 23). This process is repeated while all the tasks assigned to LE do not fit in that memory (Line 21).

As it could be observed along Subsections V.A, V.B and V.C, the static configuration mapping algorithm takes full advantage of the capacity of the on-chip modules in order to achieve important energy savings. However, as the experimental results will demonstrate, *this approach can only achieve these objectives when only one task graph (or a statically merged task graph) is active in the platform*. When several task graphs are active in a dynamic fashion, this solution may not efficient. The reason is that this algorithm assigns the maximum number of tasks to the on-chip memories, which may lead to a great number of configuration replacements. This, in turn, may lead to configuration thrashing problems where the configurations are stored in the on-chip memories and replaced before being used again, making the use of such on-chip memories inefficient. Thus, in order to prevent this situation we have developed another mapping algorithm, which exerts less pressure on the on-chip memories. This algorithm is explained in greater detail in Section VI.

D. Replacement Technique

Since it is likely that the number of configurations assigned to the on-chip memories exceed the size of these memories, we need to include a replacement technique that decides which configuration must be replaced in order to fetch a new one.

For this purpose, we have also developed a replacement technique that is used both in the configuration static and dynamic mapping algorithms (the latter is explained in the next section). This replacement technique is based on the well-known LRU (*Least Recently Used*) policy, but it includes a small modification: *a task T of a given task graph TG must not replace other tasks belonging to TG, unless all the possible victims belong to TG*. Thus, the replacement technique firstly applies LRU, and if the selected victim belongs to *TG*, this process is repeated and the next victim is selected until it finds a victim from other task graph or no more victims are available. Note that, *if all the possible victims belong to TG, then one of them is selected simply by using LRU*. As the experimental results will demonstrate, when dealing with task graphs this policy frequently performs better than LRU. The reason is that the latter is very sensitive to thrashing problems, which lead to additional reconfiguration delays and a higher energy consumption, especially for small memories. Indeed, if a task from the current task graph is replaced, it is very likely that the system will request to load again that task very soon, since periodic tasks are very frequent in embedded systems. Hence it is a good idea to assign more priority to the tasks from the current task graph.

Finally, note that the computational complexity of this technique is linear. Thus, the run-time overhead introduced by the decisions made by this policy is negligible with respect to the reconfiguration memory access times, which are in the order of milliseconds (for fine-grain task graphs) or microseconds (for coarse-grain task graphs), as it will be explained in Sections VII and VIII.

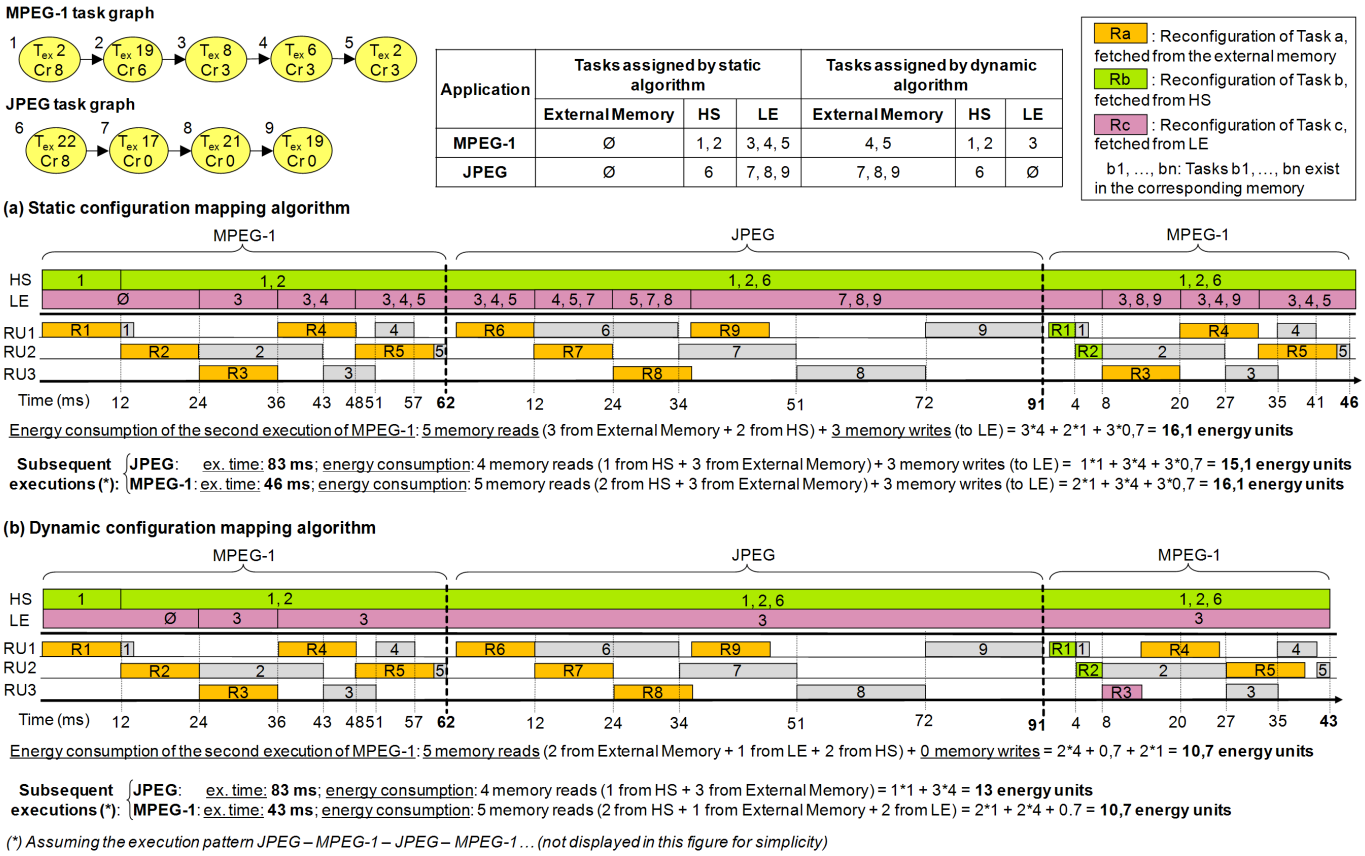
Algorithm 2 The proposed dynamic configuration mapping algorithm

```
 1: for each task graph do

#Step 1: Divide the tasks into HS and LE
 2:     assign_all_tasks_2_HS;
 3:     schedule_reconfigurations (&reference_sch);
 4:     assign_all_tasks_2_LE;
 5:     schedule_reconfigurations (&current_sch);
 6:     compare (reference_sch, current_sch, &penalty, &task);
 7:     while (penalty > 0 and (size (HS_tasks) + size (task) < size
        (HS))) do
 8:         assign_2_HS (task);
 9:         schedule_reconfigurations (&current_sch);
10:         compare (reference_sch, current_sch, &penalty, &task);
11:     end while;

#Step 2: Refine the previous solution and divide the tasks
into HS, LE and the external memory
12:     reference_sch = current_sch;
13:     assign_LE_tasks_2_EXT;
14:     schedule_reconfigurations (&current_sch);
15:     compare (reference_sch, current_sch, &penalty, &task);
16:     while (penalty > 0 and (size (LE_tasks) + size (task) < size
        (LE))) do
17:         assign_2_LE (task);
18:         schedule_reconfigurations (&current_sch);
19:         compare (reference_sch, current_sch, &penalty, &task);
20:     end while;
21: end for;
```

