A Task-Graph Execution Manager for Reconfigurable Multi-tasking Systems

Juan Antonio Clemente* Carlos González* Javier Resano+ Daniel Mozos*
ja.clemente@fdi.ucm.es *carlosgonzalez@fdi.ucm.es* *jresano@unizar.es* *mozos@fis.ucm.es*
** Computer Architecture Department. Universidad Complutense, Madrid, Spain*
+ Computer Engineering Department. Universidad of Zaragoza, Madrid, Spain

Abstract

Reconfigurable hardware can be used to build multi-tasking systems that dynamically adapt themselves to the requirements of the running applications. This is especially useful in embedded systems, since the available resources are very limited and the reconfigurable hardware can be reused for different applications. In these systems computations are frequently represented as task graphs that are executed taking into account their internal dependencies and the task schedule. The management of the task graph execution is critical for the system performance. In this regard, we have developed two different versions, a software module and a hardware architecture, of a generic task-graph execution manager for reconfigurable multi-tasking systems. The second version reduces the run-time management overheads by almost two orders of magnitude. Hence it is especially suitable for systems with exigent timing constraints. Both versions include specific support to optimize the reconfiguration process.

Keywords: FPGA; task management; reconfigurable hardware

1. Introduction

Nowadays, commercial FPGAs, such as XILINX Virtex™ series [1] or Altera® Stratix [2] can be used to implement a System-on-a-chip (SOC) system where the reconfigurable hardware collaborates with one or several embedded processors. Currently most reconfigurable hardware vendors provide special design environments to develop SOC designs using these reconfigurable platforms, like the XILINX Embedded Development Kit (EDK) [3] or the Altera SOPC builder.

In these platforms the reconfigurable hardware can be divided into a set of Reconfigurable Units (RUs) wrapped with a fixed interface (as it was initially proposed in [4]). Using partial reconfiguration [5] each of these units can be reconfigured at run-time, in order to load a new hardware task (i.e. a configuration for a given RU) without modifying the remaining ones. Hence, each RU can be considered as an independent reconfigurable processor. This approach can be used to develop a hardware multitasking system that can be included in a heterogeneous hardware/software multiprocessor system. In this system light and control tasks will be assigned to the embedded processors, whereas computing intensive tasks will be accelerated using the RUs. With this approach, the same platform can be customized at run-time, by loading the proper configurations, 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.

Figure 1 presents our target system: a heterogeneous multiprocessor SOC that includes several reconfigurable units. Similar architectures have already succeeded in the multimedia market. The best example may be the Sony PSP™ architecture that includes two embedded R4000 processors (one of them with a vector processing unit), a Graphics Processing Unit (GPU), and a VME (Virtual Mobile Engine™ [6]) that is a reconfigurable processor developed by SONY. In fact SONY has included this reconfigurable processor in several portable platforms; since they have tested that it not only provides higher performance but also consumes five times less power than their embedded processors. The VME is not based on a FPGA-like architecture, but it is more like a reconfigurable data-path. However, it shares with our approach the idea of using reconfigurable hardware as a hardware accelerator in a heterogeneous multiprocessor system.

Figure 1 also depicts our target execution scheme. Basically we assume that applications include one or several control threads that are executed in one of the embedded processors. These threads deal with dynamic events that trigger the execution of one or several task graphs, represented as Direct Acyclic Graphs (DAGs). A similar approach has been successfully applied to deal with complex MPEG-4 and MPEG-2 applications for embedded multi-processor platforms [7,8].

Each time that a task graph must be executed, the middleware decides whether this task is assigned to the RUs or to the embedded processors, and provides the task-scheduling information. For instance, a 3D game application may identify at run-time that it needs to display a 3D object. This typically involves decompressing the object information, carrying out a computing intensive rendering process, applying some specific shading and texturing improvements, and finally performing a rasterization phase. If we assume that each phase is carried out by a different task, five tasks will be needed, and their data dependencies will be easily represented as a DAG. In this case the middleware may identify that hardware acceleration is needed in order to meet the exigent timing constraints and it will assign the task graph to the reconfigurable resources.

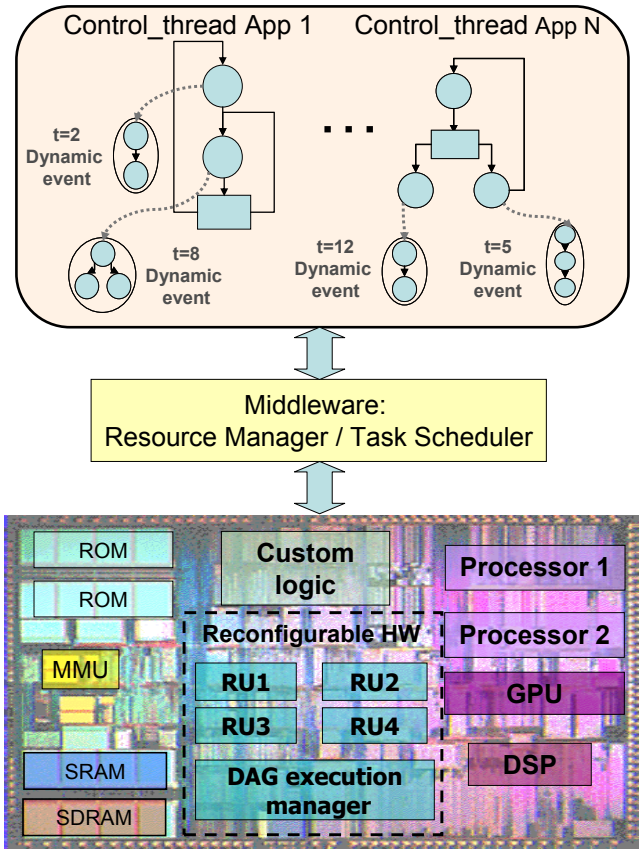

Figure 1. Target architecture and execution scheme.

Normally an embedded processor will manage the execution of these DAGs taking into account their internal dependencies. This involves dealing with complex data structures at run-time and, for a hardware multi-tasking system, frequent hardware/software communications. Since the work of this manager is very critical for the system, it is essential to evaluate its efficiency and to explore alternative implementations. In addition, this manager can greatly improve the throughput by applying run-time optimization strategies. Developing specific managers for each DAG is a complex error-prone task. Hence, developing a generic manager that can deal with any DAG can greatly simplify the design process. Moreover, since most management computations are carried out at run-time, the delays generated due to these computations must be minimized in order to prevent performance degradations.

In this regard, we have designed a generic manager (included in Figure 1 as *DAG execution manager*) that receives information from a task scheduler and guarantees the correct DAG execution taking into account both its internal dependencies and the selected schedule. In addition, the manager identifies which tasks can be reused from previous executions and provides support to apply a prefetch technique in order to hide the reconfiguration latency. These two optimizations are especially critical for reconfigurable hardware since the reconfiguration latency may have a major impact in the system performance [9]. We have developed two different implementations of our manager (a hardware version and a software one) and we have implemented them using a VIRTEX-II PRO FPGA and the EDK design environment. In both cases, we have integrated this manager in a platform with a processor and a customizable set of RUs. However, developing a hardware multi-tasking system for FPGAs involves dealing with many complex issues, such as providing inter-task communication support, or dynamically allocating memory resources to the running tasks. We believe that these issues are orthogonal to the problem that we are targeting: dealing efficiently with DAGs execution on a reconfigurable multiprocessor system; hence, we have not developed a complex hardware multi-tasking system. Instead of that, we have developed a simple hardware module that simulates the execution of tasks in the RUs using programmable counters. These counters are included in the system as peripherals to model the reconfiguration latency and the execution time of each task. This simplified environment allows us to evaluate with clock-cycle precision the penalties generated due to the graphs' management considering both the computations performed by the manager and the communications among the RUs and the processor. In this platform we have evaluated the cost (i.e. hardware area and memory resources) and the performance of both versions of our manager using a set of task graphs obtained from actual multimedia applications.

