Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures*

Juanjo Noguera
Research & Development Dept.
Hewlett-Packard InkJet Commercial Division (ICD)
jnoguera@bpo.hp.com

Rosa M. Badia
Computer Architecture Dept. (DAC)
Technical University of Catalonia (UPC)
rosab@ac.upc.es

ABSTRACT
Dynamic run-time scheduling in System-on-Chip platforms has become recently an active area of research because of the performance and power requirements of new applications. Moreover, dynamically reconfigurable logic (DRL) architectures are an exciting alternative for embedded systems design. However, all previous approaches to DRL multi-context scheduling and HW/SW scheduling for DRL architectures are based on static scheduling techniques. In this paper, we address this problem and present: (1) a dynamic scheduler hardware architecture, and (2) four dynamic run-time scheduling algorithms for DRL-based multi-context platforms. The scheduling algorithms have been integrated in our codesign environment, where a large number of experiments have been carried out. Results demonstrate the benefits of our approach.

Keywords
Dynamic run-time scheduling, reconfigurable architectures.

1. INTRODUCTION
Scheduling the tasks of an embedded system on a "*System-On-Chip (SoC)*" platform is one of the main challenges in HW/SW codesign. A scheduling policy is said to be *static* when tasks are executed in a fixed order determined at compile-time, and *dynamic* when the execution order is decided at run-time. There is a wide range of approaches to static scheduling [2]. However, recently there has been a growing interest in the development of run-time scheduling techniques for platform-based designs [9][14]. This interest is due to several reasons:

- A growing class of embedded systems need to execute *multiple applications concurrently* [9] rather than just a single application. An example is a set-top box application where audio, video and graphics applications run simultaneously. Additionally, these applications may have to be dynamically invoked (i.e. run or stopped at user request) or may have an intrinsic dynamic behavior (e.g. MPEG4) [14].
- Typical scheduling algorithms assume that the task's execution time is the worst-case execution time (WCET) [2]. However, systems designed using WCET estimates could be highly under-utilized. The execution time of a task is rarely deterministic. For instance, it could be "data-dependent" (e.g., run-length encoding of video frames depends on the information within frames). Moreover, the execution time could depend on the available resources, especially when multiple applications share a system.
- Achieving energy-efficient computation is a major challenge in embedded systems design. *Dynamic power management* [3] is a design methodology that dynamically adapts an embedded system to provide the requested services and performance levels with a minimum number of active components. This methodology is based on the idea that not all system components are always required to be in the active state, and peak performance is only required during some time intervals.

Additionally, the importance of having on-chip programmable logic regions in System-on-Chip platforms is becoming increasingly evident. Partitioning an application among software and programmable logic hardware can substantially improve performance, but such partitioning can also improve power consumption by performing computations more effectively and by allowing for longer microprocessor shutdown periods.

In this area of *Reconfigurable Computing* (RC), *Dynamic Reconfiguration* has emerged as a particularly attractive technique to increase the effective use of programmable logic blocks. *Dynamically Reconfigurable Logic* (DRL) devices allow the change of the device configuration *on the fly* during system operation. A clear example is the CS2112 chip from Chameleon Systems, Inc [1]. This device integrates a RISC core, embedded memory, and four run-time reconfigurable logic blocks.

However, this attractive idea of time-multiplexing the needed device configuration does not come for free. The *reconfiguration latency* has to be minimized to improve performance. There are two main approaches to address this challenge.

* This work is funded by CICYT-TIC project TIC2001-2476-CO3-02 and DURSI project 2001SGR00226. Juanjo Noguera acknowledges the support of Hewlett-Packard ICD in the preparation of his PhD thesis.

One of these approaches is known as *temporal partitioning*, in which the system specification must be partitioned into temporal exclusive segments (called *reconfiguration contexts*) [13]. A different approach is to find an execution order for a set of tasks that meets system design objectives (e.g. minimize the execution time), which is usually known as *DRL multi-context scheduling* [10][13].

1.1. Contributions of the paper

All existing approaches to DRL multi-context scheduling are based on static (compile time) scheduling techniques, which assume that tasks have a fixed (deterministic) execution time.

