A Hardware Task-Graph Scheduler for Reconfigurable Multi-tasking Systems

Abstract

Reconfigurable hardware can be used to build a multi-tasking system where tasks are assigned to HW resources at run-time according to the requirements of the running applications. These tasks are frequently represented as direct acyclic graphs and their execution is typically controlled by an embedded processor that schedules the graph execution. In order to improve the efficiency of the system, the scheduler can apply prefetch and reuse techniques that can greatly reduce the reconfiguration latencies. For an embedded processor all these computations represent a heavy computational load that can significantly reduce the system performance. To overcome this problem we have implemented a HW scheduler using reconfigurable resources. In addition we have implemented both prefetch and replacement techniques that obtain as good results as previous complex SW approaches, while demanding just a few clock cycles to carry out the computations. We consider that the HW cost of the system (in our experiments 3% of a Virtex-II PRO xc2vp30 FPGA) is affordable taking into account the great efficiency of the techniques applied to hide the reconfiguration latency and the negligible run-time penalty introduced by the scheduler computations.

1. Introduction

Partial reconfiguration [1] opens the possibility of developing a hardware multitasking system by dividing the entire reconfigurable area into smaller Reconfigurable Units (RUs) wrapped with a fixed interface [2]. These RUs can be reconfigured to execute tasks of the running applications; hence, with the proper configurations, the same platform can be customized at run-time to implement several application specific systems, or to adapt itself to the variable requirements of a complex dynamic application. In addition, configurations can be updated at run-time in order to extend the functionality of the system, to improve the performance, or to fix detected bugs. Nowadays, commercial FPGAs, as XILINX Virtex™ series [3] or Altera® Stratix [4] can be used to implement a HW multi-tasking system where several RUs collaborate with one or several embedded processors. Moreover, some vendors have developed special design environments that provide interesting support as the XILINX Embedded Development Kit (EDK) [5]. However, previous works [6], [7] have demonstrated that the efficiency of a HW multi-tasking system based on FPGAs can be drastically reduced due to the reconfiguration latency unless some optimization scheduling techniques are applied at run-time.

In embedded systems, tasks are frequently represented as Direct Acyclic Graphs (DAGs). Normally an embedded processor has to manage and schedule their execution. This processor must also apply the optimization techniques that, in order to deal with dynamic events, must be at least partially carried out at run-time. This involves dealing with complex data structures and, for a HW multi-tasking system, frequent HW/SW communications, which can introduce important penalties in the system execution.

Since we are targeting reconfigurable systems, we propose to use part of the reconfigurable resources to implement a HW scheduler that will efficiently carry out the scheduling computations, including the optimizations techniques. Moreover this scheduler will directly communicate with the RUs reducing the need for costly HW/SW communications.

The rest of the paper is organized as follows: next section illustrates the problem with a motivational example. Section 3 reviews the related work; Section 4 describes the structure of our scheduler and its HW implementation. Section 5 presents the experimental results and finally Section 6 explains our conclusions and future work to be done.

2. Problem overview and example

In this work we target a system where some dynamic events trigger the execution of task graphs at run-time, some of which are assigned to RUs. Our scheduler will receive the information of the graph and it will schedule and guarantee its proper execution taking into account its internal dependencies. Tasks are represented as DAGs where each node represents a task, and each edge a precedence constraint. We assume that all the task graphs are available at design-time, and we can extract some information from them in order to reduce the run-time computations. For each node we know its execution time, and using that information and analyzing the task-graph structure we assign a weight to each node. We assume that a task graph will not start execution until the previous one has finished. Once our scheduler receives the information of the task-graph, it will schedule the reconfigurations and executions needed, guaranteeing that the precedence constraints are

met, and applying a prefetch approach to carry out the reconfigurations in advance and a replacement technique to improve the task reuse. This technique will also improve the results obtained by the prefetch approach, as it is explained in the next example.

Figure 1 depicts the execution sequence for a system that executes twice the two graphs in the figure using three different replacement policies. In the LRU column (Least Recently Used), the system cannot reuse any configuration since there are 6 tasks competing for 4 RUs. However, the reconfiguration latency of 4 of the 6 tasks is hidden applying a prefetch approach. When the system applies the LFD (Longest Forward Distance) policy, which is the optimal one regarding reuse as was proved in [8], it will be able to reuse two of the six tasks in the second iteration. Moreover, the two tasks reused are the same that were introducing delays due to the reconfiguration latency; hence in the second iteration these delays disappear. Unfortunately, to apply LFD the system must have complete knowledge of the future events.

