HW implementation of an execution manager for reconfigurable systems

Javier Resano, Juan Antonio Clemente, Carlos Gonzalez, Jose Luis Garcia and Daniel Mozos
Computer Architecture Department
Universidad Complutense
Madrid, Spain

Abstract *- Reconfigurable HW can be used to build a hardware multitasking system where tasks can be assigned to the reconfigurable HW at run-time according to the requirements of the running applications. Normally the execution in this kind of systems is controlled by an embedded processor. In these systems tasks are frequently represented as subtask graphs, where a subtask is the basic scheduling unit that can be assigned to a reconfigurable HW. In order to control the execution of these tasks, the processor must manage at run-time complex data structures, like graphs or linked list, which may generate significant execution-time penalties. In addition, HW/SW communications are frequently a system bottleneck. Hence, it is very interesting to find a way to reduce the run-time SW computations and the HW/SW communications. To this end we have developed a HW execution manager that controls the execution of subtask graphs over a set of reconfigurable units. This manager receives as input a subtask graph coupled to a subtask schedule, and guarantees its proper execution. In addition it includes support to reduce the execution-time overhead due to reconfigurations. With this HW support the execution of task graphs can be managed efficiently generating only very small run-time penalties.*

Keywords: *hardware multi-tasking, run-time reconfiguration, task-scheduling, operating system support.*

1 Introduction and related work

Partial reconfiguration opens very interesting possibilities for reconfigurable systems. For instance, recently many research groups have proposed to build hardware multitasking systems using partial reconfigurable resources. The first implementation of such a system in a commercial FPGA was presented by Marescaux et al. in [1]. In this work the FPGA is partitioned into an array of identical Reconfigurable Units (RUs). These RUs are interconnected using an interconnection network implemented on the FPGA fabric. At run-time, tasks are assigned to these RUs using partial dynamic reconfiguration. Communication among the tasks is achieved by sending messages over the ICN using a fixed network interface implemented inside each tile. Other recent approaches for hardware multitasking systems are [2], [3], and [4].

In these systems normally the RUs are tightly coupled to a processor that steers the system execution [1], [3], [5]. Hence 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 some of the tasks assigned to the system [6]. Hence, if the processor is too busy managing 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 adding some HW support in order to distribute some of the OS and task scheduling computations. This approach has two main advantages. Firstly, it reduces the computational load of the processor, and secondly, it also reduces the amount of HW/SW communications, which frequently become a system bottleneck [7]. In addition, according to [8], the OS consumes a large amount of the processor power consumption, and a HW implementation will normally be more power efficient.

In System-on-a-chip environments many previous works have already proposed to distribute some of the functionalities that traditionally are carried out by a centralized OS. For instance, in [9] the authors present a hardware microarchitecture for multiprocessor synchronization. Some interesting examples for multi-processor reconfigurable system are [10] and [11]. In [10] the authors propose a distributed OS support for inter-task communications. To this end, each reconfigurable unit includes a routing table that is updated each time that a new subtask is loaded in one of the RUs. In [11] the authors propose a hardware-based dynamic scheduler for reconfigurable architectures that applies a list scheduling heuristic. However, the authors did not implement their design, but only include it in their specific simulation environment.

The rest of the paper is organized as follows. First, we will outline the contributions of this work, and present an

overview of our manager. Then, we will describe the benefits of our approach with a motivational example. After that, we will describe the main components of our system in detail, and an example of how the manager will control the execution of a given subtask graph. Finally we will present some experimental results and remark some conclusions.

2 Contributions and overview

The main contribution of this paper is to describe a HW implementation of an execution manager for a multitasking reconfigurable system. This is, to our best knowledge the first actual implementation of such a system. This manager controls the execution of a set of subtasks over the RUs following a given subtask schedule. This schedule and some other needed information are stored in an associative table and a set of FIFOs.

In addition, since the reconfigurable latency may generate significant delays in the execution (for FPGAs the delays are in the order of milliseconds [5], [12]), the manager applies two techniques to reduce this overhead. First, before reconfiguring a RU to load a subtask the manager checks if that subtasks was loaded previously. In that case, the configuration can be directly reused and no reconfiguration is carried out. Second, the manager applies a configuration prefetch technique [13] to hide the reconfiguration latency when possible. Two different prefetch modes are supported, namely greedy approach, and scheduler-based approach.

