Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 541067, 12 pages
doi:10.1155/2009/541067

Research Article

An ILP Formulation for the Task Graph Scheduling Problem Tailored to Bi-Dimensional Reconfigurable Architectures

F. Redaelli,[1] M. D. Santambrogio,[1,2] and S. Ogrenci Memik[3]

[1] *Politecnico di Milano, 20133 Milano, Italy*
[2] *Massachusetts Institute of Technology, Cambridge, MA 02139-4307, USA*
[3] *Northwestern University, Evanston, IL 60208, USA*

Correspondence should be addressed to M. D. Santambrogio, santambr@mit.edu

Received 10 March 2009; Revised 25 June 2009; Accepted 30 September 2009

Recommended by Lionel Torres

This work proposes an exact ILP formulation for the task scheduling problem on a 2D dynamically and partially reconfigurable architecture. Our approach takes physical constraints of the target device that is relevant for reconfiguration into account. Specifically, we consider the limited number of reconfigurators, which are used to reconfigure the device. This work also proposes a reconfiguration-aware heuristic scheduler, which exploits *configuration prefetching, module reuse*, and *antifragmentation* techniques. We experimented with a system employing two reconfigurators. This work also extends the ILP formulation for a HW/SW Codesign scenario. A heuristic scheduler for this extension has been developed too. These systems can be easily implemented using standard FPGAs. Our approach is able to improve the schedule quality by 8.76% on average (22.22% in the best case). Furthermore, our heuristic scheduler obtains the optimal schedule length in 60% of the considered cases. Our extended analysis demonstrated that HW/SW codesign can indeed lead to significantly better results. Our experiments show that by using our proposed HW/SW codesign method, the schedule length of applications can be reduced by a factor of 2 in the best case.

1. Introduction

Systems on a Chip (SoCs) have been evolving in complexity and composition in order to meet increasing performance demands and serve new application domains. Changing user requirements, new protocol and data-coding standards, and demands for support of a variety of different user applications require flexible hardware and software functionality long after the system has been manufactured. Inclusion of hardware reconfigurability addresses this need and allows a deeper exploration of the design space.

Nowadays, reconfigurable hardware systems, FPGAs in particular, are receiving significant attention. At first, they have been employed as a cheap means of prototyping and testing hardware solutions, while nowadays it is not uncommon to even directly *deploy* FPGA-based solutions. In this scenario, that can be termed *Compile Time Reconfiguration* [1], the configuration of the FPGA is loaded at the end of the design phase, and it remains the same throughout the whole application runtime. With the evolution of technology, it became possible to reconfigure the FPGA *between* different stages of its computation, since the induced time overhead could be considered acceptable. This process is called *Run Time Reconfiguration* (RTR) [1]. RTR is exploited by creating what has been termed *virtual hardware* [2, 3] following the concept of *virtual memory* in general computers. When an application bigger than the available FPGA area has to be executed, it can be partitioned in m partitions that fit in that area and these will be executed into numerical order, from 1 to m, to obtain the correct result. This idea is called *time partitioning*, and has been studied extensively in literature (see [4, 5]). A further improvement in FPGA technology allows novel devices to reconfigure only a portion of its own area, leaving the rest unchanged. This can be done using partial reconfiguration bitstreams. The *partial reconfiguration* time depends on the FPGA logic that needs to be changed. When both these features are available, the FPGA is called *partially dynamically reconfigurable*.

However, this scenario turns the conventional embedded design problem into a more complex one, where

the reconfiguration of hardware is an additional explicit dimension in the design of the system. Therefore, in order to harvest the true benefit from a system which employs dynamically reconfigurable hardware, existing approaches pursue the best trade-off between hardware acceleration, communication cost, dynamic reconfiguration overhead, and system flexibility. In these existing approaches the emphasis is placed on identifying computationally intensive tasks, also called kernels, and then maximizing performance by carrying over most of these tasks onto reconfigurable hardware. In this scenario, software mostly takes over the control dominated tasks. The performance model of the reconfigurable hardware is mainly defined by the degree of parallelism available in a given task and the amount of reconfiguration and communication cost that will be incurred. The performance model for software execution is on the other hand static and does not become affected by external factors. Starting from [6], HW/SW codesign researchers try to provide both analysis and synthesis methods specific for new architectures. Classical HW/SW Codesign techniques need to be improved to design reconfigurable architectures, because of a new degree of freedom. This new freedom resides in the design flow: the system can now dynamically modify the functionalities performed on the reconfigurable device. The second aim of this work is to present a model of the problem of scheduling a task graph onto a partially dynamically reconfigurable FPGA, taking into account the possibility of having both software and configurable hardware executions. The novelty of this work resides in the considered architectural model: Figure 1 shows the model. There is a processor and a reconfigurable part, each one with its own memory. The architecture is absolutely general and can be used also for a non-FPGA scenario. Furthermore, in an FPGA scenario, the processor can be within the FPGA or outside the device. What is really effective is that it has to be connected to the reconfigurable part with a channel. The channel is modeled as a bidirectional bus. Once this structure is ensured, the developed model works. With this architecture, when a hardware task needs data from the processor memory there is a latency due to this transfer.

This work provides the following contributions:

(i) an ILP formulation for the problem of minimizing the schedule length of a task graph on a 2D partially dynamically reconfigurable architecture to obtain optimal performance results,

(ii) a heuristic scheduler which takes into consideration *antifragmentation techniques* for general task graphs, a mix between classical deconfiguration policies and *antifragmentation* ones, and the use of out-of-order scheduling to better exploit *module reuse*,

(iii) an ILP formulation and a heuristic scheduler for the extended problems raised by introducing HW/SW Codesign in the initial problem.

This paper will focus on the scheduling of tasks on partially dynamically reconfigurable FPGAs in order to minimize the overall latency of the application. Section 2 proposes a description of the target architecture, describing the architectural solution on which the proposed model has been based. Section 3 describes the proposed ILP formulations and Section 4 the heuristic schedulers. Section 6 presents a set of experimental results comparing the ILP results and the heuristic ones to the results of the model presented in [7]. Finally, the conclusions regarding the proposed approach will be addressed in Section 7.

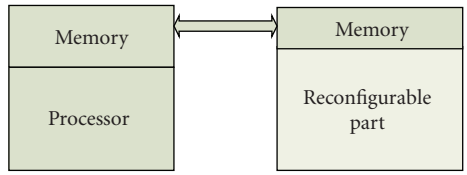
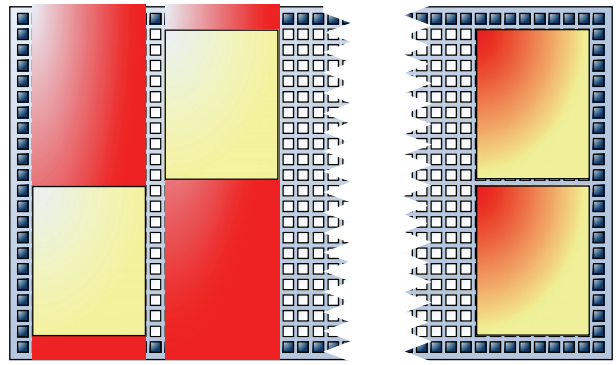


Figure 1: Considered architecture.