VI. Dynamic Configuration Mapping Algorithm

The second configuration mapping algorithm that we present in this article has been designed for dynamic scenarios. In other words, scenarios where several task graphs can be active simultaneously, and in which the actual sequence of task graphs that will be executed at run-time is unknown at design-time.

Algorithm 2 shows the pseudo-code of this algorithm. *The basic idea is to assign the minimum possible number of tasks to the on-chip memories, while achieving the same performance as if all the tasks were assigned to HS*. Hence, this algorithm also attempts to improve the performance (as our static algorithm does), but once no further performance improvements can be achieved, our dynamic algorithm stops assigning more tasks to the on-chip memories in order to reduce the pressure exerted on these memories. This is desirable



Application	Tasks assigned by static algorithm			Tasks assigned by dynamic algorithm		
	External Memory	HS	LE	External Memory	HS	LE
MPEG-1	Ø	1, 2	3, 4, 5	4, 5	1, 2	3
JPEG	Ø	6	7, 8, 9	7, 8, 9	6	Ø

Ra : Reconfiguration of Task a, fetched from the external memory
Rb : Reconfiguration of Task b, fetched from HS
Rc : Reconfiguration of Task c, fetched from LE
b1, …, bn: Tasks b1, …, bn exist in the corresponding memory

Fig. 3. Execution of the task-graph sequence: MPEG-1 - JPEG - MPEG-1 - JPEG - MPEG-1... in a system with 3 RUs and High-Speed and Low-Energy memories with a capacity of 3 tasks each, for the proposed static configuration mapping algorithm (a) and the configuration dynamic mapping one (b)

for dynamic systems since if many tasks are assigned to the on-chip modules, the system may fall into a thrashing problem with constant task replacements that may lead to an important performance degradation, and energy penalization.

This algorithm is divided into two steps: The first one (Lines 2-11) assigns as many tasks as necessary to HS (*but not a single one more*) in order to meet the temporal constraints (*penalty* > 0), and ensuring that the tasks assigned to HS do not exceed the capacity of that memory ($size(HS\_tasks) + size(task) < size\ (HS)$). Thus, at the end of this process, a number of tasks are mapped to HS (without exceeding its capacity), while the remaining ones are mapped to LE.

Next, the second step (Lines 12-21) applies again the same algorithm on the output of the first step, but this time dividing the tasks previously assigned to LE between LE and the external memory. For this purpose, it starts considering the output schedule in the first step as the new reference schedule (*reference_sch*) (Line 12). Then, it generates a new schedule assuming that all the configurations that, in the reference schedule, were fetched from LE memory are now fetched from the external memory (Lines 13-14). Next, it compares both schedules (Line 15), in order to identify if any additional delay has been generated in the new one. If this is true, the *while* loop (Lines 16-20) assigns as many tasks as necessary to LE from the external memory, while the penalty generated in *current_sch* is greater than zero and these tasks fit in the LE memory.

VII. Static vs. Dynamic Configuration Mapping Algorithms: A Practical Example

As previously explained in Section V, our static algorithm has been designed to take the most advantage of the presented memory hierarchy. Thus, the amount of tasks assigned to HS and LE is maximized, thereby minimizing the energy consumption generated by the reconfigurations, while still keeping the desired performance.

However, this algorithm only achieves good results in static scenarios; i.e., when only one task is going to be executed in the system. In contrast, in dynamic scenarios (i.e., if several task graphs are active at the same time), the static algorithm can suffer the configuration thrashing problem, since many tasks compete for the usage of the on-chip memories. We have overcome this situation with our dynamic configuration mapping algorithm (Section VI). In this section we illustrate this situation by means of a practical example.