In the greedy approach each time that the reconfiguration circuitry is idle, the manager looks for a subtask that are ready for reconfiguration. A subtask is ready if the previous subtasks assigned to the same resource have already finished its execution. If there is more than one subtask ready, the manager selects one randomly.

Greedy heuristics are fast and easy to implement, and often provide reasonable good results. However, scheduling the reconfigurations is a critical issue for a multitasking HW system, and some authors have proposed some specific run-time and design-time scheduling heuristics that achieve near optimal results [14]. To provide support for schedulers that find near-optimal reconfiguration schedules, the manager can be programmed to follow a given sequence of reconfigurations (scheduler-based approach). In this case, the manager identifies which is the following subtask that must be loaded according to the given reconfiguration schedule (that is stored in a FIFO), and starts its reconfiguration as soon as possible.

Figure 1 depicts the system organization of our execution manager. The main components are:

- Table of subtask dependencies: when a new graph must be executed the dependencies of each subtask are stored in this associative table. Each time that a subtask finished its execution, all the successors are actualized, and the subtask is removed from the table. Before starting executing a subtask, the control unit checks in this table if all the dependencies are solved. This operation is carried out in just one clock cycle.

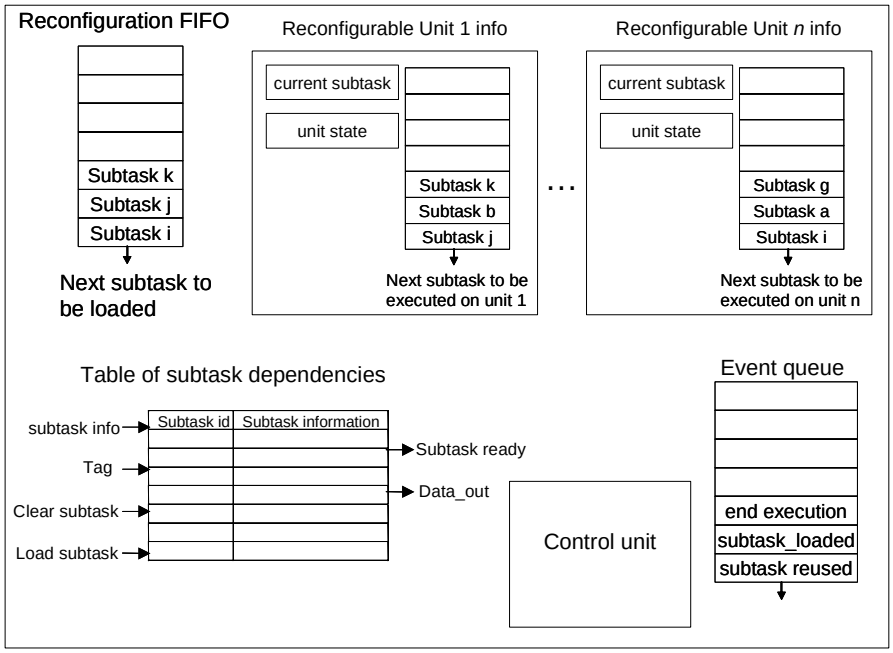

Figure 1. Overview of the execution manager

 Unit's info: the manager associates a FIFO and two registers to each reconfigurable unit. The FIFO stores the subtasks assigned to this unit following the given schedule. The registers store the state of the unit and the current loaded subtasks.

 Reconfiguration FIFO: This FIFO stores the sequence of subtask that must be reconfigured. This FIFO is only used if the scheduler-based prefetch approach is active.

 Event queue: stores the run-time events like the end of the execution of a subtask or the end of a reconfiguration.

 Control unit: this unit read the events in order to identify if any subtask can start its execution or if it is possible to prefetch a configuration.

3 Motivational Example

In HW/SW co-design tasks are frequently represented as subtask graphs, where a subtask is the basic scheduling unit representing a kernel with significant computational load (one or several important loops) that can be assigned to a reconfigurable unit, and the edges of the graph represent inter-subtask dependencies. In order to control the execution of these tasks, the processor must manage at run-time complex data structures, with great computational load. For instance, Figure 2 depicts a representation of a subtask graph using a linked list. In this case the main list includes all the subtasks of the graph, and each node contains another list with all its predecessors. If the scheduler needs to check if a given task is ready for execution, it should first look for the list node that correspond to the subtask, and then read the number of predecessors. After this, it must read the entire predecessor list, and look for the state of all of them to check if they have already finished their execution. The complexity of this operation is $\Theta(N2)$, where N is the number of subtasks in the graph.

