Run-Time HW/SW Codesign for Discrete Event Systems Using Dynamically Reconfigurable Architectures*

Juanjo Noguera
Univ. Autònoma de Barcelona, SPAIN
e-mail: juanjo@cnm.es

Rosa M. Badia
Univ. Politècnica de Catalunya, SPAIN
e-mail:rosab@ac.upc.es

Abstract

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 run-time HW/SW codesign methodology for dynamically reconfigurable architectures. Besides, all previous approaches to reconfigurable computing context scheduling are based on static scheduling techniques.

In this paper we present a run-time HW/SW codesign methodology for discrete event systems using dynamically reconfigurable architectures and a dynamic approach to reconfigurable computing multi-context scheduling. We have applied our methodology to software acceleration, and present the obtained results.

1. Introduction and motivation

There are a great number of approaches to HW/SW codesign of embedded systems, which use different techniques for partitioning and scheduling. 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 Hardware and software tasks' sequence can change dynamically in complex embedded systems (ie. control-dominated applications), since such systems often have to operate under many different conditions. Although it 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 mixed implementation of dynamic real-time schedulers in hardware and software is presented in [9]. In [1] a review of several approaches to control-dominated and dataflow-dominated software scheduling, to determine whether a given technique can satisfy deadlines, throughput and other constraints, is presented.

Reconfigurable Computing (RC) is an interesting alternative to ASICs and general-purpose processor systems, since it provides the flexibility of software processors and the efficiency and throughput of hardware coprocessors. Thanks to the advents of new Dynamically Reconfigurable Logic (DRL) devices, which are run-time reconfigurable, new and exciting challenges are presented to embedded systems designers. In order to achieve this run-time reconfiguration, system specification (usually, a tasks graph) must be partitioned into temporal exclusive segments (called *reconfiguration contexts*). This process is usually known as *temporal partitioning*, and it is one of the challenges presented by DRL. The other challenge presented by DRL is to find an execution order of a set of tasks that meets system designs objectives (i.e. minimize the total execution time), that is DRL context scheduling. Several references can be found in the literature addressing these problems, see [8] as an example. These previous approaches address the problem of temporal partitioning and DRL context scheduling, but they do not address HW/SW partitioning. In [4] an integrated algorithm for HW/SW partitioning and scheduling, temporal partitioning and context scheduling is presented.

New approaches are possible because: (1) all existing approaches to DRL context scheduling are based on static scheduling techniques, and (2) no previous work has been carried out in order to define a dynamic HW/SW codesign methodology based on DRL devices. In this paper we address these two open problems and present: (1) a novel run-time HW/SW codesign methodology for dynamically reconfigurable architectures and (2) a dynamic approach

* This work has been funded by CICYT-TIC project TIC-98-0410-CO2-01. Authors acknowledge ALTERA support within its Programmable Hardware Development program.

to DRL multi-context scheduling.

The rest of the paper is organized as follows: Section 2 introduces the run-time HW/SW codesign methodology. In section 3 we explain a list-based HW/SW partitioning algorithm. Section 4 presents a dynamic DRL context scheduling approach. In section 5, we apply our methodology to the software acceleration of telecom networks simulation, and give the obtained results. Finally, section 6 presents the conclusions of this work.

2. HW/SW codesign for discrete event systems

Discrete Event (DE) systems design has been recently addressed using HW/SW codesign techniques [6, 7, 11]. However, none of these approaches is based on DRL devices as hardware platform. The proposed methodology addresses the problem of run-time HW/SW codesign for DE systems using an 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.

2.1. Definitions

Def. 1: a 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.

Def. 2: a 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.

Def. 3: an Event E is a member of $T \times C \times O \times V$ where C is a given set of DE classes, O a set of DE objects, T a set of tags, $T \in \Re^+$ (the real numbers) and V a set of values. Tags are used to model time, and values represent operands or results of event computation.

Def. 4: an Event Stream (ES) is a list of events sequentially ordered by tag. Tags can represent, for example, event occurrence or event deadline.

Def. 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*), $p = (c_a, o_a)$.

Our methodology assumes that: (1) several DE classes could be mapped into a single DE functional unit. (2) all DE objects from a DE class are mapped into the same DE functional unit where the DE class has been mapped.