The rest of the paper is organized as follows: next section reviews the related work; section 3 presents the problem overview; section 4 describes our execution manager; section 5 analyzes the two different implementations, section 6 presents the experimental results and finally section 7 explains our conclusions and indicates some lines for future work.

2. Related work

Many research groups have proposed to build a hardware multi-tasking system in which hardware tasks are assigned to reconfigurable resources at run-time. Marescaux et al. [4] were the pioneers developing the first hardware multi-tasking platform. They divided the entire area of a FPGA into identical tiles, and developed the OS support needed to assign tasks to these tiles at run-time. In the same way, there have been other interesting hardware multi-tasking platforms, such as [10,11,12,13,14,15]. In [10] Walder et al. present some techniques to manage and schedule the execution of task graphs in a 1-dimension block-partitioned reconfigurable device. In [11] Huang et al. present a design methodology flow to extract hardware tasks from a software program and then target them into a hypothetical multi-tasking system. In [12] Noguera and Badia propose a hardware-based dynamic scheduler for reconfigurable architectures that applies a list-scheduling heuristic. However, the authors did not implement their design, but they only included it in their specific simulation environment. In [13] Qu et al. propose to include several reconfiguration controllers in a hardware multi-tasking platform in order to reduce the reconfiguration overhead. In [14] Vikram and Vasudevan propose a hardware multi-tasking specific system to take advantage of data-parallelism. Finally, in [15] Fu and Comptom propose to extend a general purpose multi-threaded OS to include reconfigurable units.

A well-known disadvantage of reconfigurable systems is that the reconfiguration latency may generate significant overheads. To try to overcome this problem, many authors have proposed techniques to reduce it. [16] is an interesting survey that gathers most of these techniques. The most interesting ones are reconfiguration caching and task prefetching: the former [17,18] consists in storing some task reconfigurations in a n-level memory (as a cache or a scratchpad) close to the reconfigurable hardware according to certain criteria, in order to reduce the reconfiguration latency. The latter [19] consists in carrying out reconfigurations as soon as possible, thus overlapping a reconfiguration latency with the execution time of previous tasks. This technique is very powerful when dealing with task graphs as it has been demonstrated in [20,21,12,13]. Another way to optimize hardware multi-tasking systems is applying a replacement technique that attempts to maximize the reuse of tasks. This was initially proposed in [22] where the authors proposed a prediction scheme to optimize the replacement. In our previous work we have also analyzed this problem developing a specific replacement heuristic for it. Further details can be found in [23].

Another management approach is to provide an Operating System (OS) with the capabilities to manage hardware resources as if they were software processes. Interesting contributions about this topic are [24,25,26]. In [24] Wigley

and Kearney review the services needed for this extension. In [25] Nollet et al. propose a distributed OS support for inter-task communications. Finally in [26] Hayden Kwok-Hay So et al. present a LINUX-based OS whose interface has been extended in order to deal with hardware processes. However, in the latter cases the current implementations were not very efficient due to the hardware-software communications that generate very significant run-time penalties. Thus there is a lot of work to be done in this regard in order to develop an OS that can manage reconfigurable resources transparently and with a good performance.

As stated before, we have implemented two versions of a generic execution manager for hardware task-graphs in multi-tasking systems based on reconfigurable hardware. As we will explain later, both implementations offer different trade-offs between hardware resources/performance. On the one hand, we believe that our manager can be integrated in most of the hardware multi-tasking systems proposed in the literature, such as [4,10,11,12,13]. However it is not compatible with [14] and [15] because [14] only targets data-parallelism, and [15] is focused on a multi-threaded environment.

On the other hand, this manager could also be part of the kernel of an OS, like in [25,26,27]. Moreover, our execution manager implements two optimization techniques, namely reconfiguration prefetch and reuse, which in our experiments hide most of the reconfiguration overheads. We have measured with clock-cycle accuracy the impact in the system performance of both versions using graphs extracted from actual multimedia applications. The results show that for certain graphs the management computations generate important overheads (up to 18% of the execution time) when using a software manager. These overheads can be reduced by almost two orders of magnitude using the hardware version. To the best of our knowledge this is the first time that a generic execution manager for DAGs execution in reconfigurable multi-tasking systems have been implemented and evaluated.

3. Problem overview

In this work, the main issue regarding task-graph management is to efficiently deal with DAGs respecting their internal dependencies and the schedule provided by a task scheduler that assigns the tasks to the RUs and specifies the execution sequence of each RU. A DAG is defined as a set of nodes, which in this case represents computational tasks and a set of vertexes, which represent the internal task dependencies. A task is the basic scheduling unit (i.e. a node of a task graph). Figure 2.a depicts an example of a simple task graph.

In our system we assume that the set of task graphs has been analysed at design-time in order to obtain accurate execution-time estimations of each computational task. If the execution time of these tasks heavily depends on the input data we propose to use the same approach as in [7,8]. In this work they include several scenarios for those task graphs with important execution-time variations. The idea is that all the scenarios share the same graph structure, but in each scenario the nodes have different weights, i.e different execution time estimations; and at run-time, when the current input data are known, the task scheduler selects the proper scenario.

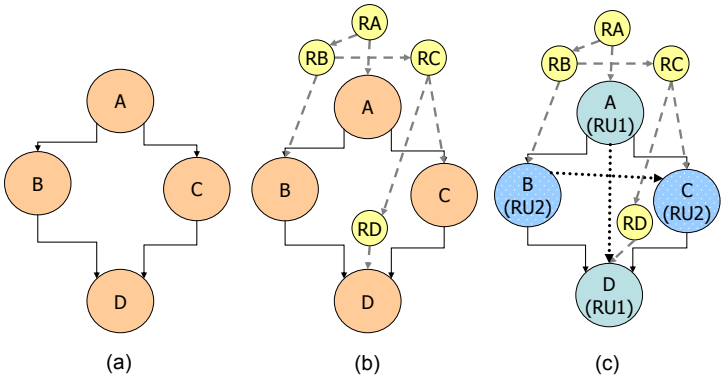

Figure 2. Example of initial task graph (a), task graph including the reconfiguration sequence (b), and final task graph including also the dependencies due to the task schedule (c).

The manager uses these execution-time estimations to select a reconfiguration sequence (this is further explained in the following section). In order to execute a task in a RU, this task must be loaded previously carrying out a run-time reconfiguration. In order to execute a task graph, it is likely that several reconfigurations will be needed.

However, current reconfigurable platforms only include one reconfiguration controller. Hence, they can only carry out one reconfiguration at a time. Our manager will update the original task graph including the reconfiguration sequence as it can be seen in Figure 2.b. To this end the original graph is extended with new nodes, which in this case do not represent computing tasks but run-time reconfigurations, and new edges, which represent the dependencies introduced due to the reconfiguration sequence selected.