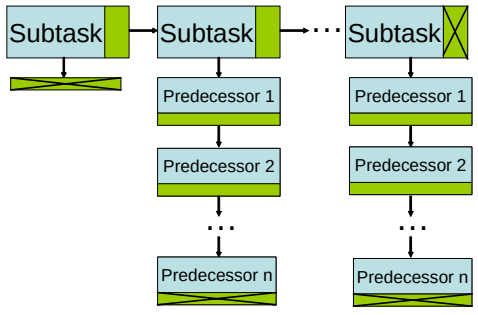

Figure 2. Subtask graph represented using a linked list.

Of course, there are many ways to improve the efficiency of this operation, like using indexed tables to allow direct access to each node of the graph instead of following the pointers, or updating the predecessor lists each time that a subtask finished its execution. However, in any case complex data management will be needed.

In addition, since the OS and the scheduler must interact with the RUs, frequent HW/SW communications will be demanded. This communication will probably involve interrupt handling, and they may need to take control of a shared bus, which frequently is a system bottleneck. Hence, they may also generate significant execution delays.

Our manager introduces two main benefits for this problem. First, the execution of a subtask graph in the RUs will be directly controlled by our manager, and operations like updating subtask dependencies, or checking if a subtask is ready, will be carried out very fast thanks to the associative table: only one clock cycle to update each dependency and one clock cycle to check if a subtask is ready. Second, our manager only needs to communicate with the processor twice during the execution of a subtask graph, one time at the beginning of the execution, in order to receive all the information, and another time at the end, to inform the processor that the execution was carried out properly.

4 Implementation details

In this section we will describe in detail the structure of the components of the execution manager and the operations that they support.

Table of subtask dependencies

This is an associative table that can be customized for different sizes and maximum number of successor per subtask. This table is used to monitor the dependencies among the subtasks of a graph. It supports three different operations: i*nsertion, deletion and update*, and *check*. In Figure 3 the structure of each entry is depicted. It includes a tag that identifies the subtask, a counter that indicates the number of unsolved dependencies (predecessor counter), and the information of the successors: number of successors and their tags.

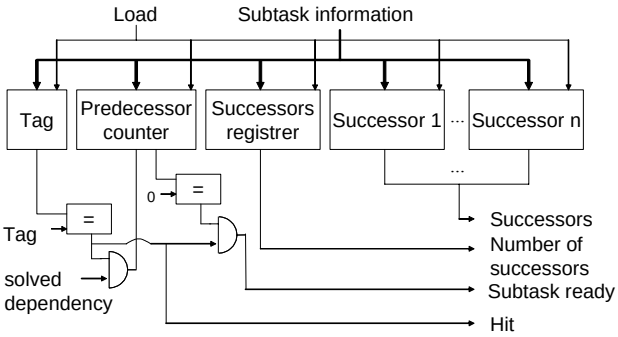

Figure 3. An entry of the table of subtask dependencies

Insertion operation writes the information of a subtask in the table. Since this is an associative table, the actual situation where the data is written is not relevant. To select the destination of the new entry we have developed a specific HW that is always pointing to the first free table entry, and generates an error if the table is full. With this approach the operation can be carried out in just one clock cycle.

When a subtask 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 sets the corresponding entry free, so it can be used in the future for other subtasks, and reads the entry of the subtask, storing the tags of the successors in the *successor register* and the number of successor in *the control counter*. Afterwards, it sequentially updates the entries of the successors. This is done in one clock cycle for each successor. Each cycle one of the successors is selected using the *control counter* and the multiplexer, the input of the associative table *solved dependency* is activated, and the *control counter* is decremented. When the control counter reach zero the operation finishes. When the table detects that the signal *solved dependency* is active, it decrements the predecessor counter of the corresponding subtask. This operation has a $\Theta(N)$ complexity, where N is the number of successors to update.

