ARCHITECTURE AND APPLICATION PARTITIONING FOR RECONFIGURABLE SYSTEM DESIGN

K. Ben Chehida, M. Auguin, S. Raimbault

I3S, University of Nice Sophia Antipolis, CNRS

ABSTRACT

This paper presents a Genetic Algorithm (GA) based approach for Hardware/Software partitioning targeting an architecture composed of a processor and a dynamically reconfigurable datapath (FPGA). From an acyclic task graph and a set of Area-Time implementation trade offs points for each task, our GA performs HW/SW partitioning and scheduling such that the global application execution time is minimized. The efficiency of our GA is established through its application to a AC-3 decoder function and its performance is compared with a greedy algorithm.

1. INTRODUCTION

The recent improvements in size, flexibility and reconfiguration speed of FPGAs make this technology very attractive for low cost and high speed embedded system design. Connecting a reconfigurable device to a programmable processor in a single chip **[1, 2],** constitutes a very flexible and efficient architecture that can be used in a wide variety of embedded devices (for example, intelligent terminals or sensors such as a networked camera **[3]**).

Rapid development of embedded systems using this software/reconfigurable technology suffers from lack of advanced system level design tools which exploit efficiently the parallelism and the dynamic reconfiguration capabilities of the architecture. The aim of the project EPICURE[1] is to introduce a design methodology for dynamically reconfigurable computing platforms composed of a general purpose processor (*CPU*) and a dynamically reconfigurable datapath (*FPGA…*). From performance/cost estimates of implementations of tasks of the application on the processor and on the reconfigurable circuit, we have developed a partitioning tool which provides a mapping and a schedule of the tasks on the architecture.

The organisation of this paper is as follow. In Section 2 we formulate our problem to match the application and the architecture models. The description of our partitioning approach based on a genetic algorithm is provided in Section 3, and in Section 4 is outlined a greedy algorithm. Results and performance comparison on a AC-3 decoder task graph example are presented in Section 5. We conclude with Section 6.

2. PROBLEM FORMULATION

The target architecture is composed of a processor connected to a dynamically reconfigurable unit. This dynamic reconfiguration technology is investigated by numerous research groups (e.g. **[4]**,**[5]**) and would be very attractive for commercial products. Exploiting dynamic reconfiguration requires rather a coarse grain parallelism to reduce the relative cost of reconfiguration and data transfers. The application model considered is a function or task level data flow graph specification. From this task graph, the goal of partitioning is to select whether to put each task into *SW* or *HW* such that the whole execution time is minimized. The partitioning algorithm takes into account the dynamic reconfiguration capabilities of the hardware unit. Currently, partial reconfigurations of the circuit are supported (the reconfiguration time depends on the number of Configurable Logic Blocks (*CLBs*) involved in the reconfiguration) without allowing overlaps between computation and reconfiguration. Complete reconfigurations of the circuit can be considered as well. The approach is based on a genetic algorithm that realizes a design space exploration by generating different mappings of the tasks on the processor and the FPGA. Evaluation of the execution time of the architecture for each mapping requires to define a schedule of the tasks including reconfigurations for context switching and data transfers between tasks. This evaluation is performed with a clustering heuristic **[6]**.

Each node of the acyclic data flow graph denotes a task that can be mapped to the *SW* or the *HW*. the amount of data (bytes) that must be transferred between two connected tasks is associated with each edge. A task can begin its execution when all its parent tasks and incoming edges have completed their executions.