(a) (b)

Figure 2: 1D and 2D placement constraints versus 1D and 2D reconfiguration.

2. Target Device and Context Description

2.1. Architecture Description. The modern FPGA devices exploit a technology that allows powerful reconfiguration features. First, it is possible to perform dynamic reconfiguration. Second, emerging technologies allow 2D reconfiguration increasing the designer's degree of freedom. The payback for this increasing freedom is the necessity of new tools capable of exploiting these features in an effective way. The possibility of having a 2D partial dynamic reconfiguration may lead to both better solutions for well-known problems and feasible solutions for new problems.

Figure 2 shows the differences between 1D and 2D reconfigurations. In a 1D scenario, a module occupying only a column portion needs the reconfiguration of the entire column, while the same module in a 2D scenario can be reconfigured in much less area. In the case of Figure 2, in a 1D scenario the two modules would occupy 16 columns, while by exploiting 2D reconfiguration only 8 columns can be used. When a portion of the FPGA has to be reconfigured, a specific file called *bitstream* is needed: this file contains the information concerning the next behavior of that portion of FPGA.

The main characteristic of bitstreams is that they have a correlation with the operation they implement: once the

bitstream is defined, the operation is defined too, while given an operation, there could exist more than one bitstream implementing it. Therefore, it is possible to assign to each bitstream an attribute called *type* used to identify the operation implemented, the area occupied on the target architecture, and the time needed to be configured and to be executed by that bitstream. The latest FPGA technology, such as the Xilinx V4 [8, 9] and V5 [10, 11] families, allows 2D partial dynamic reconfiguration. At the same time, the complexity of the problem of minimizing the schedule length of an application by exploiting reconfiguration increases. Furthermore, thanks to multiple reconfigurator devices, concurrent reconfigurations can be performed, different modules can be configured simultaneously onto the FPGA.

Let us define a set of reconfiguration features that have to be taken into account to define the schedule. *Module reuse* means that two tasks of the same type have the possibility to be executed exactly on the same module on board, with a single configuration at the beginning. The *deconfiguration policy* is a set of rules used to decide when and how to remove a module from the FPGA. *Antifragmentation techniques* avoid the fragmentation of the available space on board trying to maximize the dimension of free connected areas. *Configuration prefetching* means that a module is loaded onto the FPGA as soon as possible in order to hide its reconfiguration time as much as possible.

2.2. Formal Problem Description. The 2D reconfigurable device is modeled as a grid of *reconfigurable units* (RU) by representing rows and columns as two sets $R = \{r_1, r_2, \ldots, r_{|R|}\}$ and $C = \{c_1, c_2, \ldots, c_{|C|}\}$: each cell represented by a pair (r, c), with $r \in R$ and $c \in C$, is made up of ρ_u CLBs. Columns and rows are *linearly ordered*, by which we mean that r_k is adjacent to $r_{k\pm1}$ on the FPGA, for every $1 < k < |R|$; the same property holds also for columns. The application is provided as a task graph $\langle \mathcal{S}, \mathcal{P} \rangle$, which is a Directed Acyclic Graph (DAG). $\mathcal{S}$ is the set of tasks in the graph, while $\mathcal{P}$ is the set of precedences among them. The tasks can be physically implemented on the target device using a set E of *execution units* (EUs), which correspond to different configurations of the resources (RUs) available on the device, therefore different bitstreams. In such a scenario, the reconfigurable scheduling problem amounts to scheduling both the reconfiguration and the execution of each task according to a number of precedence and resource constraints. Resource sharing occurs at EU level: different tasks may exploit the same RUs if they are executed in disjoint time intervals. Moreover, when they also share the same EU, they can be executed consecutively with a single reconfiguration at the beginning. Given any task s and any of its feasible EU implementations i, we assume that suitable algorithms exist to readily compute the latency $l_{i,s}$, the size r_i and the reconfiguration time d_i. Therefore, it is possible to define a function that specifies for each task:

(i) the EU on which it has to be executed,

(ii) the position on the FPGA where to place the selected EU,

(iii) the reconfiguration start time for the selected EU, or the possibility of reuse if possible,

(iv) the execution start time for the task.

In this work the general problem has been simplified: for each task type there is only one available EU. In this way the problem does not lose much in generality, but becomes easier to solve. Since each EU is associated with a bitstream and due to the former simplification, the model works with a number of bitstream types equal to the number of task types.

When HW/SW Codesign is considered, there is the necessity of mapping each task on either the processor or the FPGA. This introduces complexity in the problem solution. In this case the task type concept needs to be extended: each task has both a hardware implementation, that is, a bitstream, and a software one. These two different implementations share the same task type. Moreover, when HW/SW Co-design is condidered, multiple hardware implementations are considered: each task may have more than one bitstream to be implemented on, but the task type remains just one. Another issue, with HW/SW Codesign, is that there is the necessity of moving data between the memory of the processor and the memory of the reconfigurable device. This introduces latency that must be taken into account in the scheduling process. This latency depends on the amount of data needed to be transferred from a task to one of its children.

3. The ILP Formulation for the 2D Reconfiguration and Software Executions

We consider a 2D reconfiguration scenario, as presented in [12]: the sets C and R of RUs are respectively the set of columns and the set of rows of the FPGA. Therefore, all RUs have the same ρ_u (conventionally, $\rho_u = 1$). Each task must be assigned to a rectangular set of RUs and due to the possibility of having multiple reconfigurator devices, concurrent multiple reconfiguration may be exploited. We consider the following model. The starting scenario [12] has been extended to include the possibility of having a task executed also in software, we have to extend the classical *pure* reconfigurable architecture, considering also the presence of the processor, not only to take care of the reconfiguration itself, but also as processing element. Within this scenario, we can work with a processor host in the static area and a reconfigurable area, each one with its own memory. The architecture is absolutely general and can be used also for a non-FPGA scenario. Furthermore, in an FPGA scenario, the processor can be within the FPGA or outside the device. What is really effective is that it has to be connected to the reconfigurable part with a communication channel. Such a communication channel is modeled as a bidirectional bus. Once this structure is ensured, the developed model works. With this architecture, when a hardware task needs data from the processor memory there is a latency due to this transfer. The model considers also multiple hardware implementations for each task, to explore a bigger solution space. Since there are two separated memories, one for the processor and one for the FPGA, it is needed to transfer data

between them when a task requires it. This consideration introduces the concept of communication in the model. In the following are presented only those parts of the model that need to be added to the former one.