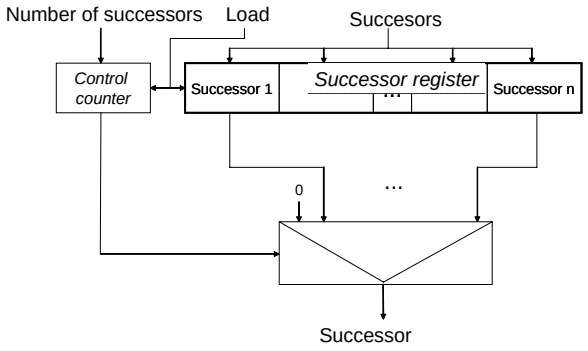

Figure 4. HW support for the *Deletion and update* operation

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

Unit's info

In our manager each RU includes a FIFO and two registers. The FIFO stores in order, following the given schedule, all the subtasks that have been assigned to this unit and are waiting for execution. The registers store the state of the unit and the current loaded subtask. In addition, there is a small controller that works as interface between the RU and the manager, generating events when a reconfiguration or an execution finishes, and updating the FIFO and the registers when a new subtask is assigned to the unit or a subtask is loaded in the RU.

Reconfiguration FIFO

This module is only used if the *scheduler-based prefetch approach* is active. In this case the reconfigurations are carried out following the order stored in this FIFO.
The data in this FIFO is loaded when the information of a given subtask graph is received, and each time that a reconfiguration is carried out the corresponding subtask is removed from the FIFO.

Event queue

This queue stores all the run-time events generated in the system until the control unit processes them. For each event, the queue stores the event code, and the tag of the subtask involved. The system identifies four different types of events:

1. *end of execution*: this event is generated by a RU controller each time that a subtask finishes.
2. *new graph:* this event is generated when the information of a new graph has been received in the input buffer.
3. *end of reconfiguration:* this event is generated when a reconfiguration finishes.
4. *reused subtask:* this event is generated when a subtask is reused. This happen when a RU controller identifies that it must start loading a subtask that it is already loaded.

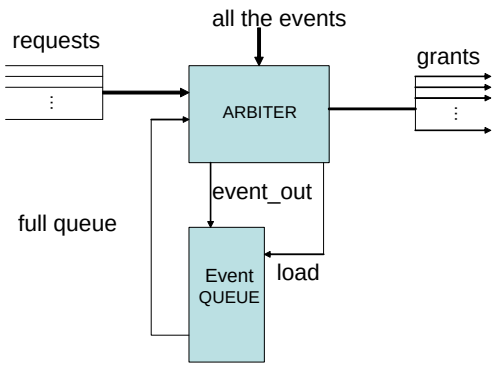

Figure 5. Organization of the event queue.

The *end of reconfiguration*, and the *reused subtask* events are similar from the system's point of view, since reusing a

subtask is the same that carrying out a reconfiguration in one clock cycle.
Since all the RU controllers may generate events, it is possible that two of them attempt to write an event in the queue at the same time. The manager prevents this by including an arbiter (Figure 5) that only allows one write per cycle by activating the appropriate grant line. In addition, this arbiter blocks the queue when it is full. The arbiter takes its decision following a fixed priority scheme.

Control unit

The control unit extract the events from the queue and carries out the proper actions. The pseudo-code of this unit is depicted in Figure 6.

When an *end of execution* event is processed the control unit updates the dependencies stored in the associative table. Then, if the reconfiguration circuitry is idle, it tries to start a reconfiguration. If the system is using the *greedy prefetch* approach it will start reading the states of the RUs and if it finds one that is idle and its subtask FIFO is not empty, it will start the corresponding reconfiguration process. If the system is using the *scheduler-based approach,* it will read the reconfiguration FIFO, and will check if it is possible to start that reconfiguration. Finally, the control unit will check if any of the subtasks that are currently loaded can start its execution.

For the *end of reconfiguration and reused subtask* events the control unit will check if the subtask that has been loaded can starts its execution. In addition, it will also try to start another reconfiguration.

```
Control unit :
    CASE event IS:
      end_of_execution:
          update_table
          look_for_reconfiguration()
          FOR i = 1 to number_of_RUs
             IF (RU_state=idle)AND(subtask_ready)
                    start_execution()
      end_of_reconfiguration:
          IF subtask_ready
              start execution()
          look_for_reconfiguration()
      new_graph:
          update_table
          update_RU_FIFOs
          if prefetch_policy=scheduler_based
               update_Reconfiguration_FIFO
           look_for_reconfiguration()
```

Figure 6. Pseudo-code of the control unit.

Finally, for the *new_graph* event, the control unit will read the data from an input buffer and store it in the associative table and the respective FIFOs. After that, if the reconfiguration circuitry is idle, it will check if it is possible to start loading one of the new subtasks.