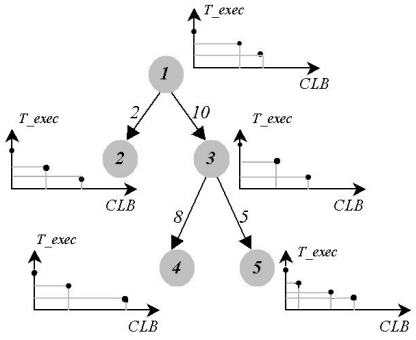

Fig. 1: Task Graph, Area-Time trade off curves

SW and *HW* runtimes of each task are estimated in terms of Area–Time trade off points. *SW* runtime performance is estimated through profiling and *HW* (*FPGA*) performance / area estimations are performed at the behavioral level. The number of implementation points can differ for each task depending on the exploitation of the available parallelism in the task **[7]**. Figure 1 shows an example of a task graph and the Area-Time implementation points for each task. The area is evaluated as a number of *CLBs*. A zero-*CLBs* implementation point of a task denotes a *SW* only implementation. We assume that a shared memory connects the processor and the reconfigurable unit with two data buses of fixed speed. This interface involves

[1] This project is supported by the French Ministry of Research and Education through the *Réseau National des Technologies Logicielles*. The partners of the project are CEA, Thales, Esterel Technologies, LESTER - Université de Bretagne Sud and I3S - Université de Nice Sophia Antipolis/CNRS.

communication times to read and write data if two tasks connected by an edge are placed one on the *HW* unit and one on the *SW* unit. The transfer time depends linearly on the amount of data bytes annotated on the edge **[6].** Let ρ_i be the number of bytes on edge e_i and λ_l be the number of bytes per packet supported by bus l. Let τ_l be the communication time of a packet on l and Ω_l be the access time per packet on that bus. Then the time to communicate the data on edge e_i is given by:

$$t_i = \left\lceil \frac{\rho_i}{\lambda_l} \right\rceil .(\tau_l + \Omega_l) \quad (1)$$

3. HW/SW PARTITIONING USING A GENETIC ALGORITHM

We model and solve our partitioning problem through a Genetic Algorithm. The encoding of any solution corresponds to the binding of each task to an implementation point. Our encoding method codes a chromosome *C* with an array of genes of length *N* where *N* is the number of tasks. Each gene *C(i)* is an integer representing a percentage. The maximum 100% value that can take *C(i)* is associated with the most *CLBs*-based expensive implementation of task *i*. The selected implementation point is the nearest point to *C(i)* on the area axis. All the solutions delivered by this encoding method are viable. The chromosome example presented in Figure 2 assigns tasks 1 and 4 to a *SW* implementation and all the others to the *HW*. Tasks mapped to *HW* have to be grouped into Contexts (or *Clusters*) to finally evaluate the effectiveness of the individual.

The N_{Indiv} individuals forming the initial population are randomly generated: the gene values of these chromosomes are randomly chosen between *0* and *100.*

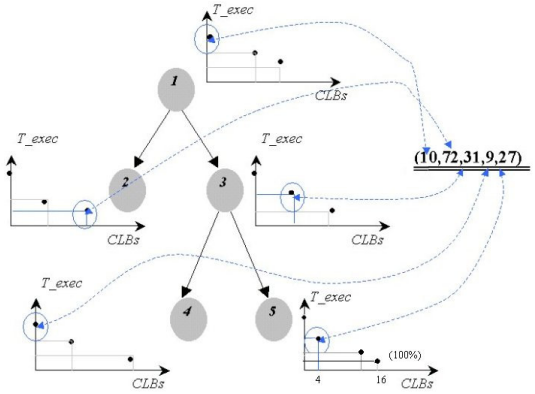

Fig. 2: Chromosome encoding

The fitness of every chromosome (solution) delivered by GA is evaluated allowing its ranking onto the current population. A solution is evaluated by its overall execution time including the reconfigurations for context switching and data transfers between tasks.

Communication times computation:

The chromosome structure provides an allocation of tasks to *HW* and *SW* so that preliminary communication times can easily be computed using (1). These communication times will be updated after clustering.

Contexts definition (Clustering):

We use a Clustering approach as addressed in **[6]** to group tasks in contexts. We first assign priority levels to tasks starting from the graph leafs. The priority level of a task is the longest path from the task to a leaf evaluated as computation and communication costs (Fig. 3). To reduce the schedule length, we need to decrease the length of the longest path by *clustering* tasks along it in order to reduce the communication costs along the path. The priority P_i of task i is computed considering the priority of its successors j and the communication time between i and j according to equation (2):

$$P_i = T\_exec(\tau_i) + SUP_{(j)}\,(P_j + T\_com(\tau_i, \tau_j)) \quad (2)$$

The cluster size S_{max} is limited to the maximum FPGA size (in practice, 80 to 85 % of the total number of *CLBs*).