To the best of our knowledge, no previous work has been carried out in order to define a dynamic run-time HW/SW scheduling approach for DRL-based multi-context platforms.

In this paper, we address this open problem and present four dynamic run-time scheduling algorithms for dynamically reconfigurable architectures. Moreover, we present a hardware architecture for the implementation of the dynamic run-time scheduler. This hardware implementation is thought to minimize run-time scheduling overheads.

The paper is organized as follows: Section 2 is an overview of previous work. Section 3 introduces our HW/SW codesign methodology, which is based on a dynamic run-time scheduling strategy. In section 4, we explain the basic architecture of the dynamic run-time scheduler. Section 5 presents four dynamic run-time scheduling algorithms. In section 6, we explain the experiments that we have carried out, and give the obtained results. Finally, section 7 presents the conclusions of this work.

2. PREVIOUS WORK

Software scheduling for real-time embedded systems have been widely covered in the literature. Balarin et al. present a survey of these techniques in [2]. Most of the work related with dynamic scheduling can be classified as *fixed priority* or *dynamic priority* assignment policies. Rate Monotonic Analysis (RMA) is an example of fixed priority dynamic scheduling. Earliest Deadline First (EDF) is an example of dynamic priority assignment policy. EDF offers attractive theoretical improvements over RMA, however EDF is not widely used in embedded systems because of its costly run-time overhead. To the best of our knowledge there is not any approach to DRL multi-context scheduling which uses a fixed priority dynamic scheduling technique.

In the other hand, several references can be found addressing temporal partitioning for reconfiguration latency minimization [13]. Moreover, *configuration prefetching* techniques are used to minimize reconfiguration overhead. They are based on the idea of loading the next reconfiguration context before it is required, hence overlapping device reconfiguration and application execution. Hauck firstly introduced configuration prefetching in [7], where a single-context prefetching technique is presented.

DRL multi-context scheduling has been addressed in many publications [10]. However, all these approaches are based on static (compile-time) scheduling techniques. Moreover, these previous approaches do not address HW/SW scheduling.

In [4] an integrated algorithm for HW/SW partitioning and scheduling, temporal partitioning and context scheduling is presented. This approach is similar to [5] and [8] which address HW/SW scheduling for dynamically reconfigurable devices. However, they are also based on static scheduling algorithms.

3. HW/SW CODESIGN METHODOLOGY

3.1. Definitions

In our approach, we model a single application as a *task graph*. Concurrent and multi-function systems are modeled as a set of several task graphs. A task graph is a directed acyclic graph where each node represents a *task.* Each task is associated with a *task type*. A task represents a coarse grained computation in an embedded system (e.g., loops are examples of tasks). An embedded system may contain more than one task of the same type (e.g. a DCT task may occur in several video applications). Each task has associated a *priority* of execution. This priority is calculated and assigned to each task statically (at compile time).

Tasks are connected using directed edges. Edges represent data dependencies between tasks. Each edge is associated with a scalar denoting the amount of data that must be transferred between the tasks it connects. A task may begin execution only after all its incoming edges have been executed.

Once a task is ready for execution, this is explicitly indicated by an *event*. An event consists of the following information: *TaskId*, *TaskGraphId*[1], *TaskPriority* and *TaskType*. Events are sequentially ordered by the *TaskPriority* field. The list of sorted events is the *Event Stream*.

A *Functional Unit* is a physical component (i.e. DRL device or SW processor) that executes tasks. A DRL device has an active *reconfiguration context*, which in our methodology, is associated to a task type. If it is required to process a new task, which has a different task type from the currently loaded in the DRL device, a *reconfiguration* will be needed.

During the processing of a task, a functional unit can be in several *states* (e.g., execution, reconfiguration, etc.). We also define the *functional unit active set* to include the following information: functional unit state, *TaskId*, *TaskType* and *TaskGraph*, of the task being processed in the functional unit.

3.2. Codesign Methodology

The proposed methodology [11][12] is divided into three stages: *Application Stage*, *Static Stage* and *Dynamic Stage*. The application stage is focused on the system specification.

