Automated Partitioning for Partial Reconfiguration Design of Adaptive Systems

Kizheppatt Vipin, Suhaib A. Fahmy
School of Computer Engineering
Nanyang Technological Univesity
Nanyang Avenue, Singapore
{vipin2,sfahmy}@ntu.edu.sg

Abstract—**Adaptive systems have the ability to respond to environmental conditions by modifying their processing at runtime. This can be implemented by using partial reconfiguration (PR) on FPGAs. However, designing such systems requires specialist architecture knowledge and an understanding of the mechanics of reconfiguration, as the design process is completely manual. One design choice that must be made, which impacts system efficiency significantly, is how to group reconfigurable modules and assign them to reconfigurable regions on the FPGA. In this paper, we present an approach, based on graph clustering, that finds a partitioning that minimises reconfiguration time, given an application description and target FPGA. The resulting allocation respects all the constraints set by the official tool flow while raising the level of design abstraction, allowing non-expert designers to leverage this capability of FPGAs.**

Index Terms—**Field programmable gate arrays; partial reconfiguration; design automation.**

I. INTRODUCTION

Adaptive systems are able to adapt their functionality to variations in their environment, leading to more sophisticated applications and improved system performance. For example, a cognitive radio can switch between sensing and transmission modes autonomously, without the need for both circuits to be on the FPGA at the same time [1]. Field Programmable Gate Arrays (FPGAs) are finding increased use in a wide range of application domains due to their high performance and flexibility. Partial reconfiguration (PR) is a promising technique for implementing dynamic hardware systems on FPGAs. It allows modification of portions of the system, while the remaining parts continue to function without interruption, through partial modification of the configuration memory. This allows for more fine-grained flexibility, time-multiplexing of multiple functions on smaller FPGAs, and hence, a reduction in power consumption and cost.

Designing PR systems, however, presents a number of challenges which mean adoption has not been widespread. Current PR tools require considerable input from the designer, and the efficiency of the implementation depends significantly on how the design is manually partitioned and floorplanned on the FPGA. Both of these design choices are closely related to the FPGA architecture and detailed aspects of the PR operation. The result is that PR is less attractive to system designers who are not FPGA experts.

A PR system consists of a number of modules, that together undertake the required processing in the system. At any point in time, a subset of these modules may be required. These modules are implemented in reconfigurable regions (RRs). These are areas of the FPGA fabric, designated at design time, which can be reconfigured at runtime. Determining the number of reconfigurable regions and allocating modules to them constitutes the design partitioning step.

When designing PR systems, one primary cost we aim to reduce is reconfiguration time. This is the time taken to configure the system from one operating mode to another. For several domains such as ad-hoc communication, space applications, real-time systems, and others, long reconfiguration times can adversely impact system performance. Since reconfiguration time is primarily proportional to the area being reconfigured, partitioning needs to be performed with due consideration of these factors.

In this paper, we propose a scheme for automatically determining the region allocation with minimal reconfiguration time for a given application and FPGA device. We consider the latest generation of FPGA architectures and development tools. The proposed algorithm uses detailed architecture information to determine how best to group modules to minimise reconfiguration time. While some related work on partitioning for time-multiplexing static task graphs on FPGAs has been presented in the past, we are interested in adaptive systems, where the pattern of reconfiguration is not known in advance. The proposed method uses the limited information available in such cases to deduce an optimal partitioning.

The rest of this paper is organised as follows: Section II discusses related work. Section III describes important factors in PR design and investigates the existing flow for Xilinx FPGAs, Section IV presents the algorithm used for partitioning. Section V presents some results for an example application and some synthetic applications, and Section VI concludes the paper.

