

MIT Open Access Articles

*Scheduling and 2D placement heuristics
for partially reconfigurable systems*

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Redaelli, F. et al. "Scheduling and 2D placement heuristics for partially reconfigurable systems." Field-Programmable Technology, 2009. FPT 2009. International Conference on. 2009. 223-230. © 2010 IEEE.

As Published: <http://dx.doi.org/10.1109/FPT.2009.5377683>

Publisher: Institute of Electrical and Electronics Engineers

Persistent URL: <http://hdl.handle.net/1721.1/60269>

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.



Scheduling and 2D Placement Heuristics for Partially Reconfigurable Systems

F. Redaelli¹, M.D. Santambrogio^{1,2}, V. Rana¹, S. Oğrenci Memik³

¹ *Dipartimento di Elettronica e dell'Informazione, Politecnico di Milano*

Milano, 20133 ITALY

rana@elet.polimi.it

fredaelli@elet.polimi.it

² *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology*

Cambridge, MA 02139, USA

santambr@mit.edu

³ *Department of Electrical Engineering and Computer Science, Northwestern University*

Evanston, IL 60208, USA

seda@eecs.northwestern.edu

Abstract—This paper proposes new scheduling and 2D placement heuristics for partially dynamically reconfigurable systems. One specific focus of this work is to deal with applications containing hundreds of tasks grouped in a few number of task types. Such a task graph structure is representative of data intensive high performance applications. We present three variations to our task management method that correspond to three possible system scenarios: (i) possessing complete static knowledge of task sequences, (ii) only having information on the maximum resource requirement by any task expected to be executed, and (iii) having no prior knowledge of any kind about the workload. Each variant of our scheduler addresses an architecture that best matches the needs of a particular configuration of the system. Together they form a complete set of techniques to serve partial dynamic reconfiguration of massively parallel computing systems.

I. INTRODUCTION

High performance computing has become essential for enabling many applications ranging from processing real-time stock trades to satellite data processing and scientific discovery in biology, astrophysics, and medicine. Traditional supercomputer architectures to tackle these tasks, which rely on software programmable processors are no longer sufficient. There is an increasing interest toward incorporating reconfigurable devices into these systems to achieve orders of magnitude performance enhancement while maintaining low performance per Watt ratings [1], [2].

It is not uncommon to expect a supercomputer to efficiently handle a diverse workload. Furthermore, several jobs of varying nature or copies of a job executing on different data sets can be launched on these systems. Reconfigurable computing resources in these systems would then be expected to serve this set of concurrent tasks. An important challenge would be to schedule this set of concurrent applications on partially dynamically reconfigurable devices (PDRDs) while fulfilling time and also area requirements for each of them. In this paper, we present three variations of a task management scheme, which performs scheduling of task requests and their physical

mapping onto a PDRD. Our proposed management scheme has features, which are specifically designed for executing data intensive high performance computing workloads. However, using the fundamental principles that we outline, the management scheme could be tailored for other workloads. Task scheduling and mapping on a partially dynamically reconfigurable device can be compared to the classical parallel machine scheduling. On the other hand the reconfiguration process also introduces time overheads that have to be taken into consideration. Specifically, there are two main issues that have to be considered in order to achieve high performance: *module reuse* and *configuration prefetching*. *Module reuse* is a technique that allows a newly launched task to use an already placed module with the same functionality. *Configuration prefetching* is a technique that allows the task scheduler to initiate the reconfiguration of a region of the device for a task to be executed in future as soon as possible, hiding partially or totally the reconfiguration time.

Our goal in this paper is to describe a management scheme that will perform the task scheduling and module mapping in an online environment while considering the challenges described above. We present three variations to our task management, where we consider three levels of knowledge about the workload: complete static knowledge of task sequences, knowledge on the maximum resource requirement by any task expected to be executed, and finally, no prior knowledge of any kind. Our proposed task management algorithms incur comparable latencies to state-of-the-art task schedulers to perform one scheduling and physical mapping decision. At the same time, considering the ultimate performance, measured as the total duration of execution for a workload, of our techniques are superior to existing schedulers. The remainder of the paper is organized as follows. Section II introduces the architectural model for this problem and an overview of how this model relates to the scheduling solutions we have developed. We present a summary of related work in Section III. Section IV describes the proposed schedulers in details.

Experimental results are presented in Section V, while Section VI summarizes our conclusions.