3.1. Constants.

(i) $a_{ij} := 1$ if tasks i and $j \in \mathcal{S}$ can be executed on the same bitstreams (by convention, $a_{ii} = 1$), if 0 they use different bitstreams;

(ii) $aM_{ij} := 1$ if task $i \in \mathcal{S}$ can be executed on bitstream $j \in M$, 0 otherwise;

(iii) $l_{ij} :=$ latency of task $i \in \mathcal{S}$ executed on an instance of bitstream j;

(iv) $lp_i :=$ latency of task $i \in \mathcal{S}$ executed on the processor;

(v) $d_i :=$ time needed to reconfigure an instance of bitstream i;

(vi) $c_i :=$ number of RU columns required by an instance of bitstream i,

(vii) $r_i :=$ number of RU rows required by an instance of bitstream i,

(viii) $cdl_{ij} :=$ time needed to transfer the data needed by task j from task i between the FPGA and the processor memories;

(ix) $NREC :=$ number of reconfigurator devices.

The scheduling time horizon $T = \{0, \ldots, |T|\}$ is large enough to reconfigure and execute all tasks. A good estimate of $|T|$ may be obtained via a heuristic.

3.2. Variables. Binary variables:

(i) $m_{ihkm} := 1$ if one instance of bitstream i is present on the FPGA starting from time h until time $h + d_i$ and cell (k, m) are the leftmost and bottommost used by i, 0 otherwise;

(ii) $b_{ihkm} := 1$ if task i is present on the FPGA at time h and cell (k, m) is the leftmost and bottommost used by i, 0 otherwise;

(iii) $p_{ih} := 1$ if task i is present on the processor starting from time h until time $h + lp_i$, 0 otherwise;

(iv) $tm_{ij} := 1$ if task i is executed on one instance of bitstream j, 0 otherwise;

(v) $cd_{ij} := 1$ if task j follows task i and there is the necessity to transfer data through the channel, 0 otherwise;

(vi) $\bar{t}_{ih} := 1$ if the reconfiguration of task i starts at time h, 0 otherwise;

(vii) $m_i := 1$ if task i exploits module reuse, otherwise 0;

(viii) $S_i^{\text{on}} :=$ arrival time of task i on the FPGA;

(ix) $S_i^{\text{off}} :=$ last time instant when task i is on the FPGA;

(x) $t_e :=$ overall execution time for the whole task graph.

3.3. Objective Function. The objective is to minimize the overall completion time of the task graph,

$$\min t_e. \tag{1}$$

3.4. Constraints. We used the *if-then* transformation (see [13]) to model the constraints marked with a*.

3.4.1. Task-to-Bitstream Assignment Constraints. Each task executed onto the FPGA must be executed on a particular bitstream:

$$tm_{ij} \geq m_{jhkm} + b_{i(h+d_j)km} - 1, \quad i \in \mathcal{S},\ j \in M, \quad k \in C, \quad m \in R, \quad h \geq 1 \wedge h \leq T - d_j, \tag{2}$$

$$tm_{jn} - tm_{in} \leq 2 - b_{ihkm} - b_{j(h-1)km}, \quad i, j \in \mathcal{S}, \quad n \in M, \quad k \in C, \quad m \in R, \quad h \in T. \tag{3}$$

When a task is executed in software, it does not need any bitstream, otherwise it needs exactly one bitstream:

$$\sum_{j \in M} tm_{ij} = 1 - \sum_{h \in T} p_{ih}, \quad i \in \mathcal{S}. \tag{4}$$

3.4.2. No-Board Constraints. When a task is executed on the processor, it cannot be placed also on the board:

$$\sum_{h \in T} \sum_{k \in C} \sum_{m \in R} b_{ihkm} \leq T \sum_{j \in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{5}$$

3.4.3. Non-Overlap Constraints. A task cannot be present on the FPGA with different leftmost and bottommost cells:

$$p_{ihkm} + \sum_{l \in C \setminus \{k\}} \sum_{j \in R \setminus \{m\}} b_{inlj} \leq 1, \quad i \in \mathcal{S}, \quad h, n \in T, \quad k \in C, \quad m \in R. \tag{6}$$

3.4.4. Single-Cell Constraints. A task cannot be present on the FPGA with different leftmost and bottommost cells:

$$b_{ihkm} + \sum_{l \in C \setminus \{k\}} \sum_{j \in R \setminus \{m\}} b_{inlj} \leq 1, \quad i \in \mathcal{S}, \quad h, n \in T, \quad k \in C, \quad m \in R. \tag{7}$$

3.4.5. Cell-on-the-Right-and-Top Constraints. The leftmost column of task i cannot be one of the last $c_i - 1$ columns; the same constraint has to be assumed for the last $r_i - 1$ rows:

$$b_{ihkm} = 0 \quad i \in \mathcal{S},\ h \in T,\ k \geq |C| - c_i + 2 \vee m \geq |R| - r_i + 2. \tag{8}$$

3.4.6. Arrival Time Constraints. The arrival time is the time in which a task comes on the FPGA. Since this time is the

first time step in which the associated p variables are set to 1, it must not exceed that time step:*

$$S_i^{\text{on}} \le h \sum_{k\in C}\sum_{m\in R} b_{ihkm} + h \cdot p_{ih} + |T|\left(1 - \sum_{k\in C}\sum_{m\in R} b_{ihkm} - p_{ih}\right), \quad i \in \mathcal{S}, h \in T. \tag{9}$$

3.4.7. Leaving Time Constraints. For each task i, the leaving time must not precede either the last instant for which b is 1 or the time p is 1 plus $lp_i - 1$:

$$S_i^{\text{off}} \ge h \sum_{k\in C}\sum_{m\in R} b_{ihkm} + (h + lp_i - 1)\, p_{ih}, \quad i \in \mathcal{S}, h \in T. \tag{10}$$

3.4.8. No-Preemption Constraints. A task is present on the FPGA in all time steps between the arrival and leaving time: this constraint works thanks to (9), (10), and (7) (No-preemption means that *once the configuration of a task begins* the configured task lasts on the FPGA until the end of its own execution). Equation (7) ensures that all the 1s of a particular task need to be on the same position of the FPGA for all the time that a task exists. This is because a task can perform its work, either be reconfigured or be executed, only if it is on the FPGA, and in specific only when its p variables are set to 1. Equation (9) ensures that the arrival time is lesser or equal to the first, in terms of time, 1 of a task. Equation (10) ensures that the leaving time is greater or equal to the last, in terms of time, 1 of a task. To ensure a task to exist on the FPGA in a single portion of time, the difference between the leaving time and the arrival time needs to be equal to the sum of all the 1s of that task. Since (7) ensures a single position, this constraint ensures that a task cannot be placed and removed and then placed again:

$$S_i^{\text{off}} - S_i^{\text{on}} + 1 = lp_i \sum_{h\in T} p_{ih} + \sum_{h\in T}\sum_{k\in C}\sum_{m\in R} b_{ihkm}, \quad i \in \mathcal{S}. \tag{11}$$