The *static stage* includes: (1) *extraction*, (2) *estimation*, (3) *HW/SW partitioning*, and (4) *HW and SW synthesis*. The extraction phase has two main objectives: (1) obtain the task graph representation from the system specification and assign to each task a priority of execution, and (2) obtain a list of independent task types found in the task graph(s). The estimation phase can use typical estimators (e.g., delay and area) that can be obtained using high-level synthesis and profiling tools. The HW/SW partitioning phase decides which *task types* will be executed in reconfigurable HW and which in SW.

HW/SW partitioning has been demonstrated to be a critical point when targeting DRL architectures [12]. The dynamic scheduling results highly depend on the quality of the HW/SW partitioning, which helps to reduce the run-time reconfiguration overhead.

The *dynamic stage* includes *HW/SW Scheduling* and *DRL Multi-Context Scheduling*. Both of them run in parallel and base their functionality on events present in the event stream. To better understand how this works, let us explain the target architecture.

[1] *TaskGraphId* identifies the task graph to which the task belongs

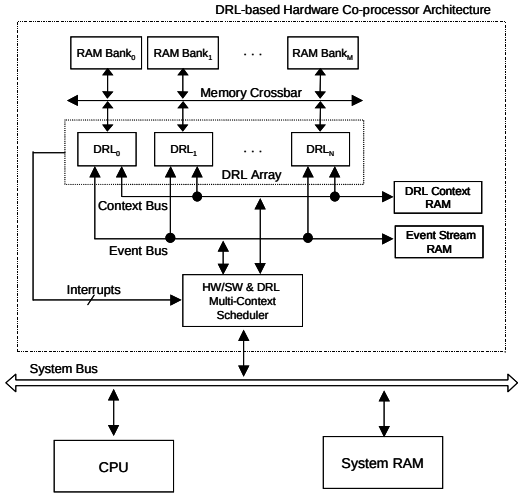

Figure 1. Target Architecture.

The target architecture is depicted in fig. 1. It is a heterogeneous architecture, which comprises a software processor, a DRL-based hardware architecture and shared memory resources. The CPU is a uniprocessing system and it can execute only one task at a time. The HW/SW and DRL Multi-Context Scheduler are mapped to hardware using a centralized control scheme. DRL contexts are stored in the DRL Context memory. The Event Stream is stored in the Event Stream memory.

Events are executed in the *DRL Array* or in the CPU. The data that must be transferred between tasks executed in the DRL Array is stored in the RAM banks. A concrete DRL may access any RAM bank using the *Memory Crossbar*. Several memory accesses to different banks are possible concurrently. A read and write operation are possible concurrently in a single bank.

The HW/SW and DRL schedulers co-operate and run in parallel during application run-time execution, in order to meet system constraints. Their functionality is based on the use of a look-ahead strategy into the event stream memory. *Event Window* (EW) is the number of events that are observed in advance.

At run-time, the HW/SW scheduler assigns events to functional units and decides the execution order of the events stored in the event window. The DRL multi-context scheduler is used to minimize reconfiguration overhead. The objective of the DRL multi-context scheduler is to decide: (1) which DRL must be reconfigured, and (2) which reconfiguration context (task type) must be loaded in the DRL. This scheduler tries to minimize this reconfiguration overhead by overlapping the execution of events with DRL reconfigurations.

4. A SCHEDULER ARCHITECTURE FOR RECONFIGURABLE PLATFORMS

4.1. Dynamic Scheduler Architecture

In this section, we will explain the internal architecture of the centralized scheduler (HW/SW and DRL multi-context). This architecture is shown in figure 2. The proposed architecture is divided in three main parts: (1) The *Dynamic Scheduling Algorithm*, (2) The *Graph Dependence Check Logic*, and (3) The *Event Stream Memory Interface Logic*.

This dynamic scheduler architecture can be seen as two processes that run concurrently and interact using a shared memory. There is a producer process (the *Graph Dependence Check Logic*) and a consumer process (the *Dynamic Scheduling Algorithms*). Both processes produce and consume events, which are stored in the event stream memory (*Event Stream Memory Interface Logic*).