II. DESIGN ASSUMPTIONS

In the reconfigurable architecture used to validate our approaches there are two distinct regions: a static and a reconfigurable region. The static side includes a General Purpose Processor used to execute the reconfiguration management and a reconfigurator device, used to internally reconfigure the system at runtime. The reconfigurable area can be viewed as a set of reconfigurable slots used to map the desired modules at different times as decided by the task scheduler. We call these slots *Reconfigurable Regions (RRs)*. A *Reconfigurable Region* is a rectangular FPGA area where two or more tasks are going to be placed and routed (at design time) and configured (at runtime) according to the application implemented in the reconfigurable system. A bus-based communication infrastructure is used to support the connections between the static and the reconfigurable areas. Starting with the general template for the physical implementation described above we have designed three different customizations based on different communication paradigms. Applications are modeled as task graphs. Each task is defined by a Reconfigurable Functional Unit (RFU), that is, a technologically mapped netlist implementing the required functionality. These are characterized by area, reconfiguration time, and execution time. The size of the tasks has been chosen to be no more than the 20% of the reconfigurable area. We targeted our approach to deal with highly parallelizable applications, like datamining applications. An important feature of these tasks is that the execution time is really short compared to the reconfiguration time. Another feature of this applications is that there are a lot of tasks doing the same operation, and, possibly, few distinct operations. Communication is one of the most important issue. In our simulation model, we made the following assumptions. Tasks communicate with each other through the static side. The processor sends data to each task and reads back the results. Task to task communication would lead to a larger idle time for tasks, waiting to send results to subsequent tasks, and possibly causing system deadlock. Saving final data on a RAM-block inside the reconfigurable side leads to two problems: not enough blocks could be available and the complexity due to RAM reachability would lead to highly complex and bad implementation results. Communication for reconfiguration data has been embedded into the reconfiguration latency of each RFU. Finally, communication between the static side and the reconfigurable side is managed in different ways for each architectural model. We will elaborate more when we introduce each architecture.

III. RELATED WORK

Most of the work proposed in literature [3], [4] has been done in offline optimization for reconfigurable embedded systems. Different design-time scheduling techniques have been proposed to optimize the execution of parallel applications demanding the use of the same limited reconfigurable resource.

However, due to the design-time nature of those approaches they can provide promising solutions only for sets of fixed and well known applications, which is in contrast to the dynamic behavior of reconfigurable devices.

An approach was proposed [5] to consider the scheduling problem of multiple applications on a runtime partial reconfigurable architecture in two phases: one at design-time and one at run-time. The design-time scheduler explores the design space for each task and generates a small set of schedules with different energy-performance trade-offs. At run-time, an online scheduler selects the most suitable schedule for each task. The presented architecture uses a Network-on-Chip (NoC) communication infrastructure, placed on the FPGA at compile time, to allow tasks to communicate. The NoC introduces extra overhead in communication delay, but this has not been taken into consideration in the scheduling phase. Furthermore, this approach cannot be applied in a context where the applications are unknown at compile time. Other recent scheduling techniques attempt to reduce the reconfiguration time overhead, while taking into consideration the communication infrastructure, [6], [7]. One of the first attempts to consider a scenario where several communication infrastructures, in terms of latency restrictions, can be supported was proposed by Fekete and Gohringer [6]. Different communication solutions have been evaluated using an ILP formulation. However, due to its high complexity this formulation cannot be used for practical purposes in online schedulers. A complete methodology for scheduling and placing tasks at runtime onto partially dynamically reconfigurable FPGA-based systems was explored in another work [7]. The scheduling problem was presented for both the 1D and 2D models, proposing two heuristics, the horizon and the stuffing techniques, [8], to tackle them. The experimental results are promising. However, the 2D approach was not validated on a real physical system, since the Xilinx Virtex II technology was being used, which does not accommodate 2D reconfiguration. Furthermore, no communication delay has been taken into account. Angermeier and Teich [9] presented a scheduling heuristic that takes the reconfiguration overhead into account. In this study, no *module reuse* is considered and the communication infrastructure has not been fully modeled. Also, the reconfigurable architecture has been modeled only as a 1D area, composed of a set of slots of fixed dimension, where each task requires exactly one slot. There are other approaches [10], [11] which try to overcome this limitation by defining a 1D model, bus-based communication infrastructure, where the reconfigurable area is organized in a given number of fixed-size reconfigurable regions. Others, assume that sufficient resources for communication are available [12], or propose to enlarge the area assigned to a task [13], to support the communication channels between tasks, but without defining any real communication infrastructure. Run-time prefetch techniques [14], similar to those used to hide the memory latency, have also been applied to hide the reconfiguration latency. Other prefetch approaches were proposed [?], where the scheduler has been implemented in hardware using a list-scheduling heuristic. Hence, it can

perform very fast online decisions (just a few clock cycles) introducing almost no time overhead.

Our approaches exploit *module reuse* and *configuration prefetching* in order to minimize the reconfiguration overhead. Furthermore, both are taken into account during the online task scheduling decision process. Most of the existing runtime management systems do not take those features into account in their decision processes. Some existing approaches that consider *module reuse* and/or *configuration prefetching*, [14], [13], [5], still lack the awareness of the communication latency. To the best of our knowledge our proposed online scheduling algorithms are the first to consider overheads associated with reconfiguration and communication latency under a unified framework.