Finally the task graph is also updated taking into account the schedule selected by the task scheduler. In the example of Figure 2.c the scheduler has assigned tasks A and D to RU1, and task B and C to RU2. In addition, the schedule specifies that A will be executed before D, and B will be executed before C. Due to this assignment two new edges are added to the original graph in order to guarantee that the task graph execution will follow the given schedule and to prevent structural conflicts.

Our manager can be seen as a Data-Flow architecture [28] that steers the task-graph execution taking advantage of the inner graph parallelism while guaranteeing that all dependencies, either due to data dependencies or structural conflicts, are met. However this manager clearly differs from classical data-flow solutions since it has to deal not only with the original task graph but also with the information provided by the scheduler and the reconfiguration sequence. In addition it aims to optimize the reconfiguration process applying both reuse and prefetch techniques.

4. The execution manager

In this work we present an execution manager that guarantees the correct execution of DAGs in hardware multi-tasking systems taking into account their internal dependencies and a schedule provided by a task scheduler.. The schedule must assign the tasks to the RUs and specify the execution sequence of each RU.

The manager also provides support to optimize the reconfiguration process applying both reuse and prefetch techniques. On the one hand, the manager identifies reusable tasks, in order to reduce the number of reconfigurations needed. On the other hand, a prefetch technique is applied to attempt to carry out the reconfigurations in advance. In order to make 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 are computed performing an ALAP (as late as possible) scheduling. They 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. With this criterion the first task in the critical path has always more weight than the others. We use this weight to generate a reconfiguration sequence that includes all the tasks assigned to a RU sorted according to their weights. Since this phase is carried out at design-time, it does not generate any run-time penalty. Hence, if needed, the reconfiguration sequence could be computed using a more complex approach. Nevertheless, according to our experiments, the results obtained with these weights are very good.

At run-time our manager will follow the reconfiguration sequence of each DAG guaranteeing that each reconfiguration starts as soon as possible according to this sequence, the state of the RUs, and the reconfiguration circuitry (a reconfiguration can start if the previous tasks assigned to the same RU have finished and the reconfiguration circuitry is free).

The manager monitors the system events and carries out the proper actions. It supports four different events:

1. *new graph:* this event is generated when the scheduler has sent the information of a new graph.
2. *end of execution*: this event is generated by a RU controller each time that a task has finished.
3. *end of reconfiguration:* this event is generated when a reconfiguration has finished.
4. *reused task:* this event is generated when a task is reused. This happens when a RU controller identifies that it has received the order of loading a task that is already loaded.

The *end of reconfiguration* and the *reused task* events are similar from the manager's point of view, since reusing a task is the same as carrying out a reconfiguration in one clock cycle.

Our manager processes sequentially the events generated in the system. Figure 3 depicts the actions triggered by each event. When an *end of execution* event is processed, the manager updates the dependencies. Then, if the reconfiguration circuitry is idle, it checks if it is possible to start a reconfiguration. Finally, the manager checks if any of the tasks that are currently loaded can start its execution. For the *end of reconfiguration* and *reused task* events, the manager checks if the task that has been loaded can start its execution. In addition, it also tries to start another reconfiguration. Finally, for the *new graph* event, the manager updates its internal structures using the data of the task graph and its schedule. After that, if the reconfiguration circuitry is idle, it will check if it is possible to start loading one of the new tasks.

```
CASE event IS:
   end_of_execution:
      IF (RC = idle)
         look_for_reconfiguration()
      update_task_dependencies(&task_ready)
      FOR i = 0 to NUMBER_OF_RUS
         IF (RU_state = IDLE) AND (task_ready)
            start_execution()
   end_of_reconfiguration or reused task:
      check_dependencies(&task_ready)
      IF (task_ready = TRUE)
         start_execution()
      look_for_reconfiguration()
   new_graph:
      IF (RC = idle)
```

Figure 3. Pseudo-code of the manager. *RC* stands for "*Reconfiguration Circuitry*"

Figure 4 describes in detail an example of the management of the execution of a DAG with 5 tasks in a system with 3 RUs. Initially the processor will send the information about the graph and the selected schedule to the manager generating a *new graph* event. In this case the manager will receive the following reconfiguration sequence: 1-3-2-4-2. Hence, the manager will start the reconfiguration of task 1. When task 1 finishes its reconfiguration, RU 1 will generate an *end of reconfiguration* event. Now, the manager will check if task 1 can start its execution. In this case it has no unresolved 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 RU 3 is ready. When that reconfiguration finishes, a new *end of reconfiguration* event will be handled, and the manager will start the reconfiguration of task 2. In addition it will also check if task 3 can start its execution, but in this case there is still one unresolved dependency. When this third reconfiguration is also finished, a new event will be processed, but this time, no new reconfiguration will start, since tasks are never replaced until they have been executed, neither any execution, since task 2 and 3 have unresolved dependencies.

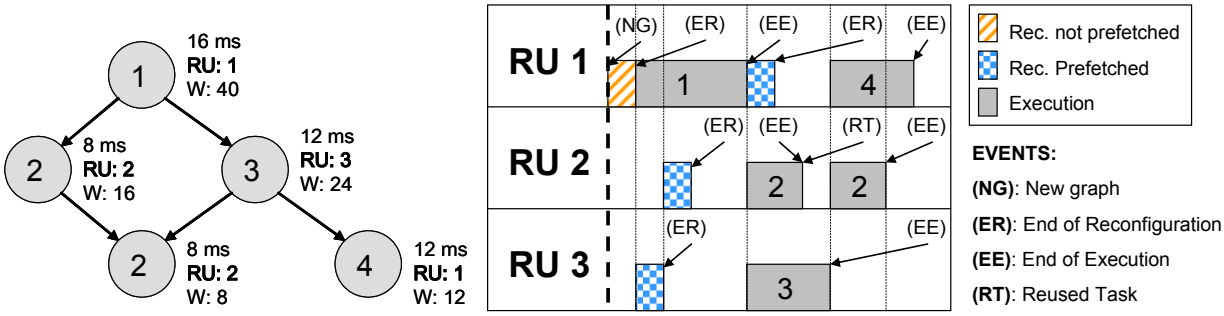

Figure 4. Example of the execution of a DAG. W: weight of the task

When the task 1 finishes its execution a new event will be generated. The first step to handle is to update the dependencies, and then to try to start a reconfiguration. In this case the system will start loading task 4. After that, the system will look for tasks ready to be executed, and will identify that task 2 and 3 are loaded and have no unresolved dependencies.

When the reconfiguration of task 4 finishes, the manager will check if task 4 can start its execution, but in this case it is not possible because it has an unresolved dependency. In addition it will try to schedule the reconfiguration of task 2, but, again, it is not possible because RU2 is busy. The next event is the end of the execution of task 2. Again, the table will be updated and the system will try to start a new reconfiguration. In this case it is possible to start the reconfiguration of task 2. However, since this task is already loaded, the system will not carry out this reconfiguration, but only generate a *reused task* event, which is similar to an *end of reconfiguration* event. Finally, when task 3 finishes, the system will update the dependencies, and will identify that task 2 and 4 can start their

execution. This example illustrates the benefits of the prefetch and reuse techniques. In this example only one of the five reconfigurations has generated a delay in the execution since one of the configurations has been reused and the remaining three have been prefetched.

