A Replacement Technique to Maximize Task Reuse in Reconfigurable Systems

Abstract—**Dynamically reconfigurable hardware is a promising technology that combines in the same device both the high performance and the flexibility that many recent applications demand. However, one of its main drawbacks is the reconfiguration overhead, which involves important delays in the task execution, usually in the order of hundreds of milliseconds, as well as high energy consumption. One of the most powerful ways to tackle this problem is configuration reuse, since reusing a task does not involve any reconfiguration overhead. In this paper we propose a configuration replacement policy for reconfigurable systems that maximizes task reuse in highly dynamic environments. We have integrated this policy in an external task-graph execution manager that applies task prefetch by loading and executing the tasks as soon as possible (ASAP). However, we have also modified this ASAP technique in order to make the replacements more flexible, by taking into account the mobility of the tasks and delaying some of the reconfigurations. In addition, this replacement policy is a hybrid design-time/run-time approach, which performs the bulk of the computations at design time in order to save run-time computations. Our results illustrate that the proposed strategy outperforms other state-of-the-art replacement policies in terms of reuse rates and achieves near-optimal reconfiguration overhead reductions. In addition, by performing the bulk of the computations at design time, we reduce the execution time of the replacement technique by 10 times with respect to an equivalent purely run-time one.**

Keywords: reconfigurable architectures, task replacement, task scheduling, Field Programmable Gate Arrays (FPGAs)

I. INTRODUCTION

In the last few years, many applications in fields such as image processing, multimedia and artificial vision have become more and more computationally intensive. Thus, embedded execution platforms have evolved into complex Systems-On-Chip (SoC), most of which can include reconfigurable resources (such as FPGAs), in order to adapt themselves to different execution contexts [1, 2].

However, a well-known drawback of FPGAs is the delay related to their reconfiguration processes, which can be of the order of hundreds of milliseconds [3], in addition to high energy consumption due the access to an external memory and the movement of a large amount of data [4]. These overheads can greatly degrade the performance of the applications if they demand reconfigurations very often. Hence, embedded systems that include reconfigurable resources must implement *a good scheduling strategy* that explicitly takes into account the impact of the dynamic reconfigurations in order to guarantee that the applications achieve their required performance. In addition, modern embedded systems are characterized by their high degree of dynamism, i.e., the status and the workload of the system can greatly vary dynamically. Hence, such a scheduling technique should be applied *at least partially at run time*, in order to be able to deal with dynamic and unexpected events.

One of the most powerful techniques proposed to reduce the reconfiguration overhead is to reuse the reconfigurations that have been loaded previously in the system, especially when dealing with tasks that are executed recurrently. In addition, this approach can be combined with other state-of-the-art scheduling techniques, such as task prefetch [5]. Hence, a good *replacement policy* that maximizes the configuration reuse may have an important additional impact in the reduction of the reconfiguration overheads. However, very few works have addressed this problem. One could mistakenly think that the reason is that replacement has been already extensively studied for other problems such as cache replacement [6]. However, configuration replacement and cache replacement are very different problems. The reason is the large reconfiguration latency. Cache replacement techniques typically must be applied within one clock cycle since otherwise they are not useful. On the contrary, more complex policies can be used for configuration replacement, since they are invoked very few times in comparison with cache replacement ones. Moreover, if applied efficiently at run time, the time needed to decide the replacement victim is almost negligible when compared with the execution time of the applications and the reconfiguration latencies (both of them typically in the order of milliseconds).

In this paper we present a hybrid design-time/run-time replacement policy for applications that run in a reconfigurable multitasking system that is composed of a set of equal-sized reconfigurable units (RUs), such as the ones proposed in [7, 8]. These applications are represented as Directed Acyclic Graphs (DAGs), where the nodes represent computational tasks and the edges, dependencies among them. We have integrated our replacement policy in an external task-graph manager [9] that applies task prefetch by loading and executing the tasks as soon as possible (ASAP). However, it is well-known that ASAP techniques often fall into locally optimal solutions. Hence in certain situations some reconfigurations are delayed in order to improve the task reuse rates (the replacement policy only delays a reconfiguration if this can be done without introducing any additional execution-time overhead). For this purpose, an important part of this technique is carried out at design time in order to extract some useful information about the incoming task graphs that will be used later at run time. Our results will illustrate that by increasing task reuse, our replacement policy achieves near-optimal reconfiguration overhead reductions while introducing a negligible run-time penalty. This also has a positive impact in the energy consumption overhead, since by increasing task reuse, fewer reconfigurations are carried out.

The rest of the paper is structured as follows. Section II illustrates the problem at hand by means of two examples. Section III overviews other relevant works about task scheduling and configuration replacement on reconfigurable systems. Section IV gives more details about the external task-graph manager upon which we have integrated our replacement technique. Section V describes our replacement policy and

Section VI presents the experimental results. Finally Section VII summarizes this article with final conclusions.

II. Work Environment and Motivational Examples

Our scheduler receives as input a set of applications to be executed (represented as one or several DAGs), and manages their execution taking into account their internal dependencies and the dynamic status of the system. The scheduler processes the task graphs sequentially. In each instant of time, it has a sorted list of enqueued applications that have to be executed next (that we have called *Dynamic List* or *DL*). However, this list can be dynamically updated with new applications, which are also inserted in *DL*, for instance, following a First In-First Out (FIFO) policy (making this decision is out of the scope of this work). In any case, *the complete list of applications* that will be executed is unknown at design time. Fig. 1 describes an example of how *DL* is updated at run time, assuming that the applications are enqueued following a FIFO policy. First of all, this sorted list is composed of 3 applications: JPEG, MPEG1 and HOUGH (Fig. 1a). After the execution of JPEG, two new instances of MPEG1 come for their execution. Hence the newly executed JPEG is deleted from *DL* and the two new instances of MPEG1 are added at the end of the list, which is updated and composed now of four items: {MPEG1, HOUGH, MPEG1, MPEG1} (Fig. 1b). Then, after the execution of the first MPEG1, the scheduler does not receive any additional application. Hence the first MPEG1 is deleted from *DL* and no additional application is enqueued (Fig. 1c). The system continues executing applications until *DL* is empty. Note that at the beginning of this process, the scheduler only knows 3 out of the whole sequence of 5 applications that will be executed. Hence, the run-time scheduler can only use the information that is available at run time in order to make the scheduling decisions. In addition, for the sake of simplicity, we assume that *DL* is updated only at the end of the execution of the applications, not during their execution.