IV. SCHEDULING TECHNIQUES FOR ONLINE 2D DYNAMIC RECONFIGURATION

Based on the general template described above we developed three scheduling algorithms, and three different architectural setup, to address each case: the *Offline/online collaborative scheduler*, *Static grid scheduler*, and *Dynamic NoC scheduler*. Figure 1 shows the taxonomy for the architectures associated with each scheduler.

A. Offline/Online Collaborative Scheduling

The **Offline/online collaborative scheduling** architecture is most suitable for the case when the task graphs are known a priori. In other words, for each task graph, the tasks properties, e.g. latency, reconfiguration time, area, and the dependencies among them are known. Also the amount of communication between two tasks is known. In this case, we can employ a sophisticated static task scheduling and a module placement tool to determine the best dimensions and shapes for the RRs. Specifically, we utilize an offline scheduler, [], to generate a schedule for one or more task graphs, which we refer to as the *basic set* to denote the representative workload. The static schedule enables us to determine the amount of reconfigurable resources needed by each application in the basic set and it is also possible to determine the size of the RFU required for each application on the target device. Next, the tasks in the basic set are placed on the FPGA, by using a simulated annealing approach. This placement tool tries to place a task in an already placed RR, if it is not possible it inserts another RR or modifies the shape of an already placed RR. Using this static design step, the number of RRs, their sizes, and their individual positions are determined. Ultimately, the run-time decisions made by the online scheduler will then be at the granularity of these statically defined RRs, mainly concerned with inserting individual RFUs from the tasks executed on the system at a given instance into one of these RRs.

In this particular architecture, each RR will establish a point to point communication with the static part using bus macros. This choice is based on the observation that in most cases the static part has to communicate with only one RR at a time. This is due to the fact that the processor is connected to an On-Chip Peripheral Bus (OPB) that can be used only for

one communication. The online scheduler will (almost) always know where to schedule each task, hence, the communication between the static side and the reconfigurable side will be handled in a very effective way, without the need of adding parallel communicator devices, e.g. switch among multiple input queues. When the *Offline/online collaborative scheduler* is used, a single RR may contain more than one communication interface. The actual number of communication interfaces, for each RR, is decided by the offline part of the scheduler, [15]. Once the initial configuration of the device has been deployed, the second step is the actual run-time execution of tasks under the guidance of the online scheduler.

In the following we refer to the scheduling algorithm *As Late As Possible* as *ALAP*.

Algorithm 1: The online step (second step) of the Offline/Online Collaborative Scheduling Algorithm (*ATQ*, *RRs*)

```

1 t ← 0 ;
2 repeat
3   Select the element with the less ALAP time, P, from
   the ATQ ;
4   Remove P from the ATQ ;
5   minET ← ∞ ;
6   ifReuse ← false ;
7   repeat
8     Select a reconfigurable region R from RRs ;
9     if (type of R) = (type of P) then
10      if ((estimated reuse time of P in R) <
        (minET)) then
11        minET ← estimated reuse time of P in R;
12        selR ← R;
13        ifReuse ← true ;
14      end
15    end
16    if (size of R) ≥ (size of P) then
17      if ((estimated end time of P in R) <
        (minET)) then
18        minET ← estimated end time of P in R;
19        selR ← R;
20        ifReuse ← false ;
21      end
22    end
23  until no more reconfigurable regions;
24  if ifReuse then
25    Reuse (P, selR, t) ;
26  end
27  else
28    Place (P, selR, t) ;
29  end
30  t ← t + TIMESTEP ;
31 until true;

```

The variable TIMESTEP in the pseudocode has been set as the average execution time of the scheduling step obtained

