A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems

Juan Antonio Clemente, Javier Resano, Carlos González and Daniel Mozos

***Abstract*—New generation embedded systems demand high performance, efficiency and flexibility. Reconfigurable hardware can provide all these features. However the costly reconfiguration process and the lack of management support have prevented a broader use of these resources. To solve these issues we have developed a scheduler that deals with task-graphs at run-time, steering its execution in the reconfigurable resources while carrying out both prefetch and replacement techniques that cooperate to hide most of the reconfiguration delays. In our scheduling environment task-graphs are analyzed at design-time to extract useful information. This information is used at run-time to obtain near-optimal schedules, escaping from local-optimum decisions, while only carrying out simple computations. Moreover, we have developed a hardware implementation of the scheduler that applies all the optimization techniques while introducing a delay of only a few clock cycles. In the experiments our scheduler clearly outperforms conventional run-time schedulers based on As-Soon-As-Possible techniques. In addition, our replacement policy, specially designed for reconfigurable systems, achieves almost optimal results both regarding reuse and performance.**

***Index Terms*— Field Programmable Gate Arrays, Reconfigurable Architectures, Task scheduling.**

I. Introduction

In the last few years embedded devices have become more and more complex, including functionality initially developed for general purpose platforms such as multimedia support (sound processing, texture rendering, image and video displaying…). In fact the new generation of portable devices has inherited the area and energy constraints of embedded systems, and at the same time they must achieve the performance required by multimedia applications. The best way to meet these constraints is to include some HW support that can speed up the execution, and even reduce the energy consumption. Traditionally this migration was carried out developing application-specific integrated circuits (ASICs), which are silicon circuits customized for a particular use. However, although using ASICs is a very efficient option three important drawbacks prevent their use as a general solution. Firstly, the HW area in embedded/portable devices is very constrained. Hence only very critical functionality can be migrated to HW. Secondly, developing a new ASIC involves an increase in time-to-market, which is frequently a key factor for the success of a platform. Finally, their functionality is fixed, and cannot be updated in order to fix some detected bugs, or improve the efficiency of the system.

One interesting option to overcome these three limitations is to include reconfigurable HW resources: run-time reconfiguration allows reusing the same HW for different functionalities in order to meet the area constraints; the time-to-market is considerably shorter for reconfigurable HW than for ASICs, because the physical platform has been already tested, and the new functionality can be tested in the target board since the beginning of the design-cycle; finally, it offers an interesting trade-off between both performance and flexibility. Thus, this technology is especially suitable for applications that have dynamic and/or unpredictable behavior. In fact Sony™ has developed its own reconfigurable architecture, and has included it in some portable devices [1].

In embedded systems, applications are often represented as one or several Direct Acyclic Graphs (DAGs), where the nodes specify computational tasks and the edges represent precedence constraints. Managing efficiently the execution of these graphs is critical for embedded systems. Therefore, it is essential to optimize it. When dealing with reconfigurable systems several issues must be taken into account in order to deal with DAGs efficiently:

- The system must manage the task-graph information and must guarantee that the execution meets the precedence constraints.
- It must schedule the task execution attempting to achieve the required performance.
- It must also efficiently schedule the run-time reconfigurations. Most of current reconfigurable devices only include one reconfiguration circuitry and the reconfiguration latencies are frequently very significant (of de order of milliseconds), hence if many reconfigurations are demanded in a short period of time, the performance of the system can be seriously affected. When this happens the reconfiguration of the

Manuscript received August 25, 2009. This work was supported by the Spanish Department of Science and Innovation under grants TIN2009-09806 and AYA2009-13300.

J. A. Clemente is with Computer Architecture Department, Universidad Complutense de Madrid, Madrid, Spain (phone: +34 620390713; fax: +34 913947510; e-mail: ja.clemente@fdi.ucm.es).

J. Resano was with Computer Architecture Department, Universidad de Complutense de Madrid, Madrid, Spain. He is now with the Computer Eng. Department, Universidad de Zaragoza, Zaragoza, Spain (e-mail: jresano@unizar.es).

C. González and D. Mozos are with Universidad Complutense de Madrid, Madrid, Spain (e-mails: carlosgonzalez@fdi.ucm.es; mozos@fis.ucm.es).

most critical tasks must be scheduled in the first place in order to minimize the reconfiguration overhead.

- Finally, it must attempt to optimize the use of the reconfigurable resources. On the one hand, the system must assign to each task a suitable resource. On the other hand, since the system usually deals with recurring tasks the scheduler must promote the reuse of the most important tasks, avoiding their delay due to the reconfiguration latency. This can be done applying the proper replacement policy.

Moreover, all these problems must be addressed in such a way that the run-time delay generated by all the management/scheduling techniques is as low as possible, in order to prevent performance degradations.

In this regard, we have developed a complete scheduling flow targeting DAGs in a HW multi-tasking system, as well as an efficient implementation of a scheduler that includes all these optimization techniques. We have also tested our implementation in a Virtex-II PRO FPGA.

The rest of the paper is organised as follows: next section describes the contributions of this article. Section III shows a motivational example and Section IV overviews the related work. Section V describes in detail the proposed scheduling flow. Section VI presents the implementation details of the proposed scheduler and Section VII presents the experimental results. Finally, Section VIII explains our conclusions and indicates some lines for future work.

II. CONTRIBUTIONS OF THIS WORK

Our target system comprises a fixed number of reconfigurable units (RUs) with similar area (as it is shown in Fig. 1), where tasks can be reconfigured and executed. These RUs are connected among them by means of an interconnection network as it was initially proposed in [2]. Our scheduler receives as input a task graph (represented as a DAG) and steers its execution in our HW multi-tasking system taking into account its internal data dependencies. A task is the basic scheduling unit (i.e. a node of a task graph). Other processing elements may be present in the platform, as processors, DSPs, GPUs… We assume that the OS or middleware assigns the task graphs to the processing elements.

The proposed algorithm is a mixed design-time/run-time approach, since we are looking for a good-quality schedule but without carrying out too many computations at run-time. At design-time, graphs are analyzed in order to extract some useful information that will be used at run-time. Basically we characterize each task with three parameters: *weight*, *criticality*, and *mobility*. At run-time the scheduler uses the *weight* parameter to decide the reconfiguration order. The idea is to reconfigure first those tasks that have a greater impact in the critical path of the graph. The *criticality* identifies the delays that the reconfiguration of each task may generate, and it is used to assign greater priority to the tasks that generate greater delays. Finally, the *mobility* is used to escape from

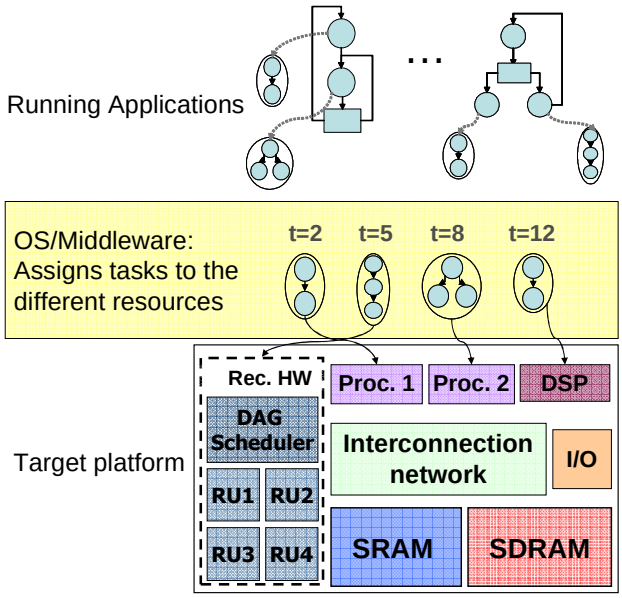

Fig. 1. Target architecture and execution scheme.

local-optimum scheduling decisions delaying at run-time some reconfigurations.

The scheduler steers the execution of the task graphs sequentially in the RUs taking into account both the precedence constraints and the available resources. In addition it applies a prefetch approach in order to carry out the reconfigurations in advance. The configuration that is fetched in advance is selected according to the weight of the candidates. Since several important scheduling decisions must be taken at run-time, we initially selected an *As Soon As Possible* (ASAP) scheduling strategy since it provides a good trade-off between the run-time complexity and the quality of the schedules. However, it is well-known that greedy ASAP strategies often fall into locally optimum decisions that can reduce the performance of the system. To attempt to escape from these local-optimum solutions, while increasing the run-time complexity as little as possible, we have developed an extended ASAP approach that sometimes delays the reconfiguration of a task taking into account its mobility and the state of the system.

In addition the scheduler also applies a replacement policy designed not only to maximize the task reuse when dealing with recurring applications, but also to improve the overall system performance collaborating with the prefetch technique. In fact, the replacement policy is the key factor to decide whether to delay the reconfiguration of a task or not.