5 Example of task execution

In this section we will describe step by step the execution of the subtask graph presented in Figure 7. This figure also depicts the final execution of the subtasks

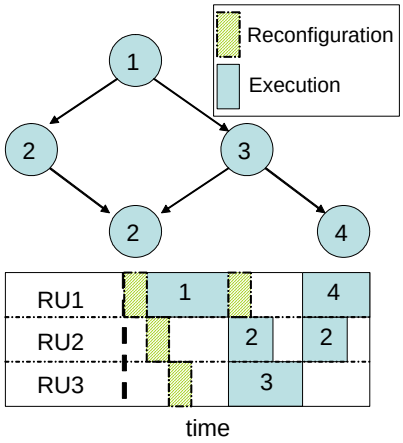

Figure 7. Example of the execution of a subtask graph

Initially the processor will send the information about the graph and the selected schedule to our manager. When this information is properly stored in an input buffer the processor will generate a *new graph* event. As soon as the control unit process this event, all this information will be stored in the associative table and the FIFOs. In addition the control unit will try to start a reconfiguration. If the system is using the *scheduler-based* prefetch approach and the reconfiguration sequence selected by the scheduler is 1-2-3-4-2, the control unit will start the reconfiguration of subtask 1. When subtask 1 finishes its reconfiguration, RU 1 will generate an *end of reconfiguration* event. Now, the control unit will check if subtask 1 can start its execution. In this case it has no unsolved dependencies, hence it will start just after the reconfiguration finishes. In addition the control unit will start the following reconfiguration since the reconfiguration circuitry is idle and nothing is being executed in RU 2. When that reconfiguration finishes, a new *end of reconfiguration* event will be handled, and the control unit will start the reconfiguration of subtask 3. In addition it will also check if subtask 2 can start its execution, but in this case the associative table will report that there is one unsolved dependency. When this reconfiguration is also finished, a new event will be processed, but this time, no new reconfiguration will be started, since subtasks are never replaced until they have been executed, neither any execution, since subtask 2 and 3 have unsolved dependencies.

When the subtask 1 finishes its execution a new event will be generated. The first step to handle it is to update the dependencies, and then try to start a reconfiguration. In this case the system will start loading subtask 4. After that, the system will look for subtasks ready to be executed, and will

identify that subtask 2 and 3 are loaded and have no unsolved dependencies.

When the reconfiguration of subtask 4 finishes, the control unit will check if subtask 4 can start its execution, but in this case it is not possible because it has an unsolved dependency. In addition it will try to schedule the reconfiguration of subtask 2, but, again, it is not possible because RU2 is busy.

The next event is the end of the execution of subtask 2. Again, the table will be updated and the system will try to start a new reconfiguration, in this case is possible to start the reconfiguration of subtask 2, however, since this subtask is already loaded, the system will not carried out this reconfiguration, but only generate a *reused subtask* event.

When subtask 3 finishes, the system will update the dependencies, and will identify that subtask 2 and 4 can start their execution. Finally, when these subtasks finish, the manager will generate an interrupt to inform the processor that the graph execution has been completed.

6 Experimental results

In this section we will first evaluate the performance of our execution manager and afterwards, its cost in terms of area used to implement it. To this end we have implemented it in a VIRTEX-II PRO xc2vp30 FPGA using the ISE 8.1 development platform.

Some of the parameters of this manager are adjustable like the size of the associative table, the maximum number of successor per task, and the number of RUs in the system. Since these parameters will influence the performance and the cost of the system, we will initially evaluate a small manager with a table with only eight entries and three RUs, and afterwards, we will scale it up.

As a first experiment we have executed our manager to schedule the subtask graph depicted in Figure 7, and some other task graphs corresponding to three multimedia applications: a JPEG decoder, a Pattern recognition application, and a MPEG encoder. Table 1 summarizes the results of these executions.

Task	Number of subtasks	Delay introduced	Rec. overhead
Figure 7	6	400 ns	- 66%
JPEG	4	450 ns	-75%
Pattern rec.	7	420 ns	-80%
MPEG	5	580 ns	-70%

Table 1. Performance evaluation.

The *Delay introduced* column includes all the delays generated in the execution of the subtask graph due to the computations carried out by the execution manager. The *Rec. overhead* column contains the reductions in the reconfiguration overhead due to the prefetch technique. We will explain the results taking the graph from Figure 7 as example. In this case, only four of the events handled by the manager have introduced any delay. These events and the delays can be seen in Table 2. These events have generated delays because all of them are working with subtasks that belong to the critical path of the system. The total delay due to the management of the subtask graph and the application of reusing and prefetching techniques is 40 clock cycles. Since for this size the clock frequency of this manager is 100 MHz, the delay is only 400 ns, which is almost negligible, especially if we take into account that in this case our manager has hidden the reconfiguration latency of three subtasks and prevent a costly reconfiguration by reusing a subtask.