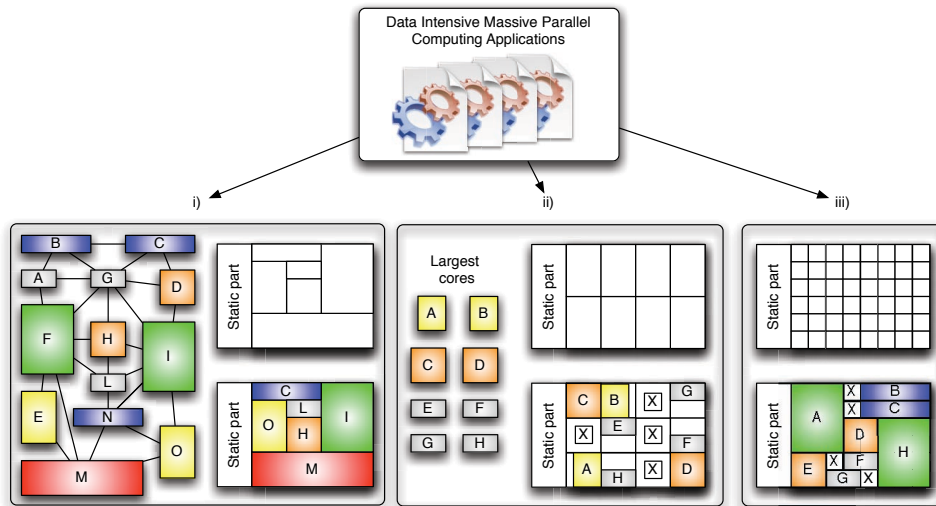


Figure 1. i) Offline/online collaborative scheduling architecture ii) static-grid scheduling architecture iii) dynamic-grid scheduling architecture

through more than ten thousand executions. The pseudocode of the online portion for the Offline/online collaborative scheduling algorithm is shown in Algorithm 1. The scheduler takes as input the *Available Task Queue (ATQ)* and the list *RRs* of reconfigurable regions placed during the offline phase (i.e. by the Offline step) onto the reconfigurable side of the device. When a task is ready to be executed it is directly inserted into the *ATQ*. At each step, the scheduler selects the task with the minimum *ALAP* time. Then, the scheduler first tries to reuse a module of the same type that is already placed. The scheduler also looks for a *RR* in which the selected task can be reconfigured and executed with the smallest timing overhead. The scheduler considers the reconfiguration and communication delays in the following way. It makes a sweep over all the *RRs* and, for each of them, it finds the first available time when the *RR* can be used (where no *RFUs* are pending to be executed after that time), determines when the reconfiguration can be performed with no-preemption and calculates the reconfiguration finish time. Considering the reconfiguration finish time of the current *RFU* and the finish time of associated parent tasks, the scheduler calculates the amount of time needed for data transmission. It models the bus on the static side of the *FPGA* and occupies it when the data travels on it. This operation takes into account transmissions that can be preempted: it is possible to send a big amount of data in not sequential time steps. For example, it is possible to send 32 bits at time 1 and other 32 bits at time 4. In this way the bus can, actually, be multiplexed between *RFUs*, increasing the throughput of the whole system. This particular step is a unique novel feature of our proposed scheduler. Once all the communication finish times have been calculated, the scheduler simulates the execution of the *RFU* starting from the latest communication finish time onwards by adding the *RFU* execution time. At the end it simulates the communication between the *RFU* and the static part, in the same way used

for the input data. With this information, the scheduler selects the mode, reused or reconfigured, for the considered task that allows the shortest termination time, thus, it decides in which *RR* it has to be placed. This process is performed for each task extracted from the *ATQ*, until *ATQ* is empty. *One specific focus of this work* is to deal with applications containing hundreds of tasks grouped in a few number of task types. Such a task graph structure is representative of data intensive high performance applications. While these tasks have a short execution time, their reconfiguration time is usually three orders of magnitude larger. This is due to the fact that all the loops in the applications have been unrolled: we know exactly the actual task graph of the applications, not a generic one. The amount of data necessary to execute a task is of the same order of magnitude as the execution time, since these tasks are massively parallelized, each to work with relatively small input sets. Therefore, each parallel instance of a task is able to generate its result quickly.

We developed a variant to our scheduling algorithm, which we refer to as the *Stack scheduler*, specifically to address this domain. The main difference between this scheduler and the previously described one is that, after the task with the minimum *ALAP* time has been selected, the scheduler will schedule *all* the other tasks of same type in *ATQ*, ordered by *ALAP* values. Thus, multiple tasks will be scheduled at each scheduling step. For example, let us consider the case when *ATQ* contains 5 tasks of type 1 and 7 of type 2:

- Offline/online collaborative scheduling schedules only one task each time it is executed, thus, 12 runs must be executed;
- Stack scheduler schedules all the task of the same type in a single run, thus, only two runs are needed.

B. Static-Grid Scheduling

The **Static-Grid** approach aims at addressing the case when only the knowledge of the maximum dimension among

