Design space exploration for dynamically reconfigurable architectures

Benoît Miramond and Jean-Marc Delosme
LaMI, Université d'Evry Val d'Essonne - UMR 8042 CNRS
{miramond, delosme}@lami.univ-evry.fr

Abstract

By incorporating reconfigurable hardware in embedded system architectures it has become easier to satisfy the performance constraints of demanding applications while lowering system cost. In order to evaluate the performance of a candidate architecture, the nodes (tasks) of the data flow graphs that describe an application must be assigned to the computing resources of the architecture: programmable processors and reconfigurable FPGAs, whose run-time reconfiguration capabilities must be exploited. In this paper we present a novel design exploration tool—based on a local search algorithm with global convergence properties—which simultaneously explores choices for computing resources, assignments of nodes to these resources, task schedules on the programmable processors and context definitions for the reconfigurable circuits. The tool finds a solution that minimizes system cost while meeting the performance constraints; more precisely it lets the designer select the quality of the optimization (hence its computing time) and finds accordingly a solution with close-to-minimal cost.

1. Introduction

Because of the quickly rising design cost of embedded systems and the difficulty to fulfill tighter and tighter time-to-market objectives, reuse between system generations must be intensified and systems must become increasingly flexible. Thus upcoming systems should exhibit heightened degrees of flexibility [7].
When performance constraints are stringent, programmable logic is of particular interest since it offers an alternative to standard application specific ICs of equivalent efficiency while being suited to more contexts.
Dynamically reconfigurable logic circuits (DRLCs), such as the *Virtex* family of products from *Xilinx*, when combined in a SoC with IP cores such as an ARM9x processor [1] or an AVR-type micro-controller [2], provide solutions that are both flexible and efficient. For reconfigurable systems, rapid system prototyping amounts essentially to programming a heterogeneous system consisting of processor(s) and dynamically reconfigurable logic. In comparison with a custom SoC, a reconfigurable SoC may be more expensive but permit a shorter time-to-market and hence potentially more units to be sold.
To perform periodic computations in a system with a DRLC, the specification of the application must be partitioned into run-time contexts. In order to meet the performance constraints in a multiprocessor system, both a schedule of the tasks assigned to the processors and a schedule of the run-time contexts must be determined. Fast development of embedded applications on reconfigurable architectures thus requires system level tools that exploit to the full the dynamic reconfiguration capabilities of the system resources and takes proper account of the reconfiguration times of the DRLCs.
This paper presents a technique for automatically mapping an application described by a precedence graph onto a heterogeneous architecture with at least one programmable processor and at least one reconfigurable circuit (RC). Section 2 summarizes the state of the art. The scheduling and partitioning problem is formulated in section 3, where the models of architecture and application are given. Our partitioning algorithm and its application to reconfigurable architectures are detailed in section 4. Experimental results are given in section 5, followed by our conclusions.

2. Related Work

To ensure that the performance constraints will be met in an embedded system with DRLCs, the computations within the application must be statically partitioned into those to be executed on the DRLCs and those to be executed on the processors (*HW/SW spatial partitioning*), the hardware tasks must be partitioned into run-time hardware contexts (*temporal partitioning*), and the software tasks, the run-time hardware contexts and the communications must be scheduled (*static software scheduling*). Therefore, finding a solution in order to run an application on a given reconfigurable architecture amounts to determining a solution to each one of the following three—coupled—subproblems: HW/SW spa-

