

Research Article

Floorplacement for Partial Reconfigurable FPGA-Based Systems

A. Montone,¹ M. D. Santambrogio,^{1,2} F. Redaelli,¹ and D. Sciuto¹

¹ *Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy*

² *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

Correspondence should be addressed to M. D. Santambrogio, marco.santambrogio@polimi.it

Received 20 August 2010; Accepted 20 December 2010

Academic Editor: Aravind Dasu

Copyright © 2011 A. Montone et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We presented a resource- and configuration-aware floorplacement framework, tailored for Xilinx Virtex 4 and 5 FPGAs, using an objective function based on *external wirelength*. Our work aims at identifying groups of *Reconfigurable Functional Units* that are likely to be configured in the same chip area, identifying these areas based on resource requirements, device capabilities, and wirelength. Task graphs with few externally connected RRs lead to the biggest decrease, while external wirelength in task graphs with many externally connected RRs show lower improvement. The proposed approach results, as also demonstrated in the experimental results section, in a shorter external wirelength (an average reduction of 50%) with respect to purely area-driven approaches and a highly increased probability of reuse of existing links (90% reduction can be obtained in the best case).

1. Introduction

Nowadays one of the most important design styles in VLSI is represented by Field Programmable Gate Arrays (FPGAs). The standard FPGA design flow starts from an RT-level description of the circuit (e.g., provided by a HDL language) and ends with a configuration file (bitstream) configuring the desired circuit on the device. Much effort has been already placed towards improving the different stages of the design flow, from logic synthesis to placement and routing. On one hand, these problems have been addressed directly by the FPGA vendors [1] as much as by academic works [2]. On the other hand, the floorplanning automation for FPGAs is a current research topic, to face the new challenges provided by FPGAs.

FPGAs particularly present two unique aspects with respect to traditional VLSI designs: resource heterogeneity and reconfigurability. FPGA devices are generally defined using several kinds of resources (e.g., programmable logic cells, memories, multipliers, DSPs, and so on). This requires to take into consideration the resource heterogeneity, to allow each architectural module to be placed in an area region containing all the needed resources. On the other hand, reconfiguration allows the possibility to change the architecture or the application implemented on the FPGA

without requiring any physical action on the device. In particular, in partial dynamic reconfigurability, an architecture may change at runtime (i.e., without having a disruption of the provided functionality) a subset of its modules in order to modify its own behavior. Reconfigurability introduces time as a new variable within the floorplanning formulation.

A floorplanner taking into account both of these aspects has to find, for each module (in this paper we use the term *module* to address an architectural component after technology mapping), an area assignment according to target device's capabilities considering that the modules can be replaced later on due to the system needs using the reconfiguration capabilities of the target device. The term floorplacer has been first introduced in the context of traditional VLSI design [3, 4] to emphasize how recently developed algorithms for VLSI design automation face placement and floorplanning tasks concurrently. In a similar spirit, resource management for partially and dynamically reconfigurable FPGAs can benefit from this paradigm. In this paper, which is an extended version of the work presented in [5], we propose to develop a floorplacer for such a device, which will identify groups of modules that are likely to be configured in the same rectangular area and, consequently, identify these chip areas according to the modules' requirements, device capabilities and design objectives. Particularly, in this work, we focus

on optimizing the wirelength since the base floorplacement framework.

The paper is structured as follows. Section 2 introduces the problem and definitions of relevant concepts. Section 3 describes the related work. The target FPGA architecture is described in Section 5, and the experimental results are presented in Section 6. Finally, conclusions and future work are outlined in Section 7.

2. Related Works

The problem of resource-aware floorplanning tailored for FPGAs has been addressed in literature [6, 7]. Montone et al. [7] proposed a floorplanning method that is only aware of resource requirements for the reconfigurable modules. Feng and Mehta [6] proposed an approach, where each architectural module has a list of required resources, hence each module has to be placed in an area region containing all the needed resources. This approach is based on a two steps algorithm: first the execution of Parquet [8] floorplanner with the following resource-aware cost function consisting of a linear combination of Parquet's objective function and the amount of satisfied resource requirements. The result is then refined solving a flow maximization with cost minimization on a purposely built graph. While this approach is currently the state of the art in resource-aware floorplanning for FPGAs, in the majority of cases, the result is not compatible with the FPGA reconfigurability process.

One of the earliest works dealing with reconfiguration-aware floorplanning is [9]. An offline floorplanner is used to decide whether a required functionality should be implemented in hardware on a reconfigurable device or has to be executed via software on a general purpose processor. Different heuristics are proposed to minimize execution time, while reducing implemented modules' fragmentation across the device area.

The first definition of *temporal floorplanning* has been introduced in [10]. According to the author a temporal floorplanning consists of two phases: (i) partitioning and sequencing design modules into design configurations (also called temporal partitioning or scheduling), and spatial positioning of design modules and wiring within the reconfigurable area, in the following named as reconfigurable region, of each configuration.

One of the most important contributions in three-dimensional floorplanning is presented in [11, 12]. They introduce a full 3D floorplanner based on simulated annealing using a three-dimensional transitive closure graph (TCG) and T-trees. They just evaluate the time required by each module to communicate with RAM chips outside the FPGA in order to store results and read input, but they do not evaluate if a found floorplan, particularly its communication infrastructure, is feasible on a given device or not. Furthermore, they do not consider other resources than logic blocks.

A more recent work [13] faces the FPGA floorplanning problem considering partial dynamic reconfiguration, in terms of module reusability. They introduce a variation to the sequence pair representation [14] in order to represent

the floorplan in different time frames. The aim of this work is to reduce the quantity of device area reconfigured between different time frames. Given two designs in two different time instants, the authors present a simulated annealing-based floorplanning able to reduce the reconfigured area between the two instants by exploiting reuse of already configured modules.

A brief comparison between previous works is provided in Table 1. The most common limitations of prior work is in their lack of control over the feasibility of the resulting communication infrastructure. Furthermore, most of the related work do not treat resource awareness as a primary goal. However, this is of utmost importance for making module allocation decisions for modern dynamically reconfigurable systems, which contain a variety of heterogeneous resources. Our proposed floorplacement method aims to address both of these two important problems, that is, enabling an interconnect and resource-aware design framework.