3.4.9. Precedence Constraints. Precedences must be respected:

$$S_i^{\text{off}} - l_j \ge S_i^{\text{off}}, \quad (i,j) \in \mathcal{P}. \tag{12}$$

3.4.10. Task Length Constraints. A task must be present on the FPGA at least for its execution time:

$$\sum_{h\in T}\sum_{k\in C}\sum_{m\in R} b_{ihkm} \ge \sum_{r\in M} (l_{ir} \cdot tm_{ir}), \quad i \in \mathcal{S}. \tag{13}$$

3.4.11. Reconfiguration Start Constraints. Each task has a single reconfiguration start time or none (if it exploits module reuse):

$$\sum_{h\in T} \bar{t}_{ih} = 1 - \sum_{j\in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{14}$$

Reconfiguration starts as soon as the task is on the FPGA, therefore, if a task needs to be configured on the FPGA, its reconfiguration will start at the first time step in which its p variables are set to 1, that is, its arrival time:*

$$-|T| \sum_{j\in M} tm_{ij} \le S_i^{\text{on}} - \sum_{h\in T} h\bar{t}_{ih} \le |T| \sum_{j\in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{15}$$

3.4.12. Reconfiguration Overlap Constraints. At most $NREC$ reconfigurations can take place simultaneously:

$$\sum_{i\in\mathcal{S}} \sum_{m=\max(1,h-d_i+1)}^{h} \bar{t}_{im} \le NREC, \quad h \in T. \tag{16}$$

3.4.13. Starting Time Constraints. This is the general formulation for this constraint, since multiple bitstreams must be considered, as for the nonoverlap constraints. This does not change the constraint formulation itself, but several instances of the constraint have to be created, one for each bitstream. Thus, their number increases, but they are written in the same way, with the obvious variable replacement. The starting instant is reserved, so that the FPGA is initially empty:

$$p_{i0km} = 0, \quad i \in \mathcal{S},\ k \in C,\ m \in R. \tag{17}$$

3.4.14. Single Processor Constraints. Only one processor is available:

$$\sum_{i\in\mathcal{S}} \sum_{l=h-lp_i+1}^{h} p_{il} \le 1, \quad h \in T. \tag{18}$$

3.4.15. Communication constraints. When two tasks i and j are linked by a precedence relation, the results of task i must be transferred to j. Due to the architectural model presented in Section 1, when two tasks are executed on the same device, either both on the FPGA or both on the processor, there is no need to transfer any data because the memories are local to the devices. When one tasks is executed on the FPGA and the other one on the processor, there is the necessity of moving the data from one to the other. For this reason a specific set of constraints have been developed:

$$\begin{aligned} -cd_{ij} &\le \sum_{l\in M} \left(tm_{il} - tm_{jl}\right) \ge cd_{ij}, \quad h(i,j) \in \mathcal{P}, \\ cd_{ij} &\le \sum_{l\in M} \left(tm_{il} + tm_{jl}\right), \quad h(i,j) \in \mathcal{P}, \\ cd_{ij} &\le 2 - \sum_{l\in M} \left(tm_{il} + tm_{jl}\right), \quad h(i,j) \in \mathcal{P} \end{aligned} \tag{19}$$

3.4.16. Definition of the Overall Latency.

$$t_e \ge S_i^{\text{off}}, \quad i \in \mathcal{S}. \tag{20}$$

3.5. Heterogeneous Case. So far, the proposed model describes the problem of scheduling a task graph onto a partially dynamically FPGA with homogeneous columns: all the FPGA cells have the same type. In the latest FPGAs devices it is possible to have columns of different types: CLBs, multiplexer, multiplier, BRAM, and so on. For this reason, a task can be implemented in different ways, due to which columns are involved in the synthesis process. Different implementations for the same tasks are now available, and the design space exploration must be more accurate: a bitstream for each one of these implementations must be created. Using the bitstream concept is very useful, because it is possible to use exactly the same ILP formulation for the extended problem. A preprocessing phase is needed, in order to avoid a bitstream to be placed on not compatible columns, and in general cells.

For each bitstream j, for each time h, for each column k, and for each row m, the variable m_{ihkm} is set to 0 if: given a couple (i,l) such that $i \geq k \wedge i \leq k + c_j - 1$ and $l \geq m \wedge l \leq m + r_j - 1$, cell (i,l) of the FPGA has a different type from cell $(i-k+1, l-m+1)$ of the bitstream. Once all these variables have been set, the proposal ILP can be applied.

4. Napoleon: A Heuristic Approach

From the results obtained through ILP solvers applied over the previous model, see Section 6.1, it is impossible to rely on it because of the huge amount of time needed. It is necessary to develop a fast technique that still obtains good results in terms of schedule length. A greedy heuristic scheduler has been selected as the best choice, and we developed it taking into account the experience achieved by writing the ILP model.

Napoleon is a reconfiguration-aware scheduler for 2D dynamically partially reconfigurable architectures. It is characterized by the exploitation of *configuration prefetching*, *module reuse* and *antifragmentation* techniques. Algorithm 1 shows the pseudocode of Napoleon. First, it performs an infinite-resource scheduling in order to sort the task set $\mathcal{S}$ by increasing ALAP values. Then, it builds subset RN with all tasks having no predecessors. In the following, RN will be updated to include all tasks whose predecessors have all been already scheduled (*available tasks*). SN, instead, is the set of scheduled tasks. As long as the dummy end task S_e is unscheduled, the algorithm performs the following operations. First, it scans the available tasks in increasing ALAP order to determine those which can reuse the modules currently placed on the FPGA. Each time this occurs, task S is placed in the position (k,m) which hosts a compatible module and is the farthest from the center of the FPGA. Unused modules can be present on the FPGA because Napoleon adopts *limited deconfiguration* as an antifragmentation technique: all modules are left on the FPGA until other tasks require their space, in order to increase the probability of reuse. The *farthest placement* criterium is also an antifragmentation technique, that aims at favoring future placements, as it is usually easier to place large modules in the center of the FPGA [14]. The execution

```
t ← 1
𝒮 ← computeALAPandSort(𝒮,𝒫)
RN ←findAvailableTasks(𝒮)
while S_e is unscheduled do
   SN ← ∅
   Reuse ← true
   for all S ∈ RN do
   (k, m) ← findFarthestCompatibleModule(S, t)
      if ∃(k, m) then
         schedule(S, t, k, m,Reuse)
         RN ← RN \ {S}
         SN ← SN ∪ {S}
      end if
   end for
   Reuse ← false
   for all S ∈ RN do
      (k, m) ← findFarthestAvailableSpace(S, t)
      if ∃(k, m) and ∃ free reconfigurators then
         schedule(S, t, k, m, Reuse)
         RN ← RN \ {S}
         SN ← SN ∪ {S}
      end if
   end for
   RN ← RN∪ newAvailableNodes(SN)
   t ← nextControlStep(t)
end while
return t_{S_e} + l_{S_e}
```