Our objective is to achieve comparable results to LFD, even when we do not have this information. To this end we have developed a replacement policy that before replacing a task takes into account two factors: *Is the task waiting for execution? Is it especially critical for the graph execution?* We have called it LF+C (Look Forward+Critical). This policy classifies the possible candidates (those that can be replaced) in three categories: *perfect candidates* (*PC*); *critical candidates* (*CC*), that have an important impact in the graph execution but the system does not know if they are going to be executed again in the future; and *reusable candidates* (*RC*) that the system knows that are currently waiting to be executed.

LF+C will always try to replace a *PC*. If there are not *PCs*, it will replace a critical one, and only if this is not possible, it will select a reusable one. We assume that our scheduler only has information of the graph that is in execution, hence those tasks that are not critical and belong to other graphs, or that have already been executed are *PCs*. In Figure 1 the first task of each graph is considered critical since our prefetch scheduler cannot hide their loading latency. Hence it is important to reuse them. These critical tasks can be identified at design-time. They will never be *PCs* but only *CCs* (If the system does not know if they are going to be executed again) or *RCs* (if the system knows that they are waiting for execution). Hence the scheduler will not replace them as long as there are *PCs* available. Assigning the maximum priority to the *RCs* is optimal for reuse, since if they are not replaced they will be reused soon.

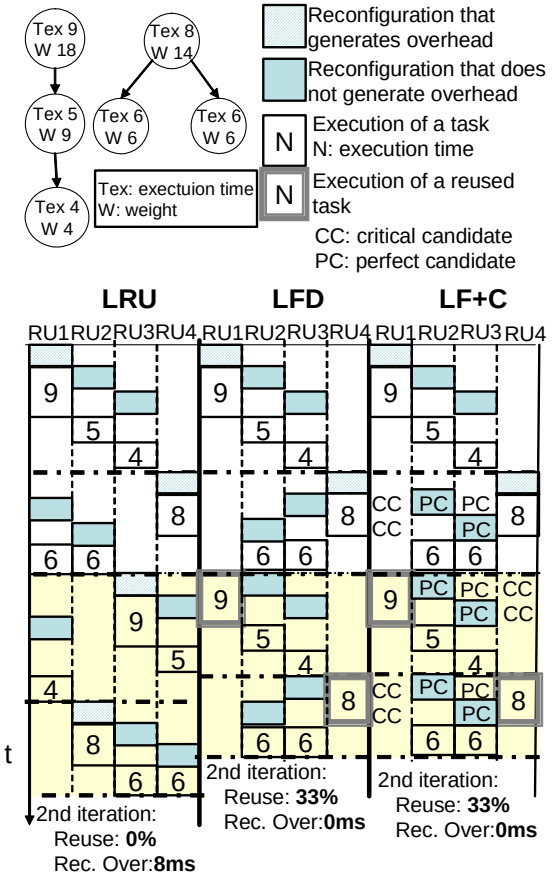

Figure 1. Motivational example for a system with 4 RUs and 4ms reconfiguration latency

3. Related work

Recently many research groups have proposed to build HW multi-tasking systems using partial reconfigurable resources. Some approaches that present HW multi-tasking systems are [9], [10], [11] and [12]. We believe that our scheduler is compatible with the HW multi-tasking systems presented in [9] and [10], but it is not with [11] and [12] because [11] only targets data-parallelism, and [12] proposes to use a general purpose multi-threaded OS, with some specific scheduling optimizations, to manage the system.

In all these HW multi-tasking systems, the RUs are tightly coupled to a processor that steers the system execution. This processor must monitor the HW execution and carry out all the Operating System (OS) and task scheduling computations. In addition, frequently this processor must also execute other tasks. Hence, if the processor is too busy managing the execution of the RUs, it will introduce important delays not only in the execution of its own tasks, but also in the whole system. One way to alleviate this problem is distributing the OS and task scheduling computations

among the processing elements. This approach both reduces the computational load of the processor and the costly HW/SW communications. Some interesting examples for HW multi-tasking systems are [9] and [10]. In [9] the authors propose a distributed OS support for inter-task communications. In [10] the authors propose a HW-based dynamic scheduler for reconfigurable architectures that applies a list-scheduling heuristic. However, they did not implement their design, but they only included it in their specific simulation environment.