To the best of our knowledge, there are no prior works dealing with the minimization of wirelength between RFUs and IOBs. The problem of minimizing internal wirelength of RFUs is considered in the past [12], but this approach is based on an estimation of the internal wirelength related to RFUs' aspect ratio and cannot be easily extended in order to support external wirelength. Similarly, by applying the approach introduced in [13], a reuse of links can be obtained, but this approach aims only at re-using RFUs implementing the same functionality, while our approach aims at re-using links between different RFUs connected to the same set of IOBs.

In the following section, we will introduce the basic concepts and definitions related to our proposed floorplacer.

3. Problem Description and Basic Definitions

Given a set of *Reconfigurable Functional Units* (RFUs) the resource- and configuration-aware floorplacement problem consists of finding, for each RFU, a chip area where it can be placed and routed offline and configured at runtime. The entire process is subject to the constraint that each RFU must be associated with an area containing all the required types of resources for its functionality.

Let a RFU be a technologically mapped netlist implementing a required functionality, and a *Reconfigurable Region* (RR) be a rectangular FPGA area where two or more RFUs are going to be placed and routed (at design time) and configured (at runtime) according to the application implemented in the reconfigurable system. The goals of the floorplacer are to

- (1) define the number of RRs and associate each RFU with one and only one RR (partitioning task),
- (2) find a position of RFUs inside the corresponding RRs (Temporal Floorplacement inside Reconfigurable Region),
- (3) find an area constraint for all the RRs inside the FPGA area (RR Floorplacement).

TABLE 1: Comparison among previous works.

Authors	Comm Infrastructure	Resources Aware	Reconfigurability Aware	Reusability Aware	Algorithm
Feng and Mehta [6]	No	Yes (high res. usage)	No	No	Sim. Annealing over seq. pairs + Flow Max over Flow Graph
Montone et al. [7]	No	Yes	Yes	For logic only	Sim. Annealing
Bazargan et al. [9]	No	No	Yes	No	Sim. Annealing over cubic modules (2d spatial, 1d temporal)
Yuh et al. [11, 12]	Limited, w/High Overhead	No	Yes	No	Sim. Annealing over cubic modules using T-trees and TCG
Singhal and Bozorgzadeh [13]	No	No	Yes	Yes	Sim. Annealing over seq. pairs

The set of input RFUs is represented by a *scheduled task graph*. Such a task graph is divided into time intervals named *static snapshot*. Each static snapshot is characterized by having the same set of configured and running RFUs, during its time interval. Given a RFU m , the function $\text{TIME}(m)$ returns the set of static snapshots containing m , that is, static snapshots requiring that m is configured and running. According to these definitions, the scheduled task graph can be considered as a finite state automaton having static snapshots as states and the reconfiguration process as transitions.

Our framework is tailored for Xilinx Virtex 4 and 5 FPGAs [15] that provide access to the off-chip logic through a set of FPGA pins dedicated for communication. The connections between the FPGA's internal logic and external pins are managed through three-state buffers named *Input Output Blocks* (IOBs).

Our proposed approach specifically deals with the interconnection optimization. For FPGAs routing resources are often the limiting resources, hence, achieving better routability in a design may in fact determine the overall feasibility of a design.

Our floorplacement framework manages an objective function based on *external wirelength*, that is, the estimated length of the nets connecting each RFU to the corresponding IOBs. This framework can be used in two different scenarios.

- (i) The designer is implementing an application on an FPGA belonging to an existing board with previously assigned IOBs. In this case the framework can help the designer to define area constraints in order to reduce external wirelength.
- (ii) The designer has to assign the IOBs and build the board from scratch. In this case the framework can provide feedback to the designer in order to identify or approximate the best IOBs assignment.

In order to formulate this objective function the distance between RFUs and IOBs need to be estimated. Experimental results proved that using the Manhattan distance between the center of the RFU and the position of the IOB provides a good approximation. This definition can be generalized as the distance between one RFU and any location on the chip

area as follows:

$$d(\text{RFU}, P) := \left| P_x - \left(\frac{1}{2} \cdot \text{RFU}_w + \text{RFU}_x \right) \right| + \left| P_y - \left(\frac{1}{2} \cdot \text{RFU}_h + \text{RFU}_y \right) \right|, \quad (1)$$

where subscripts x , y , w , and h for RFUs stand for the x coordinate, y coordinate, width and height, of the RFU. RFUs have x and y coordinates corresponding to the bottom-left most corner. P_x and P_y denote the x and y coordinates of an arbitrary location on the FPGA. Similarly the distance between one Reconfigurable Region and a point can be defined. The rationale of the wirelength-driven floorplacement is to constrain a group of RFUs characterized by being connected to a set of IOBs within the same area, thereby, creating a neighborhood. Hence, RFUs communicating with IOBs that are near each other, are kept together and constrained within the same area. This approach results in a shorter external wirelength and a highly increased probability of reuse of existing links. While the first outcome is intuitive, the second requires some elaboration. In a dynamically reconfigurable device a common approach is to allocate two main partitions on the device: the static part and the dynamically reconfigured part, where RFUs will be allocated. The communication infrastructure serving all dynamically inserted functionality is managed by the static part of the design and all the communication, both among RFUs and between RFUs and the static part, are performed by the communication infrastructure exposed through hardware macros. Also the communication between RFUs and IOB has to be managed by the static part of the design and similarly has to be exposed through hardware macros. Let us consider N RFUs accessing an external device through a set of M IOBs, in the worst case the static part has to provide one set of M hardware macros for each RFU, hence, $M \cdot N$ hardware macros. Instead, if the floorplacement is aware of IOB positions, RFUs accessing to the same set of IOBs could be constrained within the same area and hardware macros may be reused by the different RFUs. Consequently, the number of hardware macros that need to be provided by the static part of the design can be drastically lowered toward the theoretical limit of just one set of M hardware macros, that is, just one for each IOB.

Each RFU of the input scheduled task graph can be annotated with information on the position of the IOBs. In order to simplify the management of the IOBs, for each RFU n connected to M IOBs we define a point C , named *centroid*, whose coordinates are given by the arithmetic average of the coordinates of the IOBs:

$$C_x(n) = \frac{1}{M(n)} \sum_{m=1}^{M(n)} IOB_x(n, m), \quad (2)$$

where $C_x(n)$ is the x coordinate of RFU n , $IOB_x(m, n)$ is the x coordinate of the m th IOB of RFU n , and $M(n)$ is the number of IOBs of RFU n . The same holds for y coordinates. The centroid represents the ideal position where an RFU should be positioned in order to minimize the external wirelength, that is, floorplating the RFU such that its own geometrical center coincides with the centroid will result in minimizing the external wirelength. Only RFUs connected to the IOBs have an associated centroid.