We have evaluated our scheduler by executing task graphs extracted from actual multimedia applications. We have compared the results obtained by our scheduler with several reference systems that apply an ASAP scheduling approach combined with well-known replacement policies, such as LRU (*Last Recently Used*) or LFD (*Longest Forward Distance*). The latter case is especially interesting since LFD is the replacement policy that guarantees the optimal reuse percentage as it was demonstrated in [3]. However, LFD

cannot be applied in dynamic scenarios, because it can only be used in static systems where the future events are known. Nevertheless, LFD can be used as a reference to obtain an upper-bound of the reuse percentage for a given experiment. The experiments demonstrate that our replacement heuristic achieves almost as good results regarding reuse as LFD, and that our scheduler always obtains better results regarding performance than an ASAP approach, even when it applies the LFD replacement heuristic.

We have also evaluated the run-time overheads generated by our approach. For this purpose, we have developed two different versions of the system. The first one is a SW module that is executed in an embedded processor. However, for some benchmarks this version generates significant overheads, mostly due to the management of the complex data-structures and the HW/SW communications. To reduce these overheads, we have also developed an efficient implementation of the run-time scheduler using some of the reconfigurable resources. This implementation delays the execution only a few clock cycles. These two versions offer different trade-offs between the run-time management overhead and the cost needed to implement the scheduler.

There are many other areas of interest regarding HW multi-tasking systems. One of them is task placement. This may be sometimes a major problem, since a sub-optimal placement can lead to infeasible schedules that do not meet the system constraints, as it is proved in [4]. Frequently the reason is that the communication topology is not compatible with the selected mapping. In our target architecture we assume that the communication infrastructure is contention-aware and provides enough bandwidth so that even the worst-case communication is performed correctly and efficiently. This can be achieved by means of a bus with enough bandwidth or a contention-aware Network on Chip that implements a wormhole routing algorithm or any similar technique that guarantees that the latency of a transmitted message between 2 whatever nodes is almost constant, no matter the distance between them. Hence, under these assumptions we can safely assume that any task can be safely implemented on any of the RUs.

Other areas of interest regarding HW multi-tasking systems are inter-task communications, HW/SW partitioning, area fragmentation, low power concerns… However, these issues are orthogonal to the problem that we are targeting. Hence, we will also assume that the system OS or the middleware will take care of these decisions and we will just focus on what happens once a task graph has been assigned to the reconfigurable resources. To test our scheduler we have developed a simplified simulation environment and implemented it in a FPGA. This environment simulates the reconfiguration and the execution of task-graphs using programmable timers to simulate the behaviour of the RUs, and provides clock-cycle precision to measure the overheads generated by the scheduler.

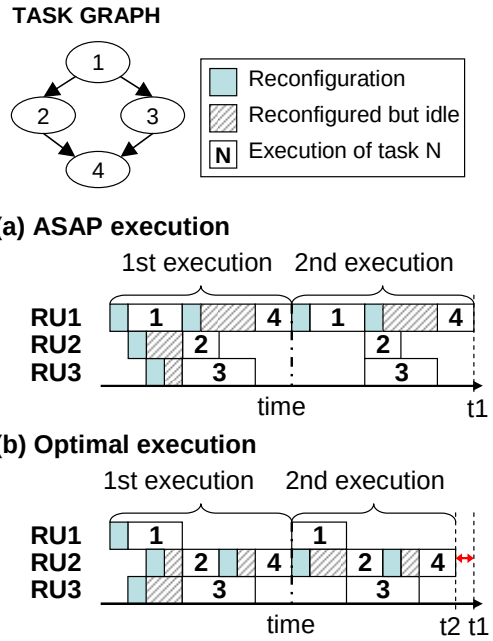

Fig. 2. Execution of a task graph in a platform with three RUs applying an ASAP approach (a) and using our scheduling technique that achieves the optimal solution delaying one of the reconfigurations (b).

III. MOTIVATIONAL EXAMPLE

Fig. 2 shows a motivational example that illustrates how our scheduler can escape from local-optimum solutions delaying some reconfigurations. In this example the task graph is executed twice in a system with 3 RUs, applying prefetch with an ASAP approach (a) and delaying some reconfigurations (b).

In both approaches it can be seen that the prefetch technique is very powerful when dealing with DAGs since it hides the latency of most reconfigurations. The simplest way to apply prefetch is using an ASAP approach, since it is very simple to implement and it generally will lead to good schedules. When more than one task can be reconfigured we select the one with greater weight (as we will describe later). In this example, both tasks 2 and 3 are ready for reconfiguration simultaneously, and task 3 is selected since it is part of the critical path. With this approach the latency of three of the four tasks is hidden. Task 1 is the only one that generates delays due to its reconfigurations. In our scheduling environment we will say that task 1 is critical whereas the others are non-critical.

However, although our replacement policy assigns greater priority to Task 1, in Fig. 2 (a) this task is replaced by Task 4, because in that instant it was the only available candidate. For this reason in the second iteration the reconfiguration of Task 1 introduces again a delay in the execution.

This delay will disappear if we delay the reconfiguration of task 4 (Fig. 2 (b)). In this case, our scheduler knows that the reconfiguration of Task 4 can be delayed without any performance degradation (in Section V we will explain how this information is obtained). Hence, when Task 1 is selected as the replacement victim, the scheduler decides to delay this reconfiguration waiting for the following event (i.e. the end of

the execution of Task 2). Thus, when Task 2 finishes its execution the replacement policy can select between two replacement victims (tasks 1 and 2), and it will select Task 2 since it is non critical. As a consequence, when the task graph is executed again, Task 1 will be directly reused and no reconfiguration will generate any delay in the execution.

IV. RELATED WORK

During recent years many research groups have developed techniques that attempt to reduce the reconfiguration overhead. And interesting survey of most of them can be found in [5].

Many authors, such as [2, 6, 7, 8, 9] have proposed to build a HW multi-tasking platform by dividing the entire reconfigurable area into smaller RUs, The first implementation of such a system in a commercial FPGA was presented by Marescaux et al. in [2]. In this work, the authors propose to divide the entire reconfigurable area into identical tiles, connected among them by an interconnection network (ICN). Recently Nollet et al. [10] have proposed to extend this approach applying the idea of configuration hierarchy to build a reconfigurable Multi-Processor System-on-a-Chip (MP-SoC). Basically, instead of executing dedicated HW tasks in the reconfigurable resources, they use these resources to implement programmable softcores that will execute SW tasks. They have developed task mapping heuristics to assign tasks to the different processors (both hard- and soft-processors), taking into account the communication topology. We believe that the ideas of this work are compatible with our approach, although we are not focusing on soft-cores.

In [6], Walder et al. present a run-time environment to execute HW tasks by partially reconfiguring a Xilinx™ Virtex II PRO. This work does not deal with task-graphs, but with independent tasks, and the main objective is to develop a partition technique to decide the optimal size of the RUs. In our work we assume that this has already been decided, hence this work is again orthogonal to our work.

In [7], Qu et al. propose adding more reconfiguration controllers to carry out several reconfigurations in parallel. This could improve the efficiency of a reconfigurable multi-tasking system, but currently the commercial platforms only include one reconfiguration controller.

In [11] and [12] the authors present two interesting approaches for HW multi-tasking. In this case the objective is to take full advantage of the data-parallelism for a given application by replicating the same task several times. This is again compatible with our work. Several versions of the same task with different level of data parallelism can be identified at design-time, and at run-time the OS or the middleware could select the appropriated one and then send it to our DAG scheduler. In fact this is what we have done in order to generate one of the benchmarks that we will use in the experimental section (Parallel-JPEG).

Other interesting contributions for HW multi-tasking systems are the research efforts to develop OS support to simplify the use of the reconfigurable HW. Some relevant examples are the works developed by Kosciuszkiewicz et al. [13] and H. Kwok-Hay So et al. [14]. They propose to extend an Embedded Linux OS to support HW tasks. The idea is to provide support to let the user transparently deal with HW tasks at run-time as they were regular threads. A related topic is how to decide, again transparently to the user, which tasks must be executed in the reconfigurable resources. HW/SW partitioning has been a very active research field during the last decade: in [15], Fu and Compton deal with a HW/SW multitasking system, which includes reconfigurable resources and a UltraSparc processor. In order to improve the performance some tasks are assigned to the reconfigurable HW. They have developed several algorithms to select these tasks taking into account the needs of each application, the available resources, and the reconfiguration overheads.