1530-1591/05 $20.00 © 2005 IEEE

tial partitioning, SW scheduling, temporal partitioning.
Since we are interested in finding the solution with minimum cost that satisfies the performance constraints, we have developed a local search method—an adaptive version of simulated annealing—that explores the space of the solutions to these three subproblems and converges to a solution close to the global optimum.
Partitioning and scheduling techniques targeting reconfigurable architectures have been recently presented, often with differing objectives. Rakhmatov et al. [13] start from a control flow formulation of the application and present a solution of the spatial partitioning problem based on a maximum flow formulation. Kaul et al. [8] introduce an ILP-based method of resolution of the temporal partitioning problem. Maestre et al. have proposed in [10] a search technique for the order of execution of the contexts of a DRLC. The algorithm is greedy and finds a temporal partitioning and a scheduling of the contexts for a partially reconfigurable circuit.
In contrast with the above three papers, Chatha et al. deal in [5] with the hardware/software partitioning problem. They propose a partitioning method by migrating tasks between hardware and software and then applying a modified list scheduling algorithm. This approach accounts for all of the characteristics of the problem but considers neither the possibility of a partial reconfiguration of the FPGA nor the overlap of processor computation (software tasks) and hardware reconfiguration.
Studies such as the one of Noguera et al. [12] consider the partitioning and the dynamic scheduling on an architecture comprising a CPU and an RC connected via a bus. The RC is composed of an array of programmable logic blocks on which the tasks waiting to be executed are dynamically assigned. Concretely, these studies do not take into account task performance estimates nor total system execution time during the partitioning step, and then determine a dynamic scheduling that decides *at run time* the order of execution of the ready tasks. Consequently the tasks with the highest computational complexity are assigned to hardware with no regard to the global effect on the system. The complexity is thus relegated to the dynamic scheduler (a centralized control scheme requiring dedicated hardware), and the exploration of the space of solutions of the three coupled subproblems described earlier is not performed.
Finally, Ben Chehida and Auguin present in [6] an approach dealing with all three sub-problems. Spatial partitioning is explored with a genetic algorithm. For each such solution, temporal partitioning is effected by means of a clustering technique and is followed by global scheduling. The two algorithms employed after spatial partitioning are deterministic and generate a single temporal partitioning and a single schedule for each spatial partitioning solution. Our contribution, discussed in the next section, is to concurrently explore all three sub-problems and hence examine different solutions for a spatial partitioning of the tasks between hardware and software.

3. Problem formulation

We present in this section our formulation of the problem of mapping an application on a reconfigurable architecture following our partitioning methodology. First we present the model used to represent the application, then the type of architecture targeted and, finally, the definition of a problem solution in terms of the three subproblems, *HW/SW spatial partitioning*, *temporal partitioning* and *HW/SW static scheduling*.

3.1. Application model

The application is described by a precedence graph (hence acyclic) $\mathcal{G} =< \mathcal{V}, \mathcal{E} >$. The graph nodes $v_i \in V$ correspond to coarse grains. Each node is characterized by an index $i \in [1, N]$, where $N = |V|$, a functionality $F(v_i)$ (FFT, DCT, FIR filter, etc.), the number $C(v_i)$ of combinational logic blocks (CLBs) for the hardware implementation, and an estimate of the execution time on the processor, t_i^{sw}, and on the RC, t_i^{hw}. An edge e_{ij}, between nodes v_i and v_j, is characterized by the amount of data transferred, q_{ij}, and by a transfer time dependent on the communication link, t_{ij}. An example of representation of an application is given in Fig. 1(a).

3.2. Reconfigurable architecture

Our method is not restricted to a particular target architecture since it can explore the types and numbers of programmable and dedicated computing resources in the system in order to minimize the global system cost while satisfying performance constraints [11]. However, as far as this article is concerned, we restrict the application of our method to a generic architecture composed of a programmable processor and a dynamically reconfigurable unit, as expounded in [6]. Thus, the criterion to be optimized becomes here the execution time.
The proposed approach applies to a partially reconfigurable architecture insofar as the FPGA reconfiguration time depends on the number of CLBs needed to perform the desired computations. If an RC does not permit multi-context execution, *i.e.* initiation of several contexts in parallel, reconfiguration cannot be overlapped with computations on the FPGA. An objective is to decompose the computations effected on the RC into different contexts, and determine their sequential execution order. Moreover, each context can carry out one or more computational tasks, according to the numbers of CLBs required for the tasks. Finally, processor and RC communicate via a shared memory connected to each one by a bus [6]. The transfer time t_{ij} of data between a

task v_i executed on the processor and a task v_j executed on the circuit is estimated in terms of the size q_{ij} of the data and the bus transfer rate D. Performance estimates are made at compile-time, in particular communication latencies are statically evaluated as ordered transactions.

