HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures

Juanjo Noguera and Rosa M. Badia

***Abstract*—Hardware/software (HW/SW) codesign and reconfigurable computing are commonly used methodologies for digital-systems design. However, no previous work has been carried out in order to define a HW/SW codesign methodology with dynamic scheduling for run-time reconfigurable architectures. In addition, all previous approaches to reconfigurable computing multicontext scheduling are based on static-scheduling techniques. In this paper, we present three main contributions: 1) a novel HW/SW codesign methodology with dynamic scheduling for discrete event systems using dynamically reconfigurable architectures; 2) a new dynamic approach to reconfigurable computing multicontext scheduling; and 3) a HW/SW partitioning algorithm for dynamically reconfigurable architectures. We have developed a whole codesign framework, where we have applied our methodology and algorithms to the case study of software acceleration. An exhaustive study has been carried out, and the obtained results demonstrate the benefits of our approach.**

***Index Terms*—Dynamic scheduling, dynamically reconfigurable architectures, HW/SW codesign, HW/SW partitioning.**

I. Introduction

THE CONTINUED progress of semiconductor technology has enabled the "system-on-chip" (SoC) to become a reality. In this sense, programmable logic manufacturers have also proposed new products. An example of this is the Altera's new device Excalibur, which integrates a processor core (ARM, MIPS, or NIOS), embedded memory and programmable logic [1]. New types of devices, which are run-time reconfigurable, have also been proposed thanks to the advents of the dynamically reconfigurable logic (DRL). An example of this, is the Virtex family from Xilinx, which is partially reconfigurable at run time [2]. A hybrid device, which combines both above explained features, is the CS2112 chip from Chameleon Systems, Inc. This device integrates a RISC core, embedded memory, and a run-time reconfigurable fabric on a single chip [3]. Clearly, all these devices could be used as the final-target architecture of a hardware/software (HW/SW) codesign methodology.

Reconfigurable computing (RC) [16] is an interesting alternative to application specific integrated circuits (ASICs) and the general-purpose processor systems, since it provides the flexibility of software processors and the efficiency and throughput of hardware coprocessors. DRL devices and architectures present new and exciting challenges to the design automation community. The major challenge introduced by DRL devices is the *reconfiguration latency*, which must be minimized in order to maximize application performance. In order to achieve run-time reconfiguration, the system specification (typically, a task graph) must be partitioned into temporal exclusive segments (called *reconfiguration contexts*). This process is usually known as *temporal partitioning* and it is a way to address the problem of reconfiguration latency. A different approach is to find an execution order for a set of tasks that meets system-design objectives (i.e., minimize the application execution time). This is known as *DRL multicontext scheduling*.

There are a great number of approaches to HW/SW codesign of embedded systems, which use different techniques for partitioning and scheduling. However, DRL devices and architectures change many of the basic assumptions in the HW/SW codesign process. The flexibility of dynamic reconfiguration (multiple configurations, partial and run-time reconfiguration, etc.) requires new methodologies and the development of new algorithms, as conventional codesign approaches do not consider the features of these new DRL devices and architectures.

A. Previous Related Work

Traditionally, HW/SW codesign challenges and DRL challenges have been addressed independently.

Earlier approaches to the HW/SW codesign, model the system based on a template of a CPU and an ASIC [11], [14]. HW/SW partitioning and scheduling techniques can be differentiated in several ways. For instance, partitioning can be classified as *fine-grained* (if it partitions the system specification at the basic-block level) or as *coarse-grained* (if system specification is partitioned at the process or task level). Also, HW/SW scheduling can be classified as *static* or *dynamic*. A scheduling policy is said to be static when tasks are executed in a fixed order determined offline, and dynamic when the order of execution is decided online. HW/SW tasks' sequence can change dynamically in complex embedded systems (i.e., control-dominated applications), since such systems often have to operate under many different conditions. Although, there has been a lot of previous work in static HW/SW scheduling, the dynamic scheduling problem in HW/SW codesign has only been addressed in a few research efforts. A strategy for the mixed implementation of dynamic real-time schedulers in HW/SW is presented in [27]. In [4] a review of several approaches to control-dominated and dataflow-dominated software scheduling is presented.

Manuscript received October 30, 2000; revised January 18, 2002. This work was supported by CICYT-TIC project TIC2001-2476-C03-02.

J. Noguera is with Hewlett-Packard Inkjet Commercial Division, Department of Research and Development, San Cugat del Valles, 08190 Spain (e-mail: jnoguera@bpo.hp.com).

R. M. Badia is with the Technical University of Catalonia (DAC-UPC), Computer Architecture Department, Barcelona, 08034 Spain (e-mail: rosab@ac.upc.es).

Digital Object Identifier 10.1109/TVLSI.2002.801575

1063-8210/02$17.00 © 2002 IEEE

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

On the other hand, several approaches can be found in the literature addressing reconfiguration latency minimization. *Configuration prefetching* techniques have been used for reconfiguration latency minimization. They are based on the idea of loading the next reconfiguration context before it is required, hence, overlapping device reconfiguration and application execution. Hauck first introduced configuration prefetching in [15], where a single-context prefetching technique is presented. Also, several references can be found in the literature addressing temporal partitioning and multicontext scheduling for reconfiguration latency minimization. See [24] as an example. All these previous approaches address the problem of reconfiguration latency minimization, but they do not address HW/SW partitioning and scheduling.

Recent research efforts have addressed this open problem. In [7], an integrated algorithm for HW/SW partitioning and scheduling, temporal partitioning, and context scheduling is presented. A more recent work [22] presents a fine-grained HW/SW partitioning algorithm (at loop level). Both previous approaches are similar to [12] which take the reconfiguration time into account when performing the partitioning, but they do not consider the effects of configuration prefetching for latency minimization. In [17], this topic is introduced, and a HW/SW cosynthesis approach for partially reconfigurable devices is presented. They do not address multicontext devices. Moreover, this approach, which is based on an ILP formulation, is limited by the high execution times of the algorithm, which hardly gives solutions for task graphs having more than ten tasks.

B. Motivation and Contributions of the Paper

New approaches are possible because 1) all existing approaches to DRL multicontext scheduling are based on static (compile time) scheduling techniques and 2) no previous work has been carried out in order to define a HW/SW codesign methodology with dynamic scheduling based on DRL architectures.

In this paper, we address these two open problems and present the following: 1) a novel HW/SW codesign methodology with dynamic scheduling for dynamically reconfigurable architectures, 2) a dynamic approach to DRL multicontext scheduling, and 3) an automatic HW/SW partitioning algorithm for DRL architectures. The proposed algorithm takes into account the reconfiguration latency when performing the HW/SW partitioning. The experiments carried out demonstrate that the benefits of using a prefetching technique, for reconfiguration latency minimization, can be improved if it is considered at the HW/SW partitioning level.

The rest of the paper is organized as follows: Section II introduces the HW/SW codesign methodology with dynamic scheduling, and the target architectures (named, local, and shared memory architectures). In Sections III and IV, we present two-different algorithms (HW/SW partitioning and DRL multicontext scheduling) for the shared and local memory architectures. In Section V, we apply our methodology and algorithms to the software acceleration of telecom networks simulation, and present the obtained results. An improved HW/SW partitioning algorithm for DRL architectures is presented in Section VI. Finally, Section VII presents the conclusions of this work.

II. HW/SW Codesign for Discrete Event Systems

Discrete event (DE) systems design has been recently addressed using HW/SW codesign techniques [13], [21]. However, none of these approaches is based on DRL devices as the hardware platform.

The methodology presented here addresses the problem of HW/SW codesign with dynamic scheduling for DE systems using a heterogeneous architecture that contains a standard off-the-shelf processor and a DRL based architecture. It is important to note that the proposed methodology follows an object orientation paradigm, and uses the object oriented concepts in all its steps (specification, HW/SW partitioning and scheduling, etc.). The majority of previous related work only uses this object oriented approach at the system specification (modeling) level. See [37], as an example.

A. Definitions

1) *Discrete event class* is a concurrent process type with a certain behavior, which is specified as a function of the state variables and input events. See Fig. 1(a).
2) *Discrete event object* is a concrete instance of a DE class. Several DE objects from a single DE class are possible. Given two DE objects (DEO_1 and DEO_2) they may differ in the value of their state variables. See Fig. 1(b).
3) *Event* E is a member of $T \times C \times O \times V$ where T is a set of tags, $T \in \Re^+$ (the real numbers), C a given set of DE classes, O a set of DE objects, and V a set of values. Tags are used to model time and values represent operands or results of event computation.
4) *Event stream* (ES) is a list of events sequentially ordered by tag. Tags can represent, for example, event occurrence or event deadline. See Fig. 1(c).
5) *Discrete event functional unit* is a physical component (i.e., DRL device or SW processor) where an event $e = (t, c_1, o_1, v_1)$ can be executed. A functional unit has an active pair (*class, object*), $p = (c_a, o_a)$. See Fig. 1(d).

   Our methodology assumes that: (1) several DE classes could be mapped into a single DE functional unit, and (2) all DE objects from a DE class are mapped into the same DE functional unit where the DE class has been mapped.
6) *Object switch* is the mechanism that allows a DE functional unit to change from one DE object to another, both DE objects belonging to a same DE class. For example, if an input event $e = (t, c_1, o_1, v_1)$ has to be processed in a DE functional unit with an active pair $p = (c_1, o_2)$ then an object switch should be performed. Object switch means a change of values in the state variables from the ones of a concrete DE object (o_1) to the others of another DE object (o_2).
7) *Class switch* is the mechanism that allows a DE functional unit to change from one DE class to another. For example, if an input event $e = (t, c_1, o_1, v_1)$ should be processed in a DE functional unit with an active pair $p = (c_2, o_2)$,

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

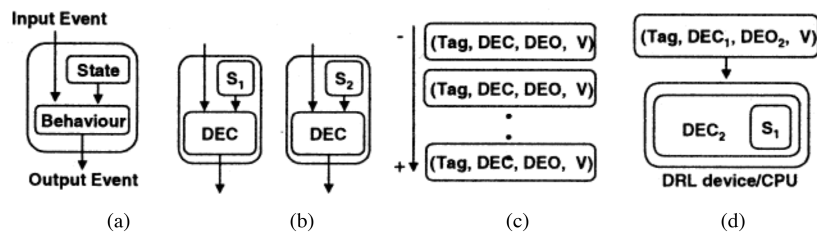


Fig. 1. HW/SW codesign methodology definitions.