The *Graph Dependence Check Logic* knows the functional unit active set (functional unit state, *TaskId*, *TaskType* and *TaskGraph*) of all DRL's and CPU. It also receives the interrupts signals indicating that a concrete execution has finished.

In case, a concrete DRL or CPU has finished the execution of a task, new tasks may become ready for execution if all its dependences have been completed. This is the main function of the *Graph Dependence Check Logic* block. This module internally has the required data structures to check task dependences. In the next section, this module will be explained in more detail. If new tasks become ready for execution this module generates new events, which are inserted in the event stream memory using *Event Stream Memory Interface Logic*.

The event stream consists of a sorted list of events. Events are sorted by the *TaskPriority* field of the event. This *TaskPriority* field is assigned to each task at compile time. Several priority functions can be used for this objective (e.g., number of output edges of the task, critical path analysis, etc.).

The event window block consumes events from the event stream memory. This will occur at the end of the execution of an event, when a concrete functional unit is available to process a new event. The event with the highest priority within the event stream will be inserted in the event window. Within the event window, there are events being processed and events waiting for execution. All events waiting for execution within the event window are candidates for execution. From all these events, the dynamic scheduling algorithm must select a concrete event for execution. This selection process changes depending on the scheduling algorithm, which may depend on several characteristics of the events (e.g., priority, task type, etc.).

The *Dynamic Scheduling Algorithm* block implements the dynamic run-time HW/SW and DRL multi-context scheduling algorithms. It assigns events to functional units (DRL's or CPU) and specifies the execution order of events present in the event window. In order to implement this functionality it knows the current functional units *active sets*. The scheduling policy depends on the event window size (i.e. number of events which are input to the dynamic scheduling algorithm). In section 5, several dynamic scheduling algorithms are explained.

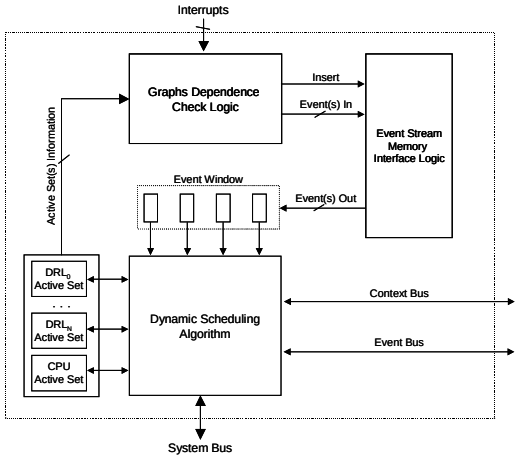

Figure 2. Dynamic Scheduler Architecture

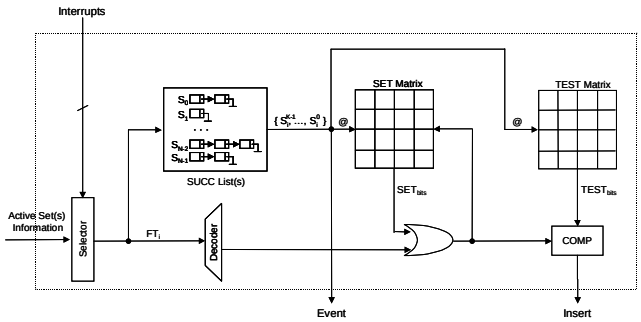

Figure 3. Graph Dependence Check Logic

4.2. Graph Dependence Check Logic

As previously introduced, the main goal of this block is to generate new events, which are inserted in the event stream. These new events are generated at the end of the execution of a task. The architecture of this block is shown in figure 3.

In this architecture, there are three main components: (1) the *Successors List* (SUCC lists), (2) the *SET Matrix*, and (3) the *TEST Matrix*. These three components must be replicated for each task graph. The successors list maintains for each task a list of all its successor tasks with their associated information (*TaskType* and *TaskPriority*). The SET matrix stores, for each task, which ones of its predecessor tasks have finished its execution. Finally, the TEST matrix stores for each task which tasks must be previously executed. This matrix should be initialized before the application begins its execution.