3.3. Formulation

The problem to be solved is to find a spatio-temporal mapping of an application, described by a task graph, onto the architecture described in section 3.2. We do this by exploring the space of feasible solutions, where a solution is defined by

- an assignment of the tasks on the processor (software) and the reconfigurable circuit (hardware),
- an assignment of the hardware tasks to time segments (logical contexts to be executed),
- an execution schedule for the software tasks and the hardware contexts, and
- an ordering of the transactions on the shared communication medium, i.e. a total order imposed on the transactions consistent with the task execution ordering.

Our method conforms to an object-oriented paradigm; the object concept is exploited at every level to achieve a high degree of flexibility and facilitate tool evolution. Class *Processing Element* belongs to the *Resource* class of the system, which is abstract and polymorphic. When several tasks are assigned to the same resource, their execution order on that resource depends on the resource type (processor, ASIC, reconfigurable IC). Indeed, at the (coarse) granularity level considered here, software task execution is sequential—the processor exhibiting actual parallelism only at a finer level. At the other extreme, the computations for several tasks could be performed with maximal parallelism on an ASIC dedicated to these computations. A dynamically reconfigurable circuit provides an intermediate solution since it can execute contexts both sequentially and concurrently; this corresponds to implementing a partial order for task execution. Therefore, a solution consists of

- a total order on the system processor,
- a globally total, locally partial (GTLP) order on the DRLC (see Fig. 1(b)) , and,
- if there were an ASIC in the system, a partial order on that circuit.

To define a total order for a processor-type resource, sequentialization edges (hence with zero computation-time) that enforce that order are inserted between the tasks assigned to the resource. The order is imposed by the search algorithm at each iteration, see section 4. The set of sequentialization edges for the processor-assigned tasks is denoted by E^{sw}. To impose the order $A \rightarrow C \rightarrow B$ on the processor in Fig. 1(b), the edge (C, B) is added. Sequentialization

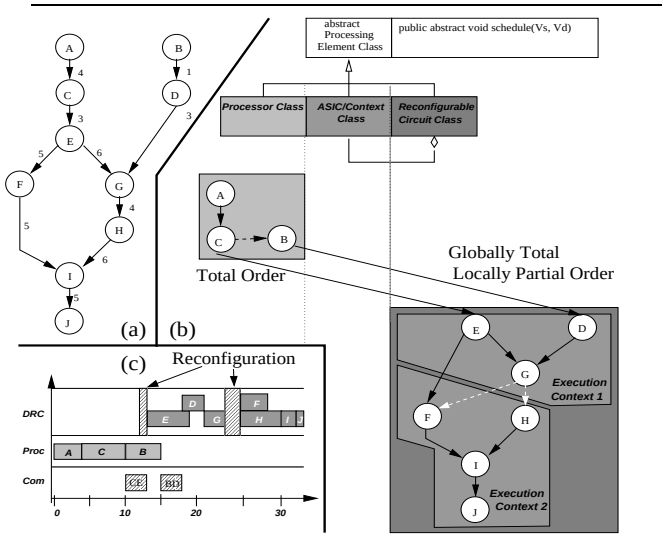

Figure 1. (a) Task graph example, (b) a spatio-temporal partitioning, and (c) its schedule.

edges are represented by black dashed arrows.
Within each context of a DRLC no edge needs to be added (Fig. 1(b)). On the other hand, to take into account the time of reconfiguration between contexts, the implementation of the abstract method *PE.Schedule* adds context sequentialization edges between the terminal nodes of a context and the initial nodes of the context that comes next. The set of all the context sequentialization edges is denoted by E^{hw}. The context sequentialization edges are represented by white dashed arrows; their weight depends linearly on the number of CLBs that must be reconfigured in the following context (partial reconfiguration case). Hence an object of *Reconfigurable* type contains

- the ordered list of its k contexts $L_c = [C_1, C_2, ...C_k]$,
- the reconfiguration time per CLB, t_R,
- the total number of CLBs in the circuit N_{CLB}.