In the following section we will first give an overview of the dynamic reconfiguration mechanism of our target FPGA architecture. Next, in Section 5 we will describe our floorplacer which can effectively manage the resources of this reconfigurable architecture.

4. FPGA Reconfiguration and Target Architecture

This section introduces the reconfiguration process from the FPGA's physical point of view. Section 4.1 presents the physical limits of the reconfiguration process, while Section 4.2 shows the design flow proposed by Xilinx for reconfigurable architectures. Finally, Section 4.3 defines the target architecture considered by this work.

4.1. Smallest Reconfigurable Region. The reconfiguration process described here relates to the latest generation of Xilinx FPGAs, that is, Virtex 5 devices. According to their datasheets [16] all the devices of this family share a common structure. The entire FPGA is made of programmable logic (commonly referred to as CLBs for this FPGA family) and periodically distributed Block RAMs (BRAMs), while all the other resources (such as DSP blocks) are placed along the vertical edges of the FPGA. User logic is implemented combining all these resources and connecting them using channels and switchboxes. The information about device configuration is described in a binary configuration file named *bitstream*: logical functions implemented by CLBs (i.e., the content of the lookup tables implementing a logical function with 6 inputs and 1 output), BRAMs content, routing information (i.e., logical status of switches managing the interconnections), and so on. The smallest reconfigurable element is 1 row high and 1 CLB wide and is referred to as *frame*, each frame is addressed by a row number (from 0 to 3) and column number (expressed in CLBs). The bitstream follows the device topology, hence inside a configuration bitstream one can easily identify data configuring a specific frame.

As previously mentioned, the frame is the smallest FPGA area that can be configured independently of the others. Given a generic module h_{module} rows high and w_{module} CLBs wide, the smallest area that can be involved in the reconfiguration process is a rectangle with $h_{\text{small-area}}$ rows height and $w_{\text{small-area}}$ CLBs width such that the following.

- (i) The height in rows is the smallest integer greater than the module's height in rows

$$h_{\text{small-area}} = \lceil h_{\text{module}} \rceil \quad [\text{rows}]. \quad (3)$$

- (ii) The width in CLBs is the smallest integer greater than the module's width in CLBs

$$w_{\text{small-area}} = \lceil w_{\text{module}} \rceil \quad [\text{CLBs}]. \quad (4)$$

For example, given a module requiring for its placement 1.5 rows height and 30 CLBs width, the smallest area that can be assigned to this module, considering the reconfiguration constraints, is a rectangle of height and width of 2 rows and 30 CLBs, respectively. Configuration and reconfiguration processes take place by writing the bitstream inside the FPGA's *configuration memory*. In the case of partial reconfiguration the bitstream carries information addressing the frames that are going to be replaced by the carried data.

4.2. Xilinx Partial Reconfiguration Design Flow. Due to the fact that Xilinx FPGA Virtex families are the target devices of this work, the Xilinx Partial Reconfiguration (PR) design flow [17] will be briefly introduced here. This flow is compatible with dynamic reconfiguration. Let a *Reconfigurable Functional Unit* (RFU) be a technologically mapped netlist implementing a required functionality and a *Reconfigurable Region* (RR) be a rectangular FPGA area where two or more RFUs are going to be placed and routed (at design time) and configured (at runtime) according to the application implemented in the reconfigurable system. PR allows the definition of a set of nonoverlapping RRs. All the *static logic* (i.e., all logic that will always remain configured, including the glue logic) is placed outside the reconfigurable regions, while they are allowed to use routing resources intersecting and even crossing RRs (the use of routing resources crossing RRs is the most relevant case). Figure 1(a) provides an example of a reconfigurable architecture developed with the PR design flow. Note how the reconfigurable regions are aligned to the grid defined by rows and CLBs according to PR requirements.

According to the PR flow, hardware macros can be placed on the boundaries of RRs in order to define *pins* where RFUs can hook themselves. Such macros are made with pairs of CLBs, one side of the CLB pair is connected to an RR signal, while the other is connected to a static logic signal. Previous design flows [18] required static logic being *placed* and also *routed* outside RRs, while the PR relaxes this constraint.

4.3. Target Architecture. The target architecture considered in this work is based on the PR design flow and has a static part implementing the communication infrastructure providing

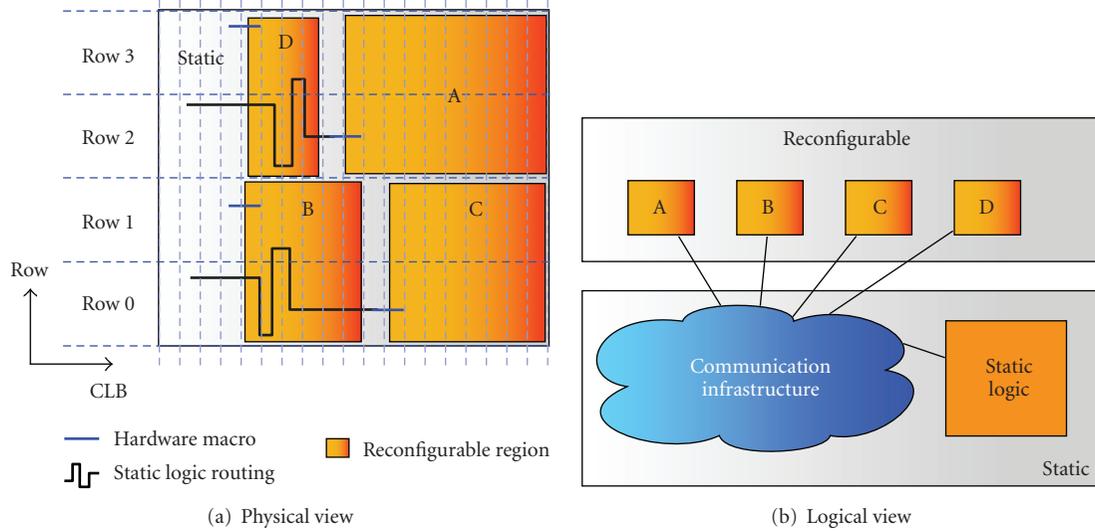


FIGURE 1: Target Reconfigurable Architecture based on PR: (a) physical and (b) logical views.

a number of interfaces at least equal to the number of RRs (each RR may provide more than one link to the communication infrastructure). Each RFU can communicate with other configured modules or with the static part by just hooking up to the hardware macros corresponding to the communication infrastructure interfaces. The most general view of the target architecture is given in Figure 1: from a physical point of view RRs communicate with the static part through to PR stand-alone static-logic nets routing, while from a logical point of view the static logic is responsible for managing intermodule communication implementing different communication infrastructures (such as bus-based, NoC-based, point to point, and so on). In conclusion, PR allows a more scalable communication infrastructure for partial dynamic reconfigurable architectures, since it allows an RFU to communicate with static logic regardless of the position of other RFUs. This approach simplifies the communication infrastructure design, while the frame size still needs to be considered during RRs floorplacement in order to prevent the configuration of an RFU inside an RR from interfering with other RFUs being executed on other parts of the FPGAs.