Algorithm 1: Algorithm Napoleon($\mathcal{S}$,$\mathcal{P}$).

starting time is tentatively set to the current time step t, but it is postponed if any predecessor has not yet terminated (see Algorithm 2 with Reuse = true). The task is also moved from the available to the just scheduled tasks (subset SN). When no further reuse is possible, Napoleon scans the available tasks in increasing ALAP order to determine those which can be placed on the FPGA in the current time step. The placement is feasible when a sufficient space is currently free or it can be freed by removing an unused module, and when a reconfigurator is available. If this occurs, the position for task S is chosen once again by the *farthest placement* criterium. The reconfiguration starting time is set to the current time step t and the execution starting time is first set to $t + d_S$ and then possibly postponed to guarantee that all the predecessors of S have terminated (see Algorithm 2 with Reuse = false). Thus, there might be an interval between the end of the reconfiguration and the beginning of the execution of a task (*configuration prefetching*). When all possible tasks have been scheduled, the set of available tasks RN is updated: Algorithm 3 does that by scanning the successors of the tasks in SN, which have just been scheduled, and determining the ones which must be added to RN. Finally, the current time step is updated by replacing it with the first time step in which a reconfigurator is available. Algorithm 1 shows the basic scheduling algorithm used, but for the sake of simplicity it does not report two optimizations to increase efficiency: if in the current time step all configured modules are in use, reuse is not possible and the first **for**

```
place(S, k, m)
if Reuse = true then
    t_S ← t
else
    t̄_S ← t
    t_S ← t + d_S
end if
for all S′ ∈ predecessors(S) do
    t_S ← max(t_S, t_S′ + l_S′)
end for
```

Algorithm 2: Procedure schedule($S, t, k, m,$ Reuse).

```
RN′ ← ∅
for all S ∈ SN do
    for all S′ ∈ successors(S) do
        if predecessors(S′) are all scheduled then
            RN ← RN ∪ {S′}
        end if
    end for
end for
return RN′
```

Algorithm 3: Function newAvailableNodes(SN).

loop can be skipped; if there is not enough available area to place any task, because no new placement is possible and the second **for** loop can be skipped.

4.1. HW/SW Extension. The scheduling algorithm schedule the task at the best possible time with respect to the schedules metric. It is simple to add the concept of HW/SW Codesign in this algorithm. Each time a task is considered to be scheduled, the algorithm computes the earliest time it can finish its execution on the processor, considering both precedences and communication delay. Then, if this time is lower than the minimum found on the FPGA device, the algorithm schedules the task on the processor, otherwise on the FPGA.

Figure 3 shows the schedule result obtained by using Napoleon in its HW/SW Codesign version. The characteristics for each task are listed in Table 1.

The number shown underneath the name of the tasks in Figure 3 represents ALAP values. Is it possible to see that Napoleon exploits the processor in an intensive way: it tries to schedule on the FPGA those task types that have more occurrences. This is done accordingly with the ALAP values and the available reconfigurable area: task B and task C share the same type, but their execution on the FPGA is not performed because it will lead to local delay in the schedule.

Table 1: Characteristics of tasks in Figure 3.

	Area	rec. time	HW time	SW time
A	2	2	1	3
B	3	3	1	2
C	3	3	1	2
D	2	2	3	4
E	4	4	2	4
F	2	2	1	3
G	2	2	3	4
H	2	2	1	3

5. Related Works

5.1. Reconfigurable Systems and Codesign Techniques. The VULCAN system [15] has been one of the first frameworks to implement a complete codesign flow. The basic principle of this framework is to start from a design specification based on a hardware description language, HardwareC, and then move some parts of the design into software. Another early approach to the partitioning problem is the COSYMA framework [16]. Unlike most partitioning frameworks, COSYMA starts with all the operations in software, and moves those that do not satisfy performance constraints from the CPU to dedicated hardware. More recent work [17] proposes a partitioning solution using Genetic Algorithms. This approach starts with an all software description of the system in a high level language like C or C++.

Camposano and Brayton [18] have been the first to introduce a new methodology for defining the Hardware (HW) and the Software (SW) sides of a system. They proposed a partitioner driven by the closeness metrics, which provides the designer with a measure on how efficient a solution could be, one that implements two different components on the same side, HW or SW. This technique was further improved with a procedural partitioning [19, 20]. Vahid and Gajski [19] proposed a set of closeness metrics for a functional partitioning at the system level.

In the context of reconfigurable SoCs, most approaches focused on effective utilization of the dynamically reconfigurable hardware resources. Related works in this domain focus on various aspects of partitioning and context scheduling. A system called NIMBLE was proposed for this task [21]. As an alternative to conventional ASICs, a reconfigurable datapath has been used in this system. The partitioning problem for architectures containing reconfigurable devices has different requirements. It demands a two dimensional partitioning strategy, in both spatial and temporal domains, while conventional architectures only involve spatial partitioning. The partitioning engine has to perform temporal partitioning as the FPGA can be reconfigured at various stages of the program execution in order to implement different functionalities. Dick and Jha [22] proposed a real-time scheduler to be embedded into the cosynthesis flow of reconfigurable distributed embedded systems. Noguera and Badia [23] proposed a design framework for dynamically reconfigurable systems, introducing a dynamic context scheduler and hw/sw partitioner. Banerjee et al. [24] introduced a partitioning scheme that is aware of the placement constraints during the context scheduling of the partially reconfigurable datapath of the SoC.

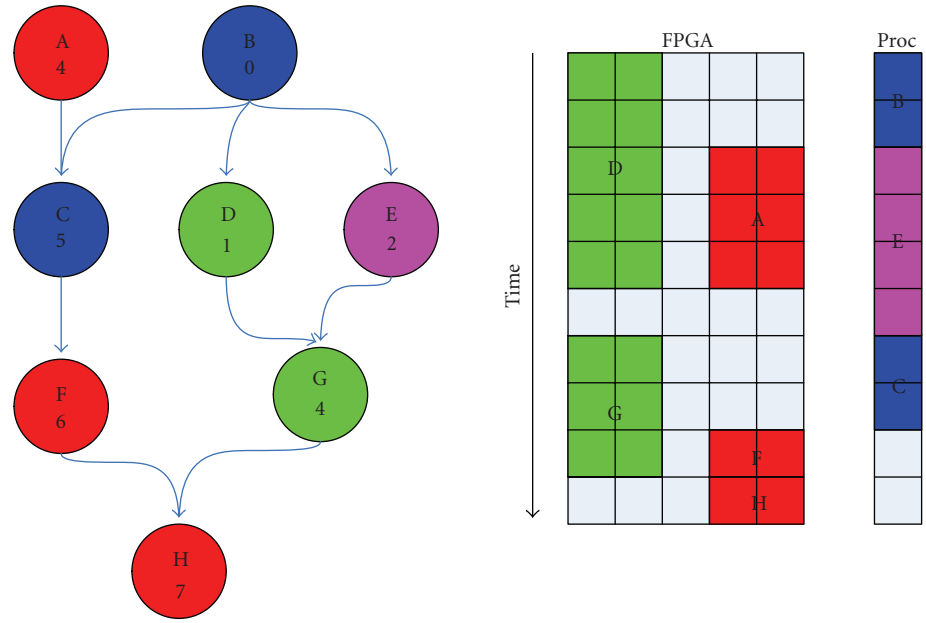


Figure 3: A scheduling example from Napoleon.