Considering a context as a resource in itself, an object of *Context* type is a *Resource* that contains in addition :

- the list I of initial nodes of the context, whose immediate predecessors are all outside the context;
- the list T of terminal nodes of the context, whose immediate successors are all outside the context;
- the number of CLBs used in the context.

The weight of a context sequentialization edge $e \in E^{hw}$ is therefore equal to $t_e = t_R \times n_{CLB}$, where n_{CLB} is the number of CLBs used by the context of the edge's head.

4. Design-space exploration

Using the characterization of solutions of section 3, we describe in this section the exploration of the space of solutions. This exploration starts from a random initial solution

and then uses a version of the local search method based on the simulated annealing meta-heuristics to reach a solution close to the global minimum of the cost function. This method being iterative, it can be interrupted by the user at any time and will then return the current solution. The transition from one solution to the next is the result of the selection and application of the so-called "moves".

4.1. Local search algorithm

To perform a search that reaches a solution at most a few percent away from the global optimum, we have pursued the work carried out by Lam, who presented in [9] both an adaptive cooling schedule and a scheme for move selection that speed up significantly the convergence of simulated annealing. Adaptive SA employs a cooling schedule whose general form is independent of the optimization problem at hand. The problem's cost function is viewed as the energy of a dynamical system whose states are the problem's solutions. The schedule is obtained by maximizing the rate at which the temperature can be decreased subject to the constraint that the system be maintained in quasi-equilibrium. The adaptive nature of the schedule comes from the fact that it is expressed in terms of statistical quantities (mean, variance, correlation) of the system's cost function. Move generation affects the correlation between consecutive cost values and the adaptive schedule specifies how to control move generation to maximize cooling speed while satisfying the quasi-equilibrium condition. This version of simulated annealing has been used in VLSI circuit place and route tools [15]. We have recently improved on the estimation procedure and refined the move selection process [11]. These modifications have been validated on several types of problems, including graph partitioning and continuous function minimization.

4.2. Move definition

For our problem, a move consists in moving a node from one resource to another. Each move is defined by randomly selecting both the task v_s to move (source object) and a destination task v_d (destination object). A simple rule consists in randomly selecting an integer between 0 and N, the number of nodes in the graph, to determine the index of v_s and an integer between 0 and N, to determine the index of v_d, and effect accordingly one of four types of moves :

- (m_1) If $R(v_s) = R(v_d)$ and the resource R is a processor, the move is a modification of the (total) execution order on the resource coherent with the precedence relationships imposed by the task graph. For instance, if $v_s = B$ and $v_d = A$ in Fig. 1(b) , the total order is modified from A, C, B to B, A, C. If R is an ASIC or an RC context no move is performed.
- (m_2) If $R(v_s) \neq R(v_d)$, the move consists in switching the assignment of the source task to the same resource as the destination task. For instance, recalling that the contexts of a DRLC are considered as resources, if $v_s = C$ and $v_d = D$ in Fig. 1(b), the task C will be executed on the DRLC and, moreover, in execution context 1.
- (m_3) If 0 is selected for the source and if there is a task that is alone on one resource, then that resource is removed from the system and the task is assigned to the same resource as the destination task.
- (m_4) If 0 is selected for the destination, there is no destination node. This is interpreted as a request for resource creation (processor, ASIC, DRLC) with assignment of the source task to that resource.

Moves m_1 and m_2 allow the simultaneous exploration of spatial partitioning (m_2), temporal partitioning and context sequential ordering (m_2 with $R(v_s)$ and $R(v_d)$ of context type), and the sequential ordering of software tasks (m_1). Moves m_3 and m_4 would allow the exploration of the system architecture if it were not fixed a priori; however, in this paper, the architecture comprises one processor and one DRLC, hence the probability of generating a 0 is set to 0.

