A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures*

Juanjo Noguera, Rosa M. Badia
Dept. Computer Architecture. Universitat Politecnica Catalunya
e-mail: {jnoguera, rosab}@ac.upc.es

Abstract

"System-On-Chip" has become a reality, and recently new reconfigurable devices have appeared. However, few efforts have been carried out in order to define HW/SW codesign methodologies and algorithms which address the challenges presented by new reconfigurable devices.

In this paper we address this open problem and present a novel HW/SW partitioning algorithm for dynamically reconfigurable architectures. The algorithm is a constructive algorithm, which obtains an initial solution and afterwards tries to optimize it. The HW/SW partitioning is done taking into account the features of the dynamically reconfigurable devices, and its final goal is minimize the reconfiguration latency.

The partitioning algorithm has been implemented and integrated into our developed codesign environment, where several experiments have been carried out. The results obtained demonstrate the benefits of the algorithm.

1. Introduction and Motivation

The continued progress of semiconductor technology has enabled the "System-On-Chip" to become a reality. Integrated circuits that merge core processors, embedded memory and custom logic have been reported by a wide range of companies. In this sense, programmable logic companies have also proposed new products [1, 2]. A clear example is the CS2112 chip from Chameleon Systems, Inc. This device integrates a RISC core, embedded memory, and a run-time reconfiguration fabric on single chip [3].

Clearly, these types of programmable devices could be used as the final target architecture in a HW/SW codesign methodology. However, Dynamically Reconfigurable Logic (DRL) changes many of the basic assumptions in the HW/SW codesign process. The flexibility of DRL devices (multiple contexts, partial and run-time reconfiguration, etc.) requires the development of new methodologies and algorithms, as conventional codesign approaches do not consider the features of these new DRL devices.

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

Traditionally, the HW/SW partitioning problem and the reconfiguration latency minimization problem (the major challenge introduced by DRL devices) have been addressed independently. In this paper, we propose a automatic HW/SW partitioning algorithm for DRL architectures. The proposed algorithm takes into account the reconfiguration prefetching when performing HW/SW partitioning. The experiments carried out demonstrate that the benefits using a prefetching technique, for reconfiguration latency minimization, can be improved if it is considered at the HW/SW partitioning level.

This HW/SW partitioning algorithm has been included within a HW/SW codesign methodology for discrete event systems, which have been recently addressed using HW/SW codesign techniques [10, 19]. Key features of this methodology are: (1) it is based on the object-oriented paradigm, and (2) the dynamic scheduling (HW/SW and DRL multi-context) approach.

The rest of the paper is organized into 6 sections. Section 2 overviews the previous related work. Section 3 shortly introduces the codesign methodology. In section 4 we present the HW/SW partitioning algorithm for DRL architectures. In section 5, we apply our algorithm to telecom networks simulation, and give the obtained results. Finally, in section 6 we present the conclusions.

2. Previous Related Work

There are a great number of approaches to HW/SW codesign of embedded systems, which use different techniques for partitioning and scheduling. Earlier approaches to HW/SW partitioning model the system based on a template of a CPU and an ASIC [8, 11]. Moreover, several types of algorithms (constructive or iterative) have been used [21].

In the other hand, *configuration prefetching* techniques can be 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 firstly introduced configuration prefetching in [13], where a single-context prefetching technique is presented. This approach addresses the problem of reconfiguration latency minimization, but it does not address HW/SW partitioning.

Recent research efforts have addressed this open

problem. In [6] an integrated algorithm for HW/SW partitioning and scheduling, temporal partitioning and context scheduling is presented. A more recent work [16] presents a fine-grained HW/SW partitioning algorithm (at loop level). This algorithm optimizes the global application execution time. Both previous approaches are similar to [9, 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 [14] this topic is introduced, and a HW/SW co-synthesis approach for partially reconfigurable devices is presented. They do not address multi-context devices. Moreover, this approach which is based on a ILP formulation, is limited by the high execution times of the algorithm, which hardly gives solutions for task graphs having more than 10 tasks.

3. Codesign Methodology

This section briefly presents the methodology we proposed in [19], which is divided into three stages: *Application Stage*, *Static Stage* and *Dynamic Stage*.

The *application stage* includes *Discrete Event (DE) System Specification* and *Design Constraints*. Initially, a set of independent DE classes are modeled. We define a DE class as a concurrent process type with a certain behavior, which is specified as a function of state variables and input events. Concrete instances of DE classes are DE objects. These DE classes are used to specify the entire system as a set of interrelated DE objects, which communicate among them using events. A DE object computation is activated upon the arrival of an event.