5. Implementation details

We have developed a hardware implementation of our manager and a software one. On the one hand, the software version is a program that runs in an embedded processor (in our case a Power PC embedded on the VIRTEX-II PRO FPGA) and that communicates with the RUs by means of their communication interface. On the other hand, the hardware version is a hardware component included as an on-chip peripheral connected with the processor via a bus and with the RUs using point to point connections. In both cases, we do not have real RUs, but we simulate their behavior (reconfiguration and execution times) using a hardware module that includes two programmable counters and a state register. When a task is assigned to one of these modules, it stores in the counters the reconfiguration latency and the execution time of the task. In addition, the task information is stored in the state register. When the module receives the order to start the reconfiguration, the corresponding counter starts the countdown, and when it finishes it will inform the manager generating an interrupt. When the module receives the order to start the execution, the other counter will behave identically. We have implemented both of them using the Xilinx EDK 9.1i environment because it provides support for easy peripheral development and integration and for compiling, simulating and debugging C and C++ software projects. Besides, it facilitates interrupt handling, hardware/software communications (using the peripherals interface and the provided bus hierarchy), DMA transactions and execution-time measurements. In addition, our managers can be easily ported to any EDK project.

5.1. Software implementation

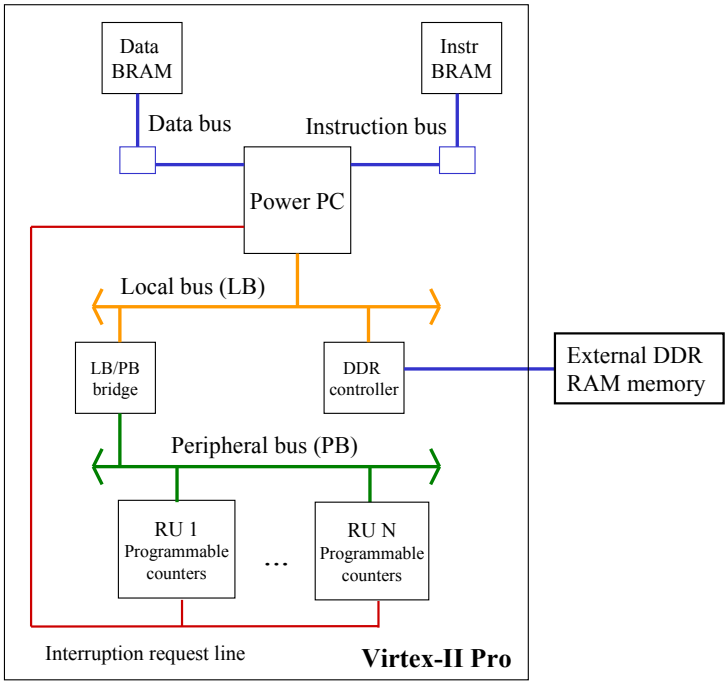

Figure 5. System with a software execution manager

In this section we will explain the software implementation of our manager. Figure 5 depicts a typical structure for our target system including a Power PC embedded processor, some on-chip RAM memories (BRAMs), that can be used as L1 caches or scratchpads, an off-chip DDR that can be used as main memory, and a bus hierarchy that

can be used to extend the system functionality by including different peripherals. In this case we have connected the RUs that include the programmable timers.

Each timer contains two identical counters, which are used to simulate reconfiguration and execution times, respectively. Both of them are addressed with the identifiers 0 and 1 in the C code of the Figure 6. The processor can communicate with them using a driver that includes start, stop and resume commands. Thus, when a counter finishes it generates an interrupt that the manager uses to identify a run-time event. For instance, if the first counter of a RU generates an interrupt the manager will deal with an *end of reconfiguration* execution event generated by the task currently assigned to this RU, whereas if the second counter generates the interrupt the manager will deal with an *end of execution* event.

The system also includes an additional timer that is used to measure the total execution time. Therefore, this software version of the manager consists of a C program that includes the platform initialization code (components initialization and interrupt configuration), a set of interrupt service routines (ISRs), and the event handling code.

```
void manager_code(){
   inicialize_data_structures();  //Initialize Graph, Events, RUs...
   while (not finished){
      if (Events not empty){
         pick_event(&ev);
         switch(ev){
            case 0: new_task_event();
            case 1: end_reconfiguration_event();
            case 2: end_execution_event();
         }
      }
   }
}
void ISR_0{ ISR_code(0); }        //ISRs for the counters
...
void ISR_N{ ISR_code(n); }
void ISR_code(int n){
   int c_address = addr(n);       //Counter address is obtained
   stop_counter(c_address);       //Stop counting -> stop simulating
   if (counter_activated(c_address) == 0){     //Counter 0 activated
      generate_end_reconfiguration_event();
   }else{                                      //Counter 1 activated
      generate_end_execution_event();
   }
   //Check if the event end_execution has been generated NUM_NODES times
   if (finished_graph()) finished = TRUE;
}
int main(){                       //MAIN PROGRAM
   initialize_peripherals();
   inicialize_timer(&timer);      //Initialize timer and start counting
   int t1 = getValue_timer(&timer);
   generate_new_task_event(graph_addr); //Generate the first event
   manager_code();
   int t2 = getValue_timer(&timer);
   printf(t2-t1);                 //Elapsed time is shown
```

Figure 6. Pseudo-code of the software implementation of our manager for a given task-graph..

The pseudo-code used to evaluate the software manager performance is described in Figure 6. It works as follows: Firstly, it initializes the peripherals and configures the interrupts in such a way that different *ISRs* (*ISR_0* … *ISR_N*) are assigned to each timer. After that, the function *manager_code()* also initialises the data structures used to represent the task graph, the queue of events and the RUs.

The main loop of the *manager_code()* function is then executed as many times as necessary in order to process all the events. Every time the manager has to send a reconfiguration or execution command to a RU, it sends a request to the appropriate counter indicating the number of clock cycles associated with the event. With this

approach the counter will simulate the reconfiguration and execution latency of the tasks. When a counter finishes, it generates an interrupt, and the execution flow switches to the proper *ISR*. Afterwards, depending on which counter has generated the interrupt, the corresponding event is generated in the *ISR_code()* function: *end of reconfiguration* or *end of execution*. When the ISR finishes, the execution flow returns to the main loop, which immediately detects that a new event has just been generated. Hence, it carries out the proper actions (Figure 3) invoking one of the following functions: *new_task_event()*, *end_reconfiguration_event()* or *end_execution_event()*. When all the tasks in the graph have been executed the value of the flag *finished* is set to one, and the execution returns to *main()*.

Finally, the manager measures and displays the total execution time. This execution time can be compared with the ideal execution time of the graph in order to identify the delays due to the management process.