The proposed example is depicted in Figure 3. In this case we are executing the task-graph sequence MPEG-1 - JPEG - MPEG-1 - JPEG - MPEG-1... in a system with 3 RUs and a configuration memory hierarchy in which both HS and LE have a capacity of 3 configurations. In this example we assume that both memories are initially empty. In addition, the memory access time is 12, 4 and 6 milliseconds and

the energy consumption is 4, 1 and 0.7 energy units for a task being fetched from the external, HS and LE memories, respectively. We have obtained these realistic figures from experimental measurements using CACTI 4.0 [49], a well-known simulator tool for cache memories that has introduced in its latest versions different technology models that can be used to simulate either HS or LE SRAMs.

The upper part of the figure shows the involved task graphs, along with the execution times (in milliseconds) and criticalities of their nodes (depicted in the figure as T_{ex} n and Cr m, respectively). The table shows the mappings obtained for these task graphs using the static and dynamic algorithms. Thus, for instance, for MPEG-1, the static algorithm assigns the configurations of Tasks 1 and 2 to HS since they are the most critical ones (with a criticality of 8 and 6, respectively), whereas the remaining ones are assigned to LE. On the contrary, again for MPEG-1, the dynamic algorithm assigns the configuration of Task 3 to LE, and those of Tasks 4 and 5 to the external memory. The reason is that fetching only one of these three tasks to LE suffices to reduce the latency incurred in the reconfiguration of these three tasks. In this way, this approach exerts less pressure on the LE memory.

Next, Figures 3.a and 3.b show the execution of the sequence MPEG-1 - JPEG - MPEG-1 - JPEG - MPEG-1... using both configuration mapping algorithms, respectively.

For the execution using the configuration static algorithm (Figure 3.a), and for the first execution of MPEG-1, all its tasks are firstly fetched from the external memory since both HS and LE are initially empty. Consequently, 5 configuration misses occur during this execution. Since our memory hierarchy behaves as a typical cache, *a configuration miss also involves writing that configuration in the corresponding memory*. Therefore Tasks 1 and 2 are written to HS, and Tasks 3, 4 and 5, to LE. Note that the writes in the on-chip memories are carried out at the same time that the corresponding tasks are sent to the reconfiguration controller. In other words, each configuration is loaded once, and it is sent to the on-chip memory and the reconfiguration controller simultaneously. Hence, storing the configuration in the on-chip memory does not introduce any additional delay.

Similarly, during the first execution of JPEG, Task 6 is written to HS and Tasks 7, 8 and 9, to LE. Note that in the latter case, these tasks replace Tasks 3, 4 and 5 (from MPEG-1), respectively, since the maximum occupancy of LE is 3.

Next, for the second execution of MPEG-1, Tasks 1 and 2 are fetched directly from HS, since they still remain in that memory as a consequence of the previous execution of MPEG-1. Hence their reconfiguration time is drastically reduced from 12 to 4 milliseconds. However, Tasks 3, 4 and 5 have to be fetched again from the external memory since they were replaced in LE during the previous execution of JPEG. Hence their configuration time is again 12 milliseconds and they consume 4 energy units per memory access. In addition, these configurations have to be copied again to the LE memory, which generates an additional energy consumption of 3*0.7 = 2.1 energy units.

In this first case, we can observe that if the system uses the static algorithm, the execution time of the second instance of MPEG-1 is 46 milliseconds. In addition, the energy consumption generated by the reconfigurations during the second execution of MPEG-1 is 16.1 energy units, which correspond to 2 and 3 tasks being fetched from HS and the external memory, respectively; and 3 tasks being written to LE.

For subsequent executions of the task graphs JPEG and MPEG-1 (explained, but not graphically displayed in the figure for simplicity), the execution times are 83 and 46 milliseconds respectively, since Tasks 1, 2 and 6 are fetched from HS and Tasks 7, 8, 9 and 3, 4, 5 are continuously replaced in LE during the execution of the corresponding task graph (and hence, fetched from the external memory and also stored in LE). On the other hand, the energy consumption is 15.1 and 16.1 energy units for JPEG and MPEG-1, respectively.

Finally, Figure 3.b shows the same execution as in Figure 3.a, but this time using the dynamic configuration mapping algorithm. In this case, *only Tasks 4 and 5 (from MPEG-1) are assigned to LE*. This prevents these tasks from being replaced during the execution of JPEG, which allows to fetch them from LE every time MPEG-1 is executed. Consequently, the total execution time of the second instance of MPEG-1 is reduced by 3 milliseconds (from 46 to 43 milliseconds, which means a reduction of almost 7%). In addition, fewer memory transactions are carried out on the configuration memories, therefore the energy consumption for the second instance of MPEG-1 is reduced from 16.1 to 10.7 energy units (which means a reduction of 33.6%).

In this case, for subsequent executions of JPEG and MPEG-1, the execution times are 83 and 43 milliseconds, respectively, since Tasks 4 and 5 are always fetched from LE. In addition, the energy consumption is 13 and 10.7 energy units for these two task graphs, respectively. This shows that not only the energy consumption is reduced for MPEG-1, but also for JPEG; in this case, from 15.1 to 13 energy units (which means a reduction of 14%).

In this example we have illustrated that in dynamic scenarios, the dynamic configuration algorithm achieves better results than the static one, both in terms of energy consumption and performance. However, it is also important to point out that *the static algorithm is still more suitable for static scenarios*, since if only one task graph is active, *or* the configurations of all the active task graphs fit in the HS and LE memories, the static algorithm achieves better energy-consumption reductions (this can easily be seen, for instance, by tuning the size of HS and LE to 10 configurations each in the example of Figure 3). We will clearly demonstrate these points with the experimental results presented in the next section.