The *static stage* includes: (1) extraction, (2) *estimation*, (3) *HW/SW partitioning*, and (4) *HW and SW synthesis*. The extraction phase obtains: (1) a list of all system instances (DE objects), and (2) a list of all different DE classes and objects connected to it. Once this phase has finished, DE classes can be viewed as a set of independent processes or tasks. The estimation phase can use typical estimators (i.e. for delay and area) that could be obtained using high-level synthesis and profiling tools. The HW/SW partitioning phase decides which classes will be executed in HW and which in SW. These classes are then packed into reconfiguration contexts.

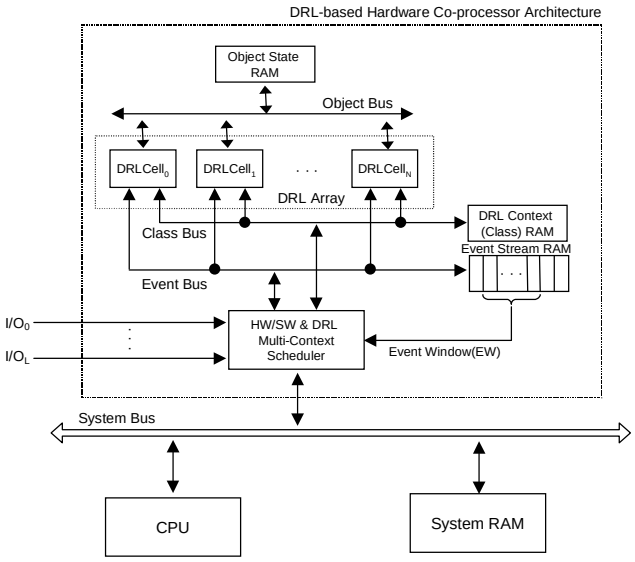

Figure 1. DRL Target Architecture.

The *dynamic stage* includes *HW/SW Scheduling* and *DRL Multi-Context Scheduling*. Both of them run in parallel and base their functionality on events present in the event stream. To better understand how this stage works, let us introduce first the target architecture, which is depicted in figure 1. It is an heterogeneous architecture which comprises a software processor, a DRL-based hardware architecture and shared memory resources.

The HW/SW and DRL Multi-Context Scheduler are mapped to hardware using a centralized control scheme. The Event Stream (a list of events sorted by time) is stored in the Event Stream memory. DRL contexts (which correspond to packed classes) are stored in the DRL Context memory. Finally, DE objects states are stored in the Object State memory. The HW/SW and DRL schedulers co-operate and run in parallel during application run-time execution, in order to meet system constraints. Events get the central scheduler through I/O ports or as a result of a previous event computation.

The HW/SW scheduler decides at run-time the execution order of the events stored in the event stream. The DRL multi-context scheduler is used to minimize packed classes switches. A packed class switch is a mechanism that allows to change from the execution in a DRL of a class (that belongs to a packed class) to another (that belongs to another packet class). A packed class switch implies a DRL reconfiguration. Another mechanism used is the object switch that allows to change from the execution of one object to another object of the same class.

Our approach will process at every moment the first event of the sorted event stream on a DRL cell or in the CPU. The scheduler tries to minimize the packed class switch (reconfiguration) overhead by overlapping the execution of events with these switches. This objective is accomplished by using a lookahead strategy into the event stream memory (see figure 1). Event Window (EW) is the number of events that are observed in advance. At the beginning of the execution of a new event, the scheduler looks ahead EW events in the event stream to see if there is any class C_i mapped to HW that will be next executed which is not currently configured in any DRL. Then the scheduler obtains the state of each DRL. In case there is any DRL that is configured with a packed class which is not going to be necessary before the execution of the class C_i, and the DRL is free (is not being reconfigured neither is executing the current event) then the DRL is reconfigured to the packed class where the class C_i belongs.

4. HW/SW Partitioning for Dynamically Reconfigurable Architectures

As introduced in the previous section, a set of independent DE classes C = (C_1, C_2, ..., C_L) is the input to the HW/SW partitioning algorithm. The partitioning algorithm throughout its execution will work with two subsets (C^{HW} and C^{SW}):

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

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

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

where:

- AET_i^{HW} stands for Average Execution Time for a hardware implementation of the DE class C_i.
- AET_i^{SW} stands for Average 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.
- NO_i stands for Number of Objects of this class.

An important comment is needed for AET_i^{HW} and AET_i^{SW}. These estimators are static estimators, which only give information about the execution time for a concrete event processing. They do not take into account the dynamic (random) behavior of the event stream. And even more important is the fact that 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. Without taking into account the features of the reconfigurable architecture, the HW/SW partitioning could give really poor results.