At run time the scheduler deals with the task-graph execution deciding in which order the tasks are going to be loaded and executed. Each time that the scheduler needs to load a new task, if none of the available RUs is free, one of the previously loaded tasks must be replaced. We propose to include a specific replacement module to make this decision. Since we are trying to maximize task reuse, we propose to apply the Longest Forward Distance (LFD) replacement policy [10] using the information about the tasks enqueued in *DL* at that moment. LFD selects the candidate that will be requested farthest in the future and, if it is applied over all the complete sequence of tasks that will be executed, it guarantees the *optimal reuse rate*. Since we apply LFD over just a subset of the total sequence of tasks (which are those that are enqueued in *DL* at the moment of performing a replacement), we have called it *Local LFD*. The reason of using it is to maximize task reuse, which has a very positive impact on the performance of the running applications. In addition, it also reduces the energy consumption and the pressure over the external memory and the system bus, since reconfigurations involve moving larges amounts of data from an external memory to the FPGA.

Fig. 2 shows an example of how our replacement works in the execution of two task graphs in a system with 4 RUs, and

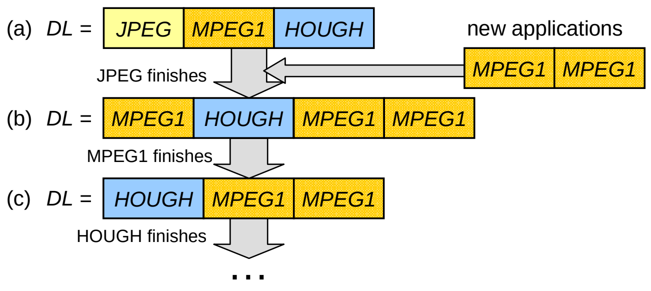
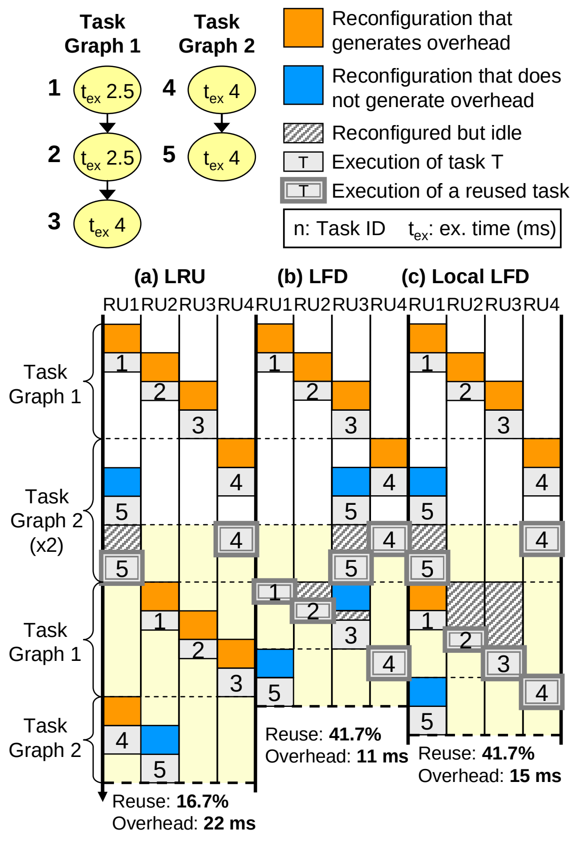


Figure 1. Example of update of the sorted list of tasks (Dynamic List, or *DL*) of our scheduler, which contain the tasks that will be executed in the near future

compares *Local LFD* (Fig. 2c) with LFD (Fig. 2b) and other well-known replacement policy, Least Recently Used (LRU, Fib. 2a). For each case, the figure shows the reuse rates and the delays generated with respect to an ideal schedule where no reconfiguration overhead is generated. In order to simplify the example, for *Local LFD* we assume that there is always only one task graph enqueued in *DL*. In other words, *Local LFD* only has the information about the following task graph. In addition, in this example we assume that all the tasks are loaded ASAP; i.e., without delaying any reconfigurations.

In all the cases, Fig. 2 shows that the prefetch technique is a powerful means to hide the reconfiguration overheads since it hides most of them either totally or partially. However, a replacement technique can be used in this context in order to achieve further overhead reductions. As the figure shows, the reuse rates greatly differ depending on the replacement policy used. Thus, if the scheduler uses LRU (Fig. 2a), the reuse rate is very low (16.7%), since there are 5 different tasks competing for 4 RUs. However, if LFD is applied (Fig. 2b), the scheduler achieves the optimal reuse rate (41.7%) and also reduces the reconfiguration overhead to half (by 11 ms). The reason is that when the first replacement has to be made (when Task 5 has to be loaded), LFD selects RU3 as victim, which is the candidate (out of all the possible ones, namely Tasks 1, 2 and 3) that will be requested farthest in the future. This makes possible to reuse Tasks 1 and 2 from the second instance of Task Graph 1. Similarly, when Task 3 has to be loaded for the second time, LFD selects RU3 (which has Task 5 loaded). This makes possible to reuse Task 4 during the following execution of Task Graph 2 and to eliminate its reconfiguration overhead.