Previously some research groups have proposed to include HW scheduling support for reconfigurable systems. In [8], the authors propose a HW micro-architecture to deal with task management at run-time efficiently applying a list-scheduling heuristic. However, the authors did not implement their design, but they only included it in their specific simulation environment. In [9] the authors have extended [8] in order to support control dependencies in the task graph. However, again they do not implement their approach. Moreover, they assume that all the reconfigurations can be carried out in parallel; hence they do not need to schedule the reconfigurations. This simplifies considerably their scheduler. However, as it was explained before, currently this is not a realistic assumption. In addition, they do not provide replacement support, and they follow a greedy list-scheduling approach that cannot escape from local-optimum decisions. In [16] the authors propose a scheduling methodology for a real-time system based on reconfigurable HW and implement it in a FPGA. They use an earliest-deadline first approach but grouping several tasks in order to apply efficiently full reconfigurations. Since they do not support partial reconfiguration, they do not have to apply any prefetch approach or a replacement policy because they only have one RU. In addition they deal with independent periodic tasks; hence the scheduling support that they have developed is completely different from our approach. Hence our work clearly provides important novel contributions. First, we not only provide a prefetch approach, which was initially proposed in [17], but also a replacement technique designed to collaborate with our scheduler and improve the overall results. Second, we have found a simple way to escape from local optimum decisions. In this regard, many researchers have proposed off-line scheduling techniques that can escape from local optimum solutions. However, when dealing with on-line systems this is much more complex and only simple (and mostly greedy) algorithms are used as list-scheduling or earliest-deadline first. Third, we have implemented our scheduler and tested it in a FPGA using graphs extracted from actual multimedia applications. Fourth, we have implemented

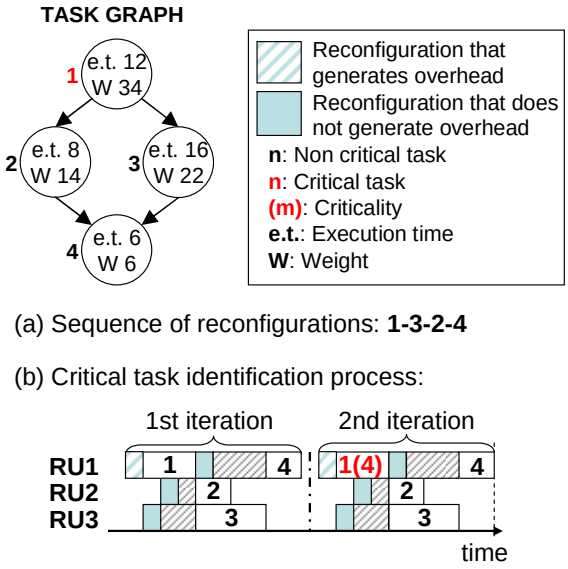

Fig. 3. Example of the algorithms for the weight calculation (a) and the critical -task identification (b). The reconfiguration latency is always 4 ms.

an equivalent SW version and analyzed the design trade-offs.

Our research group has been already active in this field for many years. We already presented a scheduling environment for reconfigurable systems in [18] and [19]. In these works we proposed several techniques that also deal with task-graphs at run-time. However, in a recent work [20] we identified that even our simple techniques generate important overheads when dealing at run-time with complex data structures, such as task-graphs, especially if they were executed in an embedded processor. Hence we decided to develop a generic HW task-graph execution manager for reconfigurable systems [21]. This module receives as inputs a task-graph and its schedule and steers the execution of that task-graph following the instructions given in the schedule. The main achievement presented here was a HW implementation of that manager, which significantly speeds up this management process with respect to an equivalent SW version.

The next step has been to add a scheduling layer to this management module, thereby developing a task-graph scheduler for reconfigurable systems. Of course, we have tried to keep those ideas that proved to be useful in our previous work, such as a replacement policy that collaborates with the scheduler, and trying to extract useful information at design-time. However, we have developed new techniques that can be efficiently implemented in HW, and we have extracted some extra information at design-time. A preliminary version of this work has been published in [22]. At that time our scheduler applied a simple ASAP approach, and our HW implementation was very area-hungry. Since then, we have developed a new scheduler that delays some reconfigurations in order to escape from local-optimum decisions, we have drastically improved the area scalability of our design, we have improved the replacement strategy, and we have carried out new experiments in order to better test our approach.

V. THE SCHEDULING FLOW

In this section we will explain the proposed scheduling flow. In the following, we will refer to a node of a task graph as a *task*.

Our scheduler flow includes two phases: *design-time* and *run-time*. At design-time, the task graphs are analyzed in order to extract some information that the scheduler will use to optimize its decisions. At run-time the scheduler steers the execution of the graph in the set of RUs) of the proposed architecture, applying a prefetch technique and a replacement policy and taking into account the mobility of the tasks.

The next sub-sections describe the *design-time* and the *run-time* phases, respectively.

A. Design-time phase

This phase is basically a compilation phase, which is necessary in order to extract useful information that will be used at run-time. Each task is characterized with three parameters: *weight, criticality* and *mobility*.

1) Weight calculation

Initially the scheduler assigns a weight to each task according to this simple algorithm: the weight of a leaf task (those tasks that have no successors) is just its execution time. For the rest of the tasks, their weight is the addition of their execution time and the maximum of the weights of all their successors. An example of this process is shown in Fig. 3 (a). Thus, according to the figure, W(task4)=6; W(task2)=8+6=14; W(task3)=16+6=22 and W(task1)= 12+max(14, 22)=34.

We assume that we have reliable estimations of the execution time of each task. Nevertheless, if the execution time of the tasks was variable (for instance, depending on the input data), the scheduler could adopt a solution similar to the one proposed in [23], where the authors suggest creating several graphs for the same task (called scenarios) and then choosing the one that best fits to the current conditions. These scenarios are identified at design-time; hence if it was necessary to apply this technique, the scheduling flow that we propose would continue being valid.

The weights represent the impact of the tasks in the critical path of the task graph. Thus, if *Weight(A) > Weight(B)*, it means that A must be loaded before B. Hence, once this process is performed, all the tasks are sorted decreasingly according to their weights to determine *the sequence of reconfigurations* that the scheduler will follow at run-time. Thus, in Fig. 3 (a), this sequence is 1-3-2-4.

2) Critical tasks identification

The second step identifies which tasks are especially critical for the system. The goal of the algorithm is to obtain the minimum set of tasks that fulfill the following condition: if they are reused (and therefore they do not generate any delay due to the reconfiguration latency), the scheduler will be able to hide the reconfiguration overhead of the remaining (i.e. non-critical) tasks. It is important to point out that at design-time

Critical tasks (CT) identification:

```
1. CT := whole_set_of_tasks (task_graph);
2. ref_sch := schedule (task_graph, CT);
3. CT := Ø;
4. end := 0;
5. WHILE (not end){
6.   current_sch := schedule (task_graph, CT);
7.   IF (ex_time (current_sch) == ex_time (ref_sch)){
8.     end := 1;
9.   }ELSE{
10.    delayed_tasks := compare (current_sch, ref_sch);
11.    t := max_weight (delayed_tasks);
12.    add_critical_task (t, CT);
13.  }
14.}
15.RETURN CT;
```

Fig. 4. Algorithm to identify the critical tasks.

we know how our scheduler will work at run-time; hence we can identify the critical tasks at design-time, saving runtime computations. These tasks will be labeled as "critical tasks", and the scheduler will assign them a value of criticality that represents the delay that they will generate when they are not reused. At run-time, the replacement policy will take into account this information assigning greater priority to the critical tasks.

Fig. 4 shows the pseudo-code of this algorithm. Firstly, the set of critical tasks (CT) is initialized to the whole set of tasks in the task-graph (line 1). Then, the function *schedule (task_graph, CT)* (line 2) is called. This function schedules the task-graph assuming that all the tasks in the CT set are reused and returns a reference schedule (*ref_sch*). Any scheduling algorithm can be used in this step. Since these computations are carried out at design-time, in this case we use a branch&bound-based scheduler that guarantees the optimal solution. Hence, as initially all the tasks have been assigned to CT, in this step we obtain an *ideal schedule* with no reconfiguration overhead. This ideal schedule is used as a reference during the critical-task identification process.

The objective of this process is to find a schedule that provides the same performance than *ref_sch*, but with the minimum number of tasks assigned to CT. We initialize CT as empty (line 3) and we start an iterative process (lines 5-12). The while loop starts computing a new schedule (*current_sch*, line 6) and compares its execution time with the execution time of *ref_sch* (line 7). If both schedules have the same execution time, the iterative process finishes. Otherwise the function *compare* identifies which tasks have been delayed in *current_sch* (line 10). In the next step (line 11) the function *max_weight* identifies the delayed task with the greatest weight, and finally this task is added to the CT set (line 12).

Fig. 3 (b) depicts an example of this process. As it can be seen in the figure, all the tasks in the graph are initially labeled as non critical. Then, the algorithm performs the first iteration of the loop and identifies a 4 ms delay (reconfiguration of Task 1) compared to the ideal execution. Hence, this task is labeled as critical, with a criticality of 4. In the following iteration of the loop, Task 1 is reused, and in this case the configurations of Tasks 2, 3 and 4 do not generate any execution time overhead. Hence the algorithm stops and tasks 2, 3 and 4 remain labeled as non critical.