Def. 6: an 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 $e = (t, c_1, o_1, v_1)$ have to be processed in a DE functional unit with an active pair $p = (c_1, o_2)$ then an object switch should be performed.

Def. 7: a 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 $p = (c_2, o_2)$, then a class switch should be performed.

Class switch, in case of a DRL device, means a context reconfiguration. Object switch means to change the values of the state variables from the ones of a concrete DE object (o_1) to the others of another DE object (o_2).

2.2. Run-time HW/SW codesign methodology

The proposed methodology is depicted in figure 1. It is divided into 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 firstly 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 (ie. 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 stated, the *static stage* handles DE classes and the system has been specified as a set of interrelated DE objects, which are instances of also 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

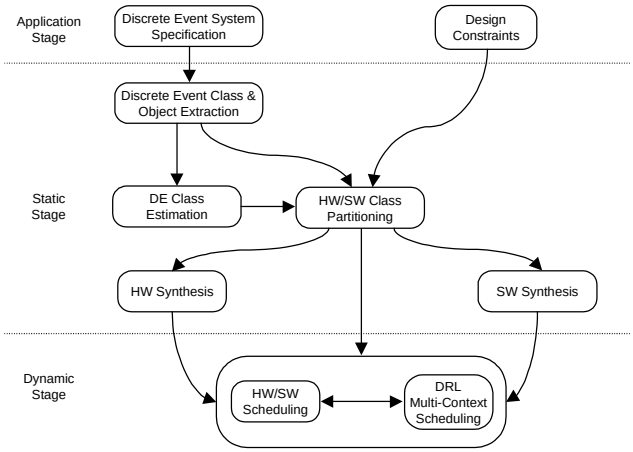


Figure 1. HW/SW codesign methodology.

(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 phase has finished, DE classes can be viewed as a set of independent processes or 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 discrete event application to be solved. The solution given by the partitioning algorithm should meet design constraints. In section 3, we propose an example of HW/SW partitioning algorithm.

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

The estimation phase also deals with DE classes, and used estimators depend on the 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 Multi-Context Scheduling*. Both schedulers base their functionality on 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 figure 1, these scheduling policies (HW/SW and DRL) co-operate and run in parallel during application run-time execution, in order to meet system constraints (i.e. minimize the total application execution time 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, in order to meet system constraints. Diverse policies could be implemented by the HW/SW scheduler based on the final application requirements (ie. earliest deadline first using or not of a pre-emptive technique).

In the other hand, the DRL multi-context 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 minimise the class switching mechanism to the HW/SW scheduler. We assume that different DRL schedulers can be defined depending on the application. In section 4, we present a dynamic DRL multi-context scheduler as an example.

2.3. Target architecture

The target architecture is depicted in figure 2. It is an architecture which comprises a software processor, a DRL-based hardware architecture and shared memory resources. The software processor is a uniprocessing system and it can execute only one event at a time. The

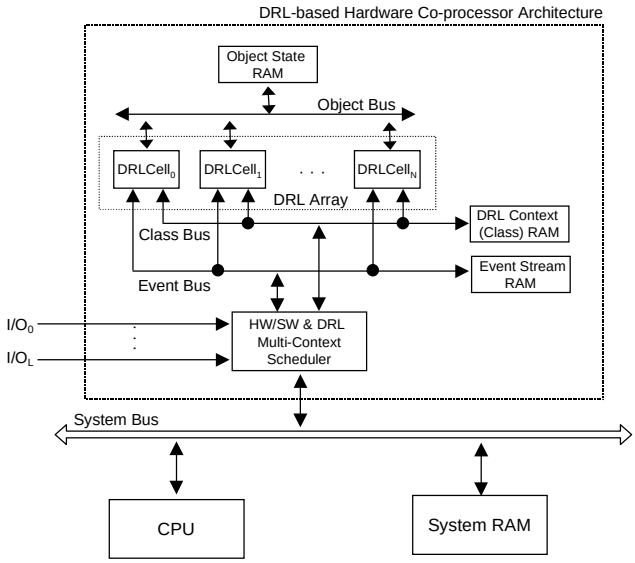


Figure 2. DRL target architecture.

DRL-based hardware co-processor can execute multiple events concurrently. Hardware and software co-operate (interact) via a DMA based memory sharing mechanism.