VIII. Experimental Results

This section evaluates the two mapping techniques proposed in this article. For this purpose, we have carried out two different representative experiments: for static and dynamic environments, respectively. For each one of them, we have compared the performance and the energy consumption of both approaches, for the execution of a set of realistic benchmarks, extracted from state-of-the-art applications designed for

TABLE I
FEATURES OF THE MEMORY MODULES FOR FINE-GRAIN RECONFIGURABLE SYSTEMS

Memory modules	*Memory access time for each configuration*	*Normalized energy consumption*
On-chip HS	4 ms	1
On-chip LE	6 ms	0,7
External memory	12 ms	4

TABLE II
FEATURES OF THE MEMORY MODULES FOR COARSE-GRAIN RECONFIGURABLE SYSTEMS

Memory modules	*Memory access time for each configuration*	*Normalized energy consumption*
On-chip HS	6 μs	1
On-chip LE	9 μs	0,7
External memory	18 μs	4

coarse-grain and fine-grain reconfigurable platforms.

Thus, Subsection VIII.A firstly describes the experimental setup that has been used to obtain the results. Next, Subsections VIII.B and VIII.C show how the two proposed approaches behave in static and dynamic environments.

A. Experimental setup

First of all, Tables I and II show the features of our memory modules, both for fine-grain and coarse-grain devices, respectively. In both tables, Column 2 shows the memory access time per configuration for each kind of memory (either HS, LE or the external one), whereas Column 3 shows their energy overhead, which in this case has been normalized to a memory access to a configuration stored in HS.

The data shown in tables I and II are based on the data obtained with CACTI 4.0 [49]. We did not want to use the results of this simulator as absolute values, since the actual features of a memory vary considerably depending on the technology used. Instead of that, we have used them to obtain the relationship in the time and energy consumed per memory access for different representative memories. For that reason, we have used normalized values for the energy consumption. For our experiments, we have selected an external memory of 1 MB, and memories HS and LE with a capacity of 64 kB, since, according to our measurements, these sizes were the ones that offered the best trade-off between memory capacity, and energy consumption and memory access time for each configuration. In this case, the HS module is 50% faster, but consumes 30% more energy per access than the LE one. In addition, HS is 3 times faster and 4 times more energy efficient than the external memory.

Also note that we are assuming that the normalized energy consumption of the three memories is exactly the same in both tables; i.e., both for fine-grain and coarse-grain devices.

In order to carry out the experiments presented in the following subsections, we have integrated our configuration mapping algorithms into a couple of external simulation environments. These two platforms are the same schedulers that were used in our configuration static and dynamic mapping algorithms (Sections V and VI). Thus, for the experiments regarding the fine-grain task graphs, we have used our previous scheduler for reconfigurable systems [25]. On the other hand, for the experiments regarding the coarse-grain task graphs, we have used the Configurable and Reconfigurable Instruction Set Processor scheduler (CRISP) [42].

On the one hand, [25] is a task scheduler for reconfigurable systems that deals with task graphs. The input of this scheduler is a sequence of events that triggers the execution of a set of task-graphs and a description of these tasks, including their execution time, their reconfiguration latency, and the RUs. The simulator assigns the tasks to the RUs and schedules their execution applying some optimization techniques to reduce the reconfiguration time overhead, such as configuration prefetch (which attempts to load the configurations in advance), and configuration reuse, which identifies when a task can be reused from a previous execution. It also applies an intelligent mapping algorithm in order to promote the task reuse.

On the other hand, CRISP [42], is an academic simulator that can be used to simulate customized coarse-grained recongurable instruction set processors. It was designed to explore the impact of the instruction memory hierarchy on the coarse-grain processors. In this case the coarse grain processor is an array of functional units and a configuration is a sequence of instructions, typically a loop, compiled for that array. Before executing these instructions they must be loaded on some specific on-chip memories. This tool allowed us to define a custom instruction memory hierarchy defining the latencies and the size of the instruction memories. Then, we were able to compile and execute C code in order to obtain several performance metrics.

As it has been explained in the Related Work Section, these two approaches, as well as the other cited techniques proposed in the literature, are compatible with the work that we present in this article since they deal with different optimization opportunities. Refs. [25] and [42] optimize the way configurations are loaded on the reconfigurable device, whereas the algorithms presented in this article optimize the way configurations are stored on the on-chip configuration memories.

These two works assume that all the RUs of the system have the same size. Many other schedulers for reconfigurable systems [19], [50], [51] work under this assumption since it simplifies the run-time management of the configurations. For that reason, the time needed to write or read a configuration on our memories is constant. However, the techniques presented in this article could easily be applied to systems with RUs with different size. In that case, we would only need to add an additional parameter indicating the size of each configuration, and to compare this size with the available space in order to know if a configuration can be loaded in a given memory.

In this work we have used [25] and [42] for practical reasons; however, any other task-graph schedulers could have been used at this point as long as they are compatible with the target architectural model described in Section II.

TABLE III
TIME OVERHEAD GENERATED BY THE PROPOSED STATIC AND DYNAMIC CONFIGURATION MAPPING ALGORITHMS, FOR STATIC ENVIRONMENTS

Granularity	*Task graphs*	*Number of tasks*	*Ideal execution time*	*External time penalty*	*LE time penalty*	*HS time penalty*	*Generated time penalty (static & dynamic)*
Fine Grain	MPEG-1	5	37 ms	+68%	+27%	+16%	+16%
	JPEG	4	79 ms	+15%	+8%	+5%	+5%
	Hough	6	94 ms	+13%	+6%	+4%	+4%
	Parallel JPEG	8	54 ms	+118%	+26%	+7%	+28%
Coarse Grain	DSP_dot_prod	3	32 us	+71%	+28%	+19%	+19%
	DSP_vec_sumq	2	32 us	+56%	+28%	+19%	+19%
	DSP_q15_tofl	3	5645 us	+0.32%	+0.23%	+0.11%	+0.11%
	DSP_neg_32	3	109 us	+18%	+8%	+8%	+8%
	DSP_min_val	3	114 us	+18%	+8%	+8%	+8%
	DSP_dotp_sqr	3	32 us	+70%	+28%	+28%	+28%
	DSP_blkmove	3	109 us	+18%	+8%	+8%	+8%
	Average			+37.55%	+15.91%	+11.09%	+13%

B. Static vs. Dynamic Mapping Algorithms in Static Environments

In a first set of experiments, we have evaluated the energy and time reconfiguration overheads generated by the two proposed approaches in static environments; i.e., when only one active task graph is executed in the system. Thus, for this experiment we evaluate both algorithms for the repeated execution of the same task graph.