5.2. Scheduling Solution. The literature solutions for the considered scheduling problem do not exploit all the possible features of partial dynamic devices where HW/SW partitioning has been taken into consideration. Actual *hardware/software codesign* approaches find a partitioning of the original specification and then schedule the partitioned application on an architecture similar to the one described in Figure 4, where both the processor unit and the FPGA can execute one or more partitions. This kind of approach can be found in [25–28] and the scheduler used is almost always based on *list-based* scheduling algorithms with the priority function given by the mobility range of nodes. All these schedulers are static schedulers: the schedule of the task graph is done only one time before the real execution starts. An existing solution to the problem of partitioning and scheduling a task graph onto an architecture containing a processor and a partially dynamically reconfigurable FPGA [28] is shown in Figure 5.

In this architecture both the processor and the FPGA have their own memory. The FPGA memory is called *shared* just because it can be accessed by all the hardware modules eventually deployed on the FPGA. The authors present an exact approach based on an ILP formulation in order to show how the partial reconfiguration and the tasks placement issue have to be considered in the solution of this problem. This ILP formulation gives as a solution the complete schedule of the task graph and for each task state if it has to be executed in SW or in HW; moreover for the HW task it gives the time in which the reconfiguration has to start and in which is position of the FPGA and the execution starting time. The formulation takes into account dependences between tasks, *configuration prefetching* for the HW tasks, communication overhead between tasks placed in different partitions, and as an improvement also multiple task implementations and

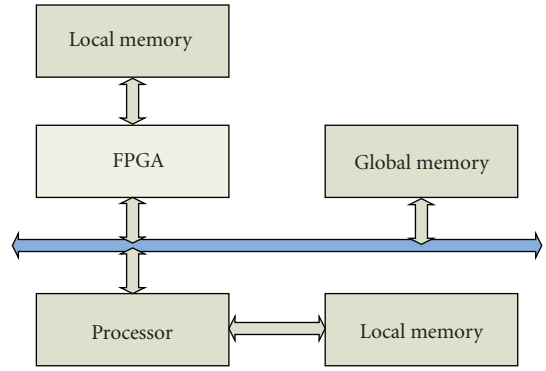


Figure 4: Architecture model for HW/SW Codesigned reconfigurable devices.

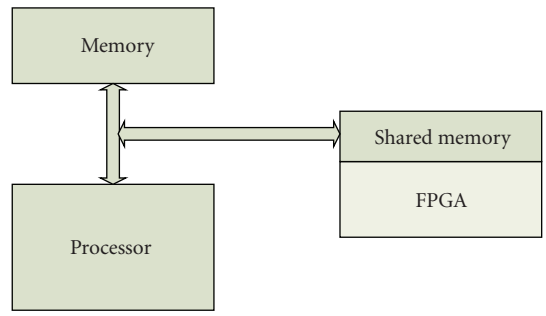


Figure 5: Two separated memories architectural model.

heterogeneity in the conformation of the columns of the FPGA. This is a model for the problem considered in this work, but *module reuse* is never considered.

A refinement of the *module reuse* concept is described in [29], where a solution to the problem of HW/SW Codesign of a task graph onto a partially dynamically reconfigurable architecture is given by an ILP formulation. The formulation is based on two concepts:

(i) early partial reconfiguration (EPR), which is similar to the concept of *configuration prefetching* and simply tries to reconfigure a hardware module as soon as possible onto the FPGA; the aim of this technique is to hide as much as possible the reconfiguration overhead;

(ii) incremental reconfiguration (IR), which is based on the concept of module reuse and states that if a new hardware module has to be placed over another already onto the FPGA, the configuration data that have to be configured are only the percentages that are not in common between the two modules.

The problem with this approach is that, in order to obtain a good IR, it requires the computation of all the possible differences between two task bitstreams and this takes a very long time. Since the developed model is required to be useful to realize a baseline scheduler, the model proposed in [29] is not suitable for the same aim: in an online scenario, all the difference bitstreams need to be in memory, thus, the total memory requirement is very large.

There is a set of works made by Teich et al., [30–32], where the authors present online heuristics for the scheduling/placement policy, taking into account the routing needed by the hardware modules to communicate among themselves. In these works, modules can communicate among themselves without sending data to the processor. This solution is good when tasks have to remain on the device for a long time and they need to frequently send data to other modules, while in a general case when the first issue is to free as soon as possible the reconfigurable area, this approach is not so interesting. Angermeier and Teich [33], present a heuristic scheduler for reconfigurable devices that minimize reconfiguration overheads. The problem with this algorithm is that it works for an architecture where tasks can be placed in a set of identical reconfigurable regions that communicate among themselves through a crossbar. Here the shape of the hardware modules must be well defined and it is impossible to place tasks bigger than a reconfigurable region. Furthermore, the complexity of the problem is reduced, and the scheduler works bad when tasks with great differences in size need to be scheduled on the device.

Some work has been done in the field of processing pipelines, where the whole application is a succession of large tasks that communicate with each other: the first task with the second one, the second with the third, and so on. Specific algorithms have been developed for managing this kind of an application, very important in image processing. In these works, such as [34, 35], the scheduling algorithm handles the HW/SW Codesign in a way that tries to minimize the overall execution time, the communication time among tasks, and to improve the throughput of the system.