II. RELATED WORK

The authors of [2] identified some of the challenges associated with PR designs and introduced partitioning, although no specific algorithm was provided. A method for reducing reconfiguration time using integrated temporal partitioning and partial reconfiguration was introduced in [3]. It describes a partially reconfigurable processor with two reconfigurable

172

regions for execution speed up. Similar work is presented in [4], in which a pre-fetching schedule is generated based on control flow graphs. Unfortunately for adaptive systems, direct scheduling based on task graphs is not possible due to the lack of a-priori knowledge of reconfiguration sequence. In these previous efforts, a static task graph is dynamically scheduled on limited resources. In the case of adaptive systems, the task graph itself is dynamic as it responds to environmental conditions.

In [5], the authors present a method for minimising reconfiguration latency based on analysing communication graphs. The algorithm tries to group modules with the most communication between them into the same reconfigurable region. However, the number of reconfigurable regions must be determined by the designer. Determining the number of regions is not straightforward, and current devices and tools do not provide any support in this respect.

In [6], the authors describe run-time temporal partitioning for reducing FPGA reconfiguration time. Here, the number of reconfigurable regions is fixed and resources are assumed to be homogeneous. The number and size of the regions needs to be determined by the designer.

A more recent paper that explores partitioning and floorplanning of PR designs is [7]. The authors describe a simulated annealing based algorithm for determining the module allocation to regions based on minimisation of area requirement variance at different time instances. This work considers the latest FPGA architectures as well as PR requirements. It is difficult to extend this work for adaptive systems because the algorithm presented makes use of a scheduled task graph. Moreover the impact on reconfiguration time is not accounted for in their method.

Most existing work we have found does not perform partitioning in a manner that considers the runtime aspects of partial reconfiguration and does not consider the latest FPGA architectures. In our previous work [8], we introduced a method for PR partitioning with the aim of minimising area and average reconfiguration time. The optimal solution was determined based on solving several exact equations. This paper improves on our previous work by allowing more flexible module arrangements, and optimises primarily for reconfiguration time, by using all the resources available in the target FPGA, since this is a key factor in the implementation of such systems. Developing a general method that can be used for adaptive systems with dynamically changing processing needs, while considering the limitations of the target architecture, is thus important to the wider adoption of partial reconfiguration.

III. PR Tool Flow

A. Present PR Tool Flow

Presently, the only vendor-supported PR flow is available from Xilinx through their PlanAhead software. Altera announced support for partial reconfiguration in their Stratix V devices [9], but the software tools are not yet publicly available. From the available documents, Altera's PR design flow is
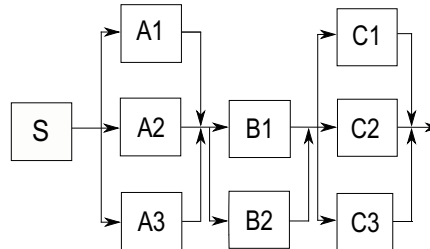

Fig. 1. An example PR design with a static region (S) and 3 modules (A, B, and C).

similar, including manual partitioning and floorplanning [10]. In our discussions, we will focus on the available Xilinx toolchain and associated devices.

Fig. 1 shows an example PR design. The design is divided into *static logic* and *reconfigurable modules*. The functionality of the static logic does not change during operation. The static logic usually contains an embedded processor, the internal configuration access port (ICAP), and other fixed circuitry. The embedded processor runs the configuration management software, which controls system transition from one configuration to another depending upon the adaptation conditions set by the application. The ICAP is used to load partial bitstreams into the FPGA configuration memory.

A PR system consists of a number of modules. At the system level, we can think of a *module* as a processing unit that may have multiple *modes*. Modes are mutually exclusive implementations of a module that might be activated at different points in time. with compatible inputs and outputs. For example, a filter module can have two modes, one acting as a high-pass filter and one acting as a low-pass filter. At runtime, a module may switch from one mode to another. In Fig. 1, module S represents the static logic and modules A, B, and C represent reconfigurable modules with modes A_1, A_2, A_3; B_1, B_2; and C_1, C_2, C_3 respectively.