5.2. Hardware implementation

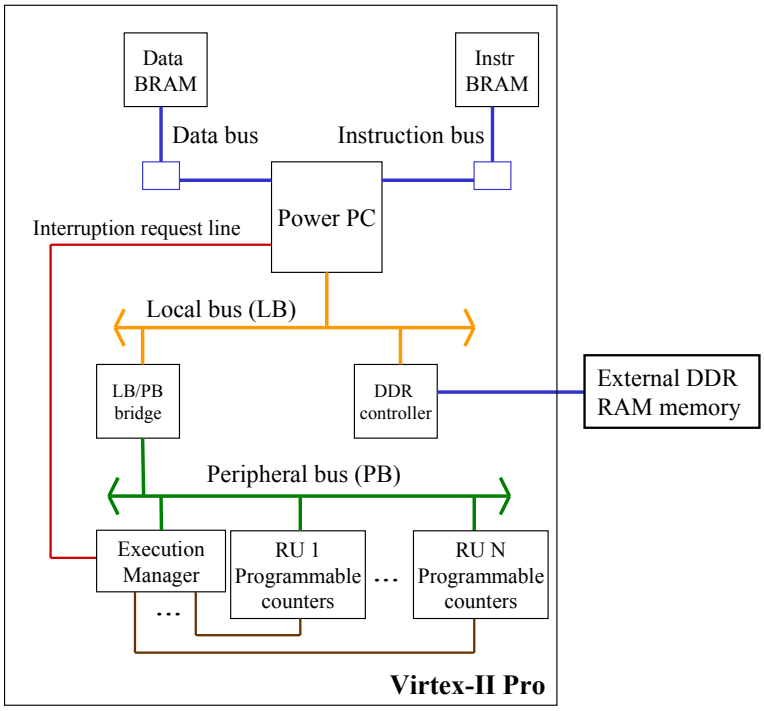

Figure 7. System with a hardware execution manager.

Figure 7 depicts the same system as Figure 5 but now with a hardware manager included as a peripheral in the system. In this case the RUs do not communicate directly with the processor, but they interact with the execution manager. This scheme drastically reduces the number of interrupts that the processor must handle. When the platform must execute a new task graph, the processor sends the information to the manager including the task graph description and the selected schedule. With this information the manager steers the graph execution, taking into account the events generated by the RUs, and at the end, it generates an interrupt to inform the processor.

The EDK environment supports two easy options to send the data to the manager: a FIFO included in the manager bus-interface or a DMA transaction. The second option is optimal for performance, since the processor does not have to write the information directly on the FIFO, but it only needs to send the initial address and the size of the data. The DMA controller generates an extra area overhead (Table 1); however, according with our experiments, a DMA transaction is almost 3.5 times faster than an equivalent standard data transfer between the processor and a local FIFO.

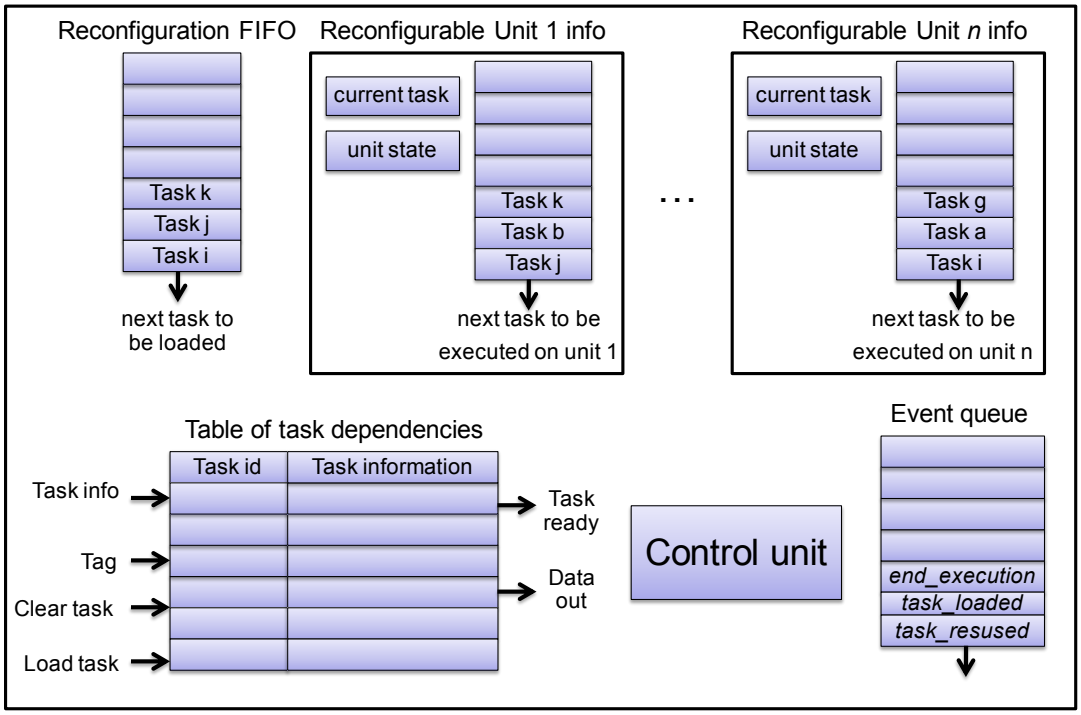

Figure 8. Hardware execution manager

Figure 8 shows the structure of the manager. It uses an associative table to store the task graph information and several FIFOs to store the schedule. In addition, it includes two registers and a small controller for each RU, an event queue that stores the run-time events and a control unit that processes these events. We will describe each of these elements in detail.

Table of task dependencies. This is an associative table that can be customized for different sizes and maximum number of successors per task. This table monitors the tasks dependencies. It supports three different operations: *insertion*, *deletion and update*, and *check*. Figure 9 describes the structure of each table entry that includes the task tag, a counter that indicates the number of unresolved dependencies (predecessor counter), and the number of successors and their tags.

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 studied two different implementation options for the design of this table. Initially we designed a fully associative table with a register pointing to the first free entry. In this version the *Insertion* operation is carried out in just one clock cycle but the delay increases significantly with the size of the table. Hence we have developed a second implementation in which we divided the table in sub-tables with 8 entries where each sub-table includes a register 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. This approach increases the latency of the operation but the size of the table does not affect the clock period. Hence it is especially suitable for large tables.

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 hardware support depicted in Figure 10. 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. 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.

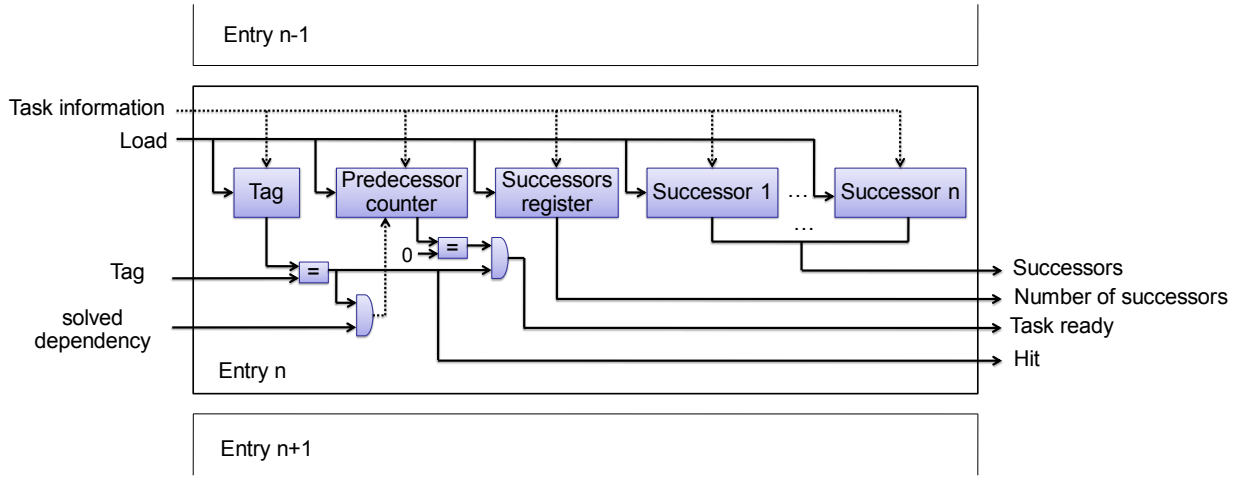

Figure 9. An entry of the table of task dependencies

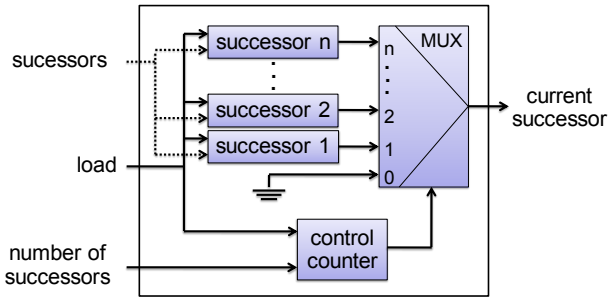

Figure 10. Hardware support for the *Deletion and update* operation

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. whether 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.