On the one hand, the evaluated applications for fine-grain devices have been: a sequential and a parallel version of the JPEG decoder, an MPEG-1 encoder, and a patter recognition application that applies the Hough transform over a matrix of pixels in order to identify geometrical figures. On the other hand, the coarse-grain applications are a set of DSP benchmarks developed by Texas Instruments [52].

Table III shows the obtained results regarding performance. Rows 2-5 refer to the applications for fine-grain devices, whereas Rows 6-12 refer to the coarse-grain ones. Finally, Row 13 shows the average results.

In this table, Column 3 shows the number of tasks of each task graph and Column 4 shows the initial execution time of the applications assuming that the reconfigurations do not generate any time penalty. Next, Columns 5-7 show the time penalty generated during the execution of the applications when *all the reconfigurations* are fetched from the external, HS and LE memories, respectively (*i.e., assuming unlimited capacity for these memories*). All these percentages represent the increment in the execution time generated with respect to the ideal one (Column 4). As it can be observed in these results, the average time penalty generated is reduced as the configurations are fetched from the LE (+15.91%) and HS (+11,09%) memories. Note that these results may not be possible to achieve due to the limited capacity of the on-chip memories in the real world. Hence we can see the results when all the tasks are fetched from HS as an upper-bound for our approaches in terms of performance.

Finally, Column 8 shows the time penalty generated when the configurations are fetched as indicated by our two configuration mapping approaches, and *assuming that HS and LE have a capacity of 3 configurations*. *In static environments, these results are the same for both algorithms* since both of them aim at reducing the generated time penalty as much as possible, and the difference between them is the way the configurations are distributed among the on-chip memories. As these results show, the average penalty generated by both approaches (13%) is very close to the upper-bound one (11.09%) in spite that the capacity of the HS and LE memories is limited to only 3 configurations. In fact, both mapping algorithms achieve the optimal results for all the evaluated benchmarks, except for the Parallel JPEG task graph. The reason is that this task graph includes 8 different tasks and only 3 of them fit in the HS and 3 in the LE memories; hence 2 tasks must be stored in the external memory (this does not apply to the other task graphs, since all their tasks are stored either in HS or LE). Finally, in comparison with fetching all the tasks from the external memory, in the table we observe an average reduction from +37.55% to just +13% in the time penalty generated, which means a reduction of 65% in the time penalty that is originally generated when fetching the configurations from the external memory.

All these numbers have been obtained assuming the reconfiguration latency only depends on the time needed to read the configurations. This is a valid assumption as long as the reconfiguration circuitry can write the data at the same speed that it receives it. In the examples analyzed in the literature ([30] and [31]) the bottleneck was always the configuration memory, and the reconfiguration circuitry was never working at full capacity. Hence we can safely assume that the configurations are written “on the fly” and the process finishes just a few clock cycles after reading the last word. Also, note that Table III does not provide any information regarding task communications latencies. The reason is that this issue is orthogonal to the problem addressed in this article. Indeed, *fetching the task configurations from different memories does not generate any additional restriction on the way the tasks communicate among them*.

TABLE IV
ENERGY CONSUMPTION DUE TO THE CONFIGURATION MEMORY ACCESSES WHEN USING THE PROPOSED STATIC AND DYNAMIC CONFIGURATION MAPPING ALGORITHMS, FOR STATIC ENVIRONMENTS (VALUES NORMALIZED TO EXT)

		Configuration static algorithm					*Configuration dynamic algorithm*				
Task graphs	*Number of tasks*	*EXT*	*HS*	*LE*	*1st execution*	*Subsequent executions*	*EXT*	*HS*	*LE*	*1st execution*	*Subsequent executions*
MPEG-1	5	0	2	3	1.21	0.20	2	1	2	1.15	0.65
JPEG	4	0	1	3	1.19	0.19	3	1	0	1.06	0.81
Hough	6	0	3	3	1.21	0.21	5	1	0	1.04	0.88
Parallel JPEG	8	2	3	3	1.16	0.41	2	3	3	1.16	0.41
DSP_dot_prod	3	0	1	2	1.23	0.23	0	1	2	1.23	0.23
DSP_vec_sumq	2	0	1	1	1.21	0.21	1	1	0	1.13	0.63
DSP_q15_tofl	3	0	1	2	1.20	0.20	1	1	1	1.14	0.48
DSP_neg_32	3	0	1	2	1.20	0.20	1	1	1	1.14	0.48
DSP_min_val	3	0	1	2	1.20	0.20	1	1	1	1.14	0.48
DSP_dotp_sqr	3	0	1	2	1.23	0.23	0	1	2	1.23	0.23
DSP_blkmove	3	0	2	0	1.25	0.25	0	2	0	1.25	0.25
Average		0.18	1.55	2.09	1.21	0.23	1.36	1.36	1.15	1.09	0.48

Next, Table IV evaluates the energy consumption generated by accesses to the configuration memories, for the same benchmarks as in Table III. In this case it is also assumed that the capacities of HS and LE are 3 configurations. In this table, Columns 3-7 show the results obtained when using the static mapper, whereas Columns 8-12 refer to the dynamic one. For each one of these two cases, Columns 3-5 and 8-10 show the number of tasks that each mapping approach assigns to the external memory (EXT), HS and LE, respectively.