Finally, Local LFD (Fig. 2c) also achieves the optimal reuse rate (41.7%), and its reconfiguration overhead (15 ms) is close to the optimal one (11 ms). In this case, the difference with respect to LFD is in the load of the first instance of Task 5, which this time selects RU1 as victim. The reason is that at the moment that replacement is carried out, *DL* contains only the second instance of Task Graph 2 and it does not know that Task Graph 1 will be executed again. Hence, in this case all the existing candidates (RU1, RU2 and RU3) have the same longest forward distance. Hence Local LFD selects the first candidate it finds, which is RU1. However, this limitation disappears if there are *two tasks* enqueued in *DL* in the moment that replacement is carried out. In this case, Local LFD achieves the same results as LFD. In any case, the example illustrates how Local LFD can achieve significant reconfiguration overhead



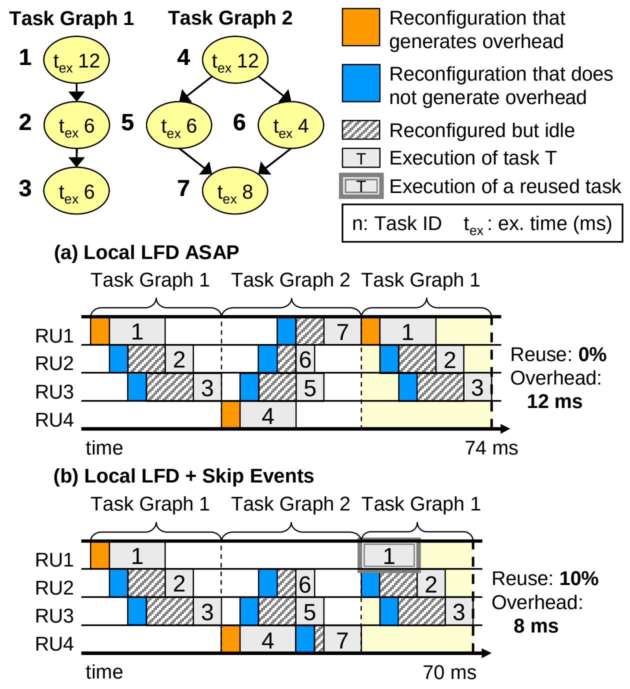
Task Graph 1
Task Graph 2
Reconfiguration that generates overhead
Reconfiguration that does not generate overhead
Reconfigured but idle
Execution of task T
Execution of a reused task
n: Task ID t_{ex}: ex. time (ms)
(a) LRU
(b) LFD
(c) Local LFD
Reuse: 16.7% Overhead: 22 ms
Reuse: 41.7% Overhead: 11 ms
Reuse: 41.7% Overhead: 15 ms

Figure 2. Motivational example for a system with 4 RUs and different replacement policies: LRU, LFD and our Local LFD. The reconfiguration latency is always 4 ms

reductions and almost optimal reuse rates even if a very limited amount of information about the future is available.

In the previous example we were making the replacements following an ASAP approach. However, as mentioned above, we want to make the replacements more flexible in order to improve the reuse rates. For this purpose, the replacement module delays a reconfiguration if it knows that the replacement victim is going to be used in the near future, and this delay is not going to introduce any additional overhead in the execution time.

Fig. 3 shows a motivational example of how Local LFD can escape from locally optimal solutions by delaying some reconfigurations. The figure shows what happens if the two task graphs are executed in a system with 4 RUs and using our *Local LFD* replacement policy (following an ASAP approach, and also allowing to delay a reconfiguration as long as it does not introduce any additional delay in the execution). In both cases we assume that *DL* always contains only one enqueued task graph. As the example shows, if an ASAP approach is used (Fig. 3a), the latencies of 7 out of 10 tasks are hidden. However, in this case when Task Graph 1 is executed for the second time, no task is reused; hence the achieved reuse rate is 0%. In addition, the total reconfiguration overhead is 12 ms. However, the results improve if the replacement module decides to delay Task 7 (Fig. 3b). In this case, the design-time scheduler has already detected that this task can be safely



Task Graph 1
Task Graph 2
Reconfiguration that generates overhead
Reconfiguration that does not generate overhead
Reconfigured but idle
Execution of task T
Execution of a reused task
n: Task ID t_{ex}: ex. time (ms)
(a) Local LFD ASAP
Reuse: 0% Overhead: 12 ms
74 ms
(b) Local LFD + Skip Events
Reuse: 10% Overhead: 8 ms
70 ms
time

Figure 3. Motivational example of our skip-event approach, for a system with 4 RUs and comparison with an purely ASAP approach. The reconfiguration latency is always 4 ms

delayed (Section IV 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 4). Thus, when Task 4 finishes its execution the replacement policy can select between two replacement victims (tasks 1 and 4), and it will select Task 4 since it is not going to be used again in the near future. As a consequence, when Task Graph 1 is executed again, Task 1 will be directly reused and its reconfiguration will not generate any delay in the execution.

III. Related Work

Much work has been published in the literature about task scheduling on reconfigurable systems. These approaches explicitly take into account the reconfiguration overhead during the scheduling process in order to minimize the impact of the dynamic reconfigurations. They can be classified into two big groups: *design-time* and *run-time scheduling*. *Hybrid techniques* (which combine the features of design-time and run-time ones) have also been proposed.