Mobility assignment:

```
1. CT := obtain_critical_tasks (task_graph);
2. NCT := obtain_non_critical_tasks (task_graph);
3. WHILE (NCT ≠ Ø){
   /* Get a task from NCT */
4.   t := get_task(NCT);
   /* Schedule with previous t.mobility */
5.   ref_sch := schedule (task_graph, CT);
6.   DO{
7.     t.mobility++;
   /* Schedule with new t.mobility */
8.     new_sch := schedule (task_graph, CT);
9.     diff := ex_time (new_sch) – ex_time (ref_sch);
10.    IF (diff > 0){
11.      t.mobility--;
12.    }
13. }WHILE (diff == 0);
14.  remove (t, NCT);
15.}
```

Fig. 5. Algorithm to assign the mobility to the non-critical tasks.

3) *Mobility calculation*

Finally, the scheduler assigns a value of mobility to each task. This parameter identifies how many times a reconfiguration can be delayed without generating any overhead in the task-graph execution. More precisely, this value represents how many events can be skipped before reconfiguring a given task without generating any delay. This means that the reconfiguration circuitry is ready, and that at least one of the RUs is available, but for some reason our scheduler decides to delay the reconfiguration. In fact the reason is always the same: the replacement policy has selected as a victim a task that the scheduler prefers not to replace. Hence the scheduler will check the mobility, and if there is still margin for delaying the reconfiguration, it will wait until the following scheduling event hoping that by that time the replacement policy will find a less important victim.

Fig. 5 shows the pseudo-code of this algorithm. Initially all the mobility of each task is initialized to 0. Then the process starts identifying the critical and the non-critical tasks of the task graph and storing them in *CT* and *NCT*, respectively (1-2). (Note that by definition the critical tasks have no mobility because even if their reconfiguration is scheduled ASAP they will generate a delay). In the main loop (3-15) the algorithm sequentially extracts a task *t* from *NCT* (4), and the function *schedule ()* obtains a first schedule (*ref_sch*) assuming that *t* has mobility 0. Then, the *do-while* loop (6-13) tentatively assigns a greater mobility value (7) to *t* and calculates a new schedule (*new_sch*), but this time delaying the reconfiguration of *t* as many times as the value of *t.mobility*. This is again done in the function *schedule ()* (8). Then, the algorithm checks if it is feasible to assign that new mobility to *t* without degrading the performance of the system (9). Thus, if *diff>0* (10), an extra overhead has been generated; hence the algorithm

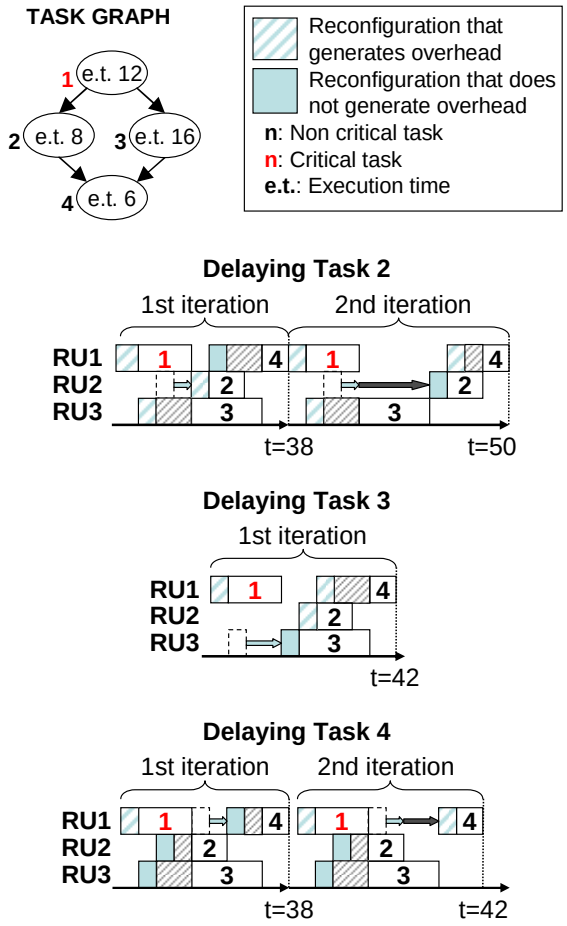

Fig. 6. Example of the mobility calculation. The reconfiguration latency is always 4 ms.

restores the former mobility value to *t* (11) and exits from the *do-while* loop (13). Finally, the task is removed from *NCT*. The while loop continues until *NCT* is empty.

Fig. 6 illustrates this process with an example. Firstly, the algorithm selects the non-critical tasks of the graph: 2, 3 and 4. For Task 3, it attempts to assign a mobility of 1. However, when the scheduler delays its reconfiguration it generates a delay in the execution of 4 ms. Hence, the mobility of Task 3 is set to 0. On the contrary, Task 2 and 4 can be delayed without any performance degradation. Hence the scheduler assigns them a mobility of 1 in the first iteration. Afterwards it attempts to assign a value of 2, but in both cases new overheads are generated. Hence, the mobility of Task 2 and 4 are set to 1.

B. Run-time phase

At run-time, the scheduler steers the execution of the task graph taking into account the information obtained at design-time and the available resources. To take all these run-time decisions as quickly as possible the scheduler only considers some discretized time values following an event-triggered approach. When certain events happen the scheduler will look for reconfigurations or tasks that are ready to be scheduled. Three different events trigger the execution of the scheduler: *new_graph*, which is generated when the information of a new task graph is received; *end_of_reconfiguration*, which is generated by a RU when a new configuration has been loaded or when the RU has identified that a configuration can be reused since it was already loaded in a previous execution; and *end_of _execution*, which is generated by a RU when a task finishes its execution. This approach greatly reduces the complexity of the run-time scheduling process, but at the same time it provides enough flexibility to optimize the execution. Basically, the scheduler processes these events sequentially and carries out the proper actions. Fig. 7 depicts this process.

```
/* task = task that triggered the event
 * RC = Reconfiguration Circuitry */
CASE event IS:
 1. new_graph:
 2.    IF (RC == idle){
 3.       look_for_reconfiguration (&rec_sequence);
 4.    }
 5. end_of_reconfiguration or reused_task:
 6.    check_dependencies (&task);
 7.      IF (is_ready (&task)){
 8.         start_execution (task);
 9.      }
10.   look_for_reconfiguration (rec_sequence);
11. end_of_execution:
12.    IF (RC == idle){
13.       look_for_reconfiguration (rec_sequence);
14.    }
15.    update_task_dependencies (&task)
16.    FOR (i := 0 TO NUMBER_OF_RUS){
17.       IF (RU_state == IDLE) AND (is_ready (&task)){
18.          start_execution ();
19.       }
20.    }
```

Fig. 7. Pseudo-code of the run-time phase.

When a new task graph arrives and the reconfiguration circuitry is free (1-3), the system attempts to schedule the reconfiguration of the first task in the reconfiguration sequence of the received graph.

For the *end_of_reconfiguration* and *reused_task* events, the system checks the dependencies of the task that has been loaded or reused in order to identify if it can start its execution. To this end, the scheduler will check if all its predecessors have already finished their execution (5). In that case, the task is ready to be executed (6) and the system starts its execution (7). Then, it attempts to schedule a new reconfiguration (8).

Finally, for the *end_of_execution* event, the scheduler checks again if the reconfiguration circuitry is idle. If so, it looks for a new reconfiguration (10-11). After that, it updates the task-graph dependencies, decreasing the number of predecessors of each successor of the finished task (12). Finally, the system checks if any of the tasks that are currently loaded in any RU can start its execution (13-15).

The following subsections describe the replacement policy and the run-time mobility management, respectively. Both techniques are applied inside the *look_for_reconfiguration* function.

1) Replacement policy

Transparently to the event management, the scheduler also

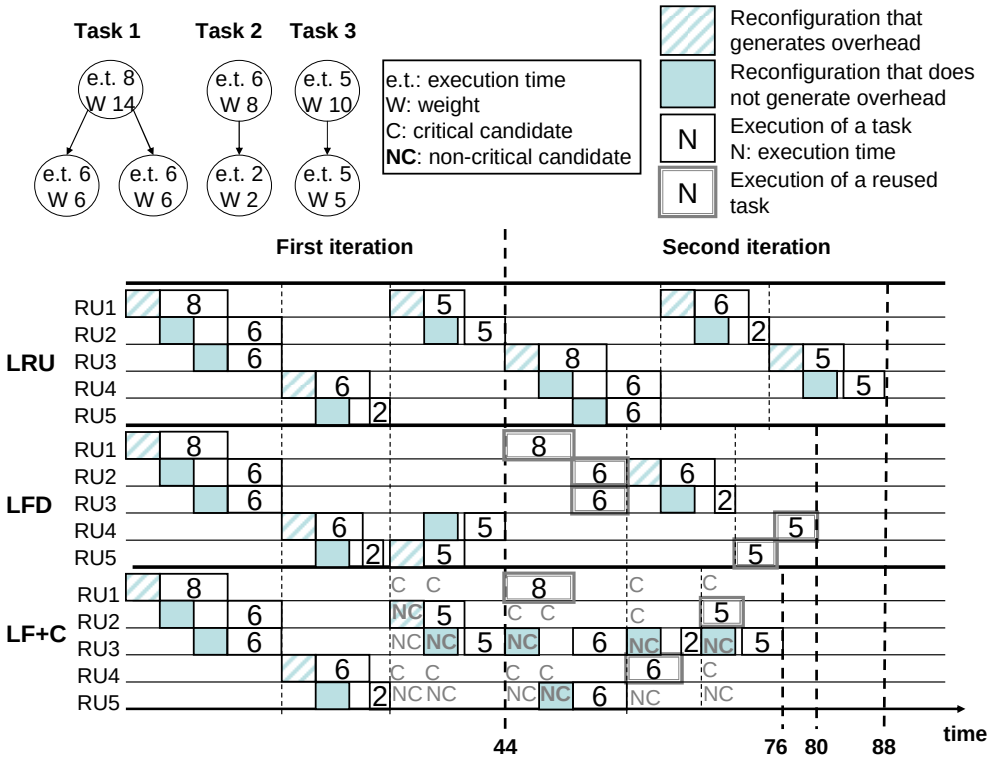

Fig. 8. Execution of three graphs in a system with 5 RUs and 4 ms of reconfiguration latency.