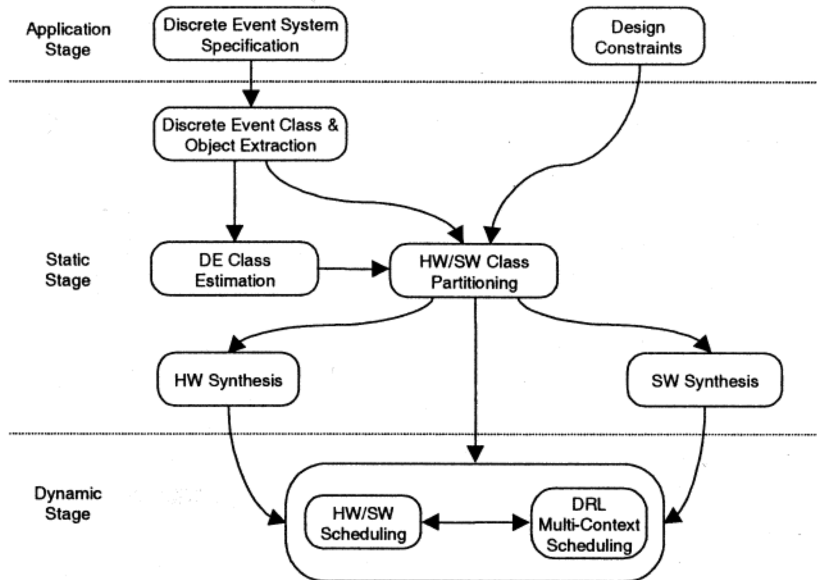


Fig. 2. HW/SW codesign methodology.

then a class switch should be performed. Class switch, in case of a DRL device, means a context reconfiguration.

B. HW/SW Codesign Methodology With Dynamic Scheduling

The methodology we propose is depicted in Fig. 2. It is divided in three stages: *application stage*, *static stage*, and *dynamic stage*. The key points in our methodology are: (1) *application* and *dynamic stages* handle DE classes and objects, and (2) *static stage* only handles DE classes.

The *application stage* includes *discrete event system specification* and *design constraints*. We assume the use of an homogenous modeling language for system specification, where a set of independent DE classes must be first modeled. Afterwards, these DE classes are used to specify the entire system as a set of interrelated DE objects, which communicate among them using events. These DE objects are interrelated creating a concrete topology. A DE object computation is activated upon the arrival of an event. By design constraints we understand any design requirement necessary when synthesizing the design (i.e., timing or area requirements).

The *static stage* includes typical phases of a codesign methodology: (1) *estimation*, (2) *HW/SW partitioning*, (3) *HW and SW synthesis*, and (4) *extraction*.

As previously stated, the *static stage* handles DE classes. However, the system has been specified as a set of interrelated DE objects, which are instances of previously specified DE classes. The final goals of the methodology's extraction phase are, for a given DE class, to obtain 1) a list of all its instances (DE objects) and 2) a list of all different DE classes and objects connected to it. Both lists are afterwards attached to each DE class found in the system specification. Once this step has finished, DE classes can be viewed as a set of independent tasks.

Note that although our methodology, addresses DRL architectures, a temporal partitioning phase can not be found. The DE object/class extraction phase should be viewed as the temporal partitioning algorithm. Indeed, the temporal partitioning algorithm is included within our concept of DE class, as DE classes are functionally independent tasks.

We classify our HW/SW partitioning approach as coarse-grained, since it works at the DE class level. Different HW/SW partitioning algorithms can be applied depending on the application. The solution obtained by the HW/SW partitioning algorithm should meet design constraints.

The estimation phase also deals with DE classes and the used estimators depend on the final application. Typically used estimators (HW/SW execution time, DRL area, etc.) can be obtained using high-level synthesis and profiling tools.

The *dynamic stage* includes *HW/SW Scheduling* and *DRL multicontext Scheduling*. Both schedulers base their functionality on the events present in the event stream. Our methodology assumes that both of them are implemented in hardware using a centralized control scheme. As it is shown in Fig. 2, these scheduling policies (HW/SW and DRL) cooperate and run in parallel during application run-time execution, in order to

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

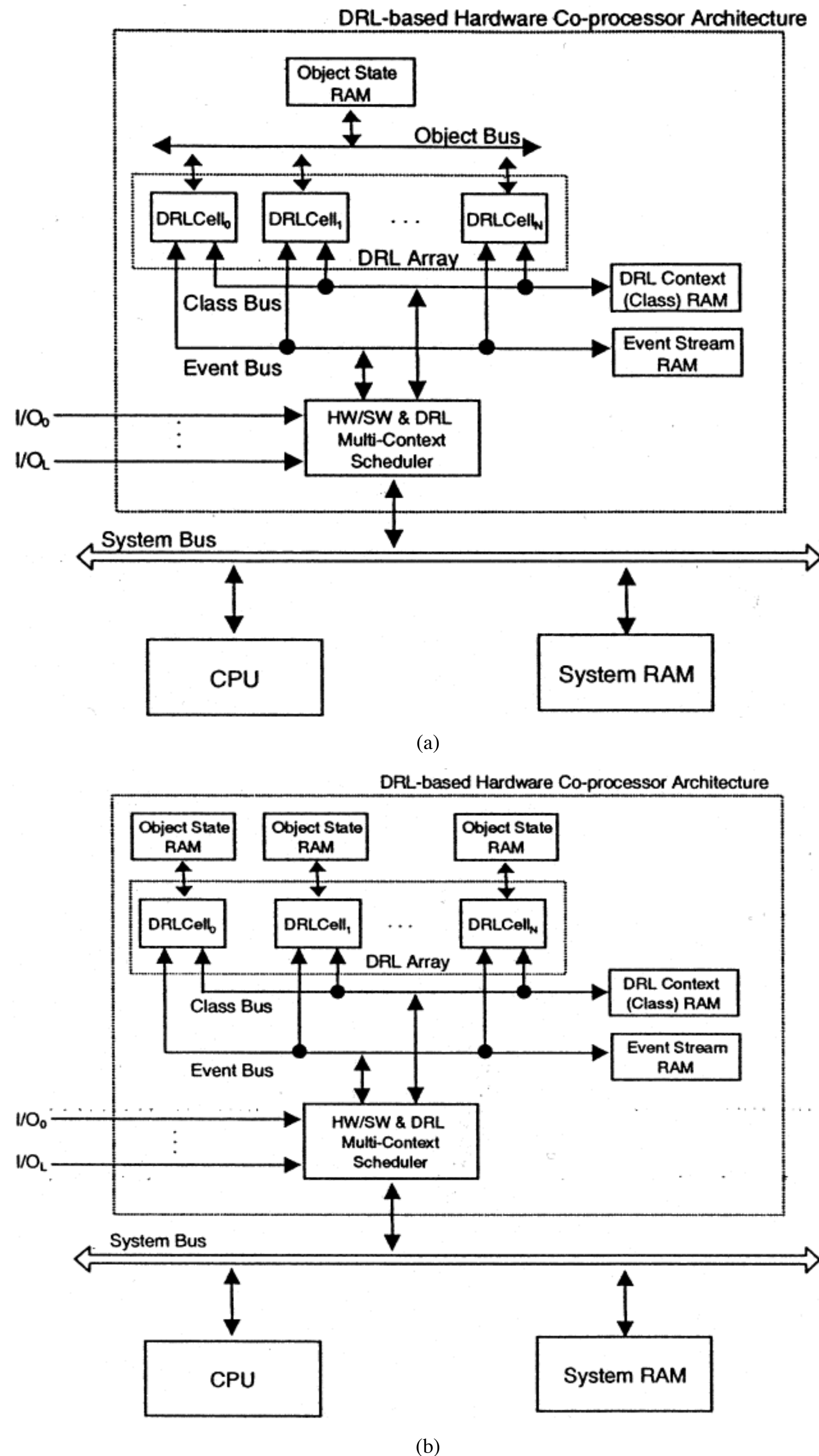


Fig. 3. DRL target architectures.

meet design constraints. With this goal, application execution time is minimized by parallelizing event executions with DRL reconfigurations.

The aim of the HW/SW scheduler is to decide at run time the execution order of the events stored in the event stream. Several policies could be implemented by the HW/SW scheduler based on the application requirements (i.e., earliest deadline first, use or not of a pre-emptive technique).

On the other hand, the DRL multicontext scheduler should be viewed as a tool used by the HW/SW scheduler. A tool in the sense that its goal is to facilitate or minimize the class switching mechanism to the HW/SW scheduler. As in the HW/SW partitioning and scheduling, we assume that different DRL schedulers can be defined depending on the application.

C. Target Architectures

The methodology we propose can be mapped to a target architecture with two variants: the shared memory architecture or the local memory architecture.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

1) Shared Memory Target Architecture: The shared memory target architecture is depicted in Fig. 3(a). This architecture includes a software processor, a DRL-based hardware coprocessor and shared memory resources.

The software processor is a uniprocessing system, which can only execute one event at the same time. The DRL-based coprocessor can execute multiple events concurrently. HW/SW cooperate (interact) via a DMA based memory-sharing mechanism.

The DRL-based coprocessor architecture is divided in: 1) HW/SW and DRL multicontext scheduler; 2) DRL array; 3) object state memory; 4) DRL context memory; and (5) event-stream memory. The HW/SW and DRL multicontext schedulers must implement functions associated to the dynamic stage of our methodology, as previously explained. Events get the central scheduler through I/O ports or as a result of a previous event computation. The event stream is stored in the event stream memory. DRL contexts (which correspond to several DE classes from an application) are stored in the DRL context memory, and they can be loaded to any DRL cell using the class bus. Finally, DE objects states (state variables) are stored in the object state memory. Any DRL cell using the object bus can access the object state variables. The proposed DRL coprocessor architecture is scalable and it is possible to apply any associative mapping between DE objects/classes and DRL cells.

The DRL array communicates with these memories and the central scheduler through several and functionally independent busses (object, class, and event busses). We assume that each DRL array element, named *DRL cell*, can implement any DE class with a required area $\cong$ 20 K gates.

2) Local Memory Target Architecture: Another possibility for the target architecture is to assume that the object state memory is distributed or local to each DRL cell, all the rest remaining as described in the Section I-C1 [see Fig. 3(b)]. As previously stated, with a shared object state memory, any associative mapping between DE objects/classes and DRL cells can be implemented. Instead, if a local object state memory is considered, the mapping between DE classes/objects and DRL cells should be direct, as the state variables for each object would be local to a concrete DRL cell. Both target architectures influence the development of the HW/SW partitioning algorithm and the scheduling algorithms (HW/SW and DRL multicontext).

III. Algorithms for the Shared Memory Target Architecture

In this section, we present two algorithms for the shared memory architecture: 1) a resource constrained HW/SW partitioning algorithm and (2) a dynamic DRL multicontext scheduling algorithm.

A. HW/SW Partitioning Algorithm

1) Problem Statement: A set of independent DE classes $C = (C_1, C_2, \ldots, C_L)$ is the input to the HW/SW partitioning algorithm, which throughout its execution will work with two subsets (C^{HW} and C^{SW}).

```
ListBasedPartitioning_ArchShared(DE_Classes) {
 P_SW = { ∅ };   P_HW = { ∅ };
 P_INITIAL = Sort_DE_Classes_List (DE_Classes, F_SORT);
 for i = 1 to L loop
   C_i = GetFirst(P_INITIAL);
   if C_i.DRL_RequiredArea > DRL_Area then
      P_SW = P_SW  U  C_i;
   else
      if AvailableResources(C_i) then
             P_HW = P_HW  U  C_i;
      else
             P_SW = P_SW  U  C_i;
      end if;
   end if;
 end loop;
}
```

Fig. 4. List-based HW/SW partitioning algorithm for the shared memory architecture.