6. Experimental Results

6.1. Pure Hardware Reconfiguration: ILP Results. This section compares the optimal results obtained by the models proposed in [7, 36] to the results of the model described in Section 3 considering input specifications characterized by only hardware and reconfigurable hardware elements. The evaluation has been performed by scheduling ten task graphs of ten nodes on an FPGA with 5 columns and 5 rows. The instances considered have small task graphs and few columns and rows because the problem is *NP-complete* and the computation time grows rapidly. Both task graphs and tasks have been generated by hand in order to verify different behaviors of the models: task graphs with tasks of different types, high *module reuse*, and high reconfiguration time. Task occupancy spreads among one column and one row, to four columns and 4 rows. The execution time is of the same order as the reconfiguration time. Furthermore, the number of dependencies goes from linear graphs to almost completely connected ones. The optimal schedule lengths are shown in Table 2. The first and second columns report the results of the proposed model, with 1 or 2 reconfigurators respectively, which correspond to a realistic scenario. The third and fourth columns report the results of the model proposed in [36], once again with 2 or 1 reconfigurators. The former is marked by an* because the original model does not accommodate more than one reconfigurator, and, hence, we have extended it to support multiple reconfigurators. The fifth column reports the results of the model proposed in [7], in which the number of reconfigurators is unlimited but the reconfiguration must immediately precede the execution and follow the end of the preceding tasks. It is possible to see that increasing the number of reconfigurator devices can improve the schedule length. This improvement is not assured because it is not always possible to hide completely the reconfiguration time. The model proposed in [36] is dominated by our proposed approach because it only allows 1D reconfiguration instead of 2D reconfiguration. Dominated means that every solution the model in [36] can find, our approach can find it too. Furthermore, our model can find and explore a bigger design space thanks to the possibility of having 2D reconfiguration. The *Fekete* model, [7], can obtain worse results because it does not exploit *module reuse* and *configuration prefetching* even if it has possibility of reconfiguring as many tasks as it needs at the same time. This is an interesting aspect of our proposed model: by modeling all the physical features recently introduced in reconfigurable devices, better results can be obtained.

6.2. Reconfigurable Hardware and Software Executions: ILP Results. This section compares the results obtained by the proposed HW/SW model, Section 3, and the one proposed in [28]. These same task graphs used in Section 6.1 have been scheduled. The only difference is that, for the HW/SW model, multiple hardware solutions have been taken into account, along with a software solution. The model described in [28] does not consider model reuse, so it is reasonable to expect a worse behavior with high possibility of reuse.

Table 2: ILP results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 [36]*	NRECS = 1 [36]	Fekete [7]
Ten1	15	17	15	17	18
Ten2	22	22	22	22	33
Ten3	16	16	16	16	25
Ten4	14	15	16	17	25
Ten5	21	21	21	21	28
Ten6	19	20	21	22	23
Ten7	20	20	20	20	28
Ten8	22	24	23	24	29
Ten9	26	26	26	26	32
Ten10	23	23	23	23	34

Table 3: ILP results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 [28]*	NRECS = 1 [28]
Ten1	13	13	14	14
Ten2	22	22	24	24
Ten3	14	14	18	18
Ten4	14	14	15	16
Ten5	16	16	17	18
Ten6	16	16	16	16
Ten7	16	16	20	21
Ten8	21	21	21	21
Ten9	22	22	22	22
Ten10	19	19	23	23

The resulting schedule lengths are shown in Table 3. The second and third columns report the results of the proposed model, with 2 or 1 reconfigurator devices respectively, which correspond to a realistic scenario. The third and fourth columns report the results of the model proposed in [28], once again with 2 or 1 reconfigurator devices. The former is marked by an* because the model had to be extended to support multiple reconfigurators. The considered FPGA has two different types of columns and so the tasks used to verify the models. Each one of these tasks can be executed to at least two different bitstreams; furthermore, the reconfiguration model considered is 2D.

It is possible to see that increasing the number of reconfigurator devices does not improve significantly the schedule length. The reason is that multiple branches of a task graph are executed on the FPGA and the other on the processor; to take advantage by using multiple reconfigurator devices, the tasks have to be on an area occupation very little with respect to the FPGA area; moreover, multiple concurrent branches have to be available. The model proposed in [28] is always dominated by the one proposed here because of the impossibility of having *module reuse*. The interesting aspect of the proposed model is that it can exploit in the best possible way the reconfiguration.

Comparing these results with the ones obtained without HW/SW Codesign, see [12], shows that the possibility of having a usable processor increases the schedule effectiveness. The schedule length decreases thanks to the possibility of having more parallelism: it is possible to execute tasks on the FPGA, in parallel, and also on the processor, saving reconfiguration time and FPGA area for subsequent tasks.

6.3. Reconfigurable Hardware and Software Executions: Heuristic Results. In this section we make a comparison between the the HW/SW Codesign model and the correspondent heuristic scheduler. These schedulers have been tested and compared on the following applications (useful to extract features from a large set of data) that have been selected from a popular data mining library, the *NU-MineBench* suite [37]:

(1) *variance* application: it receives as input of a single set of data and calculates the mean and the variance among the whole data set;

(2) *distance* application: it receives as inputs of two sets of data of equal size and calculates the distance between them;

(3) *variance1* application: it receives as input of a single set of data and calculates the mean and the variance among the whole data set. The tasks graph is different than the former variance application, since it involves different task types.

These applications are massive computing applications where there are few task types and a lot of tasks available at the same time. The developed schedulers are effective in this case due to good management of the FPGA area, the *module reuse* and *configuration prefetching* techniques. These applications are characterized by large number of tasks, grouped in two or three task types. Their graphs have the shape of a reverse tree: the same operation, task, must be done over the whole input set, then a new operation over the results, and so on. Each task does not occupy more than 5% of the reconfigurable device and its execution time is really short: two or three clock cycles. Furthermore, the communication time needed by data transfer is comparable with the execution time. The reconfiguration time is two orders of magnitude bigger than the execution time. Regarding the software implemantations, their execution time is one order of magnitude bigger than the hardware execution time. Because of the comparison among heuristics and ILPs, we choose to schedule task graphs with at most 32 tasks.

In this case, increasing the number of reconfigurator devices allows better solution in most of the cases. This is due to the fact that the parallelism can be handled in a more effective way. Napoleon, in its HW/SW Codesign version, reaches the optimal solution in just a case. With respect to the model in [28] and [28]*, the heuristic scheduler obtains always better, or at least not worse, results. This is because during the reconfiguration phases the processor can handle some tasks. In this algorithm, the reconfiguration time for each task is two orders of magnitude bigger than the execution time, thus, the scheduler decides to use the processor for a lot of tasks.

Table 4: ILP/heuristic results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 Napoleon HW/SW	NRECS = 1 Napoloen HW/SW
Ten1	13	13	13	14
Ten2	22	22	24	24
Ten3	14	14	15	17
Ten4	14	14	15	16
Ten5	16	16	18	18
Ten6	16	16	16	16
Ten7	16	16	19	22
Ten8	21	21	21	21
Ten9	22	22	22	22
Ten10	19	19	21	24
distance	520	520	520	520
variance	520	520	520	520
variance1	610	610	610	610

Table 5: ILP and heuristic execution time.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 Napoleon	NRECS = 1 Napoleon
Ten1	27 days	26 days	546 ms	477 ms
Ten2	31 days	25 days	413 ms	398 ms
Ten3	30 days	26 days	566 ms	513 ms
Ten4	27 days	25 days	578 ms	544 ms
Ten5	27 days	28 days	456 ms	401 ms
Ten6	34 days	31 days	670 ms	555 ms
Ten7	25 days	25 days	590 ms	487 ms
Ten8	23 days	20 days	602 ms	599 ms
Ten9	39 days	29 days	489 ms	433 ms
Ten10	36 days	37 days	716 ms	673 ms
distance	41 days	40 days	1,222 ms	1,001 ms
variance	35 days	36 days	1,321 ms	1,543 ms
variance1	45 days	41 days	1,561 ms	978 ms

Execution time for the experiments shown in Table 4 is shown in Table 5.