Reconfigurable Unit info. In our manager each RU includes a FIFO and two registers. The FIFO stores in order, following the given schedule, all the tasks that have been assigned to this unit and are waiting for execution. The registers store the state of the unit and the current loaded tasks. In addition, there is a small controller that works as an 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 task is assigned to the unit or a task is loaded in the RU.

Reconfiguration FIFO. This FIFO stores the reconfiguration sequence that is stored when the information of a new task graph is received. Each time that a reconfiguration is carried out the corresponding task 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 involved task. Since it is possible that two or more RU controllers attempt to write an event in the queue at the same time, the manager includes a simple arbiter with a fixed priority scheme that only allows one write per cycle by activating the appropriate grant line. In addition, this arbiter blocks the queue when it is full.

Control Unit. The control unit extracts the events from the queue and carries out the proper actions (described in Figure 3). This module is responsible for the correct operation of the system.

Table 1 presents the area used by each one of these modules and the total cost of the execution manager. We have obtained these results using the Xilinx ISE 9.1i development tool. As it can be seen in the table, this manager needs 7% of the FPGA resources but only needs 3% if we do not include the DMA controller. In this case, 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.

Table 1. Implementation cost for a manager with four RUs, a DMA controller, and an associative table with 8 entries in a Virtex-II PRO xc2vp30 FPGA.

Module	Number of slices	Slices (%)	Block RAMs	RAMs (%)
Control Unit	21	0,1%	0	0%
Rec. FIFO	8	0,1%	1	0,6%
RU info	38	0,2%	1	0,6%
Event queue	8	0.1%	1	0,6%
Associative Table	307	2%	1	0,6%
DMA controller	621	4%	0	0%
Manager	**1117 (496)**	**7% (3%)**	**4**	**2,6%**

The associative table is also the module with a greater delay; hence we have implemented this table for different sizes in order to study the evolution of its cost and its delay. The results show that the area needed grows linearly with the size of the table. As it is shown in Table 1, an associative table with 8 entries consumes 2% of the FPGA resources. Hence, a table with 16 entries demands 4% of the FPGA resources, and a table with 32 entries uses 8%. Nevertheless, the supported clock period is constant, 100MHz, as long as the table includes the modification for the *insertion* operation described previously.

In embedded systems the size of the code is also an important metric since memory space is normally very restricted. For instance, in the Virtex-II PRO xc2vp30 FPGA the size of the internal RAM is 306KB. The size of the code of the software implementation of the manager is 190KB whereas for the hardware implementation is just 82 KB. These sizes include not only the manager but all the drivers for the peripheral included in the system. Hence, although the hardware implementation of our manager has a significant hardware area cost, it greatly reduces the system memory requirements. This can be especially useful for instance to reduce the accesses to external memories.

6. Performance evaluation

In this section we will evaluate the performance of the two implementations of our manager. We have implemented both of them in a FPGA Virtex-II PRO xc2vp30 using the EDK environment, and we have evaluated them using a set of task graphs extracted from actual multimedia applications, that include two versions of a JPEG decoder (JPEG and Parallel-JPEG), a MPEG-1 encoder, a Pattern recognition application (HOUGH), and a 3D rendering application based on the open source Pocket-GL library (Pocket GL). In the latter case the application includes 20 different tasks graphs with 2, 4, 5 and 6 nodes consecutively; hence in this case the table only presents the average results of the DAGs with the same number of nodes. All these graphs have been scheduled for a platform with 4 RUs using the scheduling environment presented in [20], although any other task scheduler can be used.

Table 2 presents the delays that our manager introduces for these graphs. Columns 2 and 3 include the number of tasks of each DAG and the initial execution time assuming that the management of the execution does not generate any overheads. Then, columns 4-7 provide details regarding the overhead generated by the software version. Note that the percentage of column 7 represents the impact of the total reconfiguration overhead with respect to the initial execution time (Column 4 – Column 3) / Column 3. Finally, the last column refers to the hardware version: these delays are due to the computation times and the hardware/software communications among the processor and the RUs.

For the software implementation, the computation times include the execution time of the code that deals with the task graph management and the communication times include interrupt handling (which in turn implies switching

into the corresponding ISR, stopping/resuming the timer and enabling/disabling its interrupts) and the initialization of the RUs each time that a new task is assigned to them. Column 4 shows the actual execution times and columns 5 and 6 break down the delays introduced by the software implementation of the manager into computation and

Table 2. Performance evaluation of the task-graph execution manager

Task graph	Number of tasks	Initial execution time (ms)	MANAGER PERFORMANCE				
			Software				Hardware
			Execution time (ms)	Management time (ms)		% overhead	Execution time (ms)
				Computations	Communica-tions		
JPEG	4	79	79.87	0.573	0.298	**1.10**	79.022
PARALLEL-JPEG	8	54	55.42	0.856	0.563	**2.62**	54.023
MPEG-1	5	37	38.02	0.659	0.358	**2.75**	37.023
HOUGH	6	94	94.88	0.525	0.355	**0.93**	94.022
POCKET GL (A)	2	4.1	4.802	0.523	0.179	**17.12**	4.122
POCKET GL (B)	4	16.02	16.953	0.635	0.298	**5.82**	16.042
POCKET GL (C)	5	26.89	27.917	0.669	0.358	**3.82**	26.912
POCKET GL (D)	6	48.75	49.875	0.671	0.417	**2.31**	48.772

communication times. Finally, column 7 presents the impact that these overheads have in the overall performance of the applications.

For the hardware implementation, management time includes both the delay introduced by the logic of the manager and communication time. Column 8 shows the total delays generated by this version. In this case, the delays introduced due to management computations are just a few hundreds of cycles in a system running at 100 MHz (i.e. a few microseconds). Regarding the communications, these delays are approximately 2200 cycles when using a DMA.

As it can be seen in the table, the software implementation generates delays that go from 1% to 17% of the initial execution-time. For graphs with high execution times (like HOUGH, for instance) these delays are very low. However, there are many cases (in graphs with low execution times, like POCKET-GL, for example) in which this delay can generate a very important penalty in the performance of the system (up to almost 18%), since this overhead is quite significant with respect to the graph execution time. In the latter cases these delays are not acceptable, and we can greatly reduce both communications and management times using the hardware implementation. Thus, *communications* times are reduced on average from 0.35 ms to just 0.022 ms (i.e. between 1 and 2 orders of magnitude lower), and the *computations* time is also drastically reduced (we achieve a reduction on average from 0.64 ms to 0.002 ms, which is between 2 and 3 orders of magnitude better).