The DRL-based hardware co-processor architecture is divided into: (1) HW/SW and DRL Multi-Context Scheduler, (2) DRL array, (3) Object State memory, (4) DRL Context memory and (5) Event Stream memory.

The HW/SW and DRL Multi-Context Scheduler must implement functions associated to the dynamic stage of our methodology, as explained above. 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. Finally, DE objects states are stored into the Object State memory.

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 ≅ 20K gates.

The proposed DRL co-processor architecture is scalable, and it is possible to implement any associative mapping between DE objects/classes and DRL cells. Please, note that this mapping is not only fixed by the structure of the DRL Context memory. It also depends on the structure of the Object State memory.

3. HW/SW partitioning algorithm

In this section, we present a resources (object and class memory) constrained HW/SW partitioning algorithm as an example for our methodology.

3.1. Problem statement

Lets consider a set of independent DE classes C = (C_1, C_2, ..., C_L), where each class, lets say C_i, is characterized by a set of estimators E_i,

$$E_i = (WCET_i^{HW}, WCET_i^{SW}, SVM_i, DRLA_i)$$

where:

- $WCET_i^{HW}$ stands for Worst Case Execution Time for a hardware implementation of the DE class C_i.
- $WCET_i^{SW}$ stands for Worst Case Execution Time for a software implementation of the DE class C_i.
- SVM_i stands for State Variables Memory size required by the class.
- $DRLA_i$ stands for DE class DRL required Area.

Lets also consider the design constraints to be object memory and class (DRL context) memory constraints. That is, the total object state memory is denoted by OSMA (Object State Memory Available). CMA stands for the total amount of Class Memory Available.

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

$$Max(|C^{HW}|), s.t. \sum_{j=1}^{M} SVM_j < OSMA, DRLA_j \le CMA)$$

where:

- C^{HW} is the set of DE classes mapped to hardware, $C^{HW} = (C_1^{HW}, C_2^{HW}, ..., C_M^{HW})$, $C^{HW} \subseteq C$

3.2. List-based 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, in order to minimize the total execution time at run-time, which will be responsibility of the HW/SW and DRL context scheduler. 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.

$$F_i = \alpha \cdot (WCET_i^{HW} - WCET_i^{SW}) + \beta \cdot SVM_i$$

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

The pseudo-code of the proposed HW/SW partitioning algorithm is shown in figure 3. 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. *AvailableResources()* function is responsible of design constraints checking. Mainly, it checks that the current hardware partition plus DE class C_i complies with design constraints.

```
ListBasedPartitioningAlgorithm(ED_Classes)
{
 PSW = { ø };  PHW = { ø };
 PINITIAL = Sort_DE_Classes_List (DE_Classes,
                                  FSORT);
 Ci = GetFirst(PINITIAL);
 for i = 2 to L loop
   if Ci.DRL_RequiredArea > DRL_Area then
      PSW = PSW  U  Get(Ci, PINITIAL);
   else
      if AvailableResources(Ci)) then
         PHW = PHW  U  Get(Ci, PINITIAL);
      else
         PSW = PSW  U  Get(Ci, PINITIAL);
      end if;
   end if;
 end loop;
}
```

Figure 3. List-based partitioning algorithm.

4. Run-time DRL multi-context scheduler

In this section we present a run-time event-driven DRL multi-context scheduler. The presented scheduler assumes that the Event Stream is sorted. In this example, we also assume that only the first event of the event stream is been processed on a DE functional unit (DRL cell or CPU) at the same time. Modifications of this scheduler are possible in order to have several events being processed in parallel within the target architecture.

The key idea of the scheduler is to minimize class switching (DRL reconfiguration) overheads, in order to minimize the total application execution time. This objective is accomplished using a lookahead strategy into the event stream memory (see figure 4). Event Window (EW) describes the number of events that are observed in advance and is left as a parameter of our scheduler.

From the DRL array state (that is, from the DE classes that are active) and the event window, the DRL scheduler must decide which DE class should be removed (replaced) from the DRL array, and which DE must be loaded into.

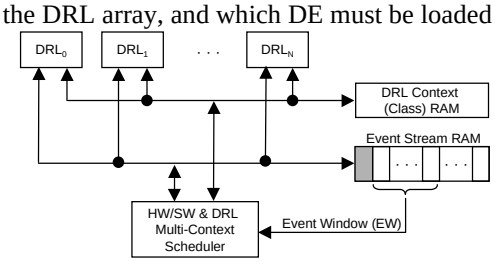


Figure 4. HW/SW & DRL dynamic scheduling.