applies a replacement policy especially designed for this system. We have called it Look Forward + Critical (LF+C) because it takes into account the *criticality* of the tasks and whether *they are going to be loaded in the near future* or not. We assume that the "near future" means "in the context of the graph currently in execution" because this is the only region of the future that the scheduler can analyze. According to this information, the scheduler classifies the RUs into 6 categories:

- **Busy**: RUs with a task in execution, or with a configuration that has recently been prefetched and that is waiting for execution. These RUs are never selected for replacement.
- **Free candidates (FC)**: These RUs do not have any task loaded. When the system is initialized, all the RUs are free.
- **Non-critical candidates (NC)**: RUs with non-critical tasks that are not going to be executed in the near future.
- **Non-critical reusable candidates (NRC)**: RUs with non-critical tasks that are going to be executed in the near future.
- **Critical candidates (CC)**: RUs with critical tasks that are not going to be executed in the near future.
- **Reusable critical candidates (RCC)**: RUs with critical tasks that are going to be executed in the near future.

The replacement technique assigns the maximum priority to the RCCs and the minimum to the FCs. Hence, it attempts to replace those tasks that are not critical and/or are not going to be executed soon. If all the possible victims of the replacement policy are critical and belong to the same category (CC or RCC), the scheduler use the criticality value to select the victim. If all of them had the same criticality, the victim is selected randomly.

It may be unclear whether is better to assign more priority to the NRC candidates or to the CC candidates. On the one hand, assigning more priority to the NRC can be a good strategy, since if those tasks are replaced the scheduler will have to load them again soon. On the other hand, according to our definition of critical tasks, replacing a critical task will generate delays in the execution if that task is needed in the future, whereas replacing a non-critical task will not generate any delay. In [22] we propose to assign more priority to the NRC category to reduce the number of replacements needed. However, after extensive simulations we have identified that assigning more priority to the CCs always provides better performance, and that the percentage of reused tasks were very similar. Hence we have modified our previous replacement module and currently we assign more priority to the CCs than to the NRCs. Hence RCCs have the maximum priority, then CCs, then NRCs, then NCs and finally FCs.

We have designed a new replacement policy for three reasons. Firstly, conventional replacement policies, such as LRU, can easily fall into a configuration trashing problem when the number of active tasks is greater than the number of RUs. Secondly, since the scheduler deals with task graphs, it has some information about the tasks that are going to be executed in the near future, and it can take advantage of that. Finally, the objective of conventional replacement policies is to improve the reuse. This is a good objective, but in our replacement policy it is only a secondary goal. In this case, the

```
void look_for_reconfiguration (){
 1. new_task := next_task_in_rec_sequence();
 2. apply_replacement (victim);
 3. IF (victim ≠ Ø){
 4.   IF (critical_candidate (victim) == TRUE){
 5.     IF (new_task.mobility > skipped_events){
 6.       skipped_events++;      /* initially 0 */
 7.     }
 8.   }ELSE{
 9.     load (&new_task);
10.    delete_task_in_rec_sequence (&new_task);
11.  }
12.}
}
```

Fig. 9. Pseudo-code of the function *look_for_reconfiguration()*.

main objective is to improve the performance by attempting to reuse as many critical tasks as possible.

Fig. 8 depicts an example that illustrates the benefits of our replacement policy. To simplify the figure we have mixed categories 3 and 4 (non-critical candidates in the figure) and categories 5 and 6 (critical candidates in the figure). The figure presents a system that executes twice three task graphs using three different replacement policies: LRU, LFD and LF+C.

The figure shows that the well-known LRU policy does not reuse any task, since there are 7 tasks competing for 5 RUs. Hence, it provides the worst execution: 88 ms. This is an example of configuration trashing. The second replacement policy (LFD) is the optimal one regarding reuse, as it was proved in [3]. Basically, this strategy replaces the task that will be requested farthest into the future. LFD cannot be applied in dynamic systems, since in order to apply LFD the system needs to know all the future events, but it can be used as a reference. In this case the system reuses 5 tasks and the execution time is 80 ms.The figure shows that the LRU policy does not reuse any task, since there are 7 tasks competing for 5 RUs. Hence, it provides the worst execution: 88 ms. The LFD replacement policy is the optimal one regarding reuse, and used here as a reference; as we explained in Section II. In this case the system reuses 5 tasks and the execution time is 80 ms.

Finally, the figure also shows what happens when the system uses LF+C. For the first two graphs, the replacement policy simply selects a Free Candidate (FC). When Graph 3 starts RU1 and RU4 are critical candidates, whereas RU2, RU3 and RU5 are non-critical candidates. For this graph the scheduler selects the first two non-critical candidates (RU2 and RU3) because critical candidates have a greater priority. When Graph 1 is executed again, the first task is reused, and the scheduler loads the two remaining nodes in RU3 and RU5 respectively, since they are the only non-critical candidates available. Although only one task has been reused, no delays have been generated due to the reconfigurations, since this was the only critical task. When graph 2 starts again its first task, which is the only critical task of the graph, is reused. For the other task the replacement policy selects the first non-critical RU (in this case RU3). Finally, when Graph 3 is executed, the system reuses its first task (which again is critical) and loads its second task in RU3, again a NC candidate. As it can be seen in the figure, LF+C does not achieve the optimal result regarding reuse. Whereas LFD reused 5 tasks, LF+C only reused 3. However, as it was explained before, reuse is only a secondary objective for our heuristic. The main objective is performance, and in this example it achieves the optimal performance, hiding the latency of all the reconfigurations in the second execution of the graphs. Using LF+C the 6 graphs are executed in 76 ms, 4 ms less than using LFD and 12 ms less than using LRU. As a conclusion the objective of LF+C is not only to reuse more, but also to reuse better.

2) *Mobility management*

In Fig. 7 there were three calls to the function *look_for_reconfiguration*(). In this function, the scheduler must decide whether to schedule the reconfiguration of the following task in the reconfiguration sequence or not. To this end it takes into account the mobility of the task and the victim selected by the replacement policy.

Fig. 9 shows the pseudo-code of this function. Firstly, the function extracts the first task in the reconfiguration sequence (1). Then, the scheduler selects a victim applying our replacement policy (2). If all the RUs are busy, this would not be possible (3). Otherwise, the function checks if the victim is critical (4). In this case the scheduler will try to delay the reconfiguration. To this end it checks if the mobility of the task is greater than the number of total skipped events at that moment (5). In that case, the system will just increase the number of skipped events (6). Otherwise, the function triggers the reconfiguration of the *new_task* (9) replacing the selected victim and removing *new_task* from the reconfiguration sequence (10).

As it was shown in the motivational example of Fig. 2, this approach allows reusing more critical tasks, leading to an improvement on the system performance.

VI. IMPLEMENTATION DETAILS

In this section we will explain in detail the proposed architecture. As mentioned before, we have implemented two different versions of our scheduler: a SW version and a HW one. In both cases, we have used a Xilinx™ Virtex II PRO FPGA and the EDK development tool to carry out our experiments. We have not implemented a complete HW multi-tasking system, but just a simulation platform where the execution of the tasks in the RUs and the reconfigurations are simulated using programmable timers. The system includes an additional timer that is used to measure the execution time of a given simulation with clock-cycle accuracy.