A *reconfigurable region* (RR) is an area on the device allocated to logic during design time, that is reconfigured during runtime. It includes different types of basic primitives such as configurable logic blocks (CLBs), BlockRAMs, and DSP Slices. Normally, it is up to the designer to specify a region large enough to implement all the modules assigned to that region, requiring FPGA architecture knowledge. The designer must then use the implementation tools to build netlists for each region, for different modes. Two standard approaches are to put all reconfigurable modules into a single reconfigurable region, or to put each module into a separate region.

The set of valid combinations of modes encountered at runtime are called the *configurations*. In the example design, $S \rightarrow A_1 \rightarrow B_1 \rightarrow C_1$ may be a possible configuration. Each configuration contains the static logic and one of the possible modes for each reconfigurable module. It is possible to have configurations, where some modules do not exist. For example, $S \rightarrow A_2 \rightarrow B_2$, in which module C does not
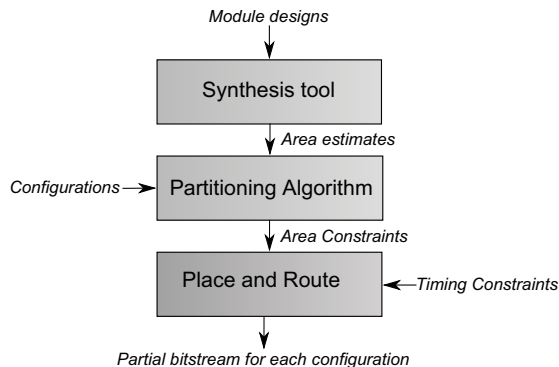
Fig. 2. Proposed partial reconfiguration tool flow.

exist. Typically the valid configurations are only a subset of the possible configurations. The example design here could be defined as having 5 configurations:

$$S \to A_3 \to B_2 \to C_3$$
$$S \to A_1 \to B_1 \to C_1$$
$$S \to A_3 \to B_2 \to C_1$$
$$S \to A_1 \to B_2 \to C_2$$
$$S \to A_2 \to B_2 \to C_3$$

In the current vendor tool flow, configurations do not play any role in synthesis, since the reconfigurable modules and the assignment of regions, are performed manually. The designer must prepare netlists for valid combinations of module modes for each region. However, these configurations play an important part in finding a good partitioning.

The designer must also decide where to place the regions on the FPGA, ensuring sufficient resources to implement all reconfigurable modules assigned to each region. Hence, PR design, as supported currently, is very much a floorplanning activity.

In our work, we consider configurations, allowing us to minimise the search space when determining a suitable partitioning. Unlike some previous work, our method does not require the the designer to set the number of PRRs to be used, or the size and composition of PRRs for partitioning.

B. Proposed Tool Flow

The proposed partitioning algorithm is part of a design flow we are developing for high-level design of adaptive systems using PR. Our proposed tool flow for PR is shown in Fig. 2. The designer provides design files for all modules (in all modes), a list of valid configurations, and design implementation constraints such as timing constraints and target FPGA device to the tool in XML format. The tools perform the following steps:

1) Xilinx *XST* is used to synthesise all the modes to determine resource requirements. If IP cores are used for some modules, resource usage is often available up front. FPGA resource numbers are obtained from a device

library that details the number of CLBs, Block RAMs and DSPs for various families and devices.

2) The resource requirements are passed to the partitioning algorithm, which performs an automated region allocation that minimises reconfiguration time while meeting the area constraints of the FPGA device selected.

3) Wrapper modules are created that group together modes that have been combined in the partitioning phase.

4) A netlist for each partition is then automatically generated using vendor synthesis tools.

5) Floorplanning of the static and reconfigurable regions must then be performed. We use the approach in [11]. Alternatively, a manual approach using Xilinx *PlanAhead* is also possible.

6) The area constraints generated by the floorplanner along with timing requirements, are used to generate