Let's consider first the AET_i^{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 processing and reconfiguration. The following expression represents how 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.

$$\overline{AET_i^{HW}} = AET_i^{HW} + \alpha_R \cdot \left(T_R - EW \cdot \overline{T_{EXE}}\right) \qquad \textbf{(1)}$$

where:

- α_R is the reconfiguration probability, which 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 (2).

$$\alpha_R = \begin{cases} 0 & if \quad |C^{HW}| \le DRL \\ \dfrac{|C^{HW}| - DRL}{|C^{HW}|} & if \quad |C^{HW}| > DRL \end{cases} \qquad \textbf{(2)}$$

- T_R is the reconfiguration time needed for a DRL cell to change its context.
- EW is the size (in number of events) of the prefetch window. We experimentally obtained that the best EW is represented by expression (3).

$$EW = \begin{cases} DRL & if \quad |C^{SW}| \le 1 \\ DRL + 1 & if \quad |C^{SW}| > 1 \end{cases} \qquad \textbf{(3)}$$

- $\overline{T_{EXE}}$ is the average executing time for all 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.

$$\overline{T_{EXE}} = \frac{\sum_{i=1}^{|C|} AET_i^{\{HW,SW\}}}{|C|} \qquad \textbf{(4)}$$

$\overline{AET_i^{HW}}$ 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 number of classes in subset C^{HW}. The reconfiguration overhead depends on the reconfiguration probability (which will be higher when more classes are present in subset C^{HW}, when fixed the number of DRL cells) and the reconfiguration time which could be reduced using the prefetching technique. As the prefetching techniques is based on the fact of overlapping the execution on a DRL cell with the reconfiguration of another DRL cell, the reconfiguration time could be reduced by a factor which is proportional to the EW and average execution time of the set of classes C.

Let's consider now the AET_i^{SW} estimator, and how it is modified to take into account the features of the event stream, software processor and HW/SW communication strategy. This is shown in the following expression:

$$\overline{AET_i^{SW}} = AET_i^{SW} + \alpha_{COM} \cdot \overline{T_{COM}} \qquad \textbf{(5)}$$

where:

- α_{COM} is the HW/SW communication probability, 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 which will be executed by the software processor in the near future (within the EW). Its value is depicted in expression (5). This probability represents the case in which two events, that have to be executed into the software processor, are consecutive in the event stream.

$$\alpha_{COM} = \frac{|C^{SW}|}{|C|}\frac{|C^{SW}|}{|C|} \quad \textbf{(6)}$$

- $\overline{T_{COM}}$ is the average H/SW communication time. It represents the average access and transfer time through the system bus.

Our partitioning algorithm is resource constrained. The design constraints are object memory and class (DRL context) memory. That is, the total object state memory is limited and denoted by OSMA (Object State Memory Available). CMA stands for the total amount of Class Memory Available. DRLA stands for the DRL device available Area. 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.

$Max(|C^{HW}|)$, *such that*:

1. $\sum_{j=1}^{M} SVM_j \le OSMA, \ \sum_{j=1}^{M} DRLA_j \le CMA, \ and \ \ DRLA_j \le DRLA$
2. $\overline{AET_j^{HW}} < \overline{AET_j^{SW}}, \forall C_j \in C^{HW}$

The partitioning algorithm that we are proposing is divided in three main phases, which are: (1) obtaining an initial solution, (2) improvement of the initial solution, and (3) Class packing into reconfiguration contexts.

In order to perform this incremental approach, classes are labelled with an active *state*. We consider these 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 using C^{SW}. *Tagged_HW* means that class belongs to subset C^{HW}, but that the class could be moved to C^{SW}. *Tagged_SW* means that class belongs to subset C^{SW}, but that the class could be moved to C^{HW}.

4.1. Obtaining the Initial Solution

Obtaining an initial solution is addressed using a list-based partitioning algorithm. The idea of this approach is to give priority to the more time consuming DE classes when mapping to hardware, in order to minimize the total execution time. Thus, the set of input DE classes is sequentially ordered and more time consuming DE classes are prioritized using a cost function. The following cost function has been used.

$$F_i = \alpha \cdot (AET_i^{HW} - AET_i^{SW}) + \beta \cdot \frac{1}{NO_i} \quad \textbf{(7)}$$

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 cost function, are better than higher values. So, the sort function classifies values from lowest to highest. Parameters α and β are used as a trade-off between the speed-up and the number of objects for the class.

The algorithm obtains the initial sequentially sorted list after the cost function has been applied to all DE classes. Afterwards, it performs a loop, and tries to set as *tagged_SW* as many DE classes as possible while memory and DRL area constraints are met. Classes that do not meet the requirements will be set as *fixed_SW*. See [20] for details on this algorithm.

4.2. 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}. The movement of a class, from the subset C^{SW} to the subset C^{HW}, is mainly determined by the expressions introduced previously at the beginning of this section (expressions 1 to 6).

The pseudo-code of the proposed algorithm is shown in figure 2. The input to this algorithm is the sorted list of classes, where each class is labeled with a state. The

```
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);
    TexecHW_i = TexecHW(C_i);
    TexecSW_i = TexecSW(C_i);
    ExecTimeHW_i = TexecHW_i + α_R * (T_R - EW * ExecTime);
    ExecTimeSW_i = TexecSW_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;
}
```

Figure 2. Improvement of the initial solution.

classes are still ordered by the 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, for a brief period of time, labels the class as *tagged_HW*. After that, the algorithm evaluates the partitioning expressions 2, 3 and 4, 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 the expression 6 assuming that the class is assigned to SW.

At this point it is possible to evaluate expressions 1 and 5 for all the preceding classes in the list, which indeed means to evaluate the influence that will have moving a new class into the reconfigurable platform 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. This mapping process is done this way to prevent the algorithm to enter into a non-converging state.

As result of applying this algorithm there will be some classes labeled as *fixed_HW*, which indeed will be the classes finally mapped into the reconfigurable HW.

4.3. Class Packing into Reconfiguration Contexts

Once the improvement of the initial solution is finished, it is possible to perform a second type of optimization to minimize reconfiguration latency. The basic idea is to reduce the number of reconfigurations that are performed during execution. This objective can be achieved if all classes labeled as *fixed_HW* are packed into the minimum number of reconfiguration contexts. 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. And, 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 estimator. 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 this 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 [15]. 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. See [20] for details on this algorithm.

5. Experiments and Results

In this section, we explain a case study of our codesign methodology and partitioning algorithm. We center this case study in the software acceleration of broadband telecom networks simulation [18]. For our case study we have chosen the SONATA network [5]. Within this network it is possible to identify six different DE classes, which have been the input to our partitioning algorithm.

We carried out several experiments on top of our codesign framework [19]. Several experiments have been performed varying the DRL architecture parameters: number of DRL cells (0, 1, 2, 3, and 4) and its associated reconfiguration time (2000ns, 1000ns and 500ns). For all experiments we assume a DRL architecture where the object state and class memory have a size of 128Kx32 bits.

Figure 3 shows the several iterations carried out in the improvement of the initial solution phase of the HW/SW partitioning algorithm. This example assumes a DRL architecture of two DRL cells, each of them with a reconfiguration time of 1000ns. In this example, we also assume that all input classes are *tagged_SW*, and there is not any class *fixed_SW*. In the first iteration, class C1 is the first class *tagged_SW*, and after evaluating the partitioning expressions it is *tagged_HW* (grey colour). In the second iteration, class C2 is the first class *tagged_SW* and after evaluating the partitioning expressions, class C1 is *fixed_HW* (black colour) and class C2 is *tagged_HW*, and so on. A special comment is needed for iteration 6, where the algorithm stops its execution, because there is not any move. This is due to the fact that when trying to move C5 to hardware, the algorithm detects that a previously class *fixed_HW* (C2) should be set as a software class. Within our algorithm this possibility is not possible (as explained in the previous section), otherwise the algorithm will enter into a non-convergence problem (if this possibility was considered then algorithm would be again in the same state as in iteration 3).

Figures 4 show the total network simulation execution time when the number of DRL cells increases. A DRL=0 value means an all software simulation execution. For each reconfiguration time, we compare the results obtained when applying the algorithm presented in this paper (S algorithm), with the results given in [19] where DRL

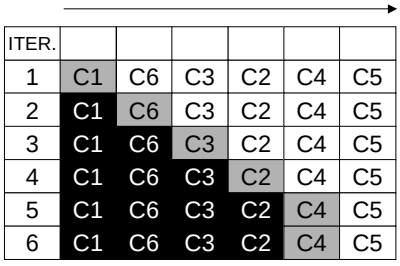

Figure 3. Partitioning Algorithm Iterations.

features are not taken into account (L algorithm) at the HW/SW partitioning level. Figure 4, shows the benefits given by the S algorithm. It does give better results even in the case of having a single DRL cell with a reconfiguration time of 2000ns. When applying the L algorithm it is seen that using a single DRL cell with a reconfiguration time of 2000ns, give worst results than an all software solution.

From figure 4, it is also seen that just having two DRL cells has a great impact into the performance. The important point here is the results given by the two partitioning algorithms. When having two DRL cells, it is possible to obtain almost the same results when using: (1) the S algorithm and slower reconfiguration time DRL cells, (2) the L algorithm and twice faster DRL cells. That is, although using faster DRL devices it is not guaranteed that best results will be obtained. The results do highly depend on the partitioning algorithm.

Finally, it can be seen that when increasing the number of DRL cells (three or four) both algorithms converge to the execution time obtained using an static hardware approach (this configuration would be the best possible one, because it means that there are six DRL cells, so there is no reconfiguration overhead). However, it can be seen that algorithm S converges with three DRL cells, while algorithm L converges with four DRL cells.

6. Conclusions

New HW/SW codesign methodologies and algorithms have to be developed, in order to take into account the features of new appearing DRL devices and architectures.

In this paper, we have presented a major contribution: a novel automatic HW/SW partitioning algorithm for dynamically reconfigurable architectures, which takes into account a configuration prefetching mechanism for reconfiguration latency minimization.

We have included this partitioning algorithm within our codesign framework and applied it to the software acceleration of telecom networks simulation. Several experiments have been carried out, and results demonstrate the benefits of our algorithm.

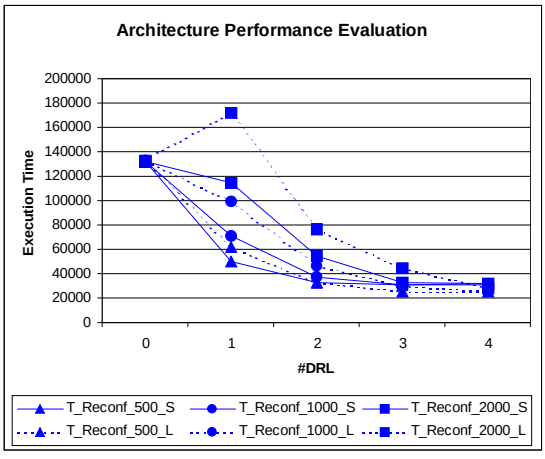

Figure 4. Comparison Results.

7. References

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

[2] http://www.xilinx.com/

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

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

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

[6] K. S. Chatta, R. Vemuri, "Hardware-Software Codesign for Dynamically Reconfigurable Architectures". Proc. of FPL'99. Glasgow, Scotland. September, 1999.

[7] R. P. Dick, N. K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems". Proc. Intl Conference on Computer-Aided Design. ICCAD'98.

[8] R. Ernst, J. Henkel, T. Benner, "Hardware-Software Cosynthesis for microcontrollers". IEEE Design and Test of Computers, vol. 10, no. 4, pp 64-75, Dec. 1993.

[9] J. Fleischman, *et al.*, "A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems". CODES'98, Seattle, USA.

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

[11] R. Gupta, G. De Micheli, "Hardware-Software cosynthesis for Digital Systems". IEEE Design and Test of Computers, vol. 10, no. 3, pp 29-41, Sept. 1993.

[12] R. W. Hartenstein *et al.*, "Two-Level Partitioning of Image Processing Algorithms for the Parallel Map-oriented Machine" CODES/CASHE'96, Pittsburgh, USA.

[13] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", ACM/SIGDA International Symposium on FPGA, pp. 65-74, 1998.

[14] B. Jeong *et al.*, "Hardware-Software Cosynthesis for Run-Time Incrementally Reconfigurable FPGAs". Proc. of Asia South-Pacific Design Automation Conf. (ASP-DAC'2000).

[15] F. J. Kurdahi, A. C. Parker, "REAL: A Program for Register Allocation", Proc. 24th Design Automation Conference, DAC'87.

[16] Yanbing Li *et al.*, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", Proc. 37th Design Automation Conference, DAC'2000.

[17] R. Maestre, F. J. Kurdahi, M. Fernandez, R. Hermida, "A Framework for Scheduling and Context Allocation in Reconfigurable Computing", Proc. ISSS'99.

[18] J. Noguera, R. M. Badia, J. Domingo, J. Sole, "Reconfigurable Computing: an Innovative Solution for Multimedia and Telecommunication Network Simulation". IEEE Proc. of the 25th Euromicro Conference. Milan. 1999.

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

[20] J. Noguera, R. M. Badia, "An Object-Oriented HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures", Research report UPC-DAC-2000-77. http://www.ac.upc.es/recerca/reports/. December, 2000.

[21] F. Vahid, "Modifying Min-Cut for Hardware and Software Functional Partitioning". Codes/CASHE'97, pages 43-48, Braunschweig, Germany, March 1997.