This architecture supports several tasks finishing at the same time. Thus, a selector block decides which finished active set is processed first. The finished task (FT_i) identifier is used as input to the successors list block. These successor tasks are the output of this block, and they are processed sequentially. The following process is repeated for each successor task (S_i^j):

- The successor task id. is used to address the SET matrix in order to know the predecessor tasks that have been executed.
- The read data from the SET matrix is used to update the completed dependences of the task, performing a bit-wise OR function with the decoded finished task identifier.
- This updated information of task's completed dependences is compared with the value read from the TEST matrix. In addition, the SET matrix is updated.
- Finally, if the values read from both matrices are equal, a new event is generated and the *insert* signal is asserted, indicating that the event must be inserted in the event stream.

5. SCHEDULING ALGORITHMS

5.1. Single In-Order Dynamic Scheduling

The first dynamic scheduling algorithms (HW/SW and DRL multi-context scheduling) were presented in [11].

In this approach, a *single* event is being executed on a functional unit (DRL or CPU) at the same time. In addition, it is *in-order* because the scheduling algorithm processes events following the order in which they are consumed from the event stream.

Hardware/Software Scheduling

This algorithm follows a *First-In-First-Out* policy for the scheduling of the events within the event window.

A second objective of the HW/SW scheduler is to manage the functional units active sets (i.e., functional unit state, *TaskId*, *TaskType* and *TaskGraph*). It is important to explain the states required to process an event (see figure 4).

In the figure 4.a., the HW/SW scheduler assigns one event to be processed in a DRL that is in the *idle* state. Depending on the active reconfiguration contexts, a DRL reconfiguration may be initiated. In addition, it is always mandatory to change the active *TaskId* (*task switch* state). Finally, it is possible that a DRL finishes reconfiguration and task switch, but the event cannot be executed because the previous events in the event window have not finished. In this case, the DRL enters into the *wait* state.

Figure 4.b., has a similar functionality for the CPU. The major changes are related with the HW/SW communication. In order to minimize communications overheads, it is possible to start the CPU communication process, while an event is being executed in the DRL array. As in the case of DRL, the CPU has a *wait* state.

DRL Multi-Context Scheduling

However, in order to minimize reconfiguration overheads to the HW/SW scheduler, it is possible to use a *reconfiguration pre-fetching* scheme, which overlaps the reconfiguration of a DRL with the event execution in another DRL. From the reconfiguration contexts that are loaded in the DRL array, and the task types which are required within the event window, the DRL multi-context scheduler decides: (1) which reconfiguration context must be loaded, and (2) in which DRL it will be loaded.

This algorithm is executed at the end of the execution of a concrete event. At that time, a new event starts its execution and a new event enters in the event window. This insertion probably means that a new reconfiguration context will be required.

The basis of the proposed DRL multi-context scheduling algorithm is to obtain an array that represents the required reconfiguration contexts within the event window. This array is obtained from the current state of the DRL's and from the event window. Afterwards, the algorithm obtains from this array the number of DRL contexts that are not required within the event window. If there is not any DRL available for reconfiguration, the algorithm selects (to reconfigure) the DRL that has an active reconfiguration context that will be required latest (remember that events are processed using a FIFO policy). Note that this is not a typical LRU replacement policy. Finally, the first reconfiguration context found in the event window, which is not loaded within the DRL array will be loaded.

5.2. Single Out-of-Order Dynamic Scheduling

As in the previous algorithm, a *single* event is executed on a functional unit at the same time. The main difference of this algorithm with the previous one is that events are executed *out-of-order*. That is, events may be executed in a different order from the one in which they enter in the event window.