In the SW version (Fig. 10 (a)) the scheduler is a program that runs in a PowerPC™, and that communicates with a set of timers representing the RUs. This program is a C-code compiled for the PowerPC architecture. The assembly code is stored in an internal memory in the FPGA. During the run-time scheduling phase the timers communicate with the processor generating interrupts and the processor can directly access to the timer interfaces using the system bus. When a task is

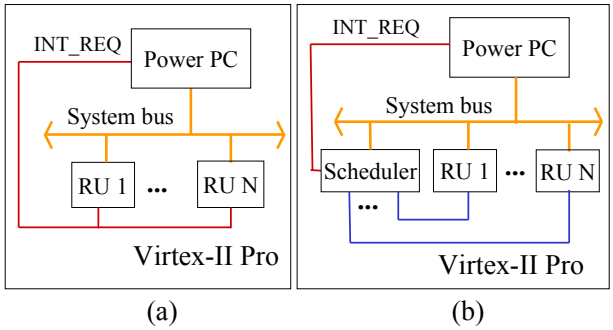

Fig. 10. Simulation platform. (a) SW implementation of the scheduler. (b) HW implementation of the scheduler.

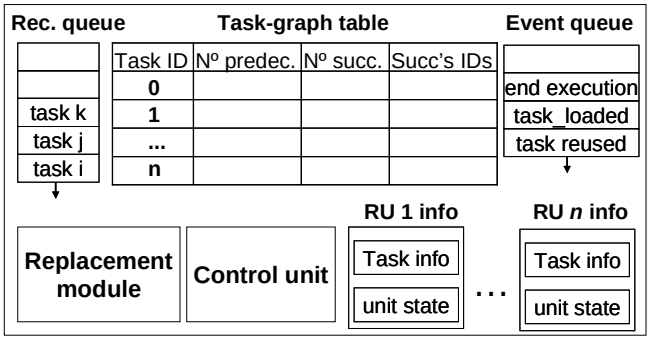

Fig. 11. Block diagram of the HW scheduler.

assigned to a RU the processor loads its information (the reconfiguration latency or the execution time) in the timer and activates it when the reconfiguration or the execution must be simulated. When the timer finishes the countdown, it generates an interrupt that will trigger the corresponding event (*end_of_reconfiguration* or *end_of_execution*). The PowerPC can then carry out the proper scheduling actions.

In the HW version (Fig. 10 (b)) the scheduler is a HW module implemented using some of the available reconfigurable resources in the device. This module only provides support for the run-time phase of our scheduling flow, since the design-time phase must be carried out at compile time. The information regarding the different task-graphs is stored in an internal memory, and when the execution of a task-graph must be simulated in this platform, the PowerPC transfers all the task graph information to the HW scheduler. When the simulation finishes, the scheduler generates an interrupt.

In this implementation the RUs interact directly with the HW scheduler using point to point communication lines. This approach reduces the penalties due to HW/SW communications from one to two orders of magnitude in our experiments. A block diagram of the HW scheduler is depicted in Fig. 11. Next we will describe these blocks in detail.

A. *Reconfiguration queue and task-graph table*

These modules are initially used to store the reconfiguration sequence and the task-graph information. During the task-graph execution this information is updated according to the run-time events.

The *Task-graph table* stores the information of the current task graph including all the precedence constraints. This table is used to guarantee the proper task execution. For this purpose, the table supports the following operations:

Insertion: used to store the information of a new task graph. The latency of this operation is NT clock cycles, where NT is the number of tasks.

Check: This operation enquires about whether a task is ready to be executed or not, which happens when all its precedence constraints have already been met. This operation just requires one clock cycle.

Hit: This operation enquires about whether a task is currently waiting for execution. This information will be used by the replacement module. This operation just requires one clock cycle.

Update: This operation updates the task graph information when a task finishes its execution. It updates the state of the task and its precedence constraints. To this end the successors of that task are extracted and sequentially updated, decrementing the value stored in *Nº predec*. This value represents the number of predecessors that have not finished their execution. The table includes some specific HW support for this operation that is not included in the figure for simplicity. The latency of this operation in *2*NS+1* clock cycles, where NS is the number of successors of the task.

Initially, we implemented an associative version of this table, which allowed having a table with a relatively small number of entries, since the particular position of a task in the table was not relevant. In addition it provided very high performance for small graphs. However, as the complexity of the tested task graphs grew, it was also necessary to use a larger table, which consumed an unacceptable amount of resources for large task-graphs (in some cases this grew up to 99% of the scheduler). Furthermore, its operation frequency greatly decreased, which limited the performance of the system. In order to solve these problems, we implemented a non-associative version of the table using one of the SRAM blocks (BRAMs) available in the FPGA and some additional HW support. In this second version, each task is stored in the table according to its ID, where the ID is its unique identifier. For instance, the information of Task N is stored in the Table[N] entry. The main drawback is that the non-associative table needs as many memory positions as different tasks can be executed in the system. However, this is not a big problem, since in our experiments just one BRAM was enough to deal with all of our tested task graphs. With this approach, the HW area required for the task-graph table has been drastically reduced. Moreover its operation frequency is more than four times faster than the associative version (up to 400 MHz), even for a large number of entries.

B. *Replacement module*

This module implements the replacement technique. It demands a very affordable HW support as it can be seen in Fig. 12. The RUs are analyzed sequentially, extracting the task

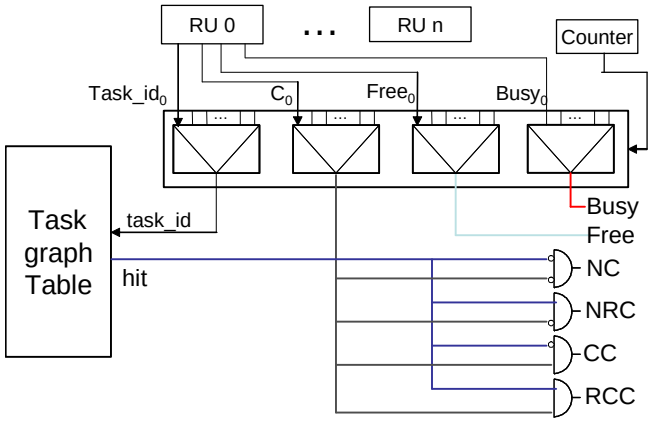

Fig. 12. HW to label the candidates.

information using a counter and some multiplexers. This information includes whether the task is critical or not. In order to check if the task is going to be executed in the near future, the replacement module uses the hit operation of the task-graph table. With this information, it is straight forward to identify the category of the task. The replacement module also includes a simple state machine and a register that stores the information of the best candidate found. Finally, it includes a comparator that identifies whether a given task can be reused (in this case the reconfiguration is cancelled and the system generates a *reused_task* event). The comparator and the implementation of the state machine are not included in the figure for simplicity.

C. Control unit and event management

As mentioned before, we have integrated our scheduler in a simulation platform that allows simulating the reconfiguration and execution of the RUs using programmable timers.

In this platform two registers are associated with each timer. These registers store the state of the RU (*idle*, *reconfiguring*, *ready to execute* and *executing*) and the information of the loaded task. The RUs also include HW support to generate some run-time events whenever the corresponding timer indicates that a task has finished its reconfiguration or execution. These events are stored in an event queue. Since it is possible that several RU modules generate events simultaneously, the queue includes an arbiter with a fixed priority scheme that prevents access conflicts.

The control unit is basically a state machine that extracts the events from the queue and carries out the actions described previously in Fig. 7. This unit also takes into account the mobility of the tasks to decide whether to delay a given reconfiguration or not. Fig. 13 depicts the HW support needed to make this decision. Basically a counter stores the number of events skipped so far and a comparator identifies if the reconfiguration can be delayed without introducing overheads (i.e. if the mobility of the task is greater then the number of skipped events). In that case, if the replacement module has selected a critical task as victim, the reconfiguration is delayed and the skipped events counter is incremented.

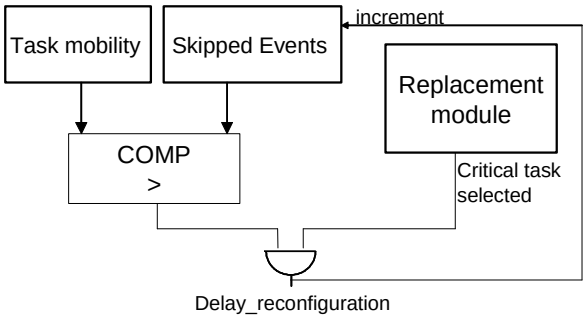

Fig. 13. HW support to implement the skip-event feature.

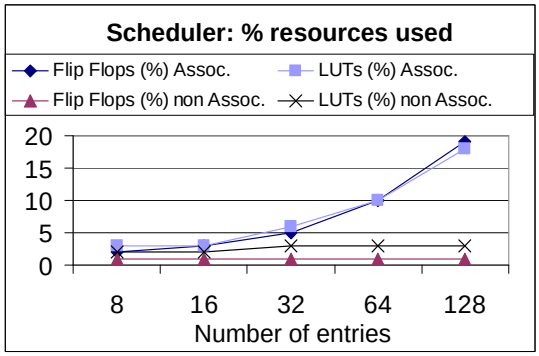

Fig. 14. Resources consumption of the scheduler both for the associative table (AT) and the non-associative one.