all tasks is available. The reconfigurable side of the device is partitioned, at compile time, in a homogeneous grid of tiles. All tiles are characterized by the same size and shape: these properties are dictated by the largest area required by the tasks. Furthermore, when all the tasks have about the same dimensions, e.g. with a bias of 5-10%, it performs in the optimal way. The communication infrastructure in this configuration is based on the Network-on-Chip paradigm, and an irregular topology is employed. The reason is as follows. Once a task completes its execution and broadcasts its output data to the network, it does not need to wait for the static side to receive this data. The logic resources occupied by this task can be released immediately and a new task could be placed in its place. In this way, tasks can be scheduled in a faster way, leaving communication responsibilities entirely to the NoC. Another advantage is that it is possible to free multiple tiles at the same time, because they can send their data in parallel to the network switches. The drawback of this architecture is that in order to implement the communication infrastructure, we need to place network switches to allow the data flow. Each RFU needs to be connected to exactly one switch in order to communicate. Hence, this places requirements on placement and resource utilization. The pseudocode is shown in Algorithm 2. The constant $CELLxSWITCH$ contains the number of cells the can be connected to a single switch. The scheduler takes as input the ATQ and the grid of RRs . After the selection of the task with the minimum ALAP time, the scheduler schedules it by first trying to reuse a module. Otherwise, the scheduler searches for a RR in which the selected task can be reconfigured and executed as soon as possible. In particular, every time a task has to be placed on the device, the algorithm determines whether a new communication switch is needed. If there is not a free tile adjacent to a switch, the task cannot be placed safely, because it would be impossible for it to communicate. Thus, when there are no available switch connections, it is mandatory to insert a new one. The insertion of a new switch does not have to delay the task for too much time: if the time needed by a RFU to terminate its work is less then the reconfiguration time of the switch, the task will be scheduled on that RR . Furthermore, when the input queue of all the placed switches are occupied, the scheduler forces the insertion of a new switch, in order to maintain the system response. When a new switch is needed, the scheduler identifies a location for the switch as well: a position with at least three free adjacent tiles will be chosen. When three free tiles are not available, the algorithm searches for two tiles, and so on. This is because, based on our experiments, it is optimal to place one neighboring switch for every three new RRs . In this approach, switches need to be reconfigured on the reconfigurable side, and occupy the reconfigurator device. The Static grid scheduler evaluates the reconfiguration overhead in the same way as the offline/online collaborative scheduler. However, the evaluation of the communication overhead needs to change since the communication delay within the NoC has been modeled differently. Each network switch in the NoC accepts multiple input queues and it sends data sequentially

Algorithm 2: Static Grid Scheduling (ATQ , RRs)

```

1  $t \leftarrow 0$  ;
2 repeat
3   Select the element with the less ALAP time,  $P$ , from
   the  $ATQ$  ;
4   Remove  $P$  from the  $ATQ$  ;
5    $minET \leftarrow \infty$  ;
6    $ifReuse \leftarrow false$  ;
7   repeat
8     Select a reconfigurable region  $R$  from  $RRs$  ;
9     if ( $type\ of\ R = type\ of\ P$ ) then
10      if ( $(estimated\ reuse\ time\ of\ P\ in\ R) <$ 
11         $(minET)$ ) then
12         $minET \leftarrow estimated\ reuse\ time\ of\ P\ in\ R$ ;
13         $selR \leftarrow R$ ;
14         $ifReuse \leftarrow true$  ;
15      end
16    end
17    if ( $(estimated\ end\ time\ of\ P\ in\ R) < (minET)$ )
18      then
19         $minET \leftarrow estimated\ end\ time\ of\ P\ in\ R$ ;
20         $selR \leftarrow R$ ;
21         $ifReuse \leftarrow false$  ;
22      end
23    until no more reconfigurable regions;
24    if  $ifReuse$  then
25      Reuse ( $P$ ,  $selR$ ,  $t$ ) ;
26    else
27      if ( $number\ of\ used\ cells > number\ of\ switches *$ 
28         $CELLxSWITCH$ ) then
29        PlaceSwitch ( $t$ ) ;
30      end
31      Place ( $P$ ,  $selR$ ,  $t$ ) ;
32    end
33     $t \leftarrow t + TIMESTEP$  ;
34  until true;

```

either to the static side or to another switch. Therefore, each RFU can immediately send its results to the switch it is connected to. As a result, taking advantage of the NoC structure, it is possible to hide part of the communication latency. This is because, based on our experiments, it is optimal to place one neighboring switch for every three new RRs . In this approach, switches need to be reconfigured on the reconfigurable side, and occupy the reconfigurator device.

C. Dynamic-NoC Scheduling

Whenever working with a completely flexible task management scheme without assuming any prior knowledge about task graphs, both the RFUs and the communication infrastructure need to be placed dynamically at runtime as decided by the **Dynamic-NoC** scheduler. Each RFU needs to be connected by switches to the static part. In order to accommodate the

complete flexibility we utilize the NoC paradigm again. Therefore, in order to enable communication between a new task and the static part, the scheduler needs to place the network switches as well. The difference between the Dynamic NoC scheduler and the *Static grid scheduler* is that the complexity of deciding where to place a RFU and a network switch depends on the size of the reconfigurable device and not on the number of grid cells. This is because the minimum amount of reconfigurable area that can be reconfigured is coincident to the one associated with the device. It is possible to increase it, in order to make faster decisions, but still it has to be not bigger than 1% of the FPGA: absolute flexibility is needed. In this approach the communication and reconfiguration overheads have been taken into account in the same way as in the *Static grid scheduler*. The switch placement policy of the scheduler is the same as well. The pseudocode is shown in Algorithm