Design-time approaches consist in applying the scheduling *before the application is executed*. Integer Linear Programming (ILP) formulations [11, 12] to schedule DAGs in reconfigurable systems fall into this group. These approaches can reach optimal results; however they are impractical even for small instances, since they do not scale well when increasing the sizes of the target architectures and of the input task graphs. This is the reason heuristic methods are usually preferred to obtain sub-optimal results in reasonable time. One of the most widespread ones is task prefetch [5], which consists in carrying out the reconfigurations in advance and hence overlapping them with the execution times of other tasks. This methodology has been applied to other techniques, such as list-based scheduling, in order to adapt them to reconfigurable hardware [13]. Design-time scheduling can also be useful to develop task replacement techniques in order to maximize task reuse. In fact, the longest forward distance replacement policy (LFD) [10] selects the candidate that will be requested farthest in the future and, as hinted above, it is proved to guarantee the *optimal reuse rate*. However, the main limitation of all these approaches is that they can only obtain good results if they deal with systems which behavior is well-defined at design time. Hence they are unsuitable for contexts with certain degree of dynamism.

This is the reason *run-time scheduling* has also been extensively studied in the literature. These techniques make the scheduling decisions *while the application is being executed*. Hence they are able to adapt themselves to highly dynamic contexts. For instance, in [14] the authors propose a run-time scheduler for HW tasks in a reconfigurable device that is partitioned into a set of reconfigurable blocks with different sizes. In this work, the authors explore different scheduling policies. Task prefetch [5] is also applied in all of them. However, their main limitation is that they do not include any specific support for task replacement in order to maximize task reuse. Hence their task reuse rates are still too low, which limits the system performance. In particular, very few techniques specifically designed for reconfigurable hardware have been proposed in the literature to address the efficient task replacement in order to maximize task reuse. In fact, the majority of them just consist in adapting general cache replacement algorithms [6] to reconfigurable hardware and to use them for the run-time task replacements [15]. For instance, in [16] the authors propose several run-time cache replacement approaches and adapt them to different reconfigurable architectures. However, their reuse rates are very far from the optimal ones that can be obtained with LFD (which they use as upper-bound), and the reconfiguration latencies that are not hidden are always from 2 to 3 times the ones that are obtained with LFD. Another interesting approach is [17], where the authors propose a run-time list-based replacement technique that collaborates with an external scheduler called Task Concurrency Manager (TCM), which applies task prefetch by loading the tasks following an ASAP approach. However, this technique greatly relies on the output of the TCM scheduler and just makes some adjustments at run time, which limits its ability to deal with the execution of several consecutive tasks under highly dynamic conditions. On the contrary, the task replacement policy that we propose in this paper is not dependent on any specific external scheduler. Its only requirement is to perform a previous design-time phase in order to analyze the incoming task graphs. In addition, the fact of carrying out an important part of the computations at design time allows to greatly reduce the run-time computations (unlike [17]). Finally, our approach makes the task replacements more flexible by delaying some of the reconfigurations depending on their mobility and the current status of the system.

As hinted above, the task replacement policy that we propose in this paper falls into the group of *hybrid design-time/run-time scheduling*; which appears as something in the middle between these two ends. The basic idea is to apply a run-time approach to deal with certain degree of dynamism, but carrying out as many computations as possible at design time in order to save run-time computations. Some general scheduling techniques have also been proposed in this field, such as [18], which main idea is to generate several schedules at design time and then select at run time the most appropriate one. In [19], the authors propose a hybrid design-time/run-time prefetch approach and integrate it in the scheduler initially proposed in [18]. However, none of them address the problem of maximizing task reuse under highly dynamic conditions. Hence, it is clear that this field needs to be further explored.

IV. Scheduling Environment

As mentioned above, we have integrated our replacement policy in an external task-graph execution manager [9]. It manages the execution of the incoming task graphs by taking into account their internal dependencies and the status of the system. For this purpose, it only considers some discretized time instants following an event-triggered approach. Thus, when *certain events* happen the manager will carry out the proper actions in order to load and execute the tasks following an ASAP approach. 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.

First of all, the manager performs a pre-processing of the task graphs at design time in order to identify in which order the tasks must be loaded in the system. Thus, the tasks are stored in a sorted *sequence of reconfigurations* that will be followed at run time. Once this sequence of reconfigurations has been obtained, the execution manager carries out the execution of the task graphs. Three different events trigger the execution of the manager: *new_graph*, which is generated when a new task graph is received; *end_of_reconfiguration*, which is generated when a new task has been loaded or when the RU has identified that a task can be reused since it was already loaded in a previous execution; and *end_of_execution*, which is generated when a task finishes its execution.

Fig. 4 depicts the actions triggered by each event. Thus, for the *new_task_graph* event (1), if the reconfiguration circuitry is free, the system attempts to trigger the reconfiguration of the first task in the reconfiguration sequence by calling the *replacement module* (2-4). For the *reused_task* and *end_of_reconfiguration* events (5), the manager checks if the task that has been loaded or reused is ready to be executed (6); i.e., if all its predecessors have already finished their execution. In that case, the system starts its execution (7). Then, the manager invokes again the replacement module (9). Finally, for the *end_of_execution* event (10), the manager checks again if the reconfiguration circuitry is idle. In that case, it invokes the replacement module (11–13). After that, it updates the task-graph dependencies, decreasing by 1 the number of predecessors of each successor of the finished task (14). Then, the manager checks if any of the tasks that are currently loaded in any RU can start its execution (15–19).

V. Our Replacement Technique

Every time the manager has to load the next reconfiguration from the reconfiguration sequence, it invokes the replacement

```
/* task = task that triggered the event */
CASE event IS:
 1. new_task_graph:
 2.    IF (reconfiguration_circuitry_idle ()){
 3.       replacement_module (&rec_sequence);
 4.    }
 5. end_of_reconfiguration or reused_task:
 6.    IF (ready (task)){
 7.       start_execution (task);
 8.    }
 9.    replacement_module (&rec_sequence);
10. end_of_execution:
11.    IF (reconfiguration_circuitry_idle ()){
12.       replacement_module (&rec_sequence);
13.    }
14.    update_task_dependencies (&task);
15.    FOR (i := 0 TO NUMBER_OF_RUS){
16.       IF (RU_idle (i)) && (ready (task (i))){
17.          start_execution (task (i));
18.       }
19.    }
```

Figure 4. Pseudo-code of the external task-graph execution manager used

module (Fig. 4, Lines 3, 9 and 12). However, a previous design-time pre-processing of the task graphs must be carried out in order to assign to each task a value of *mobility*. Hence, our replacement policy is composed of a *design-time phase* and a *run-time one*, as Fig. 5 shows.