5. The Floorplacement Framework

In this section, we introduce our proposed framework for solving the floorplacement problem targeting the dynamically reconfigurable architecture and the reconfiguration technology described in the previous section.

The floorplacement framework accepts as input a scheduled task graph (TG) composed of a node for each RFU. A task graph is a Directed Acyclic Graph whose nodes represent a single task of a given application (or part of an application). A TG representation of an application has been chosen according to most of the related works [9–11, 13]. The TG can be scheduled according to different requirements (e.g., timing requirements or target device). Dividing time into

time slots the concept of static snapshot can be defined as the set of TG's nodes (i.e., tasks) that must be configured and must be running in a given time slot. A partial dynamic reconfigurable system can be seen as a finite state automaton according to the following definition.

- (i) *States*. There is one state for each time slot (hence one for each static snapshot).
- (ii) *Transitions*. The transition is a reconfiguration process.

After presenting the chosen scheduling technique, we will describe three algorithms that comprise our framework [5]. The floorplacement process starts with a *Partitioning* step, where RFUs are first grouped into RRs according to two criteria, wirelength for external routing to IOBs and utilization of resources. In the second step, once the partitions have been computed, the position of each RFU inside the corresponding RR needs to be determined. This is performed by the *Temporal floorplacement* step inside RRs. Finally, the design is completed by placing the RRs on the FPGA using the *Reconfigurable Regions Floorplacement* step.

5.1. Static Scheduling Phase. The heuristic used to compute the scheduled task graph has been defined starting from the Napoleon scheduler [19] and the ILP formulation proposed in [20]. This heuristic is a reconfiguration-aware scheduler for dynamically partially reconfigurable architectures that exploits *configuration prefetching*, *module reuse*, and also *antifragmentation* techniques.

In the following, nodes that have to be scheduled while their ancestors have already been will be called *available nodes*. The heuristic performs a list-based scheduling using as priority function the ALAP value of the tasks. This function has been slightly changed: an available task can be scheduled if (i) it has an ALAP value greater than the minimum ALAP value of the available nodes, (ii) if the possibility of

```

sLength ← 0
t ← 1
g ← readGraph()
setALAP(g)
RNs ← getRootNodes(g)
while ∃ not scheduled tasks do
  Control possibility of reuse for available tasks in RNs
  if ∃ not scheduled tasks then
    avTask ← getFirstALAPAvailableNode(RNs)
    endT ← findEndTime(avTask,t)
    while all the available nodes in RNs have been observed do
      if ∃ a position on the FPGA for avTask then
        avTask.terminationTime ← endT
        avTask.schedulingTime ← t
        avTask.setScheduled ← true
        if sLength < endT then
          sLength ← endT
        end if
        for all avTask child nodes chTask do
          if All chTask parents have been scheduled then
            RNs ← RNs + chTask
          end if
        end for
        Control possibility of reuse for available tasks in RNs
        avTask ← getNextALAPAvailableNode(RNs)
      end if
    end while
  end if
  t ← nextControlStep
end while

```

ALGORITHM 1: Heuristic pseudocode.

scheduling for all the available tasks with an ALAP value less than its own has been verified and (iii) if there is also *enough space* onto the FPGA.

Two antifragmentation techniques have been designed.

- (i) *Farthest Placement*. When a module needs to be reconfigured, it will be placed in the farthest position with respect to the center of the FPGA. We have to do this because when a large module (a module which is demanding many hardware resources) has to be placed, maintaining the emptiness of the center of the FPGA could increase the probability of placing large modules quickly. The same concept is applied to those tasks exploiting *module reuse*: when more than one module is available to be used on the FPGA, the farthest one with respect to the center of the FPGA is selected.
- (ii) *Limited Deconfiguration*. The deconfiguration policy leaves on the FPGA all modules that are not involved directly in the cleaning process (the creation of enough contiguous space for a new task, increasing the possibility of reuse of those modules).

Algorithm 1 shows the pseudocode of the proposed algorithm.

The most important functions used in Algorithm 1 are as follows.

- (i) *∃ a Position on the FPGA for avTask*. This function involves the placer that, using *antifragmentation techniques*, tries to place the current task *avTask*. This function takes into account also that if a module is being reconfigured, no other modules can be reconfigured onto the FPGA, and in this case it returns *false*. This function has not to be confused with a later phase of the proposed framework. In the scheduling phase this function is used to make the scheduler aware of the resource, while in the next phase the it will be used to properly manage the RFUs into the RRs.
- (ii) *Control the Possibility of Reuse for Available Tasks in Root Nodes (RN)*. The pseudocode of this function is presented in Algorithm 2, and it simply considers all the available tasks in ALAP order and verifies for each one if there is a module available to be reused.
- (iii) *nextControlStep*. This function returns the next time assignable to a task. This is done to reduce the complexity of the algorithm by reducing the number of iterations in the external *while* cycle. Not all the time instants are available to assign a task:

```

avTask ← getFirstALAPAvailableNode(RNs)
while ∃ an available task not yet considered do
  endT ← findEndTimeReusedTask(avTask,t)
  if ∃ a module usable by avTask then
    avTask.terminationTime ← endT
    avTask.schedulingTime ← t
    avTask.setScheduled ← true
    if sLength < endT then
      sLength ← endT
    end if
    for all avTask child nodes chTask do
      if All chTask parents have been scheduled then
        RNs ← RNs + chTask
      end if
    end for
  end if
  avTask ← getNextALAPAvailableNode(RNs)
end while

```

ALGORITHM 2: Reuse function pseudocode.

- when a task is being reconfigured, the scheduler cannot reconfigure any other task;
- when there is not enough available area on the FPGA to place any task, the scheduler has to wait for the termination of at least one running task;
- when a module exploits the *module reuse* concept and there are no available modules of the same type on the FPGA, the scheduler has to wait for the termination of at least one of those modules to schedule the selected tasks. For this reason *nextControlStep* assigns to the current time t a value given by 1 plus the minimum time between the last time in which the reconfiguration device is used and the first termination time of the tasks running on the FPGA.