4.3. Move realization

Once a move has been selected, a search graph $\mathcal{G}' = < V, E \cup E_{sp} \cup E_{sr} >$ is deduced from the task graph by adding, temporarily, sequentialization edges. These edges are added, according to its type, by the resource on which the task is to be executed (see section 3.3). Consider the move corresponding to $v_s = G$ and $v_d = J$, which concerns the DRLC. In a context-type resource, only a partial order is imposed, and here v_s only has to precede H. The search graph $\mathcal{G}'$ is updated by calling the method for updating the source resource, $R(v_s).schedule(v_s, v_d)$, and the method for updating the destination resource, $R(v_d).schedule(v_s, v_d)$. Here, in the source resource (context $R(v_s)$), the temporary (dashed) edges (GF) and (GH) are deleted and the edge (DF) is created while, in the destination resource (context $R(v_d)$), no modification is needed since G goes from terminal node of the current context to initial node of the following context. In the destination resource, another context will be spawned if $n_{CLB}(R(v_d)) + C(v_s) > N_{CLB}$.
A move will not be performed if a cycle appears when the search graph is updated (detectable in $O(1)$ operations on the associated transitive closure matrix).

4.4. Solution evaluation

After each move, the performance of the new solution is evaluated by determining the longest path in the modified search graph (Fig. 1(c)). Exploiting the property that simulated annealing is a local search method, the longest path may in some cases be obtained incrementally by means of a Woodbury-type update formula [4].

5. Experimental results

In order to assess the effectiveness of our method, we have applied it to the motion detection application described in [6]. The application performs object labeling with a real-time constraint of 40 ms per image. Since a software implementation on an ARM922 processor leads to an execution time of 76.4 ms, some portions of the application must be hardware-accelerated to meet the time constraint [6]. The target architecture consists of an ARM922 processor and a reconfigurable FPGA of the Xilinx Virtex-E family. The reconfiguration time per CLB is $t_R = 22.5\ \mu s$. The performance estimates for the tasks of the application on this architecture were obtained in the EPICURE project [3]; several estimates are provided for each task on the FPGA, thus allowing exploration of the trade-off between number of CLBs and execution time. For each function (node) in the application, multiple implementations have been effectively synthetized. The node implementations considered form a set of dominant solutions in the area-time domain, each being characterized by the associated number of CLBs and the corresponding execution time. During exploration, SA chooses for each node implemented in hardware one of its implementations (5 or 6 synthesized solutions per function). Our method explores solutions characterized by a spatial partitioning, a temporal partitioning and a sequential ordering of the FPGA contexts. Results from a typical example of exploration of the solution space are presented in Fig. 2. The FPGA size is 2000 CLBs for this run. Both the execution time in ms and the number of FPGA contexts allocated at each iteration are plotted; the number of contexts ranges from 1 to 8 for the example. The initial solution is generated with a random hardware/software partition. A random number of tasks are moved, one by one, to the reconfigurable circuit. A new context is created when the capacity of the last context is exceeded. In the example of Fig. 2, only 9 (among 28) tasks are assigned to hardware; they require 995 CLBs, hence only one context. The execution time of this initial solution (67.9 ms) exceeds the 40 ms constraint; this poor performance is due to the excessive communication times resulting from the random nature of the initial partition. To illustrate the optimization method, the first 1200 iterations are performed at infinite temperature and the solution space is then broadly explored—the execution time ranging from 35 ms to 70 ms and the number of contexts from 1 to 8—while the average performance shows no improvement, as expected. When the method is activated and adaptive cooling starts, the execution time falls quickly below the 40 ms constraint. Between iterations 1500 and 3000 the number of contexts grows, as the relatively fine level of improvement of the partition of the tasks between contexts is then reached and explored. The final, frozen, configuration, has an execution time of 18.1 ms, well below the constraint, and uses 3 contexts. The example shows that the