D. Synthesis results

Fig. 14 shows the resources used for the HW implementation of our scheduler in the Xilinx™ Virtex II PRO XC2VP30 FPGA for both versions of the task-graph table. The resources needed by the scheduler with the associative table grow linearly with the number of table entries. This limits the scalability of the scheduler, since for large systems it consumes an important percentage of the FPGA resources. Our current implementation has solved this problem and consumes less than 5% of the available resources even for large tables. Moreover, in both cases, the implementation of the scheduler demands only 1% of the BRAMs (this is not presented in the figure for simplicity). An additional 4% of the FPGA slices has also been used to implement a DMA controller in order to optimize the transactions in the system bus. This controller is not part of our scheduler and can be used by any other component of the system.

Regarding the supported clock frequency, the current scheduler supports frequencies up to 100 MHz even for tables with a large number of entries. Our previous implementation also supported a 100 MHz clock frequency, but only for small tables. For larger tables the clock frequency was significantly reduced. For instance, the maximum frequency for the associative table with 128 was 65 MHz. Hence the new design of the task-graph table reduces the area cost and improves the performance of the scheduler.

VII. PERFORMANCE EVALUATION

In this section we will evaluate the performance of our scheduler. For that purpose, we have tested it using a set of

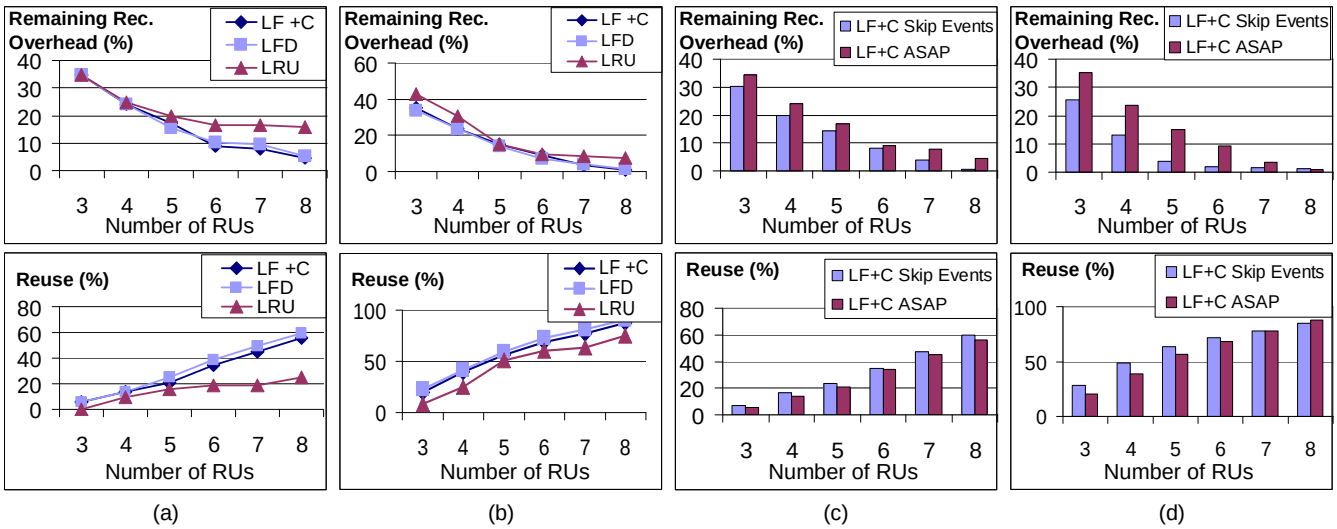

Fig. 15. Evaluation of the execution the task-graphs in groups 1 (a) and 2 (b) with different replacement policies; and comparison between an ASAP scheduler and our scheduler (Skip Event) applying in both cases the LF+C replacement policy for tasks in group 1 (c) and 2 (d).

task graphs extracted from actual multimedia applications. The applications used to extract the task graphs are: two versions of a JPEG decoder (JPEG and Parallel-JPEG), a MPEG-1 encoder, a Pattern recognition application (that applies the Hough transform), and a 3D rendering application based on the open source Pocket-GL library (Pocket GL). In the latter case the application includes 20 different task graphs. Table 1 presents the number of nodes, and the initial execution time of each graph. This initial execution time represents an ideal scenario with no delays due to the run-time management or due to the reconfigurations. We have tested our system separately for two groups of graphs. In the first group we have included the first four applications, and in the second group we have used the graphs from the Pocket-GL application. For simplicity, in Table 1 these graphs have been grouped in four categories (from A to D) taking into account the number of nodes. The table presents the average execution time for each category. It also includes the delays due to the reconfiguration overheads when using an on-demand approach, assuming a reconfiguration latency of 4 ms, which is the time needed to reconfigure the larger task used. As it can be seen in the table the original reconfiguration overheads are very important, especially for the Pocket GL application, where it can even be greater than the task-graph execution time.

As a first experiment we have tested the efficiency of our replacement policy (LF+C). To this end we have compared it with two well-known replacement policies: *Least Recently Used* (LRU) and *Longest Forward Distance* (LFD). As it was already explained in Section II, LFD is the optimal policy regarding reuse, but it can only be applied when all the future events are known, hence it cannot be applied in dynamic systems. However, it can be used to obtain an upper-bound value for the reuse. This is done recording all the events generated during the execution of a given experiment and afterwards, repeating the experiment but using LFD.

For the task graphs of the first group we have evaluated our system when executing all the possible combinations of two of these task graphs. For each combination, we execute the two

TABLE I
DETAILS OF THE TASK-GRAPHS USED AS BENCHMARKS IN THE PERFORMANCE EVALUATION

Task graph	Group	Number of tasks	Initial execution time (ms)	On-demand reconfiguration overhead (ms)
JPEG	1	4	79	16
Parallel-JPEG	1	8	54	20
MPEG-1	1	5	37	20
HOUGH	1	6	94	16
POCKET GL (A)	2	2	4	8
POCKET GL (B)	2	4	16	16
POCKET GL (C)	2	5	27	20
POCKET GL (D)	2	6	49	24

graphs alternatively simulating an execution pattern (e.g. JPEG – Parallel-JPEG – JPEG – Parallel-JPEG...). Each combination has been executed several iterations, and we have measured the average results without taking into account the first iteration, because during the first execution it is not possible to reuse anything. On average, 10.5 tasks were executed in each of these experiments; hence there were always significantly more active tasks than RUs. We are particularly interested in analyzing these situations since these experiments are so demanding that the only way to obtain a good performance is to apply an efficient replacement policy.

Fig. 15 (a) and (b) show the average results regarding the reconfiguration overhead and task reuse for a system with a variable number of RUs. The Remaining Reconfiguration Overhead percentage is defined as the percentage of the original reconfiguration overhead that remains after applying the scheduling techniques. The Reuse percentage is defined as the number of reused tasks divided by the total number of executed tasks.The system applies the same ASAP prefetch approach in the three cases. As it can be seen in Fig. 15 (a) and (b), the prefetch technique is a powerful way to reduce the reconfiguration overheads, because it can remove most of the original overhead even when no significant reuse is achieved (for instance, for three RUs using LRU no task is reused). However, the remaining overhead is still important (35% of the original overhead). Hence, a good replacement technique

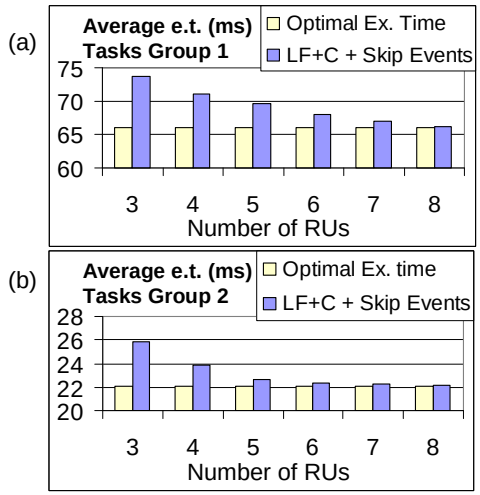

Fig. 16. Comparison of the average execution times (e.t.) in our experiments with the optimal execution times.

may have a very positive impact in the execution of these applications. Of course as more RUs are included in the system, the reuse percentage increases and this helps to reduce the overhead for the three replacement policies. Nevertheless, the results obtained when using a LRU approach are clearly suboptimal. LFD greatly increases the reuse percentage, and reduces the overheads. Finally LF+C obtains almost as good results regarding reuse as LFD, and in some cases even better performance. The reason is that although LF+C reuses a slightly smaller percentage of tasks than LFD, most of these tasks are critical, since LF+C assigns them more priority, and reusing critical tasks has a direct impact in the performance of the system.

For the Pocket GL (Fig. 15 (b)) we have simulated 500 iterations. In each iteration one of the 20 task graphs was randomly selected from the Pocket-GL library. Although there are a lot of different task graphs, reusing tasks was possible because all of them include some common tasks (in total there are 10 different tasks). The results for this experiment confirm that our replacement policy achieves almost optimal results regarding reuse (just 3% worse than LFD). In this case the prefetch approach hides 55% of the reconfiguration overheads. However, if the system includes enough RUs and it applies our efficient replacement policy the overhead disappears, whereas if it uses the LRU approach, a significant percentage (almost 10%) will penalize the system performance.