- C^{HW} is the subset of DE classes mapped to hardware, $C^{HW} = \left(C_1^{HW}, C_2^{HW}, \ldots, C_M^{HW}\right), C^{HW} \subseteq C$.
- C^{SW} is the subset of DE classes mapped to software, $C^{SW} = \left(C_1^{SW}, C_2^{SW}, \ldots, C_K^{SW}\right), C^{SW} \subseteq C$.
- $C^{HW} \bigcap C^{SW} = \emptyset, C^{HW} \bigcup C^{SW} = C$.

A concrete class (C_i) of the input set of classes, is characterized by a set of estimators E_i

$$E_i = (\mathrm{AET}_i^{\mathrm{HW}}, \mathrm{AET}_i^{\mathrm{SW}}, \mathrm{SVM}_i, \mathrm{DRLA}_i, \mathrm{NO}_i) \qquad (1)$$

where

$\mathrm{AET}_i^{\mathrm{HW}}$	average execution time for a hardware implementation of the class C_i;
$\mathrm{AET}_i^{\mathrm{SW}}$	average execution time for a software implementation of the class C_i;
SVM_i	state variables memory size required by the class;
DRLA_i	DRL required area for the class;
NO_i	number of objects of this class.

Let us also consider that design constraints are: (1) object state memory, (2) class (DRL context) memory, and (3) the DRL cell area. The total object state memory is denoted by OSMA (object state memory available). DRLA stands for the DRL cell available Area.

We state our problem as maximizing the number of DE classes mapped to the DRL architecture while meeting memory resources and DRL cell available area constraints

$$\mathrm{Max}\left(|C^{HW}|\right), \text{s.t.} \sum_{j=1}^{M} \mathrm{SVM}_j < \mathrm{OSMA}, \mathrm{DRLA}_j \leq \mathrm{DRLA}. \qquad (2)$$

2) HW/SW Partitioning Algorithm: The proposed HW/SW algorithm is a list-based partitioning algorithm. The algorithm maps more time consuming DE classes to hardware. Thus, the set of input DE classes must be sequentially ordered and more time consuming DE classes should be prioritized when mapping to hardware. This objective is implemented using a cost function. For this example, we propose the following cost function, although other cost functions could be applied

$$F_i = \alpha \cdot \left(\mathrm{AET}_i^{HW} - \mathrm{AET}_i^{SW}\right) + \beta \cdot \mathrm{SVM}_i. \qquad (3)$$

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

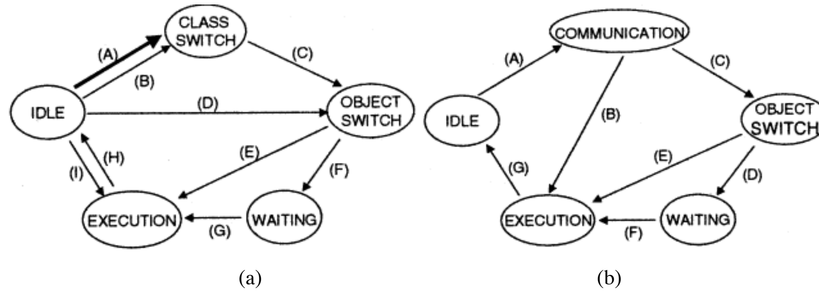


Fig. 5. DRL cell and CPU possible states.

Indeed, this cost function prioritized DE classes with significant difference in its HW and SW execution times. We assume that lower values, as a result of applying this cost function, are better than higher values. So, our sort function classifies values from lowest to highest.

The pseudocode of the proposed HW/SW partitioning algorithm is shown in Fig. 4. It obtains the initial sequentially ordered list (P_{INITIAL}) after the cost function has been applied to all DE classes. Afterwards, the algorithm performs a loop and tries to map as many DE classes to hardware as possible, while memory and DRL area constraints are met.

Available Resources (DeClass C_i) function checks that the current hardware partition plus DE class C_i complies with memory constraints. Function *GetFirst (List P_{INITIAL})* returns and extracts the first DE class C_i from the initial ordered list.

B. Dynamic DRL Architecture Management

In this section, we present the dynamic architecture management, and concretely, a dynamic event-driven DRL multicontext scheduler for the shared memory target architecture. We assume that only the first event of the event stream, which is sorted by the shortest tag, is being processed on a DE functional unit (DRL cell or CPU) at the same time. That is, the HW/SW scheduler only schedules one event at the same time, which indeed is its main objective. Modifications of this scheduler are possible in order to have several events being processed in parallel.

A second objective of the HW/SW scheduler is to manage the active objects and classes of the DRL cells and CPU, performing the class and object switches required to execute an event. This functionality can be observed in Fig. 5, which represents the finite state machines for the DRL cells, and the CPU. As introduced in Section II-B, the HW/SW scheduler is implemented using a centralized control scheme, which means that this scheduler controls all DRL cells and CPU state transitions, as shown in Fig. 5.

Let us explain Fig. 5(a). We can observe that initially a DRL cell is in the *idle* state, which represents that the DRL has finished with the execution of an event, and it is available to process a new event [see edge H, Fig. 5(a)]. Then, the HW/SW scheduler selects (following the previous commented policy) an event to execute in this available DRL cell. To schedule an event can mean to perform different tasks: 1) to enter into the *class switch* state, if the event class is not loaded into the DRL cell (edge B); 2) to enter into the *object switch* state, if the event object is not loaded into the DRL cell (edge D); and/or 3) to enter into the *execution* state if the DRL cell has already active the required

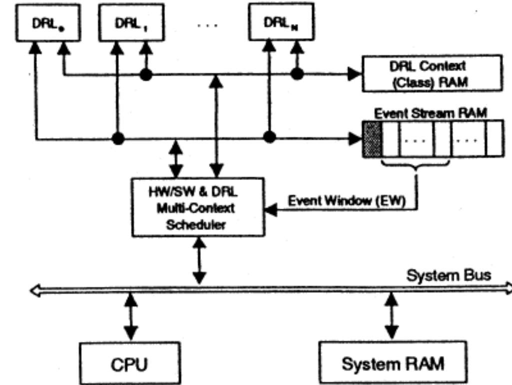


Fig. 6. HW/SW and DRL dynamic scheduling.

```
DynamicDRLSchedulingAlgorithm (EW) {
 ObtainDRLArrayUtilization(EW);
 K = NumberOfAvailableDRL();
 if K = 0 then
    DRLCell = GetLatestRequiredClass();
    Class = GetFirstClassNotInDRLArray(EW);
    if GetTag(Class) < GetTag(DRLCell) then
       DRL_Behaviour(DRLCell, Class);
    end if;
 else
    Class = CE = GetCurrentEvent();
    while (K > 0 and Class ≤ CE + EW) loop
       if ActiveClass(Class) = FALSE then
         DRLCell = GetFirstAvailableDRL(Class);
         DRL_Behaviour(DRLCell, Class);
         K = K - 1;
       end if;
       Class = Class + 1;
    end loop;
 end if;
}
```

Fig. 7. Dynamic DRL multicontext scheduling.

class and object (edge I). The *class switch*, *object switch*, and *execution* states are characterized by a processing time, which is known at compile time. That is, when a DRL cell enters into one of these three states, the time that the DRL cell will remain in the state is fixed. Whenever, one of these states has finished its execution, there is a change of the active state. For example, if the active state is *class switch*, the next active state would become *object switch* (edge C). The same idea is applied to edge E.

However, in order to minimize class switching (DRL reconfiguration) overheads to the HW/SW scheduler and improve the total application execution time, it is possible to start loading

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

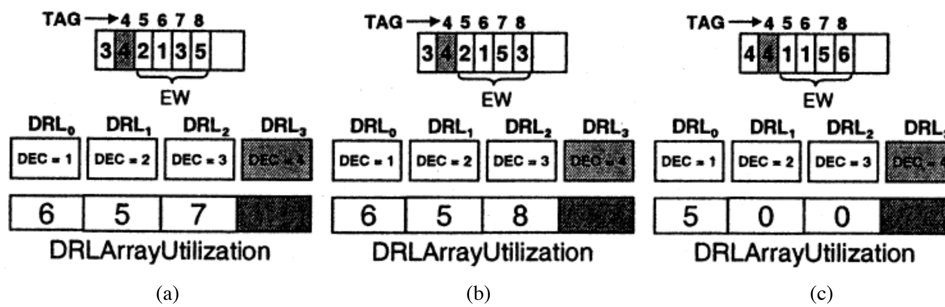


Fig. 8. DRL multicontext scheduling examples.

the required DE class into the DRL array before it is actually required by the HW/SW scheduler (edge A). This is the aim of the DRL multicontext scheduler, which will be explained in Section III-B1. This scheduling policy is based on a look-ahead strategy into the event stream memory (see Fig. 6). Event window (EW) describes the number of events that are observed in advance and is left as a parameter of our scheduler.

By introducing this class switch (reconfiguration) prefetching mechanism, it is possible to overlap the reconfiguration of a DRL cell with the event execution in another DRL cell. Using this approach, it is possible that a given DRL cell finishes with class and object switch, but the event can not be processed because the execution of the previous events in the event stream (with shortest tags) has not finished. In this case, the DRL cell enters into the *waiting* state (edge F). Finally, the DRL cell will exit this state and enter into the *execution* state (edge G), when ordered by the HW/SW scheduler. The time that a DRL cell will remain in the *waiting* state is not fixed, and it will be determined at run time, according to the dynamic behavior of the event stream.

Following, Fig. 5(b) is explained. This figure depicts the finite state machine for the CPU. As in the previous case, initially the CPU is in the *idle* state. Whenever, a concrete event must be processed by the CPU, the HW/SW scheduler will start a communication process with the CPU using the system bus (edge A). Indeed, this *communication* state means to send to the CPU, the event to be processed. The CPU will perform a class switch if it is required. If an object switch is required, the CPU will enter in this state (edge C), and if not it will enter in the *execution* state (edge B). Moreover, and with the idea of minimizing communications overheads, it is possible to start the CPU communication process, while an event is being executed in the DRL array. So, the HW/SW scheduler will start the communication process using the event window concept. In the same manner as in the case of a DRL cell, the CPU has a *waiting* state.

1) DRL Multicontext Scheduling Algorithm: The aim of the DRL multicontext scheduler is to minimize class switching (DRL reconfiguration) overheads to the HW/SW scheduler, in order to minimize the application execution time. From the DE classes that are loaded in the DRL array, and the DE classes which will be required within the event window (EW), the DRL scheduler must decide, 1) which DE class must be loaded and (2) in which DRL cell it will be loaded.

The pseudocode for the dynamic DRL multicontext scheduling algorithm is shown in Fig. 7. As stated, this scheduler depends on the size of the event window. This algorithm is executed at the end of the processing of a concrete event, but concurrently with the execution of the next event. That is, when an event finishes its execution, a new event will start to be processed. As a consequence, the event window will be moved to the next position, and it is probable that new classes are needed.