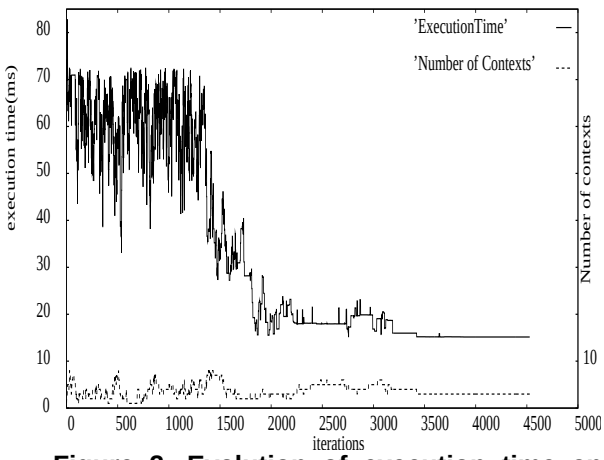

Figure 2. Evolution of execution time and number of contexts in a typical run.

performance constraint can be satisfied with an FPGA device of size 2000 CLBs. We examine next the impact of the device size (number of CLBs) on the performance that can be achieved. As a byproduct of this study we determine the size of the smallest device for which the 40 ms constraint is attained. Fig. 3 presents the results of our experiments using simulated annealing for FPGA sizes ranging from 100 to 10000 CLBs. Each value is an average of the results obtained for 100 runs. For each of the device sizes considered, the average values of the execution time, the reconfiguration time and the number of contexts are shown. As the size increases, the execution time drops quickly once the number of CLBs is large enough for a context to hold more than one task to be executed, since parallelism is then provided within the hardware. A minimum execution time is reached for about 800 CLBs and, as size increases further, execution time grows slowly and reaches a plateau around 5000 CLBs. Indeed, from this size up, all the hardware tasks can be executed in a single context. For such large devices, the optimisation method starts from random solutions with one context and an execution time exceeding 75 ms to finally reach solutions with a single context as well but whose execution time is more than halved. For small devices, holding from 400 to 1500 CLBs, the small context size leads to solutions with a large number of contexts (up to 10), number which drops steadily as size increases. Because of the compensation between number and size of contexts, total reconfiguration time remains roughly constant. Our best solutions improve on those obtained in [6], whose execution time is 28 ms. Moreover a run of our method takes less than 10 seconds, to be compared to the 4 minutes of the genetic algorithm used in [6]. In that paper, the population size is 300, hence even if it was reduced to 100, the method would still be an order of magnitude slower than ours.

Although the example seems simple, the solution space is

big. If the graph were a chain (hence with a total order), then, for 28 nodes, 2 changes of context would give 378 combinations and 6 changes 376,740 combinations. However there are many total orders possible for the actual 28-node example. Accounting just for the first 20 nodes, which form, in this case, a 7-node chain followed by a 7-node chain in parallel with a 6-node chain, there are 1716 total orders.

In fact the 6-node chain is followed by a 2-node chain in parallel with one node (3 orders) followed by 5 nodes, hence the 28 nodes form a 7-node chain followed by a 7-node chain in parallel with one of 3 14-node chains. Thus there are $3\begin{pmatrix} 21 \\ 7 \end{pmatrix}$ total orders for the example, i.e. 348,840 orders, therefore for 2 changes of context there are 131,861,520 combinations and for, say, 4 changes of context there are 7,142,499,000 combinations. This is assuming that all the processing is performed on the RC; there are many more combinations when the fact that some nodes may be executed in software is taken into account.

6. Conclusions

This paper develops an application, to reconfigurable architectures, of a general method of hardware/software partitioning optimization [11]. We have been able to quickly specialize our tool to this class of architectures, thus demonstrating the ease of incorporation of models of target architectures. This flexibility has been made possible by the object modeling effort on our application and architecture models, and by our work on acceleration of the simulated annealing algorithm.