It is possible to see that the ILP solutions are too heavy to be used in real cases, while the heuristic approaches reach good solution in a reasonable time. It is noticeble that the ILP solutions for real applications do not scale bad: this is because the solver exploits standard searching methods that lead to "fast" solutions. Furthermore, due to the reconfiguration time, the processor is used for a lot of tasks at the beginning of the schedule, and this leads to an improvement in solution time.

7. Conclusion

The main goal of this work was to introduce a formal model for the problem of scheduling in a 2D partially dynamically reconfigurable scenario. The proposed model takes into account all the features available in a partial dynamic reconfiguration scenario. The results show that a reconfiguration-aware model can strongly improve the solution. The second goal of this work was to propose a heuristic reconfiguration-aware scheduler that obtains good results, with respect to the optimal one, but in a much shorter time. In fact an ILP solver takes a very long time to solve the problem exactly, while the heuristic algorithm reaches a good solution in a very short time. The results prove that *Napoleon* can be used effectively as a baseline scheduler in an online scenario. The next step in this work is to develop an online scheduler that starting from the results obtained by *Napoleon*, finds a feasible schedule and mapping at runtime.

References

[1] B. L. Hutchings and M. J. Wirthlin, "Implementation approaches for reconfigurable logic applications," in *Field-Programmable Logic and Applications*, W. Moore and W. Luk, Eds., Lecture Notes in Computer Science, pp. 419–428, Springer, Berlin, Germany, 1995.

[2] X.-P. Ling and H. Amano, "Performance evaluation of WASMII: a data driven computer on a virtual hardware," in *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe (PARLE '93)*, vol. 694 of *Lecture Notes in Computer Science*, pp. 610–621, Munich, Germany, June 1993.

[3] W. Fornaciari and V. Piuri, "Virtual FPGAs: some steps behind the physical barriers," in *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*, vol. 1388 of *Lecture Notes in Computer Science*, pp. 7–12, Orlando, Fla, USA, March-April 1998.

[4] 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 *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99)*, pp. 616–622, IEEE Computer Society, New Orleans, La, USA, 1999.

[5] J. M. P. Cardoso, "Loop dissevering: a technique for temporally partitioning loops in dynamically reconfigurable computing platforms," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 181.2, IEEE Computer Society, Nice, France, April 2003.

[6] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, 2003.

[7] S. Fekete, E. Koler, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 658–667, Munich, Germany, 2001.

[8] Xilinx, Inc., "Virtex-4 user guide," Tech. Rep. ug70, Xilinx Inc., March 2007, http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[9] Xilinx, Inc., "Virtex-4 configuration user guide," Tech. Rep. ug71, Xilinx Inc., January 2007, http://www.xilinx.com/support/documentation/user_guides/ug071.pdf.

[10] Xilinx, Inc., "Virtex-5 user guide," Tech. Rep. ug190, Xilinx Inc., February 2007, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[11] Xilinx, Inc., "Virtex-5 configuration user guide," Tech. Rep. ug191, Xilinx Inc., February 2007, http://www.xilinx.com/support/documentation/user_guides/ug191.pdf.

[12] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *Proceedings of the International Conference on ReconFigurable Computing and FPGAs (ReConFig '08)*, pp. 97–102, December 2008.

[13] W. L. Winston, *Introduction to Mathematical Programming: Applications and Algorithms*, Duxbury Resource Center, 2003.

[14] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Considering run-time reconfiguration overhead in task graph transformations for dynamically reconfigurable architectures," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 273–274, Napa, Calif, USA, April 2005.

[15] R. K. Gupta and G. D. Micheli, "Hardware/software cosynthesis for digital systems," *IEEE Design & Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[16] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[17] Y. Zou, Z. Zhuang, and H. Chen, "HW-SW partitioning based on genetic algorithm," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '04)*, vol. 1, pp. 628–633, IEEE Press, 2004.

[18] R. Camposano and R. K. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '87)*, pp. 324–326, 1987.

[19] F. Vahid and D. D. Gajski, "Closeness metrics for system-level functional partitioning," in *Proceedings of the 37th Conference on Design Automation (DAC '95)*, pp. 328–333, IEEE Computer Society, Brighton, UK, 1995.

[20] F. Vahid and D. D. Gajski, "Incremental hardware estimatin during hardware/software functional partitioning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 3, pp. 459–464, 1995.

[21] T. C. Y. Li, E. Darnell, R. E. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th IEEE/ACM Annual Design Automation Conference*, pp. 507–512, Los Angeles, Calif, USA, 2000.

[22] R. P. Dick and N. K. Jha, "CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 62–67, ACM Press, Los Angeles, Calif, USA, 1998.

[23] J. Noguera and R. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[24] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proceedings of the 42nd Annual Conference on Design Automation (DAC '05)*, pp. 335–340, ACM Press, Anaheim, Calif, USA, 2005.

[25] J. Noguera and R. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[26] J. Noguera and R. Badia, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 729–734, Munich, Germany, March 2001.

[27] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the 11th ProRisc Workshop on Circuits, Systems and Signal Processing*, Veldhoven, The Netherlands, November 2000.

[28] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, 2006.

[29] B. Jeong, *Hardware-software partitioning for reconfigurable architectures*, M.S. thesis, School of Electrical Engineering, Seoul National University, Seoul, South Korea, 1999.

[30] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 4, p. 134, April 2004.

[31] S. Fekete, T. Kamphans, N. Schweer, et al., "No-break dynamic defragmentation of reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 113–118, Heidelberg, Germany, September 2008.

[32] S. Fekete, J. C. van der Veen, J. Angermeier, D. Ghringer, M. Majer, and J. Teich, "Scheduling and communication-aware mapping of HW-SW modules for dynamically and partially reconfigurable SoC architectures," in *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS '07)*, p. 9, Zurich, Switzerland, March 2007.

[33] J. Angermeier and J. Teich, "Heuristics for scheduling reconfigurable devices with consideration of reconfiguration overheads," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, Miami, Fla, USA, April 2008.

[34] H. Quinn, M. Leeser, and L. King, "Dynamo: a runtime partitioning system for FPGA-based HW/SW image processing systems," *Journal of Real-Time Image Processing*, vol. 2, no. 4, pp. 179–190, 2007.

[35] H. Quinn, L. King, M. Leeser, and W. Meleis, "Runtime assignment of reconfigurable hardware components for image processing pipelines," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 173–182, April 2003.

[36] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 519–522, March 2008.

[37] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, J. Pisharath, G. Memik, and A. Choudhary, "MineBench: a benchmark suite for data mining workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '06)*, pp. 182–188, October 2006.