The basis of the behavior of the proposed DRL multicontext scheduling algorithm is the use of the array *DRLArrayUtilization*, which represents the expected active DE classes and associated tags of the DRL array within the event window. This array is obtained from the current state of the DRL array and the event window, using the function *ObtainDRLArrayUtilization*.

Afterwards, the algorithm calculates the number of DRL cells that will not be required within the event window (variable K). These K DRL cells (if there is any) are available for a class (context) switch. So, this is the first condition that the algorithm checks.

If there is not any DRL cell available for a class switch, the algorithm selects (to reconfigure) the DRL cell that has an active DE class that will be required latest. Note that this is not a typical last recently used (LRU) replacement policy. The algorithm also selects a DE class to be loaded. The first DE class found in the event stream, which is not loaded within the DRL array will be selected. Finally, it will check that the event tag (associated with the first class not present in the DRL array) is lower than the DRL tag (the tag at which the loaded class into the DRL cell will be required). We assume that all tags are different, although this difference could be small. If this condition is asserted, the algorithm sets the DRL cell into the *class switch* state using the function *DRL_Behavior( )*.

On the other hand, if there are K DRL cells available for a class switch, the algorithm enters into a loop that goes through the entire event window (beginning from the current event, *CE*). If it finds a class (associated with an event), which is not loaded within the DRL array, the algorithm selects the first available DRL cell to reconfigure.

Fig. 8 shows three different possible cases for the DRL multicontext scheduler, and how does it work. In these examples, it is depicted: (1) the event stream (with associated classes and tags) and the current processed event, which is shadowed, (2) the DRL array, and for each DRL cell, the current loaded class; the shadowed DRL cell means that it is executing the current event, and (3) the array used by the DRL multicontext scheduler, *DRLArrayUtilization*.

In Fig. 8(a), it is possible to observe that all loaded classes will be required within the event window. Thus, the *DRLArrayUtilization* shows, for each DRL cell, at which tag it will

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

```
ListBasedPartitioning_ArchLocal(DE_Classes) {
  P_SW = { Ø };  P_HW = { Ø };
  P_INITIAL = Sort_DE_Classes_List (DE_Classes, F_SORT);
  ClassAssigned = FALSE;  LastAssignedDRLCell = 0;
  for i = 1 to L loop
    C_i = GetFirst(P_INITIAL);
    if C_i.DRL_RequiredArea > DRL_Area then
      P_SW = P_SW  U  C_i;
    else
      j = 0;
      while (j < NumberDRLcells and ClassAssigned = FALSE) loop
        k = (j + LastAssignedDRLCell + 1) mod (NumberDRLCells + 1);
        if AvailableResources(C_i, DRLCell_k) then
              P_HW = P_HW  U  C_i;
          LastAssignedDRLCell = k;  ClassAssigned = TRUE;
        end if;
        j++;
      end loop;
      if ClassAssigned = FALSE then
        P_SW = P_SW  U  C_i;
      end if;
    end if;
  end loop;
}
```

Fig. 9. List-based HW/SW partitioning algorithm for the local memory architecture.

be required. For example, it is observed that DRL_1 has loaded DE class 2, and the position of the *DRLArrayUtilization* associated to DRL_1 has a value of 5, which means that DRL_1 with active DE class 2, will be required at tag 5. Fig. 8(a). represents the case in which $K = 0$ because all loaded classes in the DRL array are required within the event window. In this case, the algorithm will select DRL_2 to reconfigure (it is the one which will be required latest, and DE class 5 to be loaded (it is the first class in the event stream which is not active). However, no reconfiguration will be performed because the tag of event with DE class 5 is greater than the tag at which the DRL_2 will be required. Fig. 8(b). represents a similar case, in which $K = 0$. The difference between this and the previous case, is that in this case, the reconfiguration will be performed because the tag of event with DE class 5 is smaller than the tag at which the DRL_2 will be required. Finally, Fig. 8(c). represents the case in which DRL_1 and DRL_2 are not required within the event window, so $K > 0$. In this case, several reconfigurations can be performed and DE classes 5 and 6 will be loaded.

IV. Algorithms for the Local Memory Target Architecture

In this section we present a HW/SW partitioning algorithm and a DRL multicontext scheduling algorithm for the local memory target architecture.

A. HW/SW Partitioning Algorithm

As we have explained, when the local memory target architecture is assumed, a direct mapping of DE classes to DRL cells has to be used. This section presents a HW/SW partitioning algorithm, which as in the previous considered architecture, is list based. As in the previous case, more time consuming DE classes are mapped to hardware. Moreover, this algorithm not only decides the classes that will be executed in hardware, but also decides in which DRL cell will always be executed the events of each class.

Fig. 9 shows the proposed algorithm. Initially, it sorts the list of DE classes using the same sort function (F_{SORT}) as in Section III. Afterwards, it performs a loop where for each DE class it is decided whether the class is mapped to hardware or software. If it is mapped to hardware, the class is assigned to a concrete DRL cell. The algorithm assigns DE classes to DRL cells in a cyclic way, but it should be checked that there is enough memory in the local object state memory to host the new DE class. This is performed by the function *AvailableResources(DEClass* C_i, *DRLCell* $DRLCell_k$), which checks that:

- There is enough DRL context memory to store the contexts of the classes of P_{HW} plus C_i.
- There is enough local Object State Memory to store the state of classes assigned to $DRLCell_k$ plus C_i

B. Dynamic DRL multicontext Scheduler

In this section, we present a dynamic event-driven DRL multicontext scheduler for the local memory target architecture. The basic ideas and assumptions explained in the case of the DRL multicontext scheduler for the shared memory architecture are also valid in this case. The DRL cell behavior is also the same in this scheduler [see Fig. 5(a)].

The only difference between this scheduler and the one presented for the shared memory architecture, is the mapping between classes/objects and DRL cells. In this case, the mapping is direct and decided (fixed) at compile-time within the HW/SW partitioning algorithm. So, this DRL multicontext scheduler has as input a table that contains, for all DE classes found in the system specification, the DRL cell where a concrete DE class has to be executed.

The algorithm sequentially obtains the required classes (from the events found within the event stream), and for each class it checks if the destination DRL cell (obtained from the DRL/DE class mapping table) is available or not to perform a class switch.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

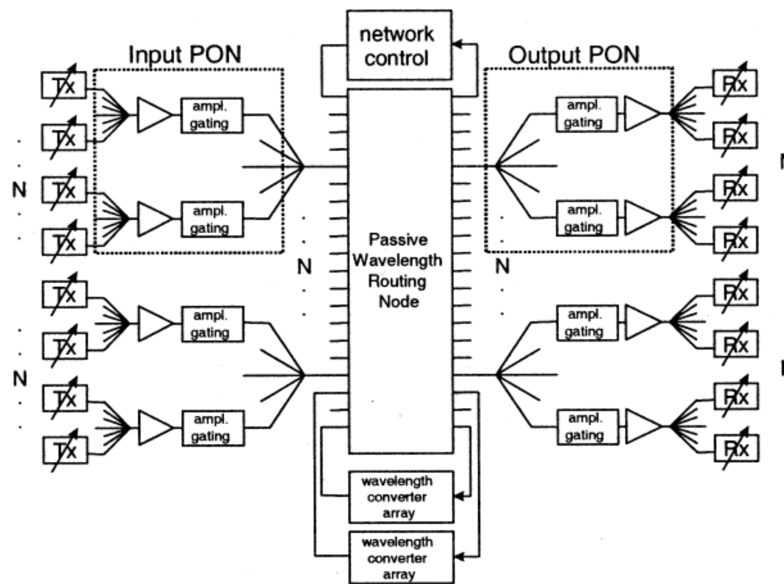


Fig. 10. SONATA network architecture.

TABLE I
DE CLASSES AND DE OBJECTS FOR THE SONATA EXAMPLE

Id.	DE Class	No. of DE objects of the class
DEC_0	Tx	N×N
DEC_1	Input PON	N
DEC_2	Network control	1
DEC_3	Passive Wavelength Routing Node	1
DEC_4	Output PON	N
DEC_5	Rx	N×N

V. A CASE STUDY: TELECOM NETWORKS SIMULATION

It is widely accepted that *software acceleration* is an important field which hardware/software codesign can address. An example of this can be found in [10]. We explain a case study of software acceleration of broad-band telecom networks simulation, which is a challenging application. Parallel computing [5] and reconfigurable computing techniques [26], [29], [33] can be used for simulation execution time improvement.

A. Introduction and Simulation Model

For our case study, we have used the SONATA[1] network [6]. It is a network based on the switchless network concept, which uses a combination of wavelength division multiple access (WDMA) and time division multiple access (TDMA) methods (see Fig. 10). Note that the proposed simulation model depends on a parameter N. This parameter will be used afterwards in order to perform several experiments to test and obtain results from applying our methodology.

The key point of this case study is how to apply the proposed methodology to the simulation of broad-band telecom networks. Specially important, is the mapping between network elements (found in the network model), and DE objects and classes which are the basic elements used in our methodology. From Fig. 10, as an example, we can affirm that there are network elements, which are instances from certain network element types. For example, from Fig. 10 it is possible to find seven different network element types: Tx, Rx, *network control*, *passive wavelength router*, etc. In this sense, these network element types should be viewed as DE classes within the scope of our methodology. In the same way, network elements should be viewed as DE objects. For this case study, we will not consider the *wavelength converter array* network element. In this case study we assume to have six different network element types (see Table I).

B. Developed Codesign Framework

In order to test our proposed methodology and algorithms, we have implemented a whole codesign framework, which is depicted in Fig. 11. In the proposed methodology, DRL target architectures, and dynamic multicontext schedulers, there are several parameters that do not have a fixed value. For example: 1) the number of DRL cells within the target architecture and its reconfiguration time and 2) the size of the event window used by the DRL multicontext schedulers. Moreover, the simulation model depends on parameter N, too. We have developed a codesign framework to study the effects of these parameters in our proposals. We have implemented two different tools: 1) a HW/SW partitioning tool and 2) a HW/SW cosimulation tool. Within both tools we have implemented the algorithms described in this paper, but new algorithms can be easily

[1]*Switchless Optical Network for Advanced Transport Architecture* is partially funded by the European Commission under ACTS program.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

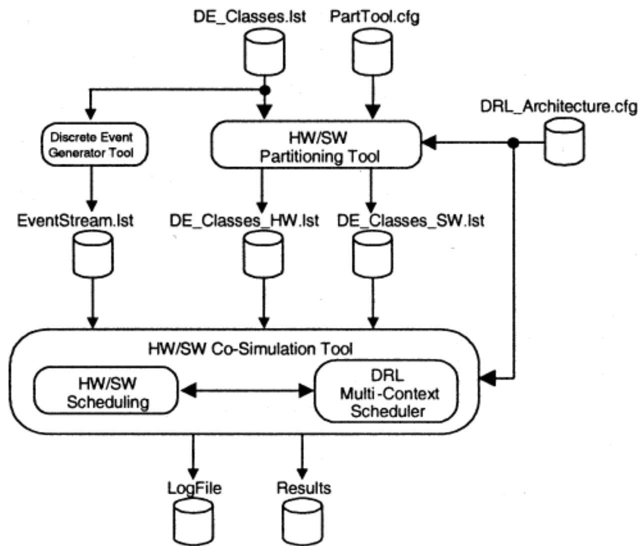


Fig. 11. Developed codesign framework.

included in these tools, as they have been implemented in a modular manner.