Finally, Columns 6-7 and 11-12 detail the energy that is consumed during the first and subsequent executions of the involved task graph in the system, respectively. In this case, contrarily to Tables I and II, *these results have been normalized to the energy consumption when all the tasks are fetched from the external memory*.

For both approaches, the energy consumption generated during the first execution is always greater than loading all the tasks from the external memory (i.e. it is greater than one). The reason is that we assume that the configurations assigned to the on-chip memories are not initially preloaded there. Therefore, they are firstly fetched from the external memories and stored in the on-chip modules simultaneously to the run-time reconfiguration, introducing an additional energy consumption (+21% and +9% on average for the static and dynamic algorithms, respectively). This initial penalty is greater for the static approach since it assigns more tasks to the on-chip modules.

However, for subsequent iterations of the task graphs, this initial energy penalty is quickly compensated with further energy savings. Thus, the static and dynamic algorithms save an average of 77% and 52% energy consumption generated by the reconfigurations respectively, with respect to fetching the tasks from the external memory. The results are better for the static approach since it maximizes the amount of tasks assigned to the HS and LE modules.

C. Static vs. Dynamic Mapping Algorithms in Dynamic Environments

In a second set of experiments we have evaluated again both mapping approaches, but this time for dynamic environments; i.e., assuming that several active task graphs are executed on the system. The objective of these experiments is to evaluate how both approaches behave in situations in which there is certain competition among the configurations that have been assigned to the same on-chip memories.

In all these experiments, we have evaluated the execution time and the energy consumption achieved by both approaches dividing the task graphs for fine-grain and coarse-grain devices in two separate groups. For each one of them, we have executed a sequence of 100 task graphs randomly selected from the corresponding group.

Figure 4 shows these results when varying the capacity of the HS and LE memories simultaneously, from 3 to 10 configurations. Figures 4.a and 4.b show the results regarding the fine-grain task graphs, whereas Figures 4.c and 4.d refer to the coarse-grain ones.

Figure 4.a shows the average task-graph execution time of the evaluated sequence of task graphs for fine-grain devices. The figure compares both the static and dynamic approaches, when using the *LRU* and *modified LRU* replacement policies (which were discussed in Section V.C). It also shows the results achieved when all the tasks are fetched from HS (labeled in the figure as *All tasks from HS*), and *assuming that this memory has unlimited capacity* (therefore for this experiment no configuration replacements are carried out due to lack of HS memory capacity). Hence this result is used as lower-bound for execution time.

In this experiment it can be observed that for all the evaluated approaches, as the capacity of the on-chip memories grows, the closer is the generated execution time to the lower-bound solution. In addition, it can also be observed that the dynamic approach always generates less average task-graph execution time than the static one (see the comparisons

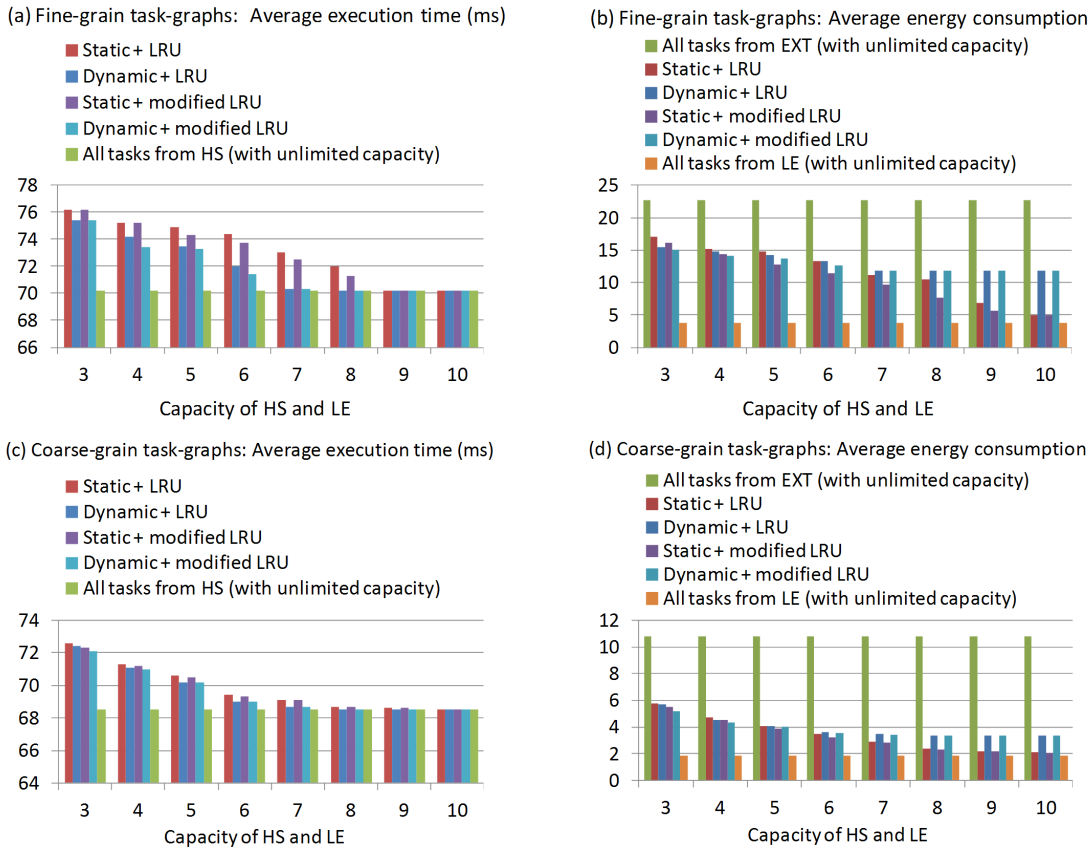


Fig. 4. Performance evaluation of the proposed static and dynamic configuration mapping algorithms, for fine-grain (a and b) and coarse-grain devices (c and d), and for dynamic environments, varying simultaneously the capacity of the HS and LE memories

between *Static + LRU* vs. *Dynamic + LRU* and *Static + modified LRU* vs. *Dynamic + modified LRU*).