Algorithm 3: Dynamic NoC Scheduling (*ATQ*)

```

1  $t \leftarrow 0$  ;
2 repeat
3   Select the element with the less ALAP time,  $P$ , from
   the ATQ ;
4   Remove  $P$  from the ATQ ;
5    $\text{minET} \leftarrow \infty$  ;
6   repeat
7     Select a reconfigurable region  $R$  from RFUs ;
8     if (type of R) = (type of P) then
9       if ((estimated reuse time of P in R) <
          (minET)) then
10         $\text{minET} \leftarrow$  estimated reuse time of P in R;
11         $\text{selR} \leftarrow R$ ;
12      end
13    end
14    until no more reconfigurable regions;
15    Find the position  $K$  that allows the minimum end
    time  $PT$  for  $P$  if  $PT \geq \text{minET}$  then
16      Reuse ( $P$ ,  $\text{selR}$ ,  $t$ ) ;
17    end
18    else
19      if (number of used cells > number of switches *
          CELLxSWITCH) then
20        PlaceSwitch ( $t$ ) ;
21      end
22      Place ( $P$ ,  $K$ ,  $t$ ) ;
23      add in RFUs the new module ;
24      Remove from RFUs the modules overlapping
        with the new one
25    end
26     $t \leftarrow t + \text{TIMESTEP}$  ;
27 until true;

```

3. The scheduler takes as input just the *Available Task Queue (ATQ)*. At each step the scheduler selects the task with the minimum ALAP time and then tries to reuse a module already placed on the device. If it is not possible to exploit *module*

reuse, the scheduler looks for a portion of the reconfigurable area in which it is possible to reconfigure and execute the selected task as soon as possible. Also in this case, the scheduler determines if another switch has to be placed on the device. The only difference is the granularity of the placement. In the case of the *Dynamic NoC* scheduler the reconfigurable side of the device is divided in a logical grid at the granularity of the minimum reconfigurable cells available. Therefore, the algorithm used for the static grid placement has been modified in order to deal with RFUs that can occupy more than one grid cell. The scheduler searches for enough reconfigurable cells to place the new RFU. The scheduler needs to scan all potential cells to decide the actual reconfiguration start time for the new RFU about to be scheduled. This process leads to a very long scheduling time step, because at this step of the work we perform an exhaustive search, but any searching algorithm could be used.

V. EXPERIMENTAL RESULTS

This section presents the results obtained from the evaluation of the different online scheduling techniques proposed in this paper. The applications used to validate the proposed approaches, have been selected from a popular data mining benchmark suite, *NU-MineBench* [16]. We have experimented with the following kernels selected from this benchmark suite:

- 1) *variance* application: it receives as input a single set of data and calculates the mean and the variance among the whole data set;
- 2) *distance* application: it receives as inputs two sets of data of equal size and calculates the distance between them;
- 3) *variance1* application: it receives as input a single set of data and calculates the mean and the variance among the whole data set. The sets of operations used to built tasks are different with respect to *variance*.

Each task graph contains hundreds of tasks grouped under a few task types. Furthermore, while these tasks have a short execution time, their reconfiguration time is usually three orders of magnitude larger. The amount of data necessary to execute a task is of the same order of magnitude as the execution time, since a high number of parallel tasks work each with a small input set. The task graphs of these applications increase in size according to the data size they have to process: the same application could have a task graph with either hundreds or thousands of nodes. Moreover, *distance* and *variance1* applications have some task types in common. The same relation holds between *variance* and *variance1* applications, since they share some task types, while *distance* and *variance* applications do not have any common task type. The experiments have been performed as simulations in a framework that allows the complete management of a 2D partially dynamically reconfigurable device. The framework receives a batch as input and it starts to send the tasks with no parents to the scheduler. Each time a task is scheduled, the scheduler notifies the framework. If the framework finds that all the parents of a task have been scheduled, it sends that

task to the scheduler to be inserted into the Available Task Queue. This is repeated until the batch is exhausted. In the following figures and tables, we will refer to the algorithms using the convention outlined in the following. *Offline/online collaborative scheduler* will be *O/OC*; *Stack scheduler* will be *Stack*; *Static grid scheduler* will be *SG*; *Dynamic NoC scheduler* will be *DNoC*. The data and application set for the experiments have been chosen to form a representative batch of different task graphs of the above mentioned applications. We took into account different orders in which the applications may be executed in an average data mining step; thus, we selected different orders and percentages for them, according to a set of initial profiling information. The clock cycle has been select to be 50 MHz for all the architectures.