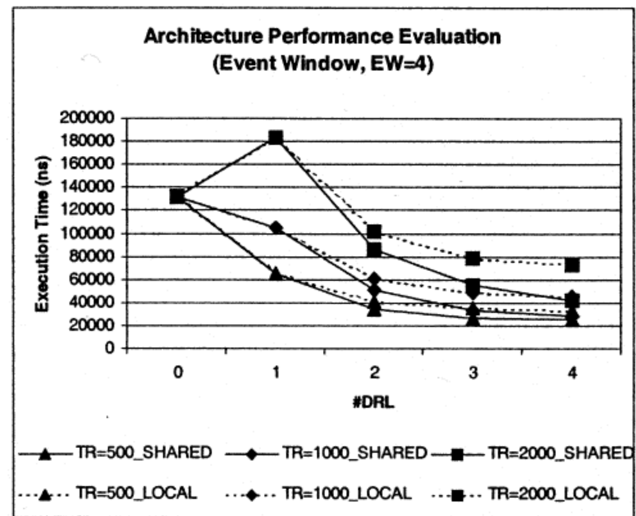
In the developed framework, all parameters can be fixed using configuration files. File *DRL_Architecture.cfg* is used to setup parameters like the number of DRL cells, their reconfiguration time, and the size of the event window used by the DRL multicontext scheduler. In the file *PartTool.cfg*, it is specified the cost function, and the parameters that HW/SW partitioning algorithm should use. The developed codesign framework assumes that the methodology's estimation and extraction phases have already been performed. So, a set of independent DE classes with its estimators (file *DE_Classes.lst*), is the input to the HW/SW partitioning tool.

The several network elements found in the SONATA network, were modeled using the telecommunication description language TeD [5]. TeD'simulator runs on top of a parallel computer, and we have performed real simulations of our SONATA model in order to obtain real simulation event traces (event stream). Afterwards, these event traces were adapted (using a DE generator tool) to be an input to our cosimulation tool. This tool is responsible for implementing the described dynamic stage of our methodology.

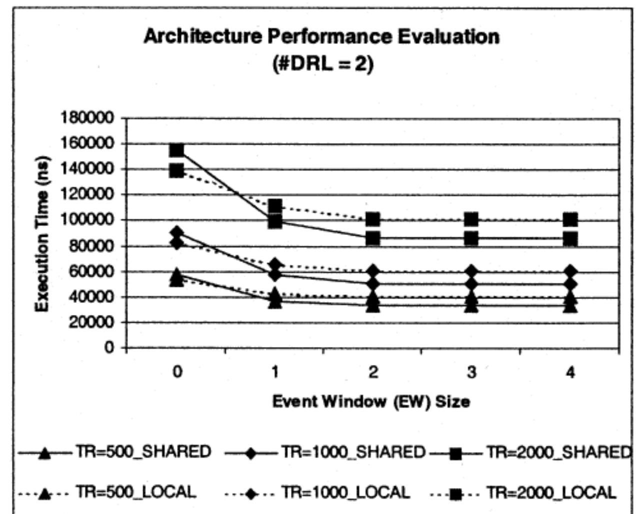
C. Experiments and Results

We carried out several experiments on top of this framework. Two groups of experiments, group I and II, have been performed varying the parameter N found in the SONATA network simulation model (see Fig. 10). Once fixed this parameter, several experiments have been performed varying the DRL architectures and its parameters (file *DRL_Architecture.cfg*).

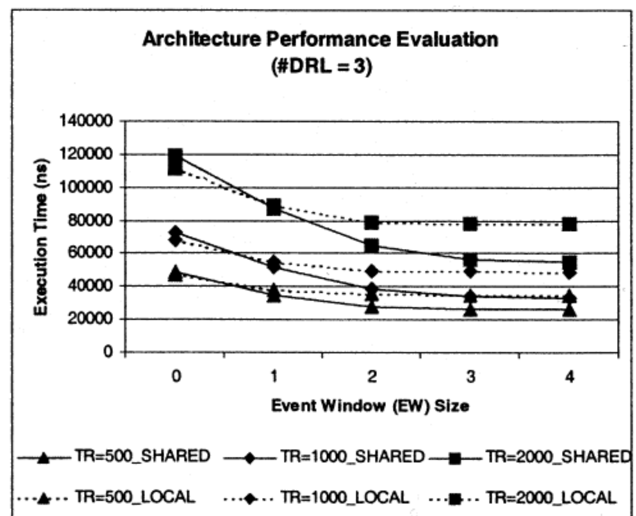
We set $N = 100$ for experiments of group I, and $N = 150$ for experiments in group II. Given these values the HW/SW partitioner for experiments of group I maps all DE classes to hardware, and for experiments of group II the HW/SW partitioner maps four DE classes to hardware and two DE classes to software. Both groups of experiments have been performed on top of the local and shared memory architectures.



(a)



(b)



(c)

Fig. 12. Architecture performance evaluation results.

Results for group I experiments are shown in Fig. 12. This shows three different reconfiguration times: 2000 ns, 1000 ns,

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

TABLE II
NUMBER OF RECONFIGURATIONS (TOTAL, HW/SW SCHEDULER, DRL MULTICONTEXT SCHEDULER) AND NUMBER OF TIMES THE *WAITING* IS REACHED

Architecture		DRL = 2					DRL = 3				
	EW	0	1	2	3	4	0	1	2	3	4
SHARED MEMORY ARCH.	$T_R = 2000$	65	63	63	63	63	47	48	50	50	50
		65	1	1	1	1	47	1	1	1	1
		0	62	62	62	62	0	47	49	49	49
		0	11	10	10	10	0	8	9	9	14
	$T_R = 1000$	65	63	63	63	63	47	48	50	50	50
		65	1	1	1	1	47	1	1	1	1
		0	62	62	62	62	0	47	49	49	49
		0	11	10	10	10	0	8	9	16	20
	$T_R = 500$	65	63	63	63	63	47	48	50	50	50
		65	1	1	1	1	47	1	1	1	1
		0	62	62	62	62	0	47	49	49	49
		0	11	17	17	17	0	8	22	33	36
LOCAL MEMORY ARCH.	$T_R = 2000$	57	57	57	57	57	43	43	43	43	43
		57	23	23	23	23	43	16	16	16	16
		0	34	34	34	34	0	27	27	27	27
		0	8	12	12	12	0	7	11	12	12
	$T_R = 1000$	57	57	57	57	57	43	43	43	43	43
		57	23	23	23	23	43	16	16	16	16
		0	34	34	34	34	0	27	27	27	27
		0	8	12	12	12	0	7	11	12	12
	$T_R = 500$	57	57	57	57	57	43	43	43	43	43
		57	23	23	23	23	43	16	16	16	16
		0	34	34	34	34	0	27	27	27	27
		0	8	14	14	14	0	7	11	15	15

and 500 ns, as we wanted to evaluate the impact of this parameter as well. Fig. 12(a) shows the total network simulation execution time when the number of DRL cells increases (EW is fixed to four). A $\mathrm{DRL} = 0$ value means an all-software simulation execution. From Fig. 12(a), it can be observed that using a single DRL cell with a reconfiguration time of 2000 ns, give worst results than an all-software solution. Clearly, with a single DRL cell, it is not possible to perform in parallel, event computation, and DRL cells reconfiguration. So, fast reconfiguration times are needed in order to obtain any improvement. When the number of DRL cells increases, event execution and DRL reconfiguration can be performed in parallel, so reconfiguration overhead effects are minimized and improvement is obtained. From Fig. 12(a) and for this example, it can be seen, that both target architectures (local and shared) obtain almost the same results when the number of DRL cells is less or equal to two.

Moreover, in this case study, when the number of DRL cells is equal or higher than three, the shared memory architecture obtains better results than the local memory architecture. This is due to the type of mapping between DE classes/objects and DRL cells. As previously introduced, using a local-memory architecture implies a direct mapping, while using a shared-memory architecture means an associative mapping. The type of mapping determines if all reconfigurations can be initiated by the DRL multicontext scheduler, and are transparent to the HW/SW scheduler. We have obtained that all reconfigurations can be initiated by the DRL multicontext scheduler when using the shared-memory architecture. When using the local-memory architecture, the DRL multicontext scheduler can not hide all reconfigurations to the HW/SW scheduler.

Finally, it is possible to observe for this example that the shared memory architecture converges faster (i.e., it requires less DRL cells) than a local-memory architecture, to achieve the same results that would be obtained using as many DRL cells as DE classes found in the system specification. Clearly, this static approach will not have reconfiguration overheads.

Fig. 12(b) and (c) show the effect of the event window size on the execution time for a fixed number of DRL cells. These results belong to group I experiments, and they have been obtained for both target architectures, too. The results obtained show that the best event window size depends on the number of DRL cells. If the size of the event window is set to zero, it indeed means that the prefetching mechanism is deactivated. In this situation, and assuming an architecture with two DRL cells [Fig. 12(b)], the local-memory architecture obtains better results than the shared-memory architecture.

However, when the prefetching mechanism is activated, the shared-memory architecture obtains better results. This same behavior can be observed for an architecture with three DRL cells [see Fig. 12(c)]. From both figures it can be observed that when a local-memory architecture is considered, results get saturated when the size of the event window equals the number of DRL cells. If a shared memory is considered, it can be observed that a high improvement in the results is obtained when the event window size equals the number of DRL cells. In this case, increasing the event window means some little improvement in the results, but not really significant [see Fig. 12(b) and (c)]

Now, results from Table II will be explained in more detail. This table shows four different values when fixed the architecture, number of DRL cells, its reconfiguration time, and

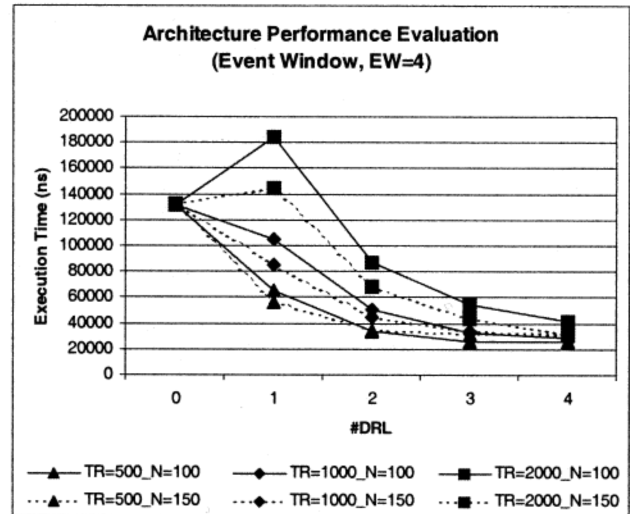
Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

the event window size. For each set, the first line indicates the total number of reconfigurations, which has been classified between the number of reconfigurations performed by the HW/SW scheduler (the second line, which is in black) and the DRL multicontext scheduler (the third line in the column). Moreover, in this table, it is shown to total number of times that all DRL cells visit the *waiting* state [remember Fig. 5(a)], which is the bottom line and is shadowed. Intuitively, it can be thought that when fewer reconfigurations are performed in total, better results (execution times) would be obtained. But, if we observe Table II, it is clear that previous statement is false. For example, if we observe results for a shared memory with three DRL cells and a reconfiguration time of 2000 ns, it is seen that increasing the event window means performing more reconfigurations in total, but also better executions times are obtained.