In fact, the dynamic approach achieves an average reduction of 40% the time overhead generated when using the static approach (we define the *time overhead* of a given approach as the difference between its execution time and the execution time of the lower-bound reference value that assumes that all the tasks are fetched from HS memories). This reduction can be very significant in some cases: for instance, when the capacity of the on-chip memories is 7 configurations, the *Dynamic + LRU* approach reduces 100% of the time overhead generated by the *Static + LRU* one for the same on-chip memories capacity.

The comparison between the two evaluated replacement techniques (*LRU* vs. *modified LRU*) also shows that the latter achieves greater execution time reductions, when combined with both the static and dynamic approaches. In this case, the average time overhead reduction is 10.4% and 7.1% when combining it with the static and dynamic approaches, respectively.

Next, Figure 4.b shows the average task-graph energy consumption achieved for the evaluated approaches, and for the fine-grain task-graph experiment. In this case, the results have been normalized to the energy consumption of fetching a configuration from HS. The four evaluated approaches are also compared with the energy consumption achieved when fetching all the tasks from the external memory and from the LE one (in both cases, it is also assumed that these two memories have unlimited capacity). Hence these two results are used for comparison as upper-bound and lower-bound solutions regarding energy consumption, respectively.

In this case we can observe that, for small capacities of the on-chip memories (3 and 4 tasks), the dynamic approach achieves greater energy savings than the static one. In fact, for these two cases, the energy consumption generated by the dynamic configurations is reduced by almost 7%, both when using the LRU and the *modified LRU* replacement policies.

However, the figure also reveals that *as the capacity of the on-chip memories grows, the static mapping leads to greater energy savings than the dynamic one*. In fact, the *Static + modified LRU* achieves up to 78% reduction in the average energy consumption with respect to fetching all the tasks from the external memory. The reason of this behavior is simple: as the capacities of HS and LE increase, there is less competition among the configurations for the usage of the on-chip memories. This is an advantage for the static mapper since it maximizes the usage of these memories. In contrast, since the dynamic approach minimizes the amount of configurations stored on the on-chip memories, some configurations will *always* be fetched from the external memory independently of the capacity of HS and LE. Hence this leads to an additional penalization in terms of energy consumption for bigger on-

chip memories (*but not in terms of execution time*, as Figure 4.a showed).

Finally, Figures 4.c and 4.d repeat the same experiments as previously shown in Figures 4.a and 4.b, but this time for the coarse-grain task graphs. In both cases the trend shown is the same as in the fine-grain experiment, both when comparing the static and the dynamic approaches, and the LRU and *modified LRU* replacement techniques.

IX. Concluding remarks

This article has proposed a novel configuration memory hierarchy that takes advantage of high-speed (HS) and low-energy (LE) features that are present separately in recent configuration SRAM memories for reconfigurable devices. It comprises two on-chip memories: one optimized for HS and other optimized for LE (which have a limited capacity), plus an off-chip memory that stores the remaining configurations. In order to take advantage of this flexibility, we have also proposed two configuration mapping algorithms that decide in which memory to store the configurations of the applications, represented as task graphs. These algorithms adapt very well to different task-graph execution contexts. The presented experimental results have shown that, in combination with our configuration mapping algorithms, the proposed configuration memory hierarchy leads to significant energy savings, while providing a performance which is very close to the optimal one.

References

[1] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1/2, pp. 7–27, May 2001.

[2] W. Chelton and M. Benaissa, "Fast elliptic curve cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 198–205, Feb. 2008.

[3] R. Marin, G. Leon, R. Wirz, J. Sales, J. Claver, P. Sanz, and J. Fernandez, "Remote programming of network robots within the UJI industrial robotics telelaboratory: FPGA vision and SNRP network protocol," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 12, pp. 4806–4816, Dec. 2009.

[4] 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.

[5] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[6] Xilinx, "Virtex-5 FPGA configuration user guide, ug191(v3.10)," 2011.

[7] 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)*, July 2010, pp. 265–272.

[8] 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, sept.-oct. 2011.

[9] M. Technology, "http://www.micron.com/products/dram/rldram-memory?&source=sibis&dd-fam=dram&dd-tech=rldram%20memory," 2013.

[10] ——, "http://www.micron.com/products/dram/mobile-lpdram?&source=sibis&dd-fam=dram&dd-tech=mobile%20lpdram," 2013.

[11] B. Blodget, P. James-roxby, E. Keller, S. Mcmillan, and P. Sundararajan, "A self-reconfiguring platform," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2003, pp. 565–574.

[12] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of IRAM architectures," *SIGARCH Computer Architecture News*, vol. 25, pp. 327–337, 1997.

[13] E. Ramo, J. Resano, D. Mozos, and F. Catthoor, "Memory hierarchy for high-performance and energy-aware reconfigurable systems," Sept. 2007.

[14] E. P. Ramo, J. Resano, D. Mozos, and F. Catthoor, "Reducing the reconfiguration overhead: a survey of techniques," in *in Proceedings of the International Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2007, pp. 191–194.