A. Analysis and Relative Comparison of Proposed Approaches

The aim of this section is to present in detail the behavior of each algorithm showing the specific cases in which each demonstrated benefits. First of all we compare the execution time of each scheduler. We measured the average time each scheduler needs to perform a single schedule step, which grows linearly with the number of available nodes in the Available Task Queue. Table I shows the average time needed by a single scheduling step, for a simulation involving 512 tasks.

Table I
AVERAGE TIME FOR A SCHEDULING STEP EXPRESSED IN CLOCK CYCLES

O/OC	Stack	SG	DNoC
140	400	160	500

Figure 2 shows a comparison among our proposed algorithms based on the total execution time needed to complete a run for a batch for different sizes of batches. The *Stack scheduler* outperforms other algorithms by at least one order of magnitude. The reason is that it takes a longer scheduling step time, but has the opportunity to schedule multiple tasks together and this leads to fewer total schedule steps. The benchmark task graphs in this particular experiment are composed of tasks of two types, so the stack scheduler is the most effective. The duration of one scheduling decision for the *Dynamic NoC scheduler* is very long and this leads to a high total execution time. However, there are situations in which the *Static grid scheduler* does not work very well. If an application has big tasks that need to be executed concurrently with small tasks, it will be impossible to generate a grid that accommodates both types of tasks efficiently. The *Dynamic NoC scheduler* is still effective in this case because it manages the whole reconfigurable side at the finest possible granularity.

Another problem that arises with the *Static grid scheduler* is related to the switch dimension and the task execution times. Each switch occupies a small amount of area, in our experiment a switch corresponds to less than 4% of the total FPGA area. Normally, this amount of area is much smaller than a single cell, hence, a lot of space is wasted. If the execution time of a single task is long enough and therefore comparable with the scheduling step time, there is no possibility of executing

the needed parallel computations. There is no such problem when using the *Dynamic NoC scheduler*. Table II shows the maximum wasted area in a set of experiments with more than 100,000 tasks, to simulate a real runtime environment.

Table II
PERCENTAGE OF WASTED AREA

O/OC	Stack	SG	DNoC
26,9%	26,9%	48,8%	0%

Another limitation on the possibility of having more concurrent tasks arises due to the necessity of having a communication infrastructure. Table III shows the percentage of area used by the communication infrastructure in the same experiment. Even in this case the *Static grid scheduler* uses most of the area and the reason is still the same: a single switch occupies an entire cell of the grid.

Table III
PERCENTAGE OF AREA USED FOR COMMUNICATION INFRASTRUCTURES

O/OC	Stack	SG	DNoC
6%	6%	23,3%	9%

Another analysis useful to compare the real necessity for having a fixed communication infrastructure is presented in Table IV.

Table IV
PERCENTAGE OF TIME USED FOR COMMUNICATION

O/OC	Stack	SG	DNoC
4,46%	90,41%	3,9%	1,2%

The data has been taken from the same experiment used for Table II. It is possible to see that the communication delay is a very interesting issue for the *Stack scheduler*. Here, communication has a significant influence on the scheduling decisions. In the other algorithms communication is not comparable with the total execution time, but still it can influence the scheduling. When a static communication infrastructure is used, most of the time it is idle, so most of the time that area is wasted. This happens even with the NoC based schedulers. Moreover, because of the reconfiguration time needed to reconfigure a switch, it is generally not desired to allow a switch to be removed and placed again.

B. Comparison with state-of-the-art solutions

In this section we compare our schedulers with existing techniques in order to show their effectiveness. The considered schedulers are: *FIFO scheduler*: it uses the classical FIFO techniques to decide which task needs to be scheduled. Furthermore, to enable a fair comparison we enhanced the basic FIFO scheduler with *collaborative* techniques for managing the communication and all the reconfiguration features exploited in our algorithms, i.e., *communication overhead*, *module reuse* and *configuration prefetching*; *FIFO Stack scheduler*: this is the stack version of the previous one using the similar principle as our Stack scheduler; *Horizon scheduler*; *Stuffed scheduler*.

Figure 2 shows a comparison among our proposed algorithms and the existing techniques based on the total execution time needed to complete a run. We first compare the