In the worst case, the algorithm assigns only one task per time instant so the external *while* is executed $O(n)$ times where n is the number of tasks in the task graph, the control for reused tasks takes $O(f_o)$ time, where f_o is the maximum fanout of the nodes of the task graph, the internal *while* is executed $O(f_o)$. The functions that return the tasks in ALAP order can be designed by implementing the binomial search in $O(1)$ time, but in this case the process of inserting a new available node into RNs will take $O(\log f_o)$. Also the *for* used to verify the availability of the children nodes of *avTask* is executed $O(f_o)$ times. Hence the complexity of the algorithm in the worst case is $O(nf_o^2 \log f_o)$.

5.2. Partitioning into RRs. Given N RRs, the N -RRs partitioning problem consists of finding a surjective binding $c_{m,n}$ of RFUs into RRs (i.e., each RFU has to be bound to one and only one RR and each RR has to contain at least one RFU). Algorithm 3 firstly aims at grouping together

```

Buckets  $B$ ;
For all Externally connected RFU  $r$  do
   $B.add(r)$ ;
end for
Wirelength-driven-partition( $B$ );
Fix-existing-associations( $B$ );
for all Remaining RFU  $q$  do
   $B.add(q)$ ;
end for
Resource-driven-partition( $B$ );

```

ALGORITHM 3: Partitioning into RRs.

externally connected RFUs (i.e., RFUs that are connected to IOBs) having nearest centroids and keeping RFUs with distant centroids in different RRs. We refer to this as the *wirelength-driven partitioning*.

Secondly, the remaining RFUs are partitioned to minimize the variance of RRs' resource requirements along different static snapshots. In other words, for a given RR, the algorithm tries to keep the amount of resources needed by the RFUs configured and running inside the considered RR constant as much as possible across different static snapshots. We refer to this as the *resource-driven partitioning*.

The problem of wirelength aware partitioning of RFUs into RRs, can be reduced to the problem of clustering the corresponding centroids in a two-dimensional space (i.e., chip area). Each identified cluster is associated with one and only one RR and the RFUs are partitioned into RRs according to the association between their corresponding centroids and clusters (a partition belongs to an RR if and only if its centroid belongs to the cluster associated with the RR). Once the wirelength-driven partitioning has been performed, the created partition is used as an initial solution by the resource-driven partitioner to further partition the RFUs that are not externally connected. This means that the surjective binding $c_{m,n}$ is no longer modified for externally connected RFUs.

While data-mining algorithms provide several tools to solve the clustering problem (like the well-known k -means or fuzzy- k -means algorithm), they are primarily geared towards very large datasets. For our purposes, real life task graphs consist of fewer than a hundred RFUs and only a few of them are connected to IOBs. Hence, we adopted a simulated annealing-based approach.

Data Structure. Let us consider a bucket data structure having a set of buckets B_n for each Reconfigurable Region n . A given RFU m belongs to a bucket B_n (and only to that one) if and only if m is going to be placed in the Reconfigurable Region n at TIME(m).

Annealer's Moves. Let the simulated annealer's moves be the following.

- Randomly Move One RFU.* Move one module between two buckets: randomly pick up a module $m \in B_n$ and move to a bucket $B_{n'}$ where $n \neq n'$. This

move can be performed if and only if B_n contains another module $m' \neq m$.

- (ii) *Swap Two RFUs*. Swap modules belonging to different buckets: randomly pick up two modules m and m' , respectively, $m \in B_n$ and $m' \in B_{n'}$ such that $B_n \neq B_{n'}$. The move consists in swapping modules' buckets such that $m \in B_{m'}$ and $m' \in B_m$.

Once the partitions have been computed, the position of each RFU inside the corresponding RR needs to be determined.

5.3. Temporal Floorplacement inside Reconfigurable Regions. The aim of the Temporal Floorplacement inside Reconfigurable Regions (TFiRR) is to compute, for each RR, a set of height-width pairs describing rectangular areas where all RFUs bound to this RR can be successfully floorplaced. In this phase the final on-chip position of the rectangular area is not considered. For a target FPGA device that is divided by up to k rows for reconfiguration, the goal of this algorithm is to determine for each RR n a set of pairs

$$\Omega = \left\{ \langle n_h^1, n_w^1 \rangle, \langle n_h^2, n_w^2 \rangle, \dots, \langle n_h^k, n_w^k \rangle, \dots \right\} \quad (5)$$

such that an eventual actual placement of this RR on the device given as $A_n = \langle n_x, n_y, n_h^i, n_w^i \rangle$ for all n_x, n_y, i , results in a feasible floorplacement independently of the final position n_x and n_y decided for this RR. Here, h , w , x , and y stand, respectively, for the height, width, and the two coordinates of the bottom-leftmost corner of the rectangular area.

Consequently, for each RR n , the set of height-width pairs Ω can be described by providing just four elements (due to technological constraints related to Xilinx Virtex 4 and 5 FPGAs that are divided in 4 rows)

$$\Omega = \{ \langle 1, n_w^1 \rangle, \langle 2, n_w^2 \rangle, \langle 3, n_w^3 \rangle, \langle 4, n_w^4 \rangle \}, \quad (6)$$

where n_w^1 is the smallest width that RR n , floorplaced in 1 row, should have in order to feasibly host all the associated RFUs, n_w^2 is the smallest width that RR n (floorplaced in 2 rows) should have in order to feasibly host all the associated RFUs, n_w^3 and n_w^4 are the smallest widths that RR n , floorplaced in 3 and 4 rows, respectively, should have in order to feasibly host all the associated RFUs.

The core of the TFiRR step is the computation of the pairs $\langle i, n_w^i \rangle$. In order to find, for a given height i , the minimum feasible width n_w^i , the algorithm has to check that every RFU can be successfully floorplaced inside the area described by the pair $\langle i, n_w^i \rangle$, that is, for each RFU the algorithm has to provide a height, width, and a position within the RR n .

Such a problem is itself three-dimensional (i.e., two spatial dimensions and a temporal one). In order to simplify the problem the following assumption is introduced: *all RFUs' heights are equal to RRs' heights*. Fixing the height dimension of the RFUs, the problem is reduced to a bidimensional packing problem such that the static snapshot and the width are the only two considered coordinates. Given a RFU m , and a height i , the smallest position-independent width required by the RFU in order to be

hosted inside an area of height equal to i rows, can be easily computed by taking into account the FPGA's resources periodic distribution. The TFiRR algorithm works as follows.