[15] M. Huang, V. Narayana, M. Bakhouya, J. Gaber, and T. El-Ghazawi, "Efficient mapping of task graphs onto reconfigurable hardware using architectural variants," *IEEE Transactions on Computers*, vol. 61, no. 9, pp. 1354–1360, sept. 2012.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] K. N. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 1010–1023, Sept. 2006.

[22] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Exploiting application data-parallelism on dynamically reconfigurable architectures: placement and architectural considerations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 234–247, Feb. 2009.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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, Jan. 2009.

[33] 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.

[34] M. Jayapala, F. Barat, T. Vander Aa, 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, Jun. 2005.

[35] L. Shang and N. K. Jha, "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2002, pp. 345–352.

[36] F. Barat, M. Jayapala, T. Vander, A. Geert, D. R. Lauwereins, and H. Corporaal, "Low power coarse-grained reconfigurable instruction set processor," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[37] L. Benini and G. De Micheli, "Networks on chip: a new paradigm for systems on chip design," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2002, pp. 418–419.

[38] Xilinx, "7 series FPGAs overview, DS180 (v1.11)," 2012.

[39] ——, "ZC702 evaluation board for the Zynq-7000 XC7Z020 extensible processing platform, user guide, UG850 (v1.0)," 2012.

[40] Altera, "http://www.altera.com/products/devices/stratix-fpgas/stratix-v/stxv-index.jsp," 2011.

[41] ——, "https://www.altera.com/download/software/quartus-ii-we," 2011.

[42] F. Barat, M. Jayapala, T. Vander, A. Geert, D. R. Lauwereins, and H. Corporaal, "Low power coarse-grained reconfigurable instruction set processor," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[43] Xilinx, "EDK concepts, tools, and techniques, UG683 (v14.1)," 2012.

[44] ——, "MicroBlaze processor reference guide, UG081 (v13.4)," 2012.

[45] ——, "AXI reference guide, UG761 (v13.1)," 2011.

[46] ——, "Local memory bus (LMB) v1.0 (v1.00a)," 2005.

[47] ——, "PlanAhead user guide, UG632 (v11.4)," 2009.

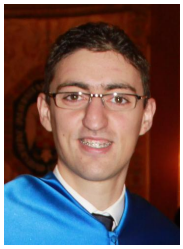
[48] Altera, "http://www.altera.com/products/ip/processors/nios2/ni2-index.html," 2011.

[49] H. Labs, "http://www.hpl.hp.com/research/cacti/," 2012.

[50] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 795–805.

[51] Z. Pan and B. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1465–1474, Nov. 2008.

[52] T. Instruments, "http://focus.ti.com/general/docs/dsnsuprt.tsp," 2010.

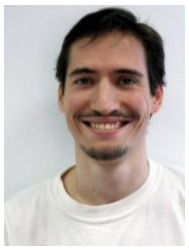


Juan Antonio Clemente (Member, IEEE) received his computer science degree from Universidad Complutense de Madrid (UCM), Madrid, Spain, in 2007; and his Ph.D. in 2011. He is Assistant Professor and Researcher with the GHADIR research Group, UCM. He also collaborates with the Embedded Systems Laboratory in the École Polytechinque Fédérale de Lausanne (EPFL), Switzerland; and the TIMA Laboratory, Grenoble, France, since 2009 and 2012, respectively.

His research interests are: FPGA design, embedded systems optimization and the development of techniques to optimize the management of task reconfigurations for reconfigurable devices. Also, his research has recently been focused on the study of Single Event Upset (SEU) tolerance of digital circuits implemented on FPGAs.

Elena Pérez Ramo received her Master Degree in Computer Engineer in 1999, and her European Ph.D. degree in 2009 at the Universidad Complutense of Madrid, Spain. Currently, she is Senior Business Intelligence Consultant at Telefónica España, in the Marketing and Business Research Department, and she is a member of the GHADIR research group, from Universidad Complutense of Madrid.

Her research has been focused on hardware/software co-design, task scheduling techniques, dynamically reconfigurable hardware and FPGA design. She has designed hardware accelerators for different fields, including digital signal processing and multimedia applications.

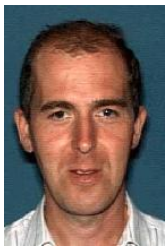


Javier Resano received his Bachelor Degree in Physics in 1997, his Master Degree in Computer Science in 1999, and the Ph.D. degree in 2005 at the Universidad Complutense of Madrid, Spain. Currently he is Associate Professor at the Computer Eng. Department of the Universidad of Zaragoza, and he is a member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

Daniel Mozos (Member, IEEE) received the B.S. degree in physics and the Ph.D. degree in computer science from the Universidad Complutense de Madrid, Spain in 1986 and 1992, respectively.

Currently, he is Full Professor with the Computer Architecture and Automation Department, Universidad Complutense de Madrid, where he leads the GHADIR Research Group on dynamically reconfigurable hardware. His research interests include design automation, hardware-software codesign, reconfigurable computing, the use of FPGAs for aerospace applications, and hyperspectral image processing. He has published more than 100 papers in international journals and conferences. He has been vice-dean for students during seven years and today he is the Dean of the Computer Science Faculty at the Universidad Complutense de Madrid.



Francky Catthoor (Member, IEEE) received his Ph.D. in Electronical Engineering from the Katholieke University of Leuven, Belgium in 1987. Between 1987 and 2000, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, including related application and deep submicron technology aspects, all at IMEC Leuven, Belgium. Currently he is an IMEC fellow. He is also part-time full professor at the EE department of the K.U.Leuven.

He has been associate editor for several IEEE and ACM journals, as IEEE Transactions on VLSI Signal Procsesing, IEEE Transactions on Multimedia, and ACM TODAES. He was the program chair of several conferences including ISSS'97 and SIPS'01. He has been elected IEEE fellow in 2005.