Hardware/Software Scheduling

The key point of this approach is the selection (within the event

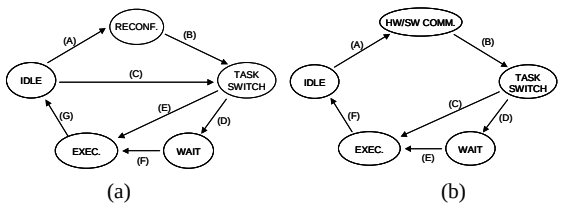

Figure 4. Single Event Execution: DRL and CPU states.

window) of the next event to be executed. In the previous case, the algorithm follows a *First-In-First-Out* policy. However, the previous approach has two main drawbacks:

- It may occur that the next event to be scheduled cannot be executed because it has not finished the DRL reconfiguration and/or task switch. In this case, no useful computation is carried out in any functional unit.
- It may also occur that the DRL array suffers an excessive number of reconfigurations, which indeed means that it spends more time reconfiguring than performing useful computations.

This single out-of-order scheduling algorithm tries to overcome these limitations. This is achieved changing the selection criteria of the next event to be executed. In this new approach, the next event selected for execution will be such that:

(1) There is an active reconfiguration context within the DRL array ready for the execution of an event.

(2) From all the events within the event window that meet the previous condition, select the event with the highest priority.

This selection criteria has as main goal to process consecutively (in the same DRL) events which require the same reconfiguration context. Thus, reconfiguration overhead can be reduced.

In this approach, the used finite state machines for the functional units (fig. 4) and the DRL multi-context scheduler algorithm are the same as the ones described in the previous section.

5.3. Concurrent Dynamic Scheduling

Our target architecture is a multi-processor architecture (fig. 1). Executing a single event at the same time prevents the architecture of achieving high throughput or utilization. This new approach schedules a new event while multiple events can be executing concurrently.

Hardware/Software Scheduling

In this approach, it is important to note that the several states required to process an event have changed. This functionality can be observed in figure 5. The major difference between figures 4 and 5 is that the *wait* state has disappeared. Having a functional unit in the wait state limits the concurrent execution capability.

This algorithm is executed at the end of the execution of an event or when a DRL finishes its reconfiguration process. In this approach, the next event selected for execution will be such that:

(1) There is an active reconfiguration context within the DRL array ready for the execution of an event.

(2) From all the events within the event window that meet the previous condition, select the event with the highest priority.

DRL Multi-Context Scheduling

This algorithm is executed at the end of the execution of a concrete event, if a new event cannot be scheduled for execution by the HW/SW scheduler.

At that time, a new event will enter in the event window, and probably, a new reconfiguration context will be required. This needed reconfiguration context will be loaded in the DRL which is in the *idle* state. It is important to note that no replacement policy is needed in this new approach, because whenever the DRL multi-context scheduler is executed always there will be a DRL available for reconfiguration.

The algorithm also selects a reconfiguration context to be loaded. The reconfiguration context (which is not currently loaded in the DRL array) associated to the highest priority event found in the event window will be selected to be loaded.

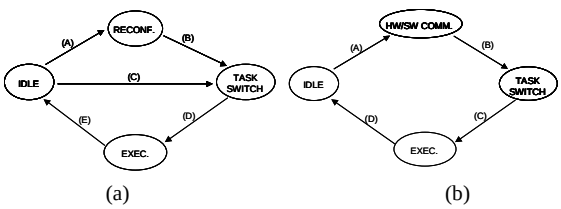

Figure 5. Concurrent Event Execution: DRL and CPU states.

5.4. Dynamic Scheduling with Replication

Due to the previous DRL multi-context scheduler policy, all events being executed use a different reconfiguration context. Thus, in the DRL array there are not two DRL's, which have loaded the same reconfiguration context.

It is possible to find an application in which multiple tasks requiring the same reconfiguration context could be executed concurrently. In our approach, this situation is shown when multiple events, which require the same reconfiguration context, are found in the event window.

Executing this situation using the concurrent dynamic scheduling algorithm has a main drawback: all events requiring the same reconfiguration context will be processed sequentially in the same DRL, while other DRL's may be in the *idle* state.

It is possible to improve the performance of the previous scheduling algorithm by having the same reconfiguration context loaded in several DRL's. This is the objective of the *concurrent dynamic scheduling algorithm with replication*.