In a FPGA-based HW multi-tasking system, optimizing the reconfiguration process is a key issue. Hence, several authors have proposed to include support to reduce the reconfiguration overhead. In [6], [7], [9], [10] and [13] the authors propose scheduling techniques that attempt to hide the reconfiguration latency. [9] and [13] target only static applications whereas the remaining techniques are applied at run-time. These techniques assume that an embedded processor will carry out the computations. However, none of these approaches have implemented their techniques in an embedded processor in order to measure their real impact on the system performance.

In our scheduler we have included a prefetch optimization very similar to the one presented in [6] but adapted to a HW implementation. This technique generates a negligible run-time overhead that, in our experiments, hides most of the reconfiguration latency. In addition our scheduler applies a replacement technique, specially designed for an efficient HW implementation which not only improves the percentage of reused tasks when task graphs are executed recurrently, but also improves the results of our prefetch technique.

4. System overview

Figure 2 presents the main HW blocks of our task scheduler. It stores the information of the graph in the associative table and the reconfiguration FIFO. In addition it includes two registers per RU, in order to store the current loaded task and its state. The RUs communicate with the scheduler generating events that are stored in the Event queue. The control unit processes these events and triggers the proper actions.

Finally, before starting a reconfiguration, the replacement module selects its destination. For each task of a task graph the scheduler receives its id, the number of predecessors and of successors and the id of these. This is stored in the associative table as can be seen in Figure 3. In addition it receives a sequence of all the tasks sorted according to their weights. This is stored in the Reconfiguration FIFO.

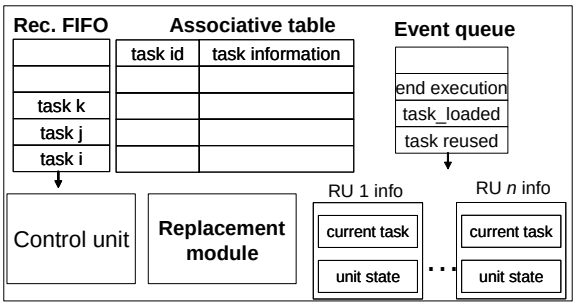

Figure 2. HW blocks of the task scheduler

4.1 Associative table

This table monitors the task dependencies of the tasks waiting for execution. It supports three different operations*: insertion, deletion and update*, and *check*.

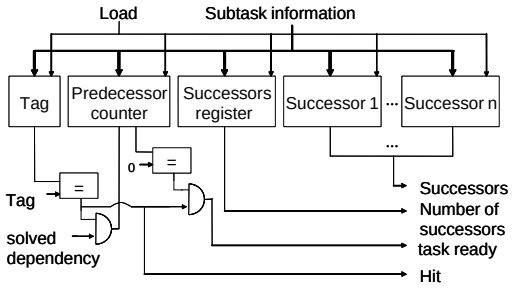

Figure 3. An entry of the associative table

The *Insertion* operation writes the information of a task in the table. Since this is an associative table, the actual situation where the data is written is not relevant. We have implemented it by dividing the table in sub-tables with 8 entries where each sub-table includes a register pointing to the first available entry. In this case the *Insertion* operation will initially look for a free entry in the first sub-table. If that table is full in the next cycle it will look for a free entry in the following sub-table. Hence, the maximum operation latency is equal to the number of subtables and its main virtue is that its size does not affect the clock period.

When a task finishes its execution it is removed from the table, and all the dependencies are updated. This is carried out with a *Deletion and update* operation, which uses the HW support depicted in Figure 4. This operation reads the entry of the task, storing the tags of the successors in the successor register and the number of successors in the control counter, and sets the corresponding entry free. Afterwards, it sequentially updates the entries of the successors. In each cycle one of the successors is selected, the input of the associative table *solved dependency* is activated; and the control counter is decremented.

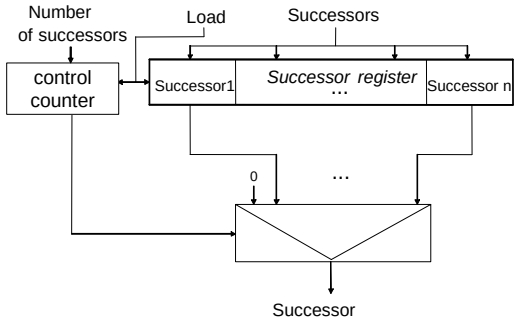

Figure 4. HW support for the Deletion and update operation

When the control counter reaches zero, the operation ends. When the table detects that the signal solved dependency is active, it decrements the predecessor counter of the corresponding task. This operation has a O(N) complexity, where N is the number of successors.