A. Design-Time Phase: Mobility Calculation

The objective of this phase is to assign to each task of the task graph a value of *mobility*, which defines how many times that reconfiguration can be delayed without generating any additional performance degradation. More specifically, this value represents how many events can be skipped before loading a task without generating any additional delay.

Fig. 6 depicts this algorithm. First of all, it obtains a *reference schedule* (*ref_sch*, Step 1) assuming that the mobility of all the tasks is 0. Then, an initial set of tasks (*TS*, Step 2) is initialized by selecting all the tasks from the task graph except the first task in the sequence of reconfigurations (since its mobility is 0). Then, if *TS* is not empty (Step 3), the algorithm extracts a task *t* from there (Step 4). Then, the iterative process (Steps 5-7) tentatively assigns a greater mobility value to *t* (Step 5) and obtains a new schedule (*new_sch*, Step 6), but assuming that *t* is delayed as many times as the value of *t.mobility*. The condition of Step 7 checks if it is feasible to assign that new mobility to *t* without generating any additional reconfiguration overhead. In that case, this iterative process is repeated at least one more time (back to Step 5). Otherwise, the algorithm restores the former mobility value to *t* (Step 8) and it removes *t* from *TS*, since its mobility has been calculated (Step 9). The external loop (Steps 3-9) is repeated for each task until *TS* is empty (Steps 3 and 10).

The proposed replacement technique

Design time:
-Mobility calculation
Run time:
-Task replacement

Figure 5. Steps of the proposed replacement technique

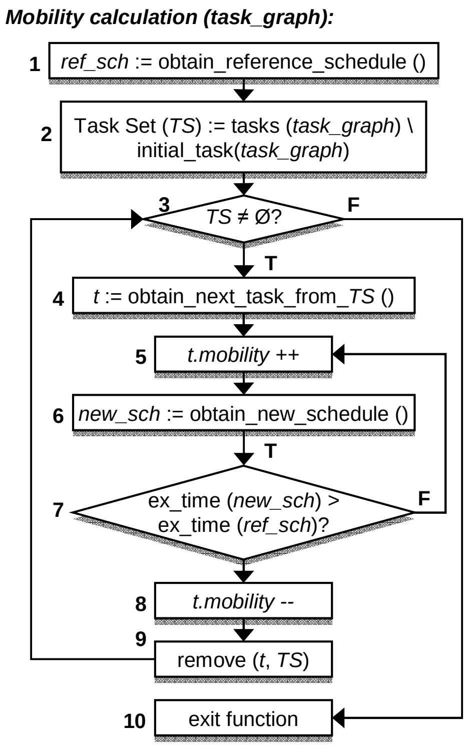

Mobility calculation (task_graph):
1
ref_sch := obtain_reference_schedule ()
2
Task Set (TS) := tasks (task_graph) \ initial_task(task_graph)
3
TS ≠ Ø?
F
T
4
t := obtain_next_task_from_TS ()
5
t.mobility ++
6
new_sch := obtain_new_schedule ()
T
7
ex_time (new_sch) > ex_time (ref_sch)?
F
8
t.mobility --
9
remove (t, TS)
10
exit function

Figure 6. Flowchart of the algorithm that assign the mobilities

Fig. 7 shows an example of the mobility calculation for the Task Graph 2 of Fig. 3. First of all, the algorithm obtains a reference schedule (Fig. 7a) assuming that all the tasks are loaded in the system following an ASAP approach. Then, the algorithm attempts to delay 1 event the reconfiguration of Task 5 (Fig. 7b). However, if this task is delayed, a new overhead of 6 ms is generated with respect to the reference schedule. Hence the mobility of Task 5 is set to 0. The mobility of Task 6 is also 0, since when the algorithm tries to delay it one event, an additional overhead of 2 ms is generated (Fig. 7c). Finally, Task 7 can be delayed once without any performance degradation (Fig. 7d). Thus, in the first iteration the scheduler firstly attempts to delay this reconfiguration one event. Since no additional reconfiguration overhead is generated, it then attempts to delay it 2 events, but in this case a new overhead of 2 ms is generated. Hence, the mobility of Task 7 is set to 1.

B. Run-Time Phase: Task Replacement

At run time, our replacement module (Fig. 4, Lines 3, 9 and 12) makes the replacement decisions each time the execution manager decides to load a new task. For this purpose, it implements the *Local LFD* replacement technique taking into account the information contained in the *Dynamic List* (*DL)* of our manager and the dynamic status of the system. In addition, if the selected victim is going to be used in the near future (i.e. inside the boundaries of *DL*) the replacement module can decide whether to delay that reconfiguration or not, depending on its mobility.

Fig. 8 depicts this function. First of all, it retrieves the next task from the reconfiguration sequence (Step 1) and selects the replacement victim for it following our Local LFD replacement

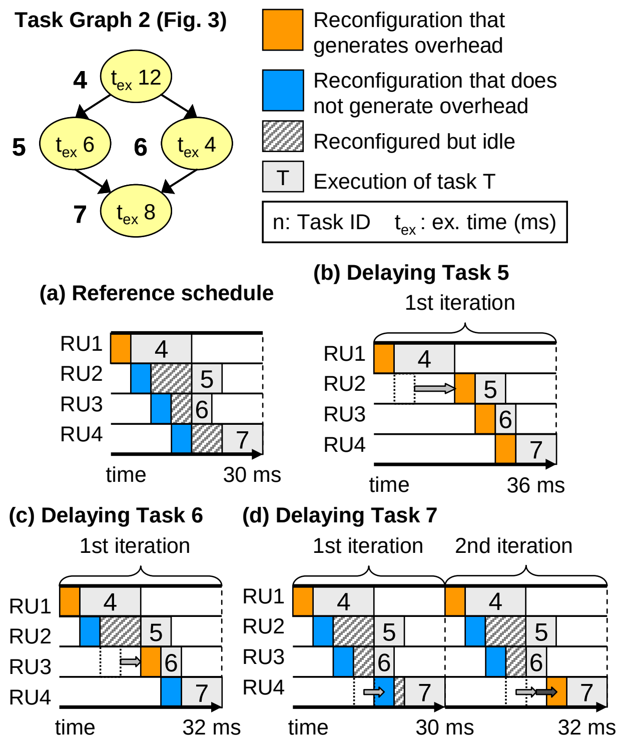
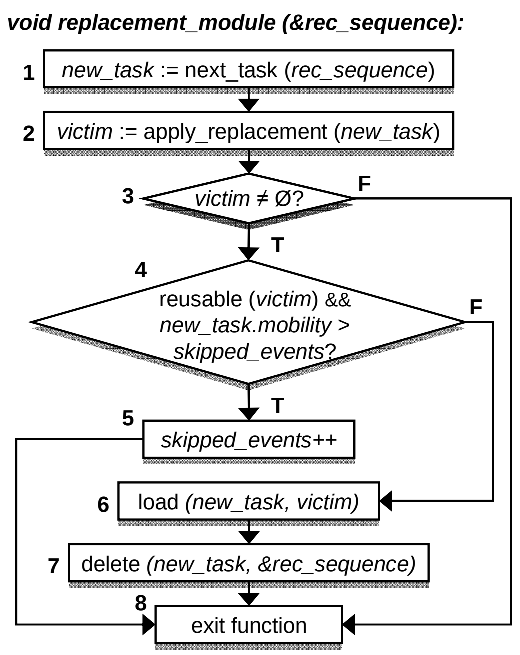


Figure 7. Example of the mobility calculation for the Task Graph 2 in the previous figure 3.The reconfiguration latency is always 4 ms

policy (Step 2). If such victim does not exist (there are no available candidates, Step 3) the replacement cannot take place and the function finishes (Step 8). Otherwise, the replacement module decides whether to load that task in the selected RU or not (Steps 4-8). For that purpose, the variable *skipped_events* keeps track of the number of events skipped so far in the execution of the current task graph (this variable is initialized externally to this function each time a new task graph starts its execution). In any case, if the replacement can take place, the function checks if the selected victim is going to be reused in the near future and if the mobility of the task is greater than the number of total skipped events at that moment (Step 4). In that case, the function just increases the number of skipped events so far (Steps 5, 8). Otherwise, it triggers the reconfiguration of *new_task* replacing the selected victim (Step 6) and deletes *new_task* from the reconfiguration sequence (Step 7).

VI. Experimental Results

We have carried out several experiments in order to test the efficiency of our replacement policy. For that purpose, we have tested it using a set of task graphs extracted from actual multimedia applications. These applications are: a JPEG decoder, a MPEG-1 encoder and a pattern recognition application (which applies the Hough transform). These task graphs are composed of 4, 5 and 6 nodes, respectively. We have evaluated the reconfiguration overheads and the reuse rates for our Local LFD technique. For that purpose, we have executed a sequence of 500 applications randomly selected from our set of benchmarks. In addition, we have also evaluated the run-time delays generated by our replacement policy in comparison with other well-known techniques: LRU and LFD.



void replacement_module (&rec_sequence):
1 new_task := next_task (rec_sequence)
2 victim := apply_replacement (new_task)
3 victim ≠ Ø?
F
T
4 reusable (victim) && new_task.mobility > skipped_events?
5 skipped_events++
6 load (new_task, victim)
7 delete (new_task, &rec_sequence)
8 exit function

Figure 8. Flowchart of our run-time replacement module

A. Performance evaluation

First of all, Fig. 9a shows the reuse rates achieved by our Local LFD replacement technique for different sizes of the *Dynamic List*: 1, 2 and 4 task graphs (displayed in *Local LFD (1)*, *Local LFD (2)* and *Local LFD (4)*, respectively). All the reuse rates are calculated as the number of reused tasks divided by the total number of executed tasks. In this case, we are assuming that the tasks are loaded following an ASAP approach; i.e., assuming that the mobility of all the tasks is 0. The figure also compares these rates with LRU and LFD. First of all, we can observe that LRU achieves poor reuse rates with respect to the optimal results of LFD. In fact, the average reuse rate for LRU is 30.06%, whereas the optimal one is 45.97%. However, these results greatly improve by using Local LFD. We can observe that in many cases the reuse rates are very close to the optimal ones if our replacement policy just knows the next task graph to be executed (*Local LFD (1)*). In addition, the more task graphs are stored in DL, the better Local LFD works. For instance, the average reuse rate for *Local LFD (4)* (45.93%) is very close to the optimal one (45.97%).

Secondly, Fig. 9b shows the reuse rates for Local LFD when it implements the *Skip Event* feature. For simplicity, this plot just illustrates what happens when the *Dynamic List* contains just the following task graph about the near future (*Local LFD (1)*). As the figure shows, the *Skip Event* feature has a positive impact on the reuse rates of the running applications. For instance, in average, *Local LFD (1) + Skip Events* reuses 48.19% of the tasks, whereas for LFD, this rate is 44.38%. Hence in this case, if Local LFD implements the *Skip Event* feature, it outperforms LFD, *which is the optimal one regarding task reuse*. And this happens when Local LFD *only* has the information about the following task graph (or course, the more task graphs are stored in the DL, the better are these reuse rates). These results may look strange, since we achieve "better results than the optimal". Clearly the reason is that we

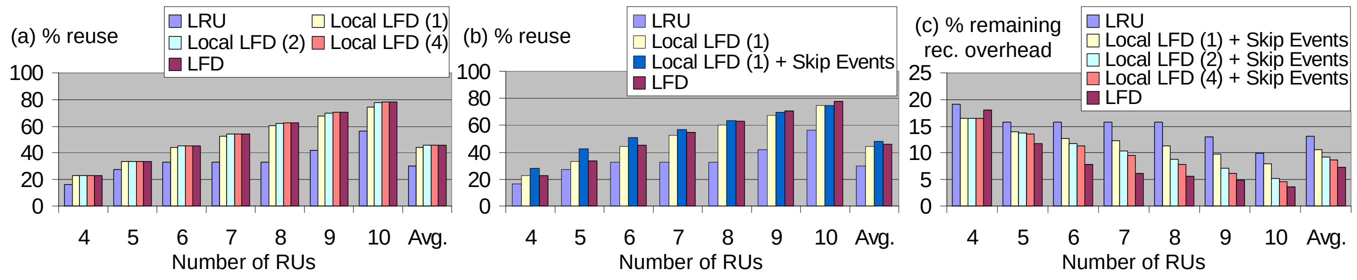


Figure 9. Performance evaluation of our Local LFD replacement technique in comparison with LRU and LFD, and for a number of RUs that range from 4 to 10. The "Avg." sections show the average results

are playing with different rules, because LFD cannot delay any reconfiguration, whereas our proposed replacement policy takes advantage of this optimization opportunity. These nice results involve other positive effects over the performance of the system. In fact, higher reuse rates reduce the system energy consumption, since a reconfiguration process consumes a large amount of energy [4]. In addition, higher reuse rates also reduce the pressure over the external memory and the system bus, since the reconfigurations involve moving large amounts of data from an external memory to the FPGA.

Finally, Fig 9c shows the remaining reconfiguration overhead that is still generated when the system applies different replacement policies: LRU, LFD, as well as *Local LFD (1)*, *Local LFD (2)* and *Local LFD (4)* when they implement the *Skip Event* feature. These results are shown as the percentages of the original reconfiguration overhead that remains after applying the different replacement policies. As the figure shows, in these experiments the remaining reconfiguration overhead is still very high even when applying a prefetch approach. For instance, for 4 RUs and applying LRU, this percentage is 19.19%. This important overhead can be reduced if we increase the number of RUs in the system and we apply a replacement technique that takes advantage of the reuse opportunities, as the replacement policy presented. LFD achieves the best *average* overhead reduction, since in this case only 7.22% of the original overhead is still present. Finally, the average results of Local LFD are between these two ones. In addition, in the latter case these overhead percentages decrease as more information about the near future is available for Local LFD, almost reaching the performance that LFD achieves. For instance, for *Local LFD (4) + Skip Events*, this average percentage is 8.9%, which is just 1.68% higher than the average one that LFD obtains. It is important to point out that these overhead reductions are applied over highly optimized solutions, since the execution manager already hides a significant amount of the reconfiguration overheads by applying prefetch.

Looking at Fig. 9c, one can observe that this trend is also repeated for all the displayed number of reconfigurable units, except for the experiment carried out with 4 RUs. In fact, this case is interesting since the *Skip Event* feature *also* reduces the remaining reconfiguration overhead with respect to LFD. The reason is the extremely high competition that is created in this experiment, since 15 different tasks compete for just 4 reconfigurable units (this also happens for 3 RUs, which is not shown in the figure for simplicity). However, as the number of RUs grows (and this competition decreases), LFD is powerful enough to outperform Local LFD. In any case, Local LFD achieves near-optimal overhead reductions, which is very interesting taking into account that this policy can be applied to dynamic systems, whereas LFD cannot.

B. *Run-time delays generated by the replacement module*

Table I shows the execution times of different replacement techniques: LRU, LFD and Local LFD with different numbers of task graphs stored in *DL*. We have measured them by running them in one of the Power PC processors embedded in a Virtex-II Pro XC2VP30. For that purpose, we have developed a SW simulator written in C and compiled for the Power PC architecture using the Xilinx™ EDK 9.1.02i development tool. In these experiments, the operating frequency of the processor is always 100 MHz. For LFD and Local LFD, this table evaluates the delays generated in a system with 4 RUs for the *worst-case scenario*. In other words, the selected replacement candidate never exists in the complete list of reconfigurations or the *Dynamic List*, respectively. Hence the replacement module always has to search in the whole list until it realizes that the given task is not going to be reused in the near future. In addition, this scenario also involves that the 4 RUs are replacement candidates (and hence this search has to be carried out 4 times).

As the table shows, LRU is the fastest replacement policy, since it barely generates a delay of 7.2 µs. On the contrary, the worst-case execution time of LFD is more than 11 ms, which is a very significant delay in the applications execution time (as it will be shown below). However, Local LFD is much faster than LFD, since it reduces its execution time by 2-3 orders of magnitude. In addition, the penalty generated by the Skip Event feature itself is negligible with respect to the Local LFD

TABLE I. RUN-TIME DELAYS GENERATED BY DIFFERENT REPLACEMENT POLICIES IN A SYSTEM WITH 4 RECONFIGURABLE UNITS

Replacement strategy	**Run-Time Worst-Case Execution Time (ms)**
LRU	0.00720
LFD	11.34983
Local LFD (1) + Skip Events	0.06028
Local LFD (2) + Skip Events	0.07412
Local LFD (4) + Skip Events	0.11020

TABLE II. IMPACT OF THE REPLACEMENT MODULE IN THE SYSTEM PERFORMANCE

Task Graph	**Initial Execution Time (ms)**	**Run-time Management in [9] (ms)**	**Replacement Module Performance**		
			Worst-case run-time performance		***Design-time performance***
			Execution time (ms)	***Overhead (%)***	***Execution time (ms)***
JPEG	79	0.87	0.08153	0.10	8.60
MPEG-1	37	1.02	0.08153	0.22	11.09
HOUGH	94	0.88	0.08153	0.09	14.48

technique when it applies a purely ASAP approach, since most of the additional computations are carried out at design time.

Finally, Table II compares the execution time of our run-time replacement module and its design-time phase. It also compares them with the execution times of the benchmarks, as well as with the execution times of the remaining task-management computations. Column 2 shows the initial execution time of the tested applications assuming that no additional overhead is generated. Column 3 shows the run-time overhead that the manager in [9] generates. Columns 4 and 6 refer to the execution time of the design-time and run-time replacement modules, respectively (these results average the performance of Local LFD with different numbers of task graphs stored in *DL*: 1, 2 and 4). Finally, Column 5 evaluates the impact of the run-time replacement module in the initial execution time of the applications. Thus, *Column 5 = Column 4 / Column 2 * 100*.

As the table shows, the run-time delays of the replacement module are 81.53 µs on average. However, they represent an overhead that ranges from 0.09% to 0.22% the initial execution times of the applications, which is almost negligible. In addition, this does not represent a significant additional overhead with respect to the delay introduced by the task-graph execution manager (Column 3), which is on average 11 times greater. On the contrary, the execution times of the design-time replacement module are from 1 to 3 orders of magnitude greater than the run-time ones. However these delays are totally acceptable for a design-time phase.

VII. CONCLUSIONS

In this paper we have presented a replacement technique to address the problem of configuration replacement for dynamically reconfigurable systems. We have integrated it in an external task-graph execution manager that carries out the reconfigurations and executions of the tasks following an event-based as soon as possible (ASAP) approach. However, in order to escape from locally optimal solutions, we have also included support to delay the reconfigurations in some cases, thereby making the replacements more flexible. In addition, the proposed approach is a hybrid design-time/run-time technique that carries out the bulk of its computations at design time in order to generate low run-time delays. Our results have shown that the proposed replacement technique achieves near-optimal reuse rates, which has a positive impact on the performance and the energy consumption of the system. In addition, we have experimentally tested in a Power PC embedded in a Virtex-II Pro FPGA that our replacement technique introduces a very low run-time delay that is negligible with respect to the execution time of the tested multimedia applications.

REFERENCES

[1] C. Chang, et al., “BEE2: A High-End Reconfigurable Computing System”, IEEE Design&Test of Computers, vol. 22, pp. 114-125, 2005.

[2] H. K.-H. So and R. W. Brodersen, “File system access from reconfigurable FPGA hardware processes in BORPH”, in Proc. of FPL 2008, pp. 567-570.

[3] *Virtex-5 FPGA User Guide*, Xilinx Corporation, May 2010.

[4] T. Becker, W. Luk and P. Cheung, “Energy-Aware Optimisation for Run-Time Reconfiguartion”, in Proc. of FCCM 2010, pp. 55-62.

[5] S. Hauck, “Configuration Prefetch for Single Context Reconfigurable Coprocessors”, in ACM/SIGDA Symposium on Field-Programmable Gate Arrays (FPGA), 1998, pp. 65-74.

[6] J. Jeong and M. Dubois, “Cache replacement algorithms with nonuniform miss costs”, IEEE Trans. on Computers, vol. 55, pp. 353-365, 2006.

[7] V. Nollet, T. Marescaux, P. Avasare, D. Verkest and J.-Y. Mignolet, “Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles”, in Proc. of DATE 2005, pp. 234-239.

[8] Y. Qu, J.-P. Soininen and J. Nurmi, “A parallel configuration model for reducing the run-time reconfiguration overhead,” in Proc. of DATE 2006, pp. 965-970.

[9] J. 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, 2010.

[10] L. A. Belady, “A study of replacement algorithms for virtual storage computers,” *IBM Syst. J.*, vol. 5, pp. 78–101, 1966.

[11] R. Cordone, F. Redaelli, M.A Redaelli, M. D. Santambrogio and D. Sciuto, “Partitioning and Scheduling of Task Graphs on Partially Reconfigurable FPGAs”, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 28, pp. 662-675, 2009.

[12] F. Redaelli, M. D. Santambrogio and S.O. Memik, “An ILP Formulation for the Task Graph Scheduling Problem Tailored to Bi-dimensional Reconfigurable Architectures”, in Proc. of ReConFig 2008, pp. 97-102.

[13] A. Popp, Y. Le Moullec and P. Koch, “Scheduling Temporal Partitions in a Multiprocessing Paradigm for Reconfigurable Architectures”, in Proc. of NASA/ESA Conference on Adaptative Hardware and Systems (AHS) 2009, pp. 230-235.

[14] H. Walder and M. Platzner, “Online scheduling for block-partitioned reconfigurable devices”, in Proc. of DATE 2003, pp. 290-295.

[15] S. Sudhir, S. Nath and S. Goldstein, “Configuration caching and swapping”, in Proc. of FPL 2001, pp. 27-29.

[16] Z. Li, “Configuration Cache Management Techniques for Reconfigurable Computing”, in Proc. of FCCM’00, pp. 22-36.

[17] J. Resano, D. Mozos, D. Verkest and F. Catthoor, “A Reconfiguration Manager for Dynamically Reconfigurable Hardware”, IEEE Design & Test of Computers, vol. 22, pp. 452-460, 2005.

[18] J. Resano et al., “A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs”, Microprocessors and Microsystems, vol. 28, pp. 291-301, 2004.

[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 Proc. of DATE 2005, pp. 106-111.