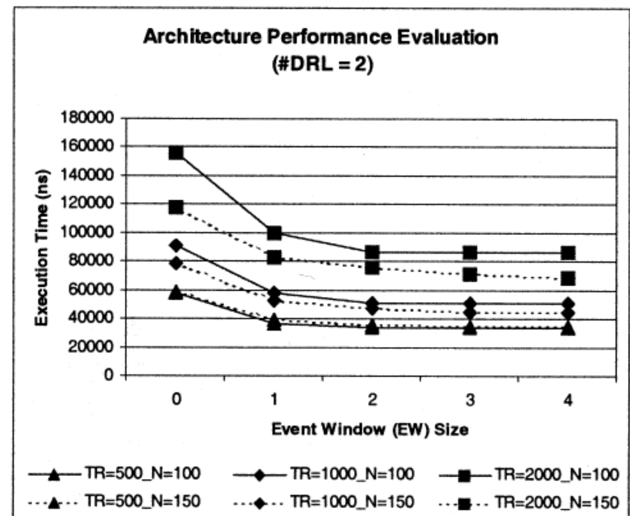
From Table II, we can observe that using the shared memory architecture, the DRL multicontext scheduler can initiate all reconfigurations (when the prefetch is active); except the correspondent to the first event, which will be performed by the HW/SW scheduler. This is not the case of a local-memory architecture, where the HW/SW scheduler must initiate multiple reconfigurations. It is important to comment the number of times that the DRL cells reach the *waiting* state. Remember that if a DRL cell visits the *waiting* state, it means that the DRL cell is ready to process an event, but it cannot be executed because previous events computation have not finished. So in our approach this is the best possible situation, which indeed means that class switch and object switch are complete before the HW/SW scheduler starts the execution of the event. From Table II, and using the shared-memory architecture with three DRL cells and a reconfiguration time of 2000 ns, we can observe that the number of times that the DRL cells reach the *waiting* state increases as the event window also increases. In this previous situation, and assuming an event window of four, the DRL multicontext scheduler initiates 49 reconfigurations and 14 of them (the number of times the *waiting* state is reached) are completely transparent to the HW/SW scheduler. The other 35 reconfigurations will only be partially overlapped with the execution of previous events. That is, the HW/SW scheduler will wait for the reconfiguration to finish. From Table II, we can also observe that reducing the reconfiguration time means more visits to the *waiting* state, that is, more reconfigurations will be completely transparent to the HW/SW scheduler.

Results for group II experiments $(N = 150)$ are shown in Fig. 13, and they have been compared to group I experiments $(N = 100)$ when a shared-memory architecture is considered. Remember that for model $N = 100$, the HW/SW partitioning algorithm mapped all classes to hardware, while in model $N = 150$ the same algorithm mapped four classes to hardware and two to software, as there are not enough memory resources (object-state memory) in the target architecture. Both experiments have been used to study the impact of the obtained HW/SW partitioning on the execution time.

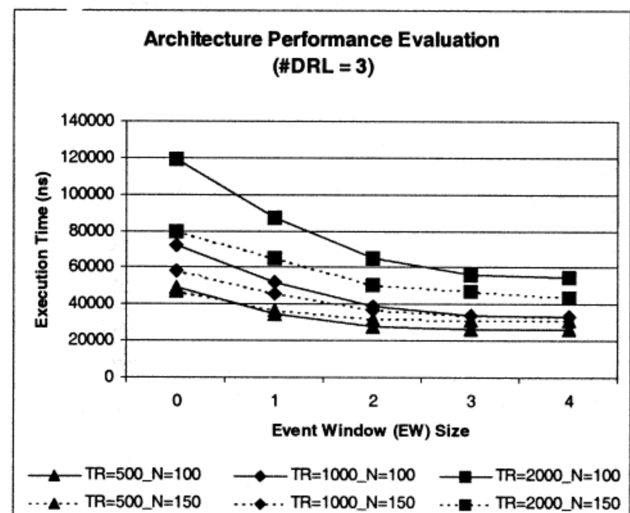
Fig. 13(a) shows the total execution time when the number of DRL cells increases. We can observe that using one DRL cell, the experiments from group II are always better than the group I experiments, independently of the reconfiguration time. That is, HW/SW partitioning for group II is better than the HW/SW partitioning for group I. However, when using two DRL cells with a



(a)



(b)



(c)

Fig. 13. Simulation models results.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

reconfiguration time of 2000 ns and 1000 ns, it can be observed that group II experiments obtain the best results. But, this is not the case if we use two DRL cells with a reconfiguration time of 500 ns, where group I experiments obtain the best results [see Fig. 13(a) carefully].

The same behavior can be clearly observed when using three or more DRL cells, where the results from group I experiments are better than results from group II experiments, if DRL cells with 1000 ns and 500 ns are used. So, from these results we can conclude that the best HW/SW partitioning not only depends on the number of DE classes, their HW/SW execution time and object-state memory requirements, but also on the number of DRL cells used and their reconfiguration time. It is clear, that a more accurate HW/SW partitioning algorithm becomes necessary. Before introducing this new algorithm in Section VI, let us comment on Fig. 13(b) and (c). They show the effect of the event window size on the execution time [as in Fig. 12(b) and (c)]. The important point here is that when having classes mapped to HW and SW (as it is the case of group II experiments) the best event window size is equal to the number of DRL cells plus one. This is clearly due to the fact that there is one more DE functional unit (CPU), than in the case of having all classes mapped to hardware. So, indeed this means that DRL reconfigurations can be overlapped with CPU executions.

VI. Improved HW/SW Partitioning Algorithm

As introduced in Section III-A, a set of independent DE classes $C = (C_1, C_2, \ldots, C_L)$ is the input to the HW/SW partitioning algorithm. A concrete class (C_i) of the input set of classes is characterized by a set of estimators E_i. Remember expression (1).

An important comment is needed for estimators $\mathrm{AET}_i^{\mathrm{HW}}$ and $\mathrm{AET}_i^{\mathrm{SW}}$. These estimators are static, which only give information about the execution time of a concrete event execution. They do not take into account the dynamic behavior of the event stream. And even more important, these estimators do not take into account the features or parameters of the dynamically reconfigurable architecture (reconfiguration time, number of contexts, etc). These parameters indeed have a direct impact into the performance given by the HW/SW partitioning.

Let us consider first the $\mathrm{AET}_i^{\mathrm{HW}}$ estimator, and how it can be modified to take into account the features of our target reconfigurable architecture. As explained in the introduction, reconfiguration latency minimization is one of the major challenges introduced by reconfigurable computing. In our approach, we propose a hardware based prefetching technique, which overlaps execution and reconfiguration. The following expression represents how it is going to be taken into account at the HW/SW partitioning level 1) the parameters from reconfigurable platform and 2) the configuration prefetching technique for reconfiguration latency minimization. We define as

$$\overline{\mathrm{AE}_i^{\mathrm{HW}}} = \mathrm{AET}_i^{\mathrm{HW}} + \alpha_R \cdot \left(T_R - EW \cdot \overline{T_{\mathrm{EXE}}}\right) \tag{4}$$

where

1) α_R is the probability of reconfiguration, i.e., the probability that when an event of class C_i is going to be executed there is not any DRL cell that has class C_i loaded. This probability is a function of the number of classes found in the set C^{HW}, and the number of DRL cells. Its value is depicted in expression (5).

$$\alpha_R = \begin{cases} 0, & \text{if } |C^{\mathrm{HW}}| \leq \mathrm{DRL} \\ \frac{|C^{\mathrm{HW}}| - \mathrm{DRL}}{|C^{\mathrm{HW}}|}, & \text{if } |C^{\mathrm{HW}}| > \mathrm{DRL}. \end{cases} \tag{5}$$

In case we have more or equal DRL cells than HW classes $(|C^{\mathrm{HW}}| \leq \mathrm{DRL})$ each class can be mapped on a different DRL cell and no reconfigurations will be required $(\alpha_R = 0)$. Otherwise, if there are more HW classes than available DRL cells, there is a nonnull probability of reconfigurations. Given $|C^{\mathrm{HW}}|$ classes and *DRL* cells, there are $C_{\mathrm{DRL}}^{|C^{\mathrm{HW}}|}$ combinations[2] of classes loaded in the DRL cells. From all these cases, reconfiguration is required if the event class is not loaded. These unfavorable cases can be calculated as $C_{\mathrm{DRL}}^{|C^{\mathrm{HW}}|-1}$. So, α_R can be calculated as

$$\alpha_R = \frac{C_{\mathrm{DRL}}^{|C^{\mathrm{HW}}|-1}}{C_{\mathrm{DRL}}^{|C^{\mathrm{HW}}|}}.$$

From this expression, it can be derived the one indicated in (5).

2) T_R is the reconfiguration time needed for a DRL cell to change its context.

3) EW is the size (in number of events) of the prefetch. We experimentally obtained that the best EW is represented by expression (3).

$$EW = \begin{cases} \mathrm{DRL}, & \text{if } \left|C^{\mathrm{SW}}\right| \leq \\ \mathrm{DRL} + 1, & \text{if } \left|C^{\mathrm{SW}}\right| > 1. \end{cases} \tag{6}$$

4) $\overline{T_{\mathrm{EXE}}}$ is defined as the average executing time for all the classes of set C. Each class belongs either to subset C^{HW} or to subset C^{SW}, so only one of its estimators will be considered to calculate the average executing time, which is given by the following expression.

$$\overline{T_{\mathrm{EXE}}} = \frac{\sum_{i=1}^{|C|} \mathrm{AET}_i^{\{\mathrm{HW,SW}\}}}{|C|}. \tag{7}$$

$\overline{\mathrm{AET}_i^{\mathrm{HW}}}$ represents the average execution time for class C_i on top of the reconfigurable architecture. This value is obtained adding to its execution time the reconfiguration overhead, which is not a fixed value and depends on the number of DRL cells, its reconfiguration time and the number of classes in the subset C^{HW}. From (4), it is possible to observe that the reconfiguration overhead depends on 1) the reconfiguration probability [which will be higher when more classes are present in subset C^{HW}, for a fixed number of DRL cells] and 2) the reconfiguration time, which could be reduced using the prefetching technique. As the prefetching techniques are based on the overlapping of the execution of an event in a DRL cell with the reconfiguration of another DRL cell, the reconfiguration time could be reduced. This reconfiguration time can be reduced by a factor that is proportional to the EW, and to the average execution time of the set of classes C.