4.1. Multi-context scheduling algorithm

The pseudo-code for the dynamic DRL scheduling algorithm is shown in figure 5. As stated above, this scheduler depends on the size of the event window.

The basis for the behavior of the proposed DRL multi-context scheduling algorithm is the use of the array *DRLArrayUtilization*, which represents the expected state (active DE classes or contexts) 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 used within the event window (variable *K*). These *K* DRL cells (if there is anyone) are available for a class (context) switch. So, this is the first condition that the algorithm checks.

If there are not any DRL cells available for a class switch, the algorithm selects (to be replaced) the DRL cell which has an active DE class that will be required latest. The algorithm also selects a DE class to be placed as active. The first DE class in the event stream which is not active within the DRL array will be selected. Finally, it performs the class switch with 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 all the event window. If it finds a DE class (associated with an event) which is not active within the DRL array, the algorithm selects the first available DRL cell to be set as active.

5. 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 [5].

In this section, we explain a case study of software acceleration of broadband telecom networks simulation. With the emergence of new packet networks and gigabit-per-second links, the network simulation community is faced with new challenges. The capabilities of sequential simulation techniques are inefficient to address such simulation requirements, due to the several days-long simulation execution time. Parallel computing [2] and reconfigurable computing techniques [10] can be used for simulation execution time improvement.

```
DynamicDRLSchedulingAlgorithm (EW)
{
 ObtainDRLArrayUtilization(EW);
 K = NumberOfAvailableDRL();

 if K = 0 then
  DRLCell = GetLatestRequiredClass();
  Class = GetFirstClassNotInDRLArray(EW);
  DRL_Behaviour(DRLCell, Class);
 else
   CE = GetCurrentEvent();
  for Class = CE to CE+EW loop
   if ActiveClass(Class) = FALSE then
      DRLCell = GetFirstAvailableDRL(Class);
      DRL_Behaviour(DRLCell, Class);
   end if;
  end loop;
 end if;
}
```

Figure 5. DRL dynamic scheduling algorithm.

5.1. Introduction and simulation model

For our case study we have chosen the SONATA[1] network [3]. It is a network based on the switchless network concept. The "switchless" network concept is based on a mixture of WDMA (Wavelength Division Multiple Access) and TDMA (Time Division Multiple Access) methods (see figure 6).

Note that the proposed simulation model has been left to depend 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 methodology proposed in section 2 to the simulation of broadband telecom networks. Specially, it is important the mapping between network elements (found in the network model) and DE objects and classes which are the basic elements that our methodology deals with. From figure 6, as an example, we can affirm that there are network elements which are instances from certain networks element types. For example, from figure 6 it is possible to find 7 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

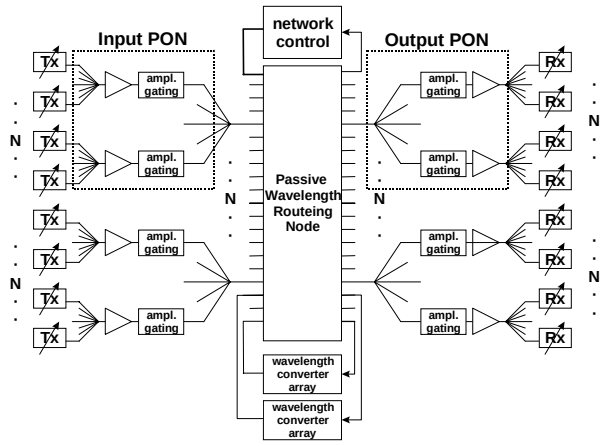


Figure 6. SONATA Network Architecture.

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

within the scope of our methodology. In the same way, network elements should be viewed as DE objects. For this case study we don't consider the *wavelength converter array* network elements, so that in this case study we assume to have 6 different network element types.

Finally, let's give a short review to DE simulation concepts. A DE simulator is a concrete application of DE systems, where the events are ordered following a policy of Shortest Tag First. Moreover, a simulation scheduler is the responsible for the execution of the simulation. Typically, a sequential scheduler is used.

5.2. Developed codesign framework

In order to test our proposed methodology, HW/SW partitioning algorithm and run-time DRL context scheduler, we have implemented a whole codesign framework, which is depicted in figure 7.