For each RR n and each possible height $i \in \{1, 2, 3, 4\}$

- (1) consider RFU m such that $c_{m,n} = 1$ (i.e., RFUs belonging to RR n), let RFUs' height be equal to RR's height, then, compute the minimum feasible width of RFUs m ;
- (2) pack all the RFUs inside the RR in order to *minimize* the maximum width of RR n .

The packing of the RFUs can be performed with a zero temperature simulated annealing (ZT-SA) algorithm. For each static snapshot p an ordered list of RFUs m , such that $p \in \text{TIME}(m)$, is kept. The RFUs are ordered from the leftmost to the rightmost with respect to the RR's area. The following moves are applied.

- (1) *Randomly Move an RFU*. Randomly pick an RFU and move to an integer position belonging to the interval $[0, \text{width}]$, where width is the current width of the RR.
- (2) *Randomly Swap Two Concurrent RFUs*. Randomly pick two RFUs m' and m'' , being concurrently configured and running in at least one static snapshot and swap their position inside RR.

In order to keep the floorplacement compact, each step of the annealer is followed by a *compression* function that computes, for each RFU, the leftmost feasible solution preventing overlaps between RFUs. The computation of the objective function is the most expensive operation, requiring in the worst case $\Theta(R \cdot P)$ time (where R is the number of RFUs), but experimental data on randomly generated partitions indicate that in practice such complexity is asymptotic $O(R \cdot \log R)$. From a memory complexity point of view the algorithm requires only the management of a list for each static snapshot, hence the memory requirements are $\Theta(R)$.

The quality of this algorithm at first seems to be strictly related to the quality of the partition provided by the previous step. Our experiments showed that TFiRR applied on nonpartitioned task graphs can reach the results of TFiRR applied to a partitioned task graph by increasing the number of iterations by at least two orders of magnitude. On the other hand, the difference between the objective functions remains fairly low. We observed degradations ranging between 1–5%.

At the end of this second step, each resulting RR is annotated with a centroid whose coordinates are given by the arithmetic average of the corresponding coordinates of the RFUs associated with the considered RR. This identifies the ideal position where each RR should be placed in order to globally minimize the external wirelength of the associated RFUs. This particular formula for computing the centroid places more emphasis on the most heavily utilized IOBs. For example, if three RFUs are connected only to the USB interface, then all of them will have the same centroid C_{USB} that will occur three times in the set of centroids associated with the RR. In the third and final step, the centroids of the RRs will be used during the final floorplacement of RRs.

5.4. Reconfigurable Regions Floorplacement. The aim of this step is to define, for each RR n , an area

$$A_n = \langle n_x, n_y, n_h, n_w \rangle. \quad (7)$$

The algorithm has to choose one $\langle n_w, n_h \rangle$ couple, for each RR n , out of the set provided by the TFiRR step. Furthermore, it has to determine the specific x and y positions on the FPGA area. According to classical floorplanning this task can be performed through simulated annealing. The RR Floorplacement algorithm is divided in two steps: the first one floorplacing the RRs connected to IOBs (wirelength-driven RRs floorplacement) and the second one floorplacing the remaining RRs (area-driven RRs floorplacement).

Data Structure. To represent the floorplacement, a Horizontal Constraint List (HCL) is used for each row of the device. The HCL for row r is a list containing all the RRs occupying row r and ordered by increasing n_x .

Objective Functions. The first step is characterized by an objective function that must take directly into account the wirelength:

$$\Gamma = \left(\sum_{r \in \text{RR}} d(r, C(r)) \cdot \#\{\text{RFU} \in r\} \right)^f, \quad (8)$$

where $d(r, C(r))$ represents the distance of the RR from its ideal position (centroid), $\#\{\text{RFU} \in r\}$ is the number of RFUs connected to IOBs belonging to the r th RFU, while f is a positive number. If f is small, this indicates that the floorplacement is feasible and if it is large the floorplacement is not feasible. The goal of the floorplacement is to minimize Γ . Note that the objective function is weighted by the number of externally connected RFUs. This means that RRs containing more RFUs connected to the external world would benefit from a partitioning in the neighborhood of the centroid. Once the wirelength-driven RR floorplacement has been performed, the remaining RRs can be floorplaced by a purely area-driven algorithm. This second step is guided by an objective function involving free area and feasibility of the final floorplacement. Given an RRs' floorplacement, the following quantities are defined: *Negative area slack* (N), that is, the area of the floorplan crossing target device boundaries, and *positive area slack* (P), that is, the greatest contiguous free area starting from the right-top most corner of the device and with nonincreasing width going bottom-ward. Figure 2 shows an example of such slacks. Given such slacks, the RR Floorplacement objective function is defined as follows (where $M \in \mathbb{N}$ and greater than the number of frames on the target FPGA area):

$$\Theta = P - M \cdot N. \quad (9)$$

This objective function Θ is positive if the floorplacement is feasible (i.e., $N = 0$), otherwise it is negative (because $P < N$). The aim of the annealer is to maximize Θ , consequently, to provide a feasible floorplacement maximizing the contiguous FPGA area left free for static logic.

Annealers' Moves. Given the HCL data structure, the following moves are defined.

- (1) *Randomly Swap Two RRs.* Randomly choose two RRs and swap their positions.
- (2) *Move an RR to a Randomly Chosen Position.* Randomly pick an RR n and two coordinates $\langle \bar{x}, \bar{y} \rangle$ belonging to the device area.
- (3) *Span a Randomly Chosen RR over Rows.* Randomly choose an RFU, having height less than number of ROWS, and increase its height by 1 row.
- (4) *Unspan a Randomly Chosen RR.* Randomly choose an RFU, having height greater than 1, and decrease its height by 1 row. It is the inverse of the span move.

It can be noticed how the floorplacement of a large number of small functional units is easier than the floorplacement of a small number of large functional units, as shown in Figure 3. A bad choice of floorplacement of a large functional unit during the early stages of the floorplacement is difficult to correct in the later steps, particularly when temperature decreases rapidly and each correcting move is likely to be rejected because it results in a worse objective function.