Figure 11 presents the reduction of the reconfiguration overhead due to the optimization techniques applied by our manager when it executes the same tasks as the ones shown in table 2. The figure includes four execution times. *On demand* represents a situation where no optimization is carried out (reconfigurations start when a task must be executed). *First execution* represents the same execution but applying our prefetch optimization. *Second execution* shows the benefits of the reuse technique if a graph is executed twice consecutively. We think that it is interesting to evaluate this point because these applications are normally cyclic and/or execute several times in a row. Thus, in this situation sometimes it is possible to reuse some of the tasks and this leads to further reconfiguration overhead reductions. Finally *Ideal* represents the execution time with no reconfiguration overheads.

In these experiments the reconfiguration latency has been set to 4 ms, which is the time needed to reconfigure one fifth of a XC2VP30 FPGA, since all the tasks considered in these experiments fit in this area. The results show that an *on demand* strategy is very inefficient, and that prefetch optimizations can greatly improve the system performance. On average the reconfigurations generate a 42% execution-time overhead when using the *on demand* approach, whereas it only generates about 13% overhead when applying our prefetch optimization. In addition, this overhead is reduced to just 9% when the manager can reuse some of the tasks. In these experiments we have used a small number of RUs to address a complex scenario in which there is certain competition among the tasks to use the available resources. That is the reason why the reconfiguration overhead is larger with only a few RUs. However, further reductions can be achieved if a certain number of RUs are included in the system, depending on the number of tasks to execute. For instance, Figure 11 shows no benefits due to task reuse for the Parallel-JPEG application with up to 4 RUs, but if we include an additional RU (i.e. five RUs in all) one of the tasks can be reused and the

reconfiguration overhead is reduced by 4 ms. The figure does not show the results for one RU because in that case no task can be reused neither prefetched; hence all the columns would show the same execution time.

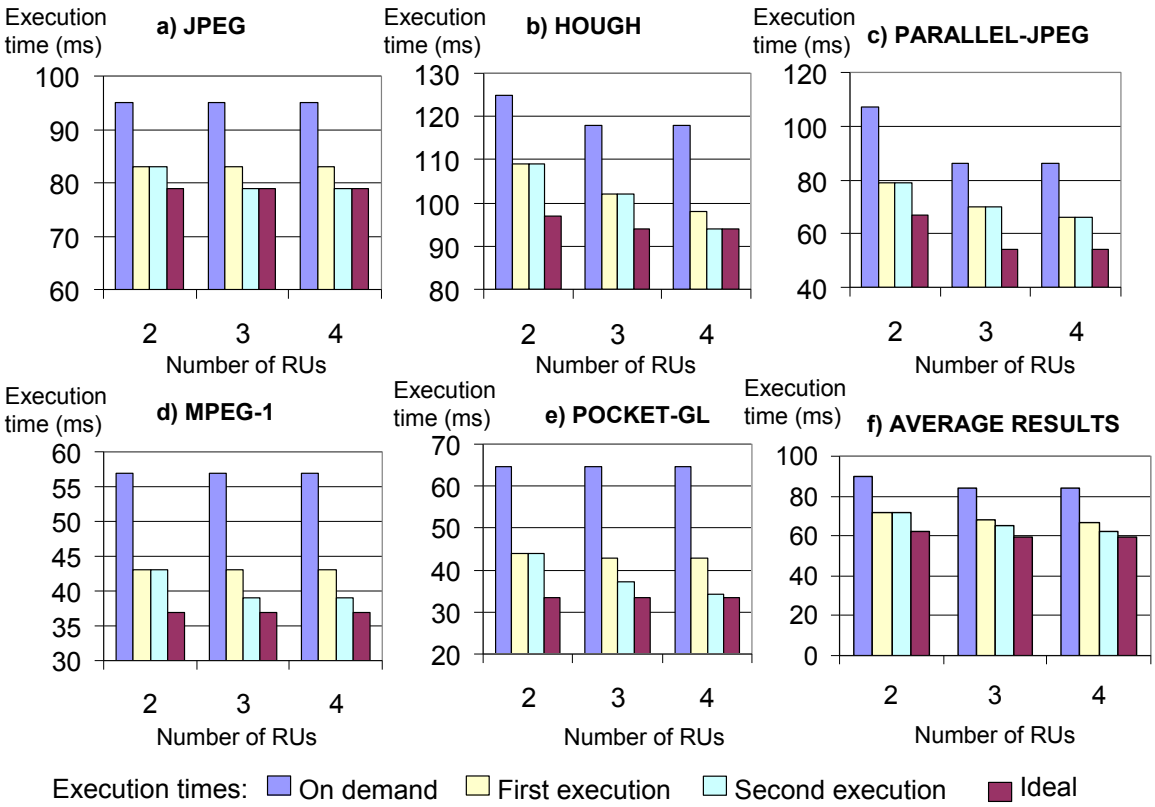

Figure 11. Impact of the prefetch and reuse optimizations in the system performance

It is important to mention that the computations regarding the prefetch and reuse optimizations are already included in the results presented previously in Table 2. Hence, although our manager applies these techniques at run-time, they generate almost no run-time penalty. This is especially true in the case of the hardware implementation of our manager. This is an important improvement when compared with most prefetch approaches presented previously, because, as we explained in the related work section, most of then cannot be applied at run-time, while others propose complex run-time computations, but they do not evaluate the run-time penalty generated by these computations.

7. Conclusions and future work

In this paper we have presented an execution manager for hardware multitasking systems. It receives scheduled task graphs as inputs, and guarantees their proper execution taking into account the selected schedule and the inter-task dependencies. Its goal is to simplify the task management in multi-tasking reconfigurable systems and improve their efficiency applying optimization techniques like configuration prefetch and reuse.

We have implemented two versions of this manager (a hardware and a software one) in a FPGA using the EDK design environment. Our software implementation consists of a program running in an embedded Power PC processor that interacts with a set of RUs; whereas the hardware version includes micro-architecture support to carry out the management operations in just a few clock cycles. In addition, it drastically reduces the need of costly hardware/software communications.

We have evaluated the cost and the performance of the two implementations using a set of task graphs from actual multimedia applications. Regarding the performance, the software version produces in some of our experiments execution time overheads that are reasonably low (1-2% of the total execution time). However, in other

cases this overhead raises up almost 18%, which is unacceptable if we want to achieve a good performance. Anyway, the hardware implementation generates negligible delays (only a few hundreds of clock cycles), it generates an extra area penalty (almost 7% of the available hardware resources) and it greatly reduces the size of the code of the embedded processor. Hence, our two implementations offer different area/performance trade-offs and the system designer can select the one that better fits his requirements depending on whether he prefers to maximize the performance or minimize the area cost.

In addition we have tested the benefits of our prefetch and reuse optimization techniques: The results demonstrate that these techniques can be very effective to reduce the reconfiguration overhead. In our experiments they have reduced an initial overhead of 42% to just 9%.

As future work we want to develop a complete hardware multi-task system and analyze the benefits of developing a hardware implementation of the task scheduler.

8. Acknowledges

This research was supported by PR34/07-15821 and TIN2006-03274.

9. References

[1] www.xilinx.com/products/silicon_solutions/index.htm.