Event	New graph	End of reconfiguration (subtask 1)	End of execution (subtask 1)	End of execution (subtask3)
cycles	16	2	11	11

Table 2. Delays introduced by the execution manager.

Module	Number of slices	Slices (%)	Block RAMs	RAMs (%)
Control Unit	21	0%	0	0%
Rec. FIFO	8	0%	1	1%
RU info	36	0%	1	1%
Event queue	8	0%	1	1%
Associative table	331	2%	1	1%
Manager	**454**	**3%**	**6**	**4%**

Table 3. Implementation cost for a manager with an associative table with eight entries and three RUs.

The cost of each one of the modules and total cost of the execution manager is depicted in Table 3. As can be seen for this small manager the manager only needs 3% of the FPGA resources. The most expensive module is the associative table because it has been designed to provide maximum performance. However, the cost is still very affordable for this size. The associative table is also the module with a greater delay, hence it is interesting to study its cost and its delay for different sizes. These data are presented in table 4. As can be seen, the cost of the associative table is linear with the number of entries, and the clock frequency supported by the system is reduced by 15-20% each time that we double the size of the table. However, it must be remarked that current FPGAs include support for multiple clocks, hence the reduction of the clock frequency only reduce the performance of our manager, but not the performance of the subtask that are being executed in the RUs, since they can use their own clock.

Size	8	16	32	64
Slices (%)	2	4	8	16
Block RAMS (%)	0	0	0	0
Clock frequency	100	85	67	56

Table 4. Cost and clock frequency supported for different sizes of the associative table.

7 Conclusions and future work

In this paper we have presented an execution manager for a HW multitasking system. Its goal is to improve the efficiency of task management in multiprocessor reconfigurable systems, reducing the costly HW/SW communications and including HW support to carry out operation with complex data structures very fast.

This manager receives as input scheduled task graphs, and guarantees their proper execution taking into account the inter-task dependencies. In addition, it applies two optimization techniques to reduce the reconfiguration overhead, namely subtask reuse and configuration prefetch, hiding in our experiments about 70% of the reconfiguration latency.

The experimental results show that this module is very affordable for small/medium systems and that provides very high performance. They also demonstrate that the associative table is the most critical component of the system. Hence, as future work we want to design a more scalable associative table, and also reduce its delay by applying carry look-ahead techniques and adding some extra cycles.

4 References

[1] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", Proc. of FPL'02, pp. 795-805, 2002.

[2] J. Noguera, M. Badia., "Power-Performance Trade-Offs for Reconfigurable Computing". CODES+ISSS.pp. 116-121. 2004

[3] K. N. Vikram, V. Vasudevan. *Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 14, Issue 9, Sept. 2006, pp.1010-1023.

[4] H. Walder, M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices". Proceedings of the Design Automation & Test in Europe Conference (DATE'03), pag. 10290-10295. 2003.

[5] 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.

[6] W. Fu, K. Compton. "An execution environment for reconfigurable computing". Proc. of Field-Programmable Custom Computing Machines (FCCM), 2005, pp.149-158.

[7] J. Javier Resano, M. Elena Pérez, Daniel Mozos, Hortensia Mecha, Julio Septién. "Analyzing Communication Overheads during Hardware/Software Partitioning". Elsevier Microelectronic Journal, vol. 34-11, pag. 1001-1007. 2003.

[8] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N.K. Jha. "Power analysis of embedded operating systems". In Proc. of Design Automation Conference (DAC),2000.

[9] B.E. Saglam, V. Andmooney. "System-on-a-chip processor synchronization support in hardware". In Proceedings of Design Automation and Test in Europe (DATE'01). 2001.

[10] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde. "Operating System controlled Network-on-Chip". Proceedings of the Design Automation Conference (DAC), pag. 256-259, 2004.

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

[12] Xilinx Inc., Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet, Xilinx, San Jose, Calif, USA, 2005.

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

[14] J. Resano, D. Mozos, F Catthoor, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-time the Reconfiguration Overhead of Dynamically Reconfigurable HW" DATE05, pp.106-111. 2005