5.5. Identifying Optimal Number of Partitions. The three steps described above comprise our floorplacement framework. The overall framework relies on the concept of partitioning, hence, some final remarks on how we control the granularity of these partitions will be useful. In order to identify the most suitable number of partitions let us consider the maximum number of concurrently configured C_C RFUs, that is, how many RFUs are present at most in one static snapshot in any partition. We define this quantity as follows:

$$C_C = \max \{ n_{p,b} \mid \#\{\text{RFU } r \mid p \in \text{TIME}(r) \wedge r \in B_b \} \}, \quad (10)$$

where $B_1, \dots, B_{\#\text{partitions}}$ represent the different output partitions. C_C can be considered as a good metric to describe the complexity of the entire partition. Let Γ be defined as the global normalized variance in resource requirements, used to represent the heterogeneity of the partitions. We have observed experimentally that the number of partitions minimizing the product of Γ and C_C provides (in resource-driven partitioning algorithm) a good tradeoff between partition complexity and intra-partition variance.

6. Experimental Results

Our proposed approach for the wirelength and resource management has been validated both on randomly generated task graphs and real-world applications from the domain of data processing for biomedical applications (i.e., collecting data from sensors, performing some preprocessing like FIR filtering, computing error detection codes, and sending data through a network). General consideration about the performance in finding the optimal results can be found

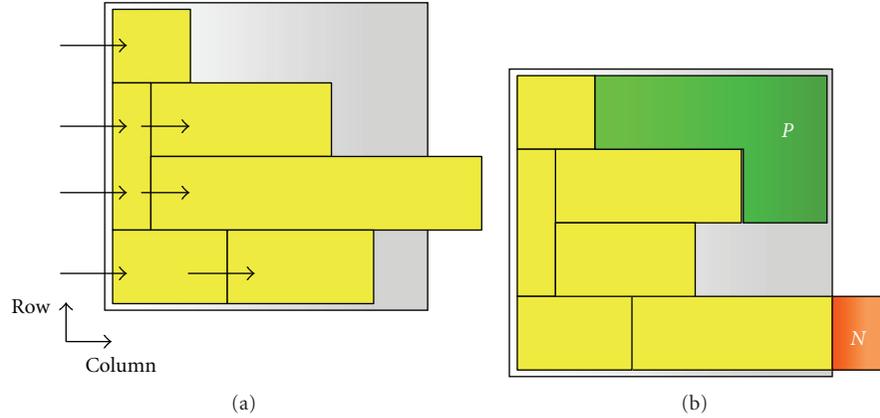


FIGURE 2: (a) A variation of horizontal constraint graph used for floorplan representation. (b) Negative (N) and positive (P) area slacks (the empty space on the second row is not included in P because it has a width greater than one of the empty areas in the upper row).

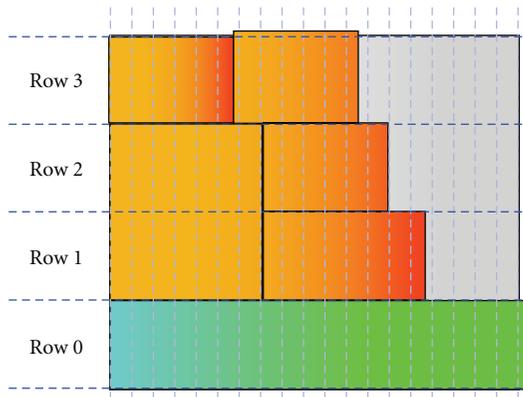


FIGURE 3: Limits of the floorplacement of big modules.

in [21]. The results introduced by this new approach are application dependent, therefore we will describe three different metrics used to evaluate the results. For each metric, we will provide the description of the task graphs. (Table 2 shows a summary of the metrics and a summary of our comparison with an existing floorplacement method that is only resource aware, but does not consider the wirelength implications of resource management [7].)

Our results confirm that introducing the wirelength awareness indeed improves the interconnect cost significantly. We observed a reduction of 90% in external wirelength in the best case and an average reduction of 50%. Task graphs with few externally connected RRs lead to the biggest wirelength reduction. On the other hand, the reduction for external wirelength in task graphs with many externally connected RRs with near centroids is less. In such task graphs only few RRs can be placed near their centroids and the other RRs have to be placed far away. The number of hardware macros provided by the static part can be reduced by 90% in the best case. Task graphs containing several RFUs accessing (in different static snapshots) the same set of IOBs (e.g., RFUs connected to the same external interface like USB) would particularly benefit from our approach, while task

TABLE 2: Quality metrics and the summary of the variation in these metrics compared with existing work [7].

Metric	Variation/Value
External wirelength	Reduction in external wirelength ranging between (90, 30)% compared to existing area-driven method
Links	Reduction in number of links required ranging between (90, 0)% compared to existing area-driven approach
Blank Area	(5, 35)% of the final floorplacement

graphs having all the RFUs connected to a distinct sets of IOBs would not benefit from links reduction (this is the 0% reduction case referred in Table 2).

Figure 4 shows how our proposed approach drastically reduces the number of required links.

In this figure we also observe one weakness of our approach. The most relevant drawback of our approach is referred to as the *blank area* problem (i.e., the amount of area being surrounded by RRs but not assigned to any RR). Let us consider a set of RRs, each one being externally connected and apply our approach several times, each time increasing the percentage of RRs that are considered (by the floorplacer) as attached to IOBs. During the first iteration no RR is considered as attached to IOBs, while during the last iteration all RRs are considered as attached to IOBs. Figure 5 plots the percentage of the final floorplacement that remains as blank area with respect to the percentage of RRs considered as attached to IOBs as a result of this experiment.

When no RR is considered as externally connected a 5% blank area is obtained (same as the purely area-driven approach). The peak is obtained when half of the RRs are managed by the wirelength-driven algorithm and the other half by the area driven, in such a case the blank area may reach 30–35% of the final floorplacement. On the other hand, the blank area is generally divided in no more than three or four areas that are wide enough to be used by the static part of the design according to the PR design

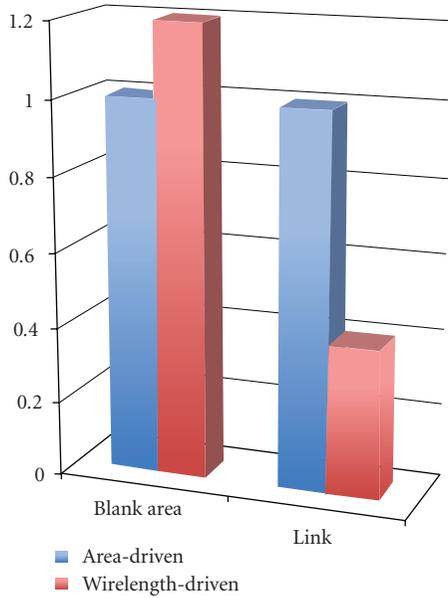


FIGURE 4: Comparison of the area and wirelength metrics among the area-driven and wirelength-driven approaches (normalized w.r.t. area-driven approach).

flow. Figure 5 shows also how the first externally connected RRs obtain a great wirelength improvement, while the latest one cannot obtain such improvement due to the previously introduced nonoverlapping constraints.