Compared to other optimization methods, this approach does not require either tuning, as one can find in tabu search (tabu list sizes) or genetic algorithms (population size, reproduction rate...), or problem specific adaptation effort (chromosome encoding). With our method, adaptation to new models of computation and target arhitectures only requires the definition of simple simulated annealing moves. This study confirms that our method is not only flexible but also efficient since the execution times for the motion detection application improve on those available on the same benchmark. We are currently working on developing simulated annealing moves for systems described by multiple models of computation, including SDF and CFSM.

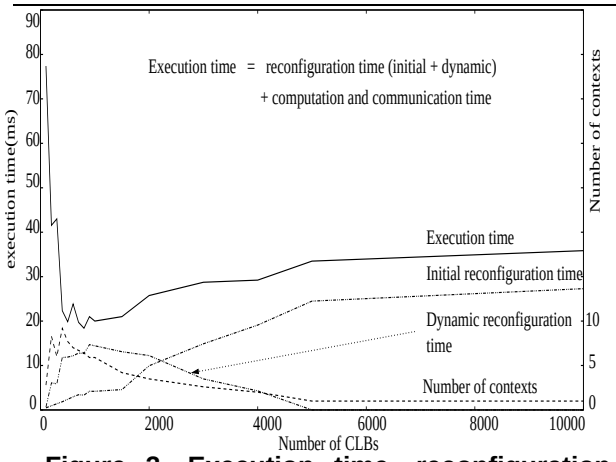

Figure 3. Execution time, reconfiguration times, and number of contexts vs. FPGA size.

References

[1] Altera. Excalibur architecture, www.altera.com, 2004.

[2] Atmel. AT94K architecture, www.atmel.com, 2004.

[3] M. Auguin and K. Ben Chehida et al. Partitioning and codesign tools & methodology for reconfigurable computing: the EPICURE philosophy. In *Proc. 3rd Workshop on Systems, Architectures, Modeling Simulation*, pages 46–51, July 2003.

[4] B. Carré. *Graphs and Networks*. Oxford Univ. Press, 1985.

[5] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In P. Lysaght et al., editor, *Field-Programmable Logic and Applications*, pages 175–184. Springer-Verlag, Berlin, 1999.

[6] K. B. Chehida and M. Auguin. HW/SW partitioning approach for reconfigurable system design. In *Proc. Intnl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 247 – 251, 2002.

[7] C. Haubelt, J. Teich, K. Richter, and R. Ernst. System design for flexibility. In *Proc. Design Automation and Test in Europe (DATE'02)*, pages 854–861, March 2002.

[8] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *Proc. of Design Automation Conf.*, pages 616–622, 1999.

[9] J. Lam. *An efficient simulated annealing schedule*. PhD thesis, Computer Science, Yale University, 1988.

[10] R. Maestre and F. Kurdahi et al. Kernel scheduling in reconfigurable computing. In *Proc. Conf. on Design, Automation and Test in Europe (DATE'99)*, pages 90–110, March 1999.

[11] B. Miramond. *Optimization method for hw/sw partitioning of systems described with multiple models of computation*. PhD thesis, Université d'Evry, France, 2003, (in French).

[12] J. Noguera and R. Badia. HW/SW codesign techniques for dynamically reconfigurable architectures. *IEEE Transactions on VLSI Systems*, 10(4):399–415, August 2002.

[13] D. N. Rakhmatov and S. B. Vrudhula. Hardware-software bipartitioning for dynamically reconfigurable systems. In *Proc. 10th Intnl. Worshop on Hardware/Software Codesign, CODES'02*, pages 145 – 150, May 2002.

[14] L. Shang and N. K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In *15th IEEE Intnl. Conf. on VLSI Design*, pages 345–352, Jan. 2002.

[15] W. Swartz. *Automatic layout of analog and digital mixed macro/standard cell integrated circuits*. PhD thesis, Electrical Engineering, Yale University, 1993.