In this new approach, the HW/SW scheduler has the same functionality as the one presented in the previous subsection. The DRL multi-context scheduler has been modified (in the selection of the next reconfiguration context to be loaded) to allow the same reconfiguration context to be loaded in several DRL's.

6. EXPERIMENTS AND RESULTS

We have implemented the four explained dynamic run-time scheduling algorithms in our HW/SW codesign framework [11].

Our HW/SW co-simulation tool accepts task graphs generated by TGFF [6]. In these experiments, we used the HW/SW partitioning algorithms proposed in [12]. In order to test the presented scheduling algorithms, we have performed a large number of experiments (more than 3000 simulations).

Important parameters to study and its effect on the scheduling algorithms are: the number of DRL's, its reconfiguration time (and its relation to the tasks' average execution time), the size of the event window (EW) and the used priority function.

With the idea to cover a wide range of applications, we have generated synthetically task graphs using TGFF. We have generated four different test-benches. Each one of these test-benches has three task graphs, and each task graph has in average 25 tasks. The number of task types (in each one of the test-benches) is 5, 10, 15 and 20, respectively.

The number of DRL's used in the experiments is 2, 4 and 8. The reconfiguration time of the DRL's is a value, which is relative to the tasks average execution time. Thus, the used reconfiguration times are 4x, 2x, 1x, ½x and ¼x the tasks' average execution time in a DRL. The event window size is another parameter we have tested. It has been tested for sizes between 1 and 16. Finally, we have used two functions to assign the TaskPriority field of the events. These two functions are: (1) critical path analysis *(cp)*, and (2) number of output edges *(oe)* of the task.

6.1. Obtained Results

Samples of the obtained results are shown in figure 6. The pictures show performance when the event window (EW) increases. The EW size has been found to be one of the key parameters of the dynamic scheduling algorithms. A trade-off must be performed when selecting the EW size. Scheduling algorithms need big EW sizes to perform a better scheduling. As events are inserted in order in the event stream but no in the EW, smaller EW sizes allow to maintain more sorted the event stream.

We cannot compare our techniques to any other approach, since previous approaches to HW/SW scheduling for DRL-based architectures and DRL multi-context scheduling are based on static scheduling techniques.

Figures 6.a. shows the execution time when two DRL's with a 4x reconfiguration time are used. It is possible to observe the results obtained when using both single execution schedulers (*v1* and *v2* respectively) and the concurrent execution scheduler (*v3*) without replication. Moreover, the results of using both priority functions are shown. From figure 6.a., it may be concluded that when few DRL's with slow reconfiguration time (4x) are used, then:

(1) There is a great impact of the priority function in all schedulers. The number of output edges (*oe*) priority function obtains better results than using a critical path analysis function.

(2) The out-of-order and concurrent dynamic schedulers require events windows of large size in order to improve performance.

However, the benefits of the out-of order scheduler compared to the in-order scheduler may be reduced when: (1) the number of DRL increases, or (2) DRL's with a fast reconfiguration time are used. These both conditions mean a perfect overlapping of execution and reconfiguration when using the in-order scheduler.

The results for the concurrent dynamic scheduler are presented in figure 6.b. We can observe the results obtained when using four DRL's. Results for reconfiguration times 4x, 1x and ¼x, are presented and compared to an all HW solution, where no reconfiguration overheads exits (i.e. lower bound). The optimal EW size also depends on the DRL reconfiguration time. When using DRL's with slow reconfiguration time, bigger EW sizes are required. However, if DRL's with fast reconfiguration times are used, the EW size may be reduced.

Finally, figure 6.c. compares the concurrent dynamic scheduling (*v3*) and the dynamic scheduling algorithm with replication (*v4*). In this picture, the results correspond to an architecture with 8 DRL's with different reconfiguration times (4x and ¼x). The replication strategy obtains better results if DRL's with fast reconfiguration time are used. In case that DRL's with slow reconfiguration time are used, the scheduler with replication obtains worst results than the concurrent scheduler.