In the proposed methodology, DRL target architecture and run-time context scheduler, several parameters where left without 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 context scheduler. Moreover, the simulation model depends on parameter *N*, too. So, it is obvious that a framework where to study the effects and impact of these parameters into our proposals is necessary. We mainly have implemented two different tools: (1) a HW/SW partitioning tool, and (2) a HW/SW co-simulation tool. Within both tools we have implemented the algorithms described in this paper, but new algorithms can be easily included into 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 set-up parameters like the number of DRL cells, their reconfiguration time and the size of the event window used by the DRL context scheduler. In file *PartTool.cfg* it is specified the cost function and parameters that HW/SW partitioning algorithm should use.

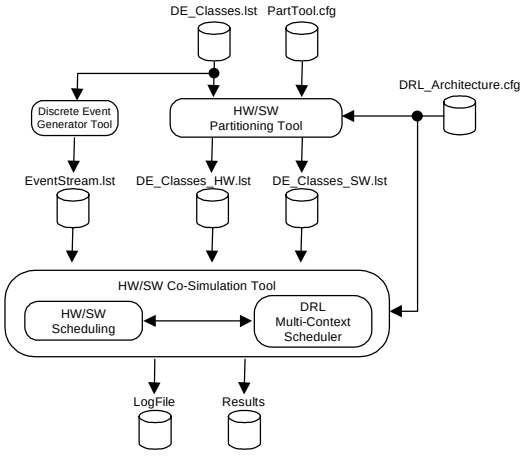


Figure 7. Developed codesign framework.

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.

Each one of the several network elements found in the SONATA network has been modeled using the telecommunication description language TeD [2]. 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 co-simulation tool. This tool is responsible to implement the described dynamic stage of our methodology.

5.3. Experiments and results

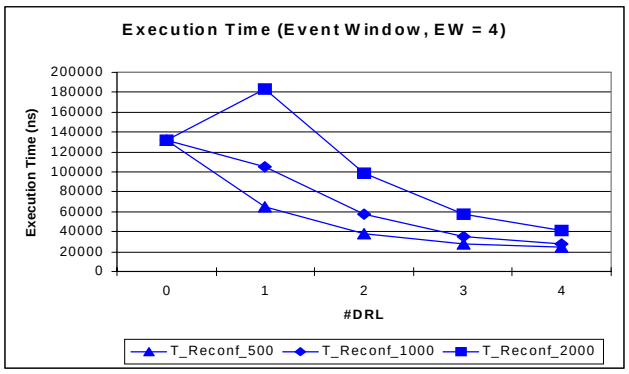
We carried out several experiments on top of this framework. In this sense, two groups of experiments (named, group I and II) have been performed varying the parameter *N* found in the SONATA network simulation model (see figure 6). Once fixed this parameter, several experiments have been performed varying the DRL architecture parameters (file *DRL_Architecture.cfg*). For all experiments the the object state memory and class (context) memory have a size of 128Kx32 bits.

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 group I experiments maps all DE classes to hardware. In the other hand, for group II experiments the HW/SW partitioner maps 4 DE classes to hardware and 2 DE classes to software.

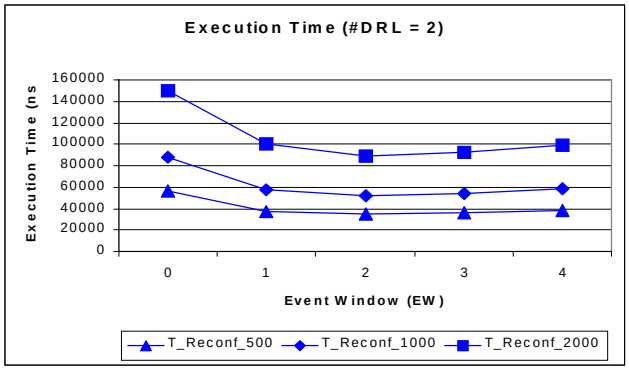
Results for group I experiments are shown in figure 8. This figure shows three different reconfiguration times: 2000ns, 1000ns and 500ns, as we wanted to evaluate the impact of this parameter, too. Figure 8.a shows the total network simulation execution time when the number of DRL cells increases (EW is fixed to 4). A DRL=0 value means an all software simulation execution. From figure 8.a, it is seen that using a single DRL cell with a reconfiguration time of 2000ns, 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 both event execution and DRL reconfiguration can be performed in parallel, so reconfiguration overhead effects are minimized and improvement is obtained. Specially interesting are figures

8.b and 8.c. They show the effect of the event window size on the execution time for a fixed number of DRL cells.