Initially, all the tasks are sorted in the decreasing order of their priority levels. We pick the unclustered task $\tau_i(t_i,S_i)$ with the highest priority level, where t_i is the execution time and S_i the number of *CLBs* defined by the implementation pointed by *C(i)* in the chromosome, and mark it clustered. The available resources of the current cluster C_{curr} (initially to *Smax*) are decreased by S_i. This context building is iterated with tasks $\tau_j(t_j,S_j)$ assigned to *HW* while:

$$S_j \leq Ress(C_{curr}) \quad (3)$$

Else, a new cluster is created and the process is repeated until all the *HW* tasks are assigned to clusters. The reconfiguration time depends on the quantity N_k of logic cells (*CLBs*) needed for mapping the context k on the FPGA. Let T_F be the time for a full reconfiguration then the reconfiguration time per *CLB* is given by:

$$T_{reconf/CLB} = \frac{T_F}{S_{\max}} \quad (4)$$

We evaluate the reconfiguration time of the context k by:

$$T_{reconf}\,(k) = N_k \,.\, T_{reconf/CLB} \quad (5)$$

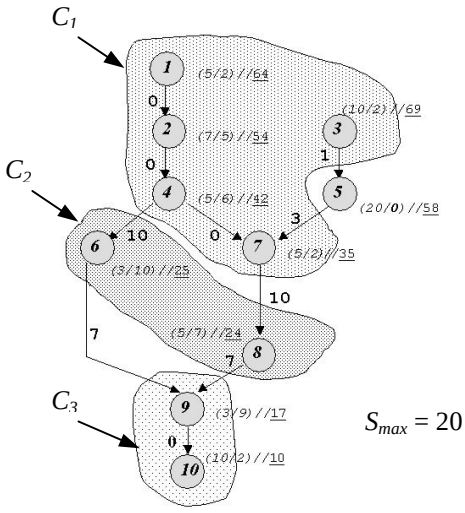

Fig. 3: Clustering and Communication time update

Tasks of the graph shown in figure 3 are annotated with the parameters $(t_i\ /S_i)//P_i$. The priorities are computed before communication times update. The GA allocates only task 5 on the processor. The reconfiguration times of the three contexts are $T_{reconf}\,(C_1) = T_{reconf}\,(C_2) = 10$ and $T_{reconf}\,(C_3) = 7$.

Communication time updates:

Once the contexts are defined, the algorithm updates the *intra-Context* (within a context) and *inter-Contexts* (between different contexts) communication times. *Intra-Context* communication times are set to zero.

Let $E_i(k)$ and $E_o(k)$ be respectively the incoming and outgoing edges of context *k.* for each edge $e_j \in E_o(l) \cap E_i(k)$ of contexts *l* and *k.* The communication time is updated by:

$$T_{com}(e_j) = Max\,(T_{com}(e_j),\ T_{reconf}(k)) \quad (6)$$

Where T_{com} is the communication time computed using (1) (see fig. 3). Hence, after updating communication times the global execution time for the example given in fig. 3 is 84: that is the cost of this solution.

Selection of solutions by GA is performed by the *Tournament* technique. A number ($N_{parents}$) of tournaments are performed, each one opposes a given number of individuals randomly chosen in the current population to finally select the fittest to be one of the parents allowed to reproduce.

Genetic operators are used on the $N_{parents}$ individuals selected by the *Tournament* technique to generate the $N_{children}$ solutions representing the new offspring. We perform a dynamic control on the number of the individuals created by each operator based on its efficiency over the previous generations. These operators are:

Mutation operators*:*

Mutation randomly selects a gene (or a set of genes) and changes its value. The mapping of a task can change from a *SW* to a *HW* implementation, *HW* to *SW*, or the task may remain in *HW* but using a different implementation point. Five mutation operators are used in our algorithm. Two operators *2-Opt* and *3-Opt* from the *k-Opt* family: neighbourhood search operators performing an exploitation process by local optimization.