7. CONCLUSIONS

Dynamic run-time scheduling for SoC platforms has become an important field of research. However, no previous work has been carried out in dynamic DRL multi-context scheduling and dynamic HW/SW scheduling for DRL-based architectures. We have addressed this open problem and we have presented: (1) a dynamic scheduler hardware architecture, and (2) four dynamic scheduling algorithms for DRL multi-context platforms. These algorithms cover a wide range of designs. Out-of-order dynamic scheduling can be applied to low-power designs (*idle* and *wait* states can represent low-power states). Concurrent dynamic scheduling should be applied to designs where both power and performance are critical. Finally, concurrent dynamic scheduling with replication can be used in high-performance designs.

An exhaustive study of these scheduling algorithms has been performed, and the effect of the algorithms parameters has been studied. Results demonstrate the benefits of our approach.

REFERENCES

[1] http://www.chameleonsystems.com/

[2] F. Balarin, *et al.*, "Scheduling for Embedded Real-Time Systems", IEEE Design and Test, Jan-March, 1998.

[3] L. Benini, A. Bogliolo, G. De Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management". IEEE Transactions on VLSI Systems. Vol. 8. Issue 3. June 2000.

[4] K. Chatta, R. Vemuri, "Hardware-Software Codesign for Dynamically Reconfigurable Architectures". Proc. of FPL'99.

[5] R. P. Dick, N. K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems". Proc. of ICCAD'98.

[6] R. P. Dick, D.L. Rhodes, W. Wolf, "TGFF: Task Graphs For Free", in Proc. Int. Workshop Hardware/Software Codesign, Mar. 1998.

[7] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", ACM Int. Symp. on FPGA, 1998.

[8] B. Jeong *et al.*, "Hardware-Software Cosynthesis for Run-Time Incrementally Reconfigurable FPGAs". Proc. ASP-DAC' 2000.

[9] A. Kalavade *et al*., "Software Environment for a Multiprocessor DSP". Proc. of Design Automation Conference (DAC), 1999.

[10] R. Maestre *et al*, "Kernel Scheduling in Reconfigurable Computing", Proc. of DATE'99.

[11] J. Noguera, R. M. Badia, "Run-Time HW/SW Codesign for Discrete Event Systems using Dynamically Reconfigurable Architectures", Proc. of ISSS'2000.

[12] J. Noguera, R. M. Badia, "A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures", Proc. of DATE'2001

[13] K. Purna, D. Bhatia, " Temporal Partitioning and Scheduling Data Flow Graphs for Re-configurable Computers", IEEE Trans. on Computers, vol. 48, No. 6. June 1999.

[14] C. Wong *et al*, "Task Concurrency Management Methodology to Schedule the MPEG4 IM1 Player on a Highly Parallel Processor Platform". Proc. CODES'01.

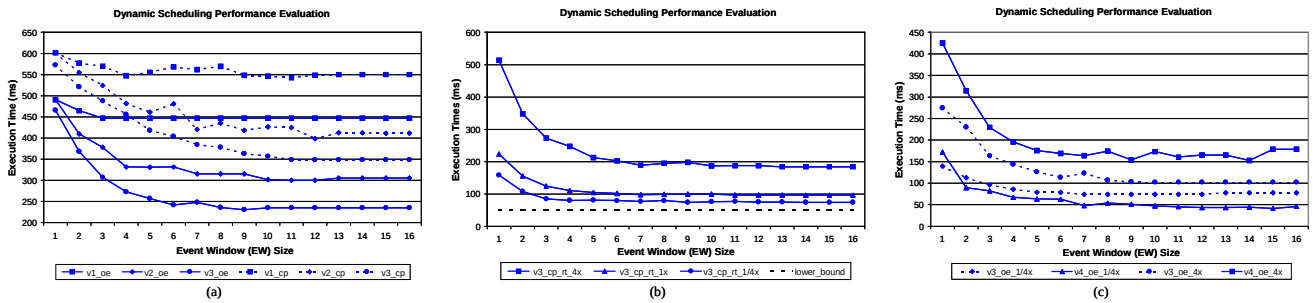

Figure 6. Obtained Results.