All these results have been obtained combining the replacement policies with an ASAP scheduling approach. These techniques are broadly used at-run time because they provide good results and they do not generate an excessive penalty due to the run-time computations. However, they often lead to local-optimum decisions. Hence we have included in our scheduler some simple support to delay certain reconfigurations in order to find better schedules as explained in Section *V.a.2*. Figures 15 (c) and 15 (d) present the benefits of this approach both regarding reuse and performance.

The results demonstrate that delaying some reconfigurations can improve significantly the performance of the system, and even increase the percentage of reused tasks. The results of Fig. 15 (d) are especially good, since our scheduler clearly outperforms the purely ASAP approach.

Finally, Fig. 16 shows the results of our experiments regarding execution time. Fig. 16 (a) and (b) show the average execution times for the task graphs of group 1 and 2, respectively In both cases, we compare our results with the *optimal* execution times that have been obtained assuming that *there is not any reconfiguration overhead*, and using a branch&bound scheduler. Of course these optimal schedules cannot be obtained at run-time, since the branch&bound scheduler needs from hundred of milliseconds to seconds to find the optimal schedules even for these relatively small graphs. In any case, these results demonstrate that our scheduler provides near-optimal results as long as enough RUs are available to take advantage of our replacement policy. The average number of tasks executed in the experiment with the first group of tasks is 11.5, whereas in the second group is 10. Hence we are obtaining near-optimal results even when the number of task doubles the number of RUs.

We have also evaluated the delays that the run-time scheduler generates in the task-graph executions due to the run-time computations. The SW implementation of our scheduler generates on average an execution-time overhead of only 1% for the graphs in the group 1. However, it generates almost a 10% overhead in the execution of the Pocket GL graphs. On average 36% of these overheads are due to the HW/SW communications and the remaining ones are due to the run-time scheduling computations that must deal with complex data structures such as graphs and lists. If the overheads generated by our SW scheduler are not acceptable for a given system, the best solution is to include the HW implementation that can almost completely eliminate them. In this case the scheduling computations are carried out at run-time in just a few clock cycles, and the average delay generated in the execution of a graph is 0.02 ms including the time needed to send the information of the graph from the PowerPC to our scheduler.

VIII. CONCLUSIONS

An efficient HW multitasking system based on reconfigurable HW needs some specific support to reduce the reconfiguration overheads. We have developed a scheduler that provides this support applying a prefetch scheduling technique, in order to load most configurations in advance, and a replacement technique, which reduces the number of reconfigurations needed and even improves the results obtained by the prefetch technique assigning more priority to the most critical tasks.

As the experimental results have shown, the cooperation between the prefetch and the replacement techniques provides very good results hiding most of the reconfiguration delays. Moreover, the possibility of delaying some reconfigurations in order to escape from local-optimum decisions can greatly improve the results for some run-time conditions (as in Fig. 15 (d)). Finally, with the developed HW support, and using the information extracted at design-time, all these techniques can be applied at run-time while generating a negligible delay due

to their computations. Hence our scheduler provides a transparent and efficient management of the execution of task-graphs for reconfigurable multi-tasking systems. This management can significantly improve the performance of the system, and even reduce the energy consumption, because the replacement technique can largely reduce the number of reconfigurations needed.

IX. REFERENCES

[1] www.sony.net/Products/SC-HP/cx_news/vol42/pdf/sideview42.pdf

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

[3] L.A. Belady, "A study of replacement algorithms for virtual storage computers", *IBM Systems Journal*, vol. 5, pp. 78-101, no. 2, 1966.

[4] S. Banerjee, E. Bozorgzadeh and N. Dutt, "Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration", *IEEE Trans on VLSI Systems*, vol. 14, pp. 1189-1202, November 2006.

[5] E. Pérez-Ramo, J. Resano, D. Mozos and F. Catthoor, "Reducing the reconfiguration overhead: a survey of techniques", in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, USA, 2007, pp. 191- 194.

[6] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices", in *Design, Automation & Test in Europe (DATE)*, Munich, Germany, 2003, pp. 290-295.

[7] Y. Qu, J.-P. Soininen and J. Nurmi, "A parallel configuration model for reducing the run-time reconfiguration overhead", in *Design, Automation & Test in Europe (DATE)*, Munich, Germany, 2006, pp. 1-6

[8] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling", *ACM Trans. on Embedded Computing Systems*, vol. 3, pp. 385 – 406, May 2004.

[9] Z. Pan and B. E. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures", *IEEE Trans. on VLSI Systems*, vol. 16, pp. 1465 – 1474, November 2008.

[10] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles", *IEEE Trans. on VLSI Systems*, vol. 16, pp. 24 – 33, January 2008.

[11] K. N. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures", *IEEE Trans. on VLSI Systems*, vol. 14, pp. 1010-1023, September 2006.

[12] S. Banerjee, E. Bozorgzadeh and N. Dutt, "Exploiting application data-parallelism on dynamically reconfigurable architectures: placement and architectural considerations", *IEEE Trans. on VLSI Systems*, vol. 17, pp. 234 – 247, issue 2, 2009.

[13] K. Kosciuszkiewicz, F. Morgan and K. Kepa, "Run-time management of reconfigurable hardware tasks using embedded Linux", in *International Conference on Field Programmable Technology (ICFPT)*, Kitakyushu, Japan, 2007. pp. 209 – 215.

[14] H. Kwok-Hay So and R. W. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH", *ACM Transactions in Embedded Computing Systems*, vol. 7, pp. 259-264, 2008.

[15] W. Fu and K. Compton, "Scheduling intervals for reconfigurable computing", in *IEEE Symposium on field-programmable custom computing machines (FCCM)*, Palo Alto, CA, USA, 2008, pp. 87 – 96.

[16] K. Danne, R. Miihlenbernd and M. Platzner, "Executing hardware tasks on dynamically reconfigurable devices under real-time conditions", in *International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 2006, pp. 1 – 6.

[17] 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)*, Monterey, CA, USA, pp. 187-195

[18] J. Resano, D. Mozos, D Verkest and F Catthoor, "A reconfiguration manager for dynamically reconfigurable hardware", *IEEE Design&Test*, vol. 22, pp. 452-460, September 2005.

[19] J. Resano, D. Mozos and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware", in *Design, Automation & Test in Europe (DATE)*, Munich, Germany, 2005, pp. 106-111.

[20] J. Resano, J.A. Clemente, C. González, D. Mozos, and F. Catthoor. "Efficiently scheduling runtime reconfigurations", *ACM Trans. Design Automation of Electronic Systems*, Vol. 13, pp. 58-70, September 2008.

[21] J. A. Clemente, C. González, J. Resano and D. Mozos. "A task graph execution manager for reconfigurable multi-tasking systems", Microprocessors & Microsystems, vol. 34, pp. 73 – 83, June 2010.

[22] J. A. Clemente, C. González, J. Resano and D. Mozos, "A hardware task-graph scheduler for reconfigurable multi-tasking systems", in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 2008, pp.79-84.

[23] C. Wong, P. Marchal and P. Yang, "Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform", in *International Conf. on HW/SW Codesign and System Synthesis (CODES),* Copenhagen, Denmark, 2001, pp. 170-175.

Juan A. Clemente was born in 1984. He started studying a Computer Science Degree at Universidad Complutense de Madrid (UCM), Spain, in 2002 and finished in 2007.

Since then he is a PhD student and works there as a teaching assistant and also as a researcher in the GHADIR group. He has also an active collaboration with the Embedded Systems Laboratory in the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. In his work he mainly focuses on developing scheduling techniques to hide reconfiguration latencies in dynamically reconfigurable systems.

Javier Resano received the Bachelor Degree in Physics in 1997, a Master Degree in Computer Science in 1999, and the PhD 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 also collaborates with the Digital Design Technology Group from IMEC-laboratory since 2002. His research has been focused in hardware/software co-design, task scheduling techniques, Dynamically Reconfigurable Hardware and FPGA design.

Carlos González was born in 1984. He started studying a Computer Science Degree at Universidad Complutense de Madrid (UCM) in 2002 and finished in 2007.

Since then he is a PhD student and since 2008 works there as a teaching assistant. In his work he mainly focuses on applying run-time reconfiguration in aerospace applications. He has recently started with this topic, working with algorithms that deal with hyperspectral images. He is also currently collaborating with Juan Antonio Clemente on developing techniques to hide reconfiguration latencies in dinamically reconfigurable systems.

Daniel Mozos obtained a B.S. in physics and a Ph.D. in computer science from the Universidad Complutense de Madrid.

He is a permanent professor in the Computer Architecture and Automation Department of the Universidad Complutense de Madrid, where he leads the GHADIR research group on dynamically reconfigurable architectures. His research interests include design automation, computer architecture, and reconfigurable computing.