[2] Combinations of $|C^{\mathrm{HW}}|$ elements taken from DRL to DRL

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

```
ImproveInitialSolution(P_INITIAL) {
 Movement = TRUE;
 while Movement == TRUE loop
    Movement = FALSE;
    FirstTaggedSW = GetFirstClassTaggedSW();
    C_F = GetFirstClass(P_INITIAL, FirstTaggedSW);

    SetState(C_F, tagged_HW);
    ExecTime = AverageExecutionTime();
    α_R = ReconfigurationProbability();
    EW = EventWindowSize();

    SetState(C_F, tagged_SW);
    α_COM = CommunicationProbability();
    MediumCommTime = AverageCommTime();

    for i = 0 to FirstTaggedSW loop
       C_i = GetClass(P_INITIAL);
       ExecTimeHW_i = TexecHW(C_i) + α_R . (T_R - EW * ExecTime);
       ExecTimeSW_i = TexecSW(C_i) + α_COM * CommTime;
       if (ExecTimeHW_i < ExecTimeSW_i) then
          if (GetState(C_i) == tagged_SW) then
             SetState(C_i, tagged_HW);
             Movement = TRUE;
          elsif (GetState(C_i) == tagged_HW)
             SetState(C_i, fixed_HW);
             Movement = TRUE;
          end if;
       else
          Movement = FALSE;
          Break;
       end if;
    end loop;
 end loop;
}
```

Fig. 14. Improvement of the initial solution.

Let us now consider the $\overline{\mathrm{AET}_i^{\mathrm{sw}}}$ estimator, and how it is modified to take into account the features of the event stream, the software processor, and the HW/SW communication strategy. This is shown in the following expression:

$$\overline{\mathrm{AET}_i^{SW}} = \mathrm{AET}_i^{SW} + \alpha_{\mathrm{COM}} \cdot \overline{T_{\mathrm{COM}}} \tag{8}$$

where

1) α_{COM} is the probability of HW/SW communication, which is a function of the number of classes found in the set C^{SW}. In our approach, we assume that HW/SW communication could also be improved using a prefetching technique, which overlaps an event execution on the DRL architecture with the HW/SW communication, for an event that will be executed by the software processor in the near future (within the EW). Its value is depicted in (9). This probability represents the case in which two events, that have to be executed into the software processor, are consecutive in the event stream, and thus HW/SW communication can not be hidden

$$\alpha_{\mathrm{COM}} = \frac{|C^{SW}|}{|C|}\frac{|C^{SW}|}{|C|}. \tag{9}$$

2) $\overline{T_{\mathrm{COM}}}$ is the average HW/SW communication time. It represents the average transfer time using the system bus.

We have already introduced that our partitioning algorithm is resource constrained. The design constraints are object state memory and class (DRL context) memory. We formulate our problem as maximizing the number of DE classes mapped to the subset C^{HW} while 1) meeting memory and DRL area constraints and 2) the average execution time for all classes present in C^{HW} is less than its average software execution time

$$\begin{aligned}&\mathrm{Max}\left(|C^{\mathrm{HW}}|\right), \text{ such that:}\\ &1.\ \sum_{j=1}^{M}\mathrm{SVM}_j \le \mathrm{OSMA} \text{ and } \mathrm{DRLA}_j \le \mathrm{DRLA}\\ &2.\ \overline{\mathrm{AET}_j^{\mathrm{HW}}} < \overline{\mathrm{AET}_j^{\mathrm{SW}}}, \forall\, C_j \in C^{\mathrm{HW}}.\end{aligned} \tag{10}$$

This new HW/SW partitioning algorithm that we are proposing is divided in three main steps: 1) obtaining an initial solution; 2) improvement of the initial solution; and 3) class packing in reconfiguration contexts.

In order to perform this incremental approach, classes are labeled with an active *state*. We consider the following states: 1) *free*; 2) *fixed_HW* and *fixed_SW*; and 3) *tagged_HW* and *tagged_SW*. *Free* state means that the class is not assigned to any subset (C^{HW} or C^{SW}). Initially all classes are *free*. *Fixed_HW* means that the class belongs to subset C^{HW}, and that it can not be moved from this subset. *Fixed_SW* means the same as *fixed_HW* but with respect to C^{SW}. *Tagged_HW* means that the class belongs to subset C^{HW}, but that the class could be moved to C^{SW}. *Tagged_SW* means the same as *tagged_HW* but with respect to C^{SW}.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

A. Obtaining the Initial Solution

Obtaining an initial solution is addressed using the list-based partitioning algorithm presented in Section III-A. In this case, the following cost function has been used.

$$F_i = \alpha \cdot \left(\mathrm{AET}_i^{\mathrm{HW}} - \mathrm{AET}_i^{\mathrm{SW}}\right) + \beta \cdot \frac{1}{NO_i}. \qquad (11)$$

The difference is that in Section III-A, the algorithm was used to perform the HW/SW partitioning, and in this case, the algorithm is used to decide which classes will be definitively mapped to SW (*fixed_SW*) because of the limited resources. The rest of classes will be classified as *tagged_SW*, and they will be the input to the following step of the algorithm.

B. Improvement of the Initial Solution

Improvement of the initial solution is achieved using an iterative algorithm. This algorithm is based on the idea of moving classes from the subset C^{SW} (concretely, the ones labeled as *tagged_SW*) to the subset C^{HW}. This movement of a class is mainly determined by the expressions introduced previously at the beginning of this section [see (4) to (9)].

The pseudocode of the proposed algorithm is shown in Fig. 14. The input to this algorithm is the sorted list of classes, where each class is labeled with a state. Classes are still ordered by the same cost function. The algorithm iterates within a loop while there is any movement. When trying to perform the movement, the algorithm initially gets the first class in the list that is labeled as *tagged_SW*, and momentarily labels the class as *tagged_HW*. After that, the algorithm evaluates the partitioning of (5)–(7) assuming that the class is mapped to HW. Once this process has finished, it returns the class to its initial label (*tagged_SW*) and evaluates (9) assuming that the class is assigned to SW.

At this point it is possible to evaluate (4) and (8), for all the preceding classes in the list. This means, to evaluate the influence that will have moving a new class into the reconfigurable hardware, on the average executing time of the other classes. For each class, the algorithm checks if the average execution time in HW is less than the average execution time in SW. If this condition is not asserted the algorithm stops, otherwise it checks the state of the class in order to move the class to HW. This process of moving a class to HW is performed in two steps; 1) the class changes its state from *tagged_SW* to *tagged_HW* and 2) the class changes its state from *tagged_HW* to *fixed_HW*. Each one of these steps will be performed in different iterations of the algorithm. The mapping process is done this way to prevent the algorithm to enter into a nonconverging state. The result of applying this algorithm will be some classes labeled as *fixed_HW* and *tagged_HW*. These classes will be finally mapped to the reconfigurable HW. The rest of classes will be mapped to SW.

C. Class Packing in Reconfiguration Contexts

Once the improvement of the initial solution is finished, it is possible to perform a second type of optimization. Previously, the reconfiguration-latency problem has been addressed using a prefetching technique. However, a second possibility for minimizing the reconfiguration latency is to reduce the number of reconfigurations that are performed.

```
LeftEdgeClassPacking(FixedHWClasses) {
 Final = FALSE;
 AreaFPGA = GetAreaFPGA();
 SortFixedHWClasses_AreaFPGA();
 LastAssignedRC = 1;
 AreaRC = AreaFPGA;
 while !Final  loop
   Final=TRUE;
   for i=0 to |FixedHWClasses| loop
     Ci = GetClass(FixedHWClasses);
     if GetAssignedToRC(Ci) == 0 then
       if AreaRC >= GetClassAreaFPGA(Ci) then
          SetAssignedToRC(Ci, LastAssignedRC);
          AreaRC = AreaRC - GetClassAreaFPGA(Ci);
       end if;
       Final=FALSE;
     end if;
   end loop;
   LastAssignedRC++;
   AvailableAreaRC = AreaFPGA;
 end loop;
}
```

Fig. 15. Left-edge class packing.

This objective can be achieved if all classes labeled as *fixed_HW* and *tagged_HW* are packed into the minimum number of reconfiguration contexts. In this case, a reconfiguration context represents the implementation of several classes into a single DRL cell. In the worst case, each reconfiguration context will implement a single class. In the best case, a single reconfiguration context will be needed for all classes. Classes are packed into reconfiguration contexts according to their DRL area. A reconfiguration context can implement N classes if the sum of the DRL required area of these N classes does not exceed the area of the DRL cell.

We have addressed the problem of obtaining the minimum number of reconfiguration contexts using a left-edge based algorithm. The left-edge algorithm is well known for its application in channel-routing tools for physical-design automation. It has been also adapted to solve the register allocation problem in high-level synthesis [20]. We have adapted and used this algorithm to address our problem. Using this approach we always get optimal results for the number of reconfiguration contexts.

The pseudocode for the left-edge class packing is shown in Fig. 15. The basic idea of this algorithm is to sort the input classes (labeled as *fixed_HW* and *tagged_HW*), using their area as the ordering factor. Once this is done, the algorithm searches sequentially for the next class that can be included within the current reconfiguration context. Function *GetAssignedToRC*(C_i) indicates if class C_i has already been assigned to any reconfiguration context or not.

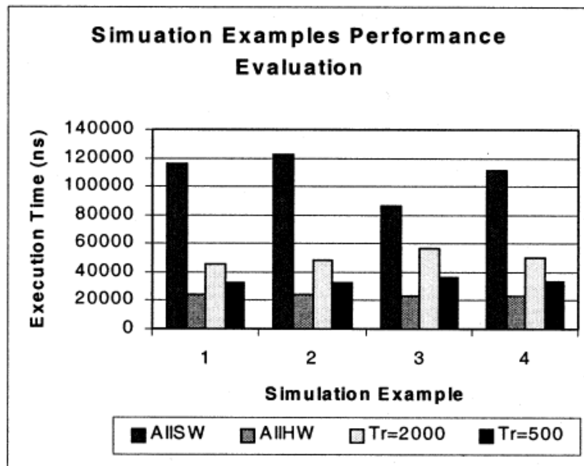
D. Experiments and Results

In order to test this novel HW/SW partitioning algorithm, we have applied the algorithm to four different network configurations (named, simulation example 1, 2, 3, and 4). The used simulation examples have different features, which are defined in the following: Simulation examples 1 and 2, have 7 DE classes, while simulation examples 3 and 4, have 8 DE classes. Simulation examples 1 and 2 differ because of the DE classes DRL required area. In simulation example 1, the DE classes require

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

SIMULATION EXAMPLE		HW	SW	
	1	{1}, {3, 7}, {6}	2, 4, 5	$T_R = 2000$
		{1}, {2, 6}, {3, 7}, {4}	5	$T_R = 500$
	2	{1}, {3}, {6}, {7}	2, 4, 5	$T_R = 2000$
		{1}, {2}, {3}, {4}, {6}, {7}	5	$T_R = 500$
	3	{1}, {2}, {4}	3, 5, 6, 7, 8	$T_R = 2000$
		{1}, {2, 4}, {5, 7}, {6}	3, 8	$T_R = 500$
	4	{1}, {3, 6}, {8}	2, 4, 5, 7	$T_R = 2000$
		{1}, {2, 8}, {3, 6}, {4}	5, 7	$T_R = 500$

(a)



(b)

Fig. 16. Final results. (a) HW/SW partitioning and (b) execution times.

a DRL area that facilitates class packing into reconfiguration contexts. Simulation example 2 is the opposite case, where each class has an area equivalent to a DRL context. Finally, the simulation examples 3 and 4 differ in the HW and SW execution times. In simulation example 3, the difference between the HW and the SW execution time is not significant, while in simulation example 4, this difference is considerable.