Therefore, we observe that our approach can yield significant improvements in the cost of the communication infrastructure by tradingoff a reasonable amount of blank area. Using our framework, the designer can chose between (a) considering more RRs as externally connected, thereby, decreasing external wirelength or (b) considering that beyond a certain point blank area overcomes benefits provided by the wirelength-driven approach. Hence, the designer may decide to floorplace in a wirelength-driven way only for the most relevant RRs leaving the others to be floorplaced by an area-minimizing approach. Finally, we observe that the blank area problem is not an issue for task graphs requiring most of the resources of the target FPGA (because the feasibility of the floorplacement requires as much area as possible to be used, hence blank area is reduced as a consequence) and for task graphs having few RFUs connected to IOBs (or many of them connected to the same IOBs).

7. Conclusions and Future Work

In this paper, we presented a resource- and configuration-aware floorplacement framework, tailored for Xilinx Virtex 4 and 5 FPGAs, using an objective function based on *external wirelength*. The proposed approach has achieved a shorter external wirelength and a highly increased probability of reuse of existing communication links. The reduction in wirelength ranges from 30% to 90% in comparison to a purely area-minimizing approach. Task graphs with few

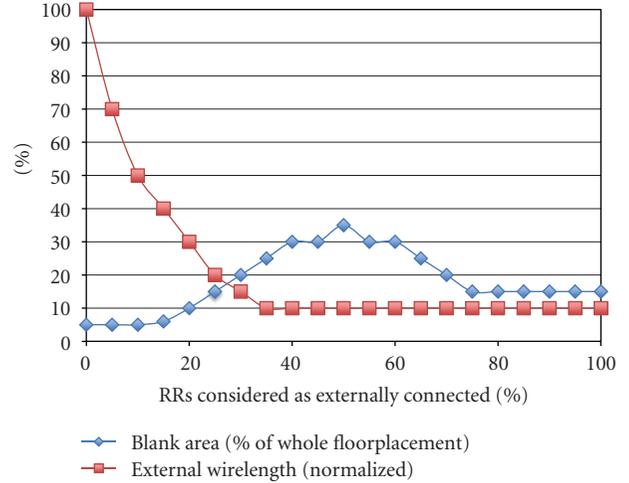


FIGURE 5: Example of wirelength reduction (normalized) and blank area left (percentage of the final floorplacement) for the floorplacement of only externally connected RRs plotted with respect to the percentage of RRs that are considered as externally connected by the algorithm.

externally connected RRs lead to the biggest decrease, while external wirelength in task graphs with many externally connected RRs show lower improvement. Future improvement for the work presented in this paper can be done in considering a hybrid approach between the area and the wire length solution. Furthermore we aim at directly addressing the blank area problem by modifying the objective function or floorplacing all the RRs, not connected to the IOBs, around the RRs connected to IOBs, trying to keep the RRs as close as possible in order to further reduce the blank area.

References

- [1] *ISE 9.2i Manual*, Xilinx Incorporation, 2007.
- [2] V. Betz and J. Rose, *VPR: A New Packing, Placement and Routing Tool for FPGA Research*, Springer, London, UK, 1997.
- [3] J. A. Roy, S. N. Adya, D. A. Papa, and I. L. Markov, “Min-cut floorplacement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1313–1326, 2006.
- [4] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa, and I. L. Markov, “Unification of partitioning, placement and floorplanning,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers (ICCAD ’04)*, pp. 550–557, November 2004.
- [5] A. Montone, M. D. Santambrogio, and D. Sciuto, “Wirelength driven floorplacement for FPGA-based partial reconfigurable systems,” in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW ’10)*, pp. 1–8, 2010.
- [6] Y. Feng and D. P. Mehta, “Heterogeneous floorplanning for FPGAs,” in *Proceedings of the IEEE 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, vol. 2006, pp. 257–262, 2006.
- [7] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik, “A reconfiguration-aware floorplacer for FPGAs,” in

- Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 109–114, December 2008.
- [8] S. N. Adya and I. L. Markov, “Fixed-outline floorplanning: enabling hierarchical design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1120–1135, 2003.
- [9] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “3-D floorplanning: simulated annealing and greedy placement methods for reconfigurable computing systems,” in *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping (RSP '99)*, pp. 38–43, June 1999.
- [10] M. Vasilko, “Dynasty: a temporal floorplanning based cad framework for dynamically reconfigurable logic systems,” in *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL '99)*, pp. 124–133, Springer, London, UK, 1999.
- [11] P. H. Yuh, C. L. Yang, and Y. W. Chang, “Temporal floorplanning using the T-tree formulation,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers (ICCAD '04)*, pp. 300–305, November 2004.
- [12] P. H. Yuh, C. L. Yang, and Y. W. Chang, “Temporal floorplanning using the three-dimensional transitive closure sub-Graph,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 4, article 37, 2007.
- [13] L. Singhal and E. Bozorgzadeh, “Multi-layer floorplanning on a sequence of reconfigurable designs,” in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–8, 2006.
- [14] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, “VLSI module placement based on rectangle-packing by the sequence-pair,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 12, pp. 1518–1524, 1996.
- [15] *Virtex 5—Family Overview*, Xilinx Incorporation, 2007.
- [16] Xilinx Inc., “Virtex-5 user guide,” Tech. Rep. ug190, Xilinx Inc., 2007, <http://www.xilinx.com/bvdocs/userguides/ug190.pdf>.
- [17] *Partial Reconfiguration User Guide*, Xilinx Incorporation, 2010.
- [18] *Xilinx Application Note 290*, Xilinx Incorporation, 2007.
- [19] 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 Design, Automation and Test in Europe (DATE '08)*, pp. 519–522, March 2008.
- [20] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, “Partitioning and scheduling of task graphs on partially dynamically reconfigurableFPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, 2009.
- [21] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik, “A reconfiguration-aware floorplacer for FPGAs,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 109–114, 2008.



Hindawi

Submit your manuscripts at
<http://www.hindawi.com>