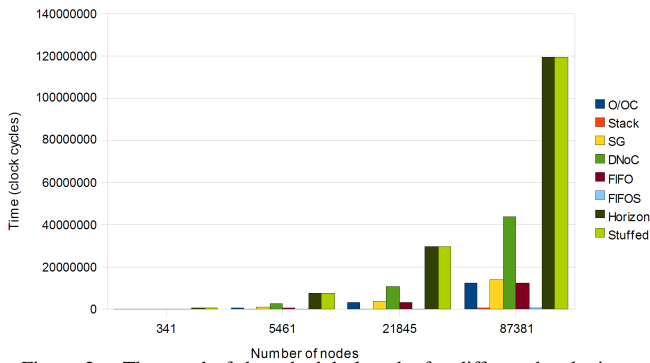


Figure 2. The trend of the schedule lengths for different batch sizes

performance of the algorithms that possess full knowledge on the task graphs. These are our Offline/online collaborative scheduler and Stack scheduler versus the FIFO scheduler. Our Offline/online collaborative scheduler yields better results than the FIFO scheduler because the *ALAP* choice exploits *reconfiguration prefetching* in a more effective way. This becomes more clear when scheduling a general application. The *FIFO stack scheduler* and our *Stack scheduler* obtain almost always the same results. Next, we compare the schedulers that do not operate with full knowledge on the task graphs. *Horizon scheduler* and *Stuffed scheduler* do not consider *module reuse*, so their high execution time is due to the repeated reconfigurations. Each reconfiguration takes more than 1,000 clock cycles. The authors [8] do not consider reconfiguration time at all and this has been the general practice in existing works so far. Furthermore, they implicitly assume the possibility for executing multiple reconfigurations in parallel, which is not possible in practice. For this reason we further assumed that the communication has zero cost for the *Horizon scheduler* and *Stuffed scheduler* in this experiment, but still their overall performance is significantly worse than our algorithms.

VI. CONCLUSIONS

The aim of this work is to propose novel online scheduling techniques useful in a 2D partially dynamically reconfigurable system scenario. Four algorithms have been developed: *Offline/online collaborative scheduling*, *Stack scheduler*, *Static grid scheduler*, and *Dynamic NoC scheduler*. In presence of complete knowledge over the task graphs that have to be scheduled on the system, the *Offline/online collaborative scheduling* approach is the best choice. The *Stack scheduler* will become the best possible choice in the case each task graph contains a lot of tasks with few distinct types, such as in the case of data mining applications. If at design time, only the maximum dimension of a task is known, the *Static grid scheduler* is the best choice. Finally, when no information regarding the applications that will be executed on the system is available, or when these applications change

very frequently during the system lifetime, or when the other scheduling algorithms encounter their limitations, the *Dynamic NoC scheduler* allows a complete freedom of placement. The scheduling time is much higher than the other schedulers, but the performance is still acceptable being comparable with other state-of-the-art approaches.

REFERENCES

- [1] M. Gokhale, J. Cohen, A. Yoo, W.M. Miller, A.C. Jacob, and R. Ulmer, C. Pearce. Hardware technologies for high-performance data-intensive computing. *IEEE Computer* 41(4), pages 60–68, 2008.
- [2] Fpga high performance computing alliance. <http://www.fhpca.org/maxwell.html>.
- [3] R. Maestre, F.J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, and M. Fernandez. Kernel scheduling in reconfigurable computing. *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 90–96, 1999.
- [4] F. Redaelli, M. D. Santambrogio, and D. Sciuto. Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems. In *DATE 08: Proceedings of the conference on Design, Automation and Test in Europe*, March 2008.
- [5] J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A reconfigurable manager for dynamically reconfigurable hardware. *Design & Test of Computers, IEEE*, 22(5):452–460, Sept.-Oct. 2005.
- [6] S. P. Fekete and D. Ghringer. Scheduling and communication-aware mapping of hw/sw modules for dynamic partial reconfigurable soc architectures. In *Architecture of Computing Systems*, pages 151–159, 2007.
- [7] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *Computers, IEEE Transactions on*, 53(11):1393–1407, Nov. 2004.
- [8] C. Steiger, H. Walder, and M. Platzner. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *Proceedings of the 13rd International Conference on Field Programmable Logic and Application*, pages 575–584. Springer, 2003.
- [9] J. Angermeier and J. Teich. Heuristics for Scheduling Reconfigurable Devices with Consideration of Reconfiguration Overheads. In *Proceedings 15th Reconfigurable Architectures Workshop*, April 2008.
- [10] J. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, pages 116–122, 2002.
- [11] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *In Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures*, pages 284–287. CSREA Press, 2003.
- [12] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable fpgas. *Computers and Digital Techniques, IEE Proceedings -*, 147(3):181–188, May 2000.
- [13] E.P. Ramo, J. Resano, D. Mozos, and F. Catthoor. A configuration memory hierarchy for fast reconfiguration with reduced energy consumption overhead. *International Parallel and Distributed Processing Symposium*, April 2006.
- [14] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *International symposium on Field-programmable gate arrays*, pages 187–195, 2002.
- [15] Juanjo Noguera and Rosa M. Badia. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *Trans. on Embedded Computing Sys.*, 3(2):385–406, 2004.
- [16] A. Montone, F. Redaelli, M.D. Santambrogio, and S.O. Memik. A reconfiguration-aware floorplacer for fpgas. In *Reconfigurable Computing and FPGAs, 2008. ReConFig 08. International Conference on*, pages 109–114, Dec. 2008.
- [17] 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 International Symposium on Workload Characterization*, pages 31–36, 2006.