The proposed partitioning algorithm has been implemented within our codesign framework, and the DRL multicontext scheduler for a shared-memory architecture has been adapted to deal with groups of classes, as the reconfiguration contexts. A whole bunch of cosimulations, varying the number of DRL cells and their reconfiguration times, have been carried out.

The obtained results for this HW/SW partitioning algorithm are shown is Fig. 16. They are a sample of the obtained results. Fig. 16(a), shows for each simulation example the HW/SW partitioning results when three DRL cells with two different reconfiguration times are considered. Classes mapped to HW, are grouped to DRL contexts. In this table, it can be observed the effect of the reconfiguration time in the HW/SW partitioning. That is, when the reconfiguration time is lower, more classes are mapped to HW.

In the other hand, Fig. 16(b). shows the obtained simulation execution time. For each simulation example, four execution times are presented: 1) an all SW implementation; 2) a HW implementation with unlimited number of DRL cells and object state memory; 3) a mixed HW/SW implementation with 3 DRL cells with a reconfiguration time of 2000 ns, and (4) a mixed HW/SW implementation with 3 DRL cells with a reconfiguration time of 500 ns. These last two results were obtained using the previously obtained HW/SW partitions.

From this Fig. 16(b), the great difference can be observed between the all SW implementation and the rest of implementations (even in the case of simulation example 3, because of the HW/SW communication overhead). In general, we can observe that the obtained results, for a mixed HW/SW implementation (for both reconfiguration times), do not have a really significant difference compared to an all HW implementation.

VII. Conclusion

DRL devices and architectures change many of the basic assumptions in the HW/SW codesign process. The flexibility of DRL architectures requires the development of new methodologies and algorithms. In this paper, we have presented three major contributions: 1) a new HW/SW codesign methodology with dynamic scheduling for discrete event systems using dynamically reconfigurable architectures; 2) a novel approach to dynamic DRL multicontext scheduling; and 3) a HW/SW partitioning algorithm for dynamically reconfigurable architectures.

We have applied our methodology to the software acceleration of broad-band telecom networks simulation. We have developed a whole codesign framework, in order to perform an exhaustive study of our methodology and proposed algorithms and schedulers. This exhaustive study has been carried out, and the obtained results demonstrate the benefits of our approach.

Acknowledgment

The authors acknowledge the Department of Research and Development of Hewlett-Packard Inkjet Commercial Division, Barcelona, Spain, for its support in the preparation of his Ph.D. dissertation.

References

[1] [Online]. Available: http://www.altera.com/
[2] [Online]. Available: http://www.xilinx.com/
[3] [Online]. Available: http://www.chameleonsystems.com/
[4] F. Balarin, L. Lavagno, P. Murthy, and A. S. Vincentelli, "Scheduling for embedded real-time systems," *IEEE Design and Test*, Jan–Mar. 1998.
[5] S. Bhatt, R. Fujimoto, A. Ogielski, and K. Perumalla, "Parallel simulation techniques for large-scale networks," *IEEE Commun. Mag.*, vol. 46, pp. 42–47, Aug. 1998.
[6] N. Caponio *et al.*, "Single layer optical platform based on WDM/TDM multiple access for large scale switchless networks," *Euro. Trans. Telecom.*.
[7] K. Chatta and R. Vemuri, "Hardware–software codesign for dynamically reconfigurable architectures," in Proc. of FPL'99, Glasgow, Scotland, Sept. 1999.
[8] D. Deshpande, A. Somani, and A. Tyagi, "Configuration caching vs data caching for striped FGPA's," in *Proc. ACM/SIGDA Int. Symp. on FPGA*, Monterey, CA, Feb. 1999, pp. 206–214.
[9] R. P. Dick and N. K. Jha, "CORDS: Hardware–software co-synthesis of reconfigurable real-time distributed embedded systems," in Proc. Int. Conf. Computer-Aided Design '98, San Jose, CA, Nov. 1998.
[10] M. D. Edwards *et al.*, "Acceleration of software algorithms using hardware/software co-design techniques," *J. Syst. Architecture*, vol. 42, no. 9/10, p. 1997.
[11] R. Ernst, J. Henkel, and T. Benner, "Hardware–software cosynthesis for microcontrollers," *IEEE Design Test Comput.*, vol. 10, pp. 64–75, Dec. 1993.
[12] J. Fleischman *et al.*, "A hardware/software prototyping environment for dynamically reconfigurable embedded systems," in Proc. CODES'98, Seattle, WA, Mar. 1998.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.

[13] R. Gerndt and R. Ernst, "An event-driven multi-threading architecture for embedded systems," in *Proc. Codes/CASHE '97*, Braunschweig, Germany, Mar. 1997, pp. 29–33.

[14] R. Gupta and G. De Micheli, "Hardware–software cosynthesis for digital systems," *IEEE Design Test Comput.*, vol. 10, pp. 29–41, Sept. 1993.

[15] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *Proc. ACM/SIGDA Int. Symp. FPGA*, Monterey, CA, Feb. 1998, pp. 65–74.

[16] R. Hartenstein, "A decade of reconfigurable computing: A Visionary retrospective," in Proc. DATE'01, Munich, Germany, Mar. 2001.

[17] B. Jeong *et al.*, "Hardware–software cosynthesis for run-time incrementally reconfigurable FPGA's," in Proc. of Asia South-Pacific Design Automation Conf. (ASP-DAC'2000), Yokohama, Japan, Jan. 2000.

[18] A. Kalavade and E. A. Lee, "The extended partitioning problem: Hardware/software mapping, scheduling and implementation-bin selection," *J. Design Automation Embedded Syst.*, vol. 2, pp. 163–226, Mar. 1997.

[19] M. Kaul, R. Vemuri, R. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in Proc. Design Automation Conf. (DAC), New Orleans, LA, June 1999.

[20] F. J. Kurdahi and A. C. Parker, "REAL: A program for register allocation," in Proc. 24th Design Automation Conf., Miami, FL, June 1987.

[21] E. A. Lee, "Modeling concurrent real-time processes using discrete events," in *Annuals Software Engineering, Special Volume on Real-Time Software Engineering*. Norwell, MA: Kluwer, 1998.

[22] Y. Li *et al.*, "Hardware–software co-design of embedded reconfigurable architectures," in Proc. 37th Design Automation Conf, DAC'2000..

[23] Z. Li and S. Hauck, "Don't care discovery for FPGA configuration compression," in *Proc. ACM/SIGDA Int. Symp. FPGA*, Monterey, CA, Feb. 1999, pp. 91–98.

[24] R. Maestre, F. J. Kurdahi, N. Bagerzadeh, H. Singh, R. Hermida, and M. Fernandez, "Kernel scheduling in reconfigurable computing," in Proc. DATE, Munich, Germany, Mar. 1999.

[25] R. Maestre, F. J. Kurdahi, M. Fernandez, and R. Hermida, "A framework for scheduling and context allocation in reconfigurable computing," in *Proc. Symp Syst Synthesis*, San Jose, CA, Nov. 1999, pp. 134–140.

[26] D. McConnell and P. Lysaght, "Queue simulation using dynamically reconfigurable FPGA's," in Proc. U.K. Teletraffic Symp., Scotland, U.K., Mar. 1996.

[27] V. Mooney and G. De Micheli, ""Real time analysis and priority scheduler generation for hardware-software systems with a synthesized run-time system," in *Proc. Int. Conf. Computer-Aided Design (ICCAD'97)*, San Jose, CA, Nov. 1997, pp. 605–612.

[28] R. Nieman and P. Marwedel, "Hardware/software partitioning using integer programming," in Proc. European Design and Test Conf., Paris, France.

[29] J. Noguera, R. M. Badia, J. Domingo, and J. Sole, "Reconfigurable computing: An innovative solution for multimedia and telecommunication network simulation," in Proc. 25th Euro. Conf., Milan, Italy, Sept. 1999.

[30] ——, "Run-time HW/SW codesign for discrete event systems using dynamically reconfigurable architectures," in Proc. ISSS'2000, Madrid, Spain, Sept. 2000.

[31] ——, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in Proc. DATE'01, Munich, Germany, Mar. 2001.

[32] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for re-configurable computers," *IEEE Trans. Comput.*, vol. 48, pp. 579–590, June 1999.

[33] A. Touhafi, W. F. Brissinck, and E. F. Dirkx, "Simulation of ATM switches using dynamically reconfigurable FPGA's," in *Proc. FPL'98*, vol. 1482, Lecture Notes in Computer Sciences, Tallin, Estonia, Sept..

[34] F. Vahid, "Modifying min-cut for hardware and software functional partitioning," in *Codes/CASHE'97*, Braunschweig, Germany, Mar. 1997, pp. 43–48.

[35] ——, "A three-step approach to the functional partitioning of large behavioral processes," in *Proc. Int. Symp. Syst. Synthesis*, Dec. 1998, pp. 152–157.

[36] M. Vasilko and D. Ait-Boudaoud, "Scheduling for dynamically reconfigurable FPGA's," in Proc. Int. Workshop on Logic and Architecture Synthesis., Grenoble, France, Dec. 1995, IFIP TC10 WG10.5, pp. 328–336.

[37] W. Wolf, "Object-oriented cosynthesis of distributed embedded systems," *ACM Trans. Design Automation Electron. Syst.*, vol. 1, no. 3, pp. 301–314, July 1996.

Juanjo Noguera received the B.Sc. degree in computer science from the Autonomous University of Barcelona, Barcelona, Spain, in 1997. He is currently working toward the Ph.D. degree at the Technical University of Catalonia, Barcelona, Spain.

Since June 2001, he has been with Hewlett-Packard Inkjet Commercial Division, Research and Development Department, San Cugat del Valles, Spain. He was formally with the Spanish National Center for Microelectronics, Barcelona, Spain, and was an Assistant Professor with the Computer Architecture Department, Technical University of Catalonia. His research interests include HW/SW codesign, reconfigurable architectures and system-on-chip design techniques. He has published papers in international conference proceedings.

Rosa M. Badia received the B.Sc. and Ph.D. degrees in computer science from the Technical University of Catalonia, Barcelona, Spain, in 1989 and 1994, respectively.

Currently, she is an Associate Professor with the Department of Computer Architecture, Technical University of Catalonia and Project Manager at the CEPBA-IBM Research Institute, Barcelona, Spain. Her research interests include computer-aided design tools for very large scale integration (VLSI), reconfigurable architectures, performance prediction, analysis of message-passing applications, and GRID computing. She has published papers in international journals and conference proceedings.

Authorized licensed use limited to: University of Florida. Downloaded on March 17, 2009 at 18:35 from IEEE Xplore. Restrictions apply.