The *Double Bridge* operator permits large jumps in the solution space assuring a pure exploration process. Two simple operators are also used: the *CutAndPaste* operator (we cut the chromosome at a random point and we swap the two portions) and the *Scramble* operator (the genes between two randomly chosen points are scrambled).

Crossover operators:

Two parent's chromosomes are cut at the same offset(s) (randomly set) from their starting points and the portions following the cut are swapped. Two crossover operators are used in our algorithm. A simple point *(1p-Cross)* and a double point *(2p-Cross)* crossover operator.

After generation of the new offspring, the renewal of the population is performed according to the *elitism* principle. *Clones* are not allowed in our renewal procedure because they can invade the whole population leading to a genetic drift. When a number of generations N_{gen} has passed without improvements of the best individual, GA halts and displays the best encountered solution. The user-specified parameters of this algorithm are N_{Indiv} (initial population size), $N_{children}$ (offspring size) and N_{gen} (termination condition: number of populations without improvements).

4. GREEDY PARTITIONING ALGORITHM

The partitioning problem can be addressed also using scheduling heuristics. The greedy algorithm presented in **[8]** targets a multiprocessor architecture. We have adapted this algorithm to deal with an architecture composed of a processor connected to a dynamically reconfigurable unit. Compared to the original algorithm, we use an ALAP scheduling and a backward critical path length evaluation. This ALAP scheduling algorithm operates backward through the graph, starting from the terminal tasks. It considers at each step the set of tasks that have their successors allocated and scheduled. The algorithm selects from this set the most time critical task, whatever the hardware or software context available for this task.

The most critical task is allocated and scheduled to the context that minimizes the estimated distance (in time) between the end of execution of the task and the earliest start time of the initial tasks of the graph (figure 4). Distances from each task to initial tasks of the graph are calculated before partitioning using a critical path length evaluation recursive algorithm.

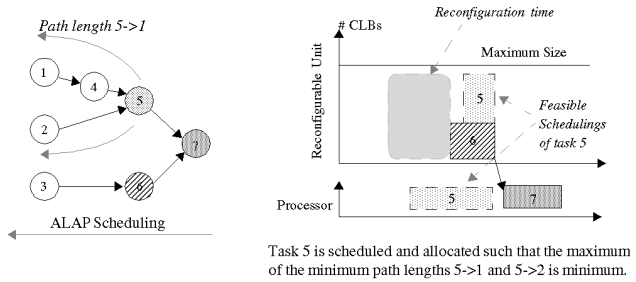

Fig. 4: Scheduling and allocation in the greedy algorithm

These path lengths are evaluated for every implementation points available for the task. Path length evaluation and task scheduling take into account communication times and reconfiguration times when there are not enough free *CLBs* to allocate a new task in the reconfigurable unit.

5. EXPERIMENTAL RESULTS

In this section results of the genetic algorithm for HW/ SW partitioning are compared to those given by the greedy algorithm. Our partitioning algorithms (GA and Greedy) are implemented in C++ on an Ultra Sparc 5 Unix workstation.

The benchmark used to evaluate the result quality of our partitioning algorithm is the AC-3 decoder application. It includes functions such as Inverse Discrete Cosine Transform (IDCT), Bit Allocation (BA), Decoupling (DC).A simplified task graph of this application is presented in fig. 5.