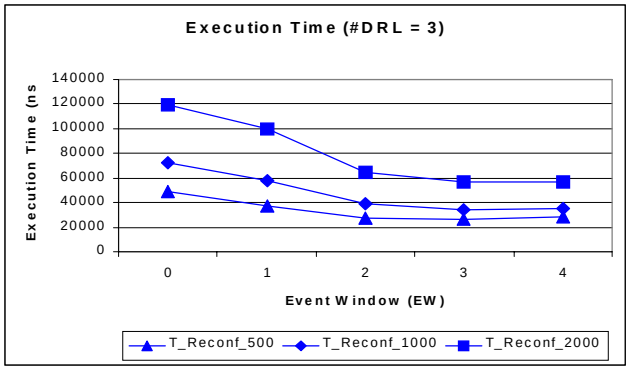
The obtained results demonstrate that the optimum event window size depends on the number of DRL cells. They do present really different behaviors for DRL=2 and DRL=3 cells. For DRL=2 cells, the event window size must keep under 3 events, otherwise execution time gets worst due to an excessive reconfiguration overhead. However, when DRL=3 cells simulation execution gets betters results as EW size increases. It is important to note that these results are not affected by the DRL cell reconfiguration time. We can conclude from these results that our dynamic DRL multi-context scheduler requires, at least, DRL=3 cells in order to exploit its possibilities.



(a)



(b)



(c)

Figure 8. Results for group I experiments

6. Conclusions

In this paper, we have presented two major contributions: (1) a novel run-time HW/SW codesign methodology for discrete event systems using dynamically reconfigurable architectures, and (2) a novel approach to dynamic DRL multi-context scheduling.

We have applied our methodology to the software acceleration of broadband 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, performing two major groups of experiments. Results demonstrate the benefits of our approach. Further research will be carried out, in order to propose alternative HW/SW and DRL context schedulers.

7. References

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

[2] S. Bhatt, R. Fujimoto, A. Ogielski, K. Perumalla, “Parallel Simulation Techniques for Large-Scale Networks”. IEEE Communication Magazine, pp. 42-47. August 1998.

[3] N. Caponio *et al.*, "Single Layer Optical Platform Based on WDM/TDM Multiple Access for Large Scale Switchless Networks", European Transactions on Telecommunications.

[4] K. S. Chatta, R. Vemuri, “Hardware-Software Codesign for Dynamically Reconfigurable Architectures”. Proc. of FPL’99. Glasgow, Scotland. September, 1999.

[5] M. D. Edwards *et al.*, “Acceleration of software algorithms using hardware/software co-design techniques”, Journal of Systems Architecture, Vol. 42, No. 9/10, pp. 1997.

[6] R. Gerndt, R. Ernst “An Event-Driven Multi-Threading Architecture for Embedded Systems”. Codes/CASHE '97, pages 29-33, Braunschweig, Germany, March 1997.

[7] E. A. Lee, “Modeling Concurrent Real-Time Processes using Discrete Events”. Annuals of Software Engineering, Special Volume on Real-Time Software Engineering. 1998.

[8] R. Maestre, F. J. Kurdahi, M. Fernandez, R. Hermida, “A Framework for Scheduling and Context Allocation in Reconfigurable Computing”, Proc. of the International Symposium on System Synthesis, pp. 134-140. 1999.

[9] V. Mooney and G. De Micheli, ``Real Time Analysis and Priority Scheduler Generation for Hardware-Software Systems with a Synthesized Run-Time System," (ICCAD'97), 605-612, November 1997.

[10] J. Noguera, R. M. Badia, J. Domingo, J. Sole, “Reconfigurable Computing: an Innovative Solution for Multimedia and Telecommunication Network Simulation”. IEEE Proc. 25th Euromicro Conference. Milan, Italy. 1999.

[11] Stefan Petters et al. “The REAR framework for emulation and analysis of embedded hard real-time systems”. IEEE Proc. Int. Workshop RSP'99, pp. 100-107, Florida. 1999.