[2] www.altera.com/products/devices/stratix-fpgas/about/stx-about.html.

[3] www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[4] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde and R. Lauwereins, Interconnection Networks enable Fine-Grain Dynamic Multi-Tasking on FPGAs, in the 12th International Conference on Field-Programmable Logic and Applications (FPL); September 2-4, 2002; Montpellier, France; pp. 795-805.

[5] P. Lysaght, B. Blodget, J. Mason, J. Young and B. Bridgford, Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs, in the 16th International Conference on Field-Programmable Logic and Applications (FPL); August 28-30, 2006; Madrid, Spain; pp. 1-6.

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

[7] C. Wong, P. Marchal and P. Yang, Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform, in the 9th International Symposium on Hardware/Software Codesign (CODES); April 25-27, 2001; Copenhagen, Denmark; pp. 170–175.

[8] P. Yang and F. Catthoor, Dynamic mapping and ordering tasks of embedded real-time systems n multiprocessor platforms, in the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES); September 2-3, 2004; Amsterdam, The Netherlands, Lecture Notes in Computer Science 3199, pp. 167–181.

[9] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu, An Overview of Reconfigurable Hardware in Embedded Systems, EURASIP Journal on Embedded Systems, vol. 2006, issue 1, pp. 13-13, 2006.

[10] H. Walder and M. Platzner, Online Scheduling for Block-partitioned Reconfigurable Devices, in the Conference on Design, Automation and Test in Europe (DATE); March 3-7, 2003; Munich, Germany; pp. 290-295.

[11] Z. Huang and S. Malik, Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks, in the Conference on Design, Automation and Test in Europe (DATE); March 12-16, 2001; Munich, Germany; pp. 735-740.

[12] J. Noguera and M. Badia, Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling, ACM Transactions on Embedded Computing Systems (TECS), vol. 3, issue 2, pp. 385-406, 2004.

[13] Y. Qu, J.-P. Soininen and J. Nurmi, A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead, in the Conference on Design, Automation and Test in Europe (DATE); March 6-10, 2006; Munich, Germany; pp. 1-6.

[14] K. N. Vikram and V. Vasudevan, Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures, IEEE Transactions on VLSI Systems (TVLSIS), vol. 14, issue 9, pp. 1010-1023, 2006.

[15] W. Fu and K. Compton, An execution environment for reconfigurable computing, in the 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM); April 17-20, 2005; Napa, California, USA; pp. 149-158.

[16] E. Pérez-Ramo, J. Resano, D. Mozos and F. Catthoor, Reducing the reconfiguration overhead: a survey of techniques, in the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA); June 25-28, 2007; Las Vegas, Nevada, USA; pp. 191-194.

[17] S. Sudhir, S. Nath and S. Goldstein, Configuration Caching and Swapping, in 11th International Conference on Field-Programmable Logic and Applications (FPL); August 27-29, 2001; Belfast, Northern Ireland, UK; pp. 192-202.

[18] E.P. Ramo, J. Resano, D. Mozos and F. Catthoor, Memory hierarchy for high-performance and energyaware reconfigurable systems, Computers & Digital Techniques, IET, vol. 1, issue 5, pp. 565-571, 2007.

[19] Z. Li and S. Hauck, Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation, in the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA); February 24-26, 2002; Monterey, California, USA; pp. 187-195.

[20] J. Resano, D. Mozos, D Verkest and F Catthoor, A Reconfiguration Manager for Dynamically Reconfigurable Hardware, IEEE Design&Test, vol. 22, issue 5, pp. 452-460, 2005.

[21] L. Shang and N.K. Jha, Hardware/Software Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs, in the Asia and South Pacific Design Automation Conference (ASP-DAC); January 7-11, 2002; Bangalore, India; pp. 345-360.

[22] Z. Li, K. Compton, S. Hauck, Configuration Caching Management Techniques for Reconfigurable Computing, in the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM); April 17-19, 2000; Napa Valley, California, USA; pp. 22-36.

[23] J. A. Clemente, C. González, J. Resano and D. Mozos, A Hardware Task-Graph Scheduler for Reconfigurable Multi-tasking Systems, in the International Conference on ReConFigurable Computing and FPGAs (ReConFig); December 9-11, 2008; Cancun, Mexico; pp.79-84.

[24] G. B. Wigley and D. A. Kearney, Research Issues in Operating Systems for Reconfigurable Computing, in the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA); June 24-27, 2002; Las Vegas, Nevada, USA; pp. 10-16.

[25] V. Nollet, T. Marescaux, D. Verkest, J.Y. Mignolet and S. Vernalde, Operating System controlled Networkon-Chip, in the 41st Design Automation Conference (DAC); June 7-11, 2004; San Diego, California, USA; pp. 256-259.

[26] H. Kwok-Hay So and R. W. Brodersen, ‚A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH, ACM Transactions on Embedded Computing Systems (TECS), vol. 7, issue 2, pp. 259-264, 2008.

[27] C. Haubelt, D. Koch and J. Teich, Basic OS Support for Distributed Reconfigurable Hardware, in the International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS); July 19-20, 2004; Samos, Greece; Lecture Notes in Computer Science 3133, pp. 30-38.

[28] J.B. Dennis and D.P. Misunas, A Preliminary Architecture for a Basic Data-flow Processor, in the 2nd International Symposium on Computer Architecture (ISCA), December 1974, pp. 126 – 132.

Biographical notes of all authors

Juan Antonio Clemente

Juan Antonio Clemente was born in 1984. He started studying a Computer Science Degree at Universidad Complutense de Madrid (UCM) in 2002 and finished in 2007. Since then he is a PhD student and works there as a teaching assistant and also as a researcher in the GHADIR group. In his work he mainly focuses on developing techniques to hide reconfiguration latencies in dynamically reconfigurable systems. He is also currently collaborating with Carlos González in the development of algorithms dealing with hyperespectral imagery, as well as an efficient implementation of them in a FPGA.

Carlos González

Carlos González was born in 1984. He started studying a Computer Science Degree at Universidad Complutense de Madrid (UCM) in 2002 and finished in 2007. Since then he is a PhD student and since 2008 works there as a teaching assistant. In his work he mainly focuses on Applying run-time reconfiguration in aerospace applications**.** He has recently started with this topic, working with algorithms that deal with hyperspectral images. He is also currently collaborating with Juan Antonio Clemente on developing techniques to hide reconfiguration latencies in dinamically reconfigurable systems.

Javier Resano

Javier Resano received the Bachelor Degree in Physics in 1997, a Master Degree in Computer Science in 1999, and the PhD degree in 2005 at the Universidad Complutense of Madrid, Spain. Currently he is Associate Professor at the Computer Eng. Department of the Universidad of Zaragoza, and he is a member of the GHADIR research group, from Universidad Complutense, and the GAZ research group, from Universidad de Zaragoza. In collaborates with the Digital Design Technology Group from IMEC-laboratory since 2002. His research has been focused in hardware/software co-design, task scheduling techniques, Dynamically Reconfigurable Hardware and FPGA design.

Daniel Mozos

Daniel Mozos is a permanent professor in the Computer Architecture and Automation Department of the Universidad Complutense de Madrid, where he leads the GHADIR research group on dynamically reconfigurable architectures. His research interests include design automation, computer architecture, and reconfigurable computing. Mozos has a B.S. in physics and a Ph.D. in computer science from the Universidad Complutense de Madrid.