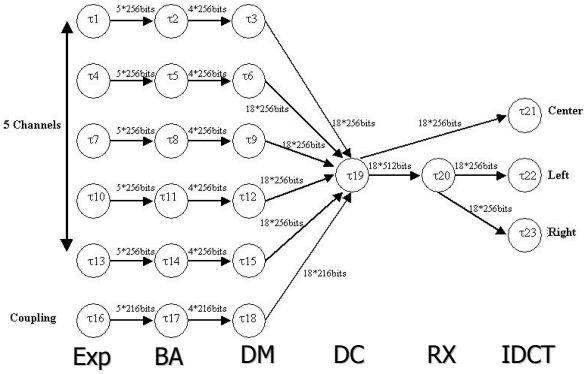

Fig. 5: A simplified AC-3 decoder task graph

The deadline given in the AC-3 norm is 5.33 *ms.* A *SW* only implementation on a simple DSP processor leads to an execution time of 5.57 *ms*. So we need to accelerate some portions of the application on *HW* to fit the deadline constraint. We don't target a specific commercial FPGA component, so we consider its attributes as parameters of the whole architecture: the FPGA total size (in terms of number of *CLBs*), the reconfiguration time per *CLB*, the buses speed and width and the memory access time. We fix the buses speed and width (bus 1 between the processor and the interface: λ_1 = 128, τ_1 = 10 *ns;* bus 2 between the FPGA and the interface: λ_2 = 256, τ_2 = 15 *ns*) and the memory access time Ω = 2 *ns.*

GA is executed with an initial population size N_{Indiv} of 600 and an offspring size $N_{children}$ of 200. The GA terminates when N_{gen} = 100 generations have passed without improvements of the best solution. Towards the end of the run, a convergence is observed as displayed in Figure 6. This figure shows the evolution of the best individual's cost and the mean cost over several generations. The CPU run time on the Ultra 5 workstation of the GA on the AC-3 application is in the range of 4 to 6 *minutes* and 0.03 *s* for the greedy algorithm.

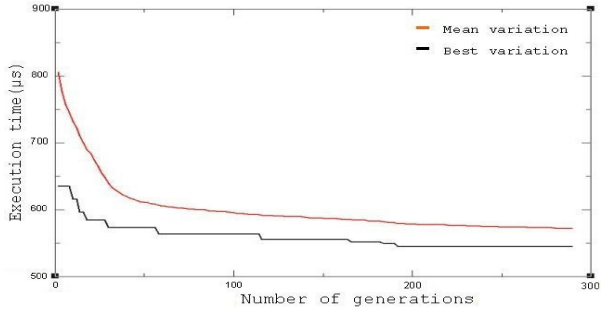

Fig. 6: Best individual's cost and the mean cost values

Given a reconfiguration time per *CLB* of 0.2 *µs* (ATMEL AT6K), we evaluate the influence of the FPGA total size on the entire execution time of the application for the GA and the greedy algorithms.

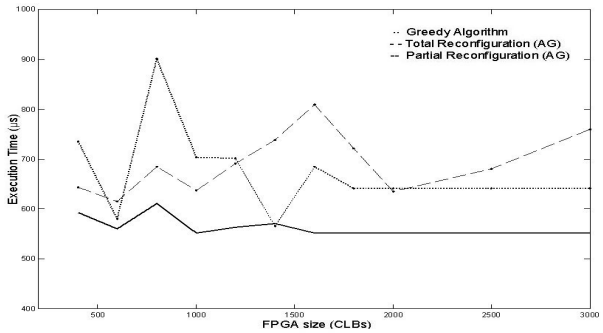

Fig. 7: Impact of FPGA size on execution time

Figure 7 shows that the increase in the size of the reconfigurable unit is useless since it does not provide any more speed up due to the too large reconfiguration time penalties (obviously for the total reconfiguration curve). The upper limits are 1600 *CLBs* for GA and 1800 *CLBs* for the greedy algorithm. Except the point of 1400 *CLBs*, the genetic algorithm is able to do better exploitation of the number of available *CLBs* giving lower execution times than the greedy algorithm. This figure illustrates also that the two algorithms cannot handle efficiently some sizes of the FPGA (in the range 600-800 for the GA and the greedy algorithm). The behaviour of the greedy algorithm and the clustering/scheduling algorithm in GA consists in exploiting the available *CLBs* in the FPGA to parallelize and to speed up executions of tasks. However, the allocation of a new task to a HW context leads to delay the executions of the other tasks already allocated in that context since the reconfiguration time of the context is augmented. This effect is not really handled in the algorithms as it is illustrated in fig.7: considering a total reconfiguration of the FPGA, we notice that the gap between the two curves given by the GA is not too large mainly for small FPGA sizes. Note that the results in fig.7 are obtained for a fixed granularity of *CLBs*. Considering an FPGA with a different granularity requires new estimations and partitioning.