Finally the *check* operation enquires about whether a given task is ready to start its execution, i.e. all its dependencies are solved. This is done in just one clock cycle, introducing the task tag as input to the table and reading the output *task ready* in the following cycle. A task is ready if its predecessor counter is zero.

4.2 Prefetch approach

In order to take good prefetch decisions without carrying out complex run-time computations, our prefetch technique is based on weights assigned to each task at design-time. These weights represent the longest path (in terms of execution time) from the beginning of the execution of the task to the end of the execution of the whole graph (Figure 1 includes an example of task weights). We use this weight to sort all the tasks in the graph and generate the reconfiguration sequence. Since this phase is carried out at design-time, it does not generate any run-time penalty.

At design-time we also identify those nodes that will generate a reconfiguration overhead unless they are reused (critical nodes), following the process presented in [14]. This process starts assuming that no task is reused, and the tasks that generate any delay due to their reconfiguration are identified. The one with the greatest weight is tagged as critical. Then the process continues iteratively assuming that the critical tasks are reused. When no task generates any delay the process finishes.

At run-time our scheduler will follow the reconfiguration sequence of each graph guaranteeing each reconfiguration starts as soon as possible, if there is a RU available and the reconfiguration circuitry is free.

4.3 Control unit

The control unit processes sequentially the run-time events that are stored in the event queue. It supports four different events:

1. *new graph*: the system generates this event when it sends the information of a new graph.
2. *end of execution*: a RU generates this event when a task finishes its execution.
3. *end of reconfiguration*: a RU generates this event when a reconfiguration process finishes.
4. *reused task*: this event is generated when a task is reused. This happens when the following task in the Reconfiguration FIFO is already loaded in a RU.

The *end of reconfiguration* and the *reused task* events are similar from the control unit point of view, since reusing a task is the same as carrying out a reconfiguration in one clock cycle. Figure 5 depicts the actions triggered by each event.

```
CASE event IS:
 end_of_execution:
     update_task_dependencies
     look_for_reconfiguration()
     FOR i = 0 to number_of_units
        IF(RU_state=idle)AND(task_ready)
                  start_execution()
  end_of_reconfiguration or reused task:
     IF task_ready
         start execution()
         look_for_reconfiguration()
  new_graph:
     update_ task_dependencies
     update_schedules
     look for reconfiguration()
```

Figure 5. Pseudo-code of the control unit

When an *end of execution* event is processed, the scheduler updates the dependencies. Then, if the reconfiguration circuitry is idle, it checks if it is possible to start a reconfiguration. Finally, the scheduler checks if any of the tasks that are currently loaded can start its execution. For the *end of reconfiguration* and *reused task* events, the scheduler checks if the task that has been loaded can start its execution. It also tries to start another reconfiguration. Finally, for the *new_graph* event, the scheduler updates the reconfiguration FIFO and the associative table with the information received. After that, if the reconfiguration circuitry is idle, it will check if it is possible to start loading one of the new tasks.

4.4 Replacement module

We have developed a replacement heuristic that classifies the possible candidates in three categories: perfect candidates (*PC*), critical candidates (*CC*), and reusable candidates (*RC*). This replacement policy demands a very affordable HW support as can be seen in Figure 6. It uses a counter and 3 multiplexers to look for candidates in the RUs. For each RU it checks if the task loaded is waiting for execution using the associative table (if *hit*=1) , if it is critical (if C_i=1) and if the RU can be used (if $free_i$=1). With this information three gates identify the candidates. If a *PC* is found it is

directly selected. Otherwise, if a *CC* or a *RC* is found it is stored in a register (not shown in the figure for simplicity). If all the RUs have been analyzed without finding any *PC*, the module will select the *CC* stored in the register. If there is not any *CC* it will select the *RC*. If there are no candidates the reconfiguration is not carried out.

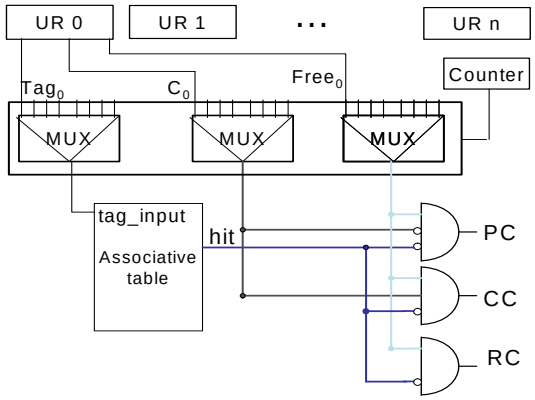

Figure 6. HW that selects the candidates

5. Experimental results

Since we do not have an actual HW multi-tasking system available, we have developed a HW simulation platform that uses two programmable counters to simulate the reconfiguration latency and the execution time of a task in a RU. Figure 7 presents the system overview that is controlled by a Power PC embedded on a VIRTEX-II PRO xc2vp30 FPGA. The scheduler and the RUs have been included as on-chip peripherals connected with the processor via a bus. The scheduler interacts with the RUs using point to point connections, and with the Power PC using the interruption line. We have implemented the system using the Xilinx EDK 9.1i environment because it provides support for easy peripheral development and HW/SW integration.

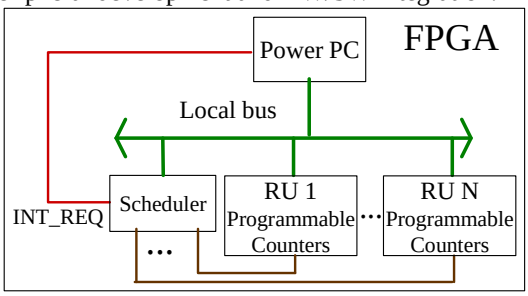

Figure 7. System overview

When a task is assigned to a RU, the scheduler loads its execution time and its reconfiguration latency in the two counters. When the first counter finishes, it generates an *end_of_reconfiguration* event, and when the second one finishes it generates an *end_of_execution event.* The counters only count when the scheduler identifies that the reconfiguration or the execution can start. The system also includes an additional counter that is used to measure the total execution time.

We have tested our scheduler using two task graphs extracted from two actual multimedia applications: A JPEG decoder with four tasks and a MPEG encoder with five tasks. To evaluate the results we have also implemented four additional schedulers that apply different replacement policies. These policies are the well-known *First-Free* (FF), *Least Recently Used* (LRU), an optimization of LRU (that assigns more priority to the task waiting for execution (LRU+LF)); and LFD, mentioned in section 2. We have carried out three different experiments assuming that these tasks are executed recurrently. In the first one (a) we firstly execute the JPEG task and afterwards the MPEG task. To evaluate the reuse we have done this twice, and we have only measured the second iteration since it was impossible to reuse anything during the first time. In the second experiment (b) we assume that JPEG is executed twice, and then MPEG is executed once, and again we repeat this pattern two times consecutively. In the third experiment (c) JPEG is executed once and MPEG twice. Figure 8 presents the results for these experiments. On the left it includes the percentage of tasks reused, and on the right, the normalized reconfiguration overhead. In this case, 1 means that the reconfiguration overhead is not reduced due to the replacement policy. The initial reconfiguration overhead without applying the prefetch and replacement optimizations was almost 25% of the execution time. Just applying our prefetch approach it was reduced to 8%. And, as it is depicted in the figures, when the task graphs are executed recurrently our replacement policy can totally eliminate it in most cases.

The experimental results show that an efficient replacement policy is very important. Thus, a simple FF approach does not take advantage of the reuse possibilities. An LRU approach is also not efficient when there are more tasks than RUs. If the LRU is optimized to look-forward and identifies which tasks are going to be executed soon, the percentage of reuse is greatly improved, but the reconfiguration overhead does not always decrease. This happens because the number of critical tasks reused is the same, and those are the ones that generate the overhead. The LFD approach achieves the maximum reuse. However, it cannot eliminate all the overhead because sometimes replaces some critical tasks. Our replacement policy reuses slightly less tasks than the LFD but it reuses more critical tasks since they have more priority than non-critical ones. Hence for 6 RUs or more it eliminates all the execution time overhead due to the reconfigurations.

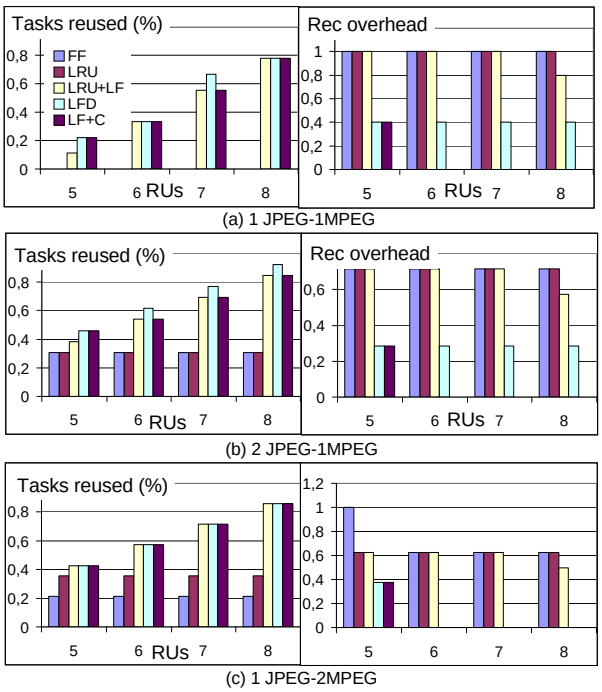

Figure 8. Tasks reuse and reconfiguration overhead when executing the MPEG and JPEG tasks graphs with different replacement policies

Our HW generates a negligible run-time penalty due to its computations (just a few clock cycles). The only thing that generates a significant overhead is the new-graph event, since the processor must send the task-graph information. Using a DMA this is done in 0.002 ms, otherwise it consumes 0.014 ms. As a final experiment we have implemented an equivalent SW version of our scheduler and we have repeated the experiments and measured the delays introduced due to the scheduler computations. In this case the delay generated due to the scheduler computations is around 1 ms, and there is an additional 0.5 ms delay due to the communications among the RUs and the processor (in this case the RUs use interrupts to generate events). Hence the HW version with a DMA is three orders of magnitude faster than the SW version. Regarding the HW cost of our scheduler, in these experiments it uses 3% of the FPGA slices, and 1% of the BlockRAMs. The number of slices needed grows linearly with the size of the graphs supported (in this version it just supports graphs with 8 tasks or less). The scheduler operates at 100 MHz even when dealing with larger graphs.

6. Conclusions and Future work

We have developed a HW implementation of a task-graph scheduler for HW multi-tasking systems that applies two optimization techniques (a prefetch approach and a replacement policy) to reduce the impact of the reconfiguration latency. The results demonstrate that this scheduler can drastically reduce the reconfiguration overhead when dealing with tasks that are executed recurrently. These techniques have been designed to demand a very affordable amount of HW resources while dealing with task graphs at run-time in just a few clock cycles. As a result this scheduler is three orders of magnitude faster than an equivalent SW scheduler. As future work we want to extend the scheduler to deal with several tasks in parallel and also to support the execution of tasks graphs with internal control dependencies.

7. References

[1] P. Lysaght et al.," Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs", FPL'06, p. 1-6, 2006.

[2] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", FPL,pp. 795-805, 2002.

[3] www.xilinx.com/products/silicon_solutions. May, 2008.

[4] www.altera.com/products/devices/stratix-fpgas/about/stx-about.html. May, 2008.

[5] www.xilinx.com/ise/embedded/edk_docs.htm., 2008.

[6] J. Resano, D. Mozos, D Verkest, F Catthoor, "A Reconfiguration Manager for Dynamically Reconfigurable Hardware", IEEE Design&Test, Vol. 22, Issue 5, pp. 452-460. 2005.

[7] Z. Li, S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation", FPGA'02, pp. 187-195. 2002.

[8] L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," IBM Systems J., vol. 5, no. 2, 1966, pp. 78-101.

[9] Y. Qu, J. Soininen, J..Nurmi, "A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead", DATE'06, pp. 965- 969. 2006.

[10] J. Noguera, M. Badia., "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling". ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, p. 385-406. 2004

[11] K. N. Vikram, V. Vasudevan. "Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures". IEEE Trans. on VLSI Systems, Volume 14, Issue 9, Sept. 2006, p.1010-1023.

[12] W. Fu, K. Compton. An execution environment for reconfigurable computing. FCCM'05, p.149-158. 2005.

[13] L. Shang, N.K. Jha, "Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs", ASP-DAC'02, pp. 345-360, 2002.

[14] J. Resano, D. Mozos and F. Catthoor, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-time the Reconfiguration Overhead of Dynamically Reconfigurable HW". Proc of the DATE'05. p. 106-111. 2005