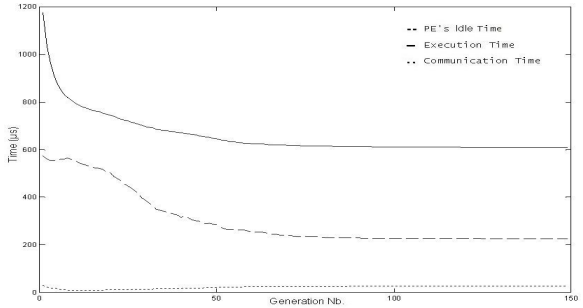

Fig. 8: The mean cost, Com. mean time and PE's mean Idle time values

As communications play an increasing role in today's SOC components, it is also interesting to see the variation of the mean communication time over several generations. Figure 8 shows that, after a short decrease, the mean communication time increases as the overall execution time is dropped. That means that the refinement procedure of the GA tries to exploit at best the available parallelism between the Processing Elements (*PEs*) leading to extra communication times that remain 'reasonable' comparing with the overall execution time. Figure 8 shows also the variation of the mean *PE*'s Idle time which is the mean over the individuals of a given generation of the Idle times on the two *PEs* (the FPGA and the processor). This mean time decreases drastically as the GA proceeds, which is also due to the refinement procedure capability to use at best the available gaps in the timing charts of the two *PE*s.

6. CONCLUSION

The iterative partitioning process of the greedy algorithm optimizes locally the allocation and the scheduling of the tasks on the units of the architecture. No backtracking is introduced in this algorithm avoiding to consider different allocation of tasks. The scheduling/clustering process in the fitness evaluation step of the genetic algorithm is also a greedy algorithm. These two algorithms must be tuned to take into account the delay introduced in the executions of tasks in a context due to the allocation of a new tasks in that context.In the genetic algorithm allocation and scheduling are separated: allocation is included in the design space exploration while scheduling allows the evaluation of each solution. The genetic approach for *HW/SW* partitioning with a dynamically reconfigurable unit is really effective compared with a scheduling-based greedy algorithm. The genetic partitioning approach provides an efficient assistance to the designer in the investigation of a balanced architecture. It allows various parameters of the architecture to be optimized, such as the number of available *CLBs* in the reconfiguration unit, the reconfiguration time per *CLB*, the data transfer rates on the buses, the relative speeds of the processor and the reconfiguration unit.

7. REFERENCES

[1] O. Brosch, J. Hesser, "ATLANTIS – A Hybrid FPGA/RISC Based Reconfigurable System", *Reconfigurable Architectures Workshop, Cancun; Mexico May 2000.*

[2] Excalibur backgrounder, *Altera Corporation, June 2000.*

[3] P. Six, "Designing Reconfigurable Networked Appliances using C++", *Vendor presentation at DAC 2001, june 18-20, Las Vegas.*

[4] C. Ebeling, D. Cronquist and P. Franklin, "Configurable Computing: The Catalyst for High-Performance Architectures", *Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 364-72, July 1997.*

[5] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh and E. M. C. Filho, "MorphoSys : An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications". *University of California, Irvine, CA 1999.*

[6] B.P. Dave, G. Lakshminarayana, N. Jha, "COSYN: hardware/software co-synthesis of embedded systems", *Design Automation Conference, Anaheim, 1997.*

[7] S. Bilavarn, G. Gogniat, J. L. Philippe. "Area Time Power Estimation for FPGA Based Designs at a Behavioral Level", *ICECS, Beyrouth, December 2000, Kaslik, Lebanon.*

[8] B.Jorgensen, P.Madsen "Critical path driven heterogeneous target architectures", 5th Workshop Codes / CASHE'97 ,15-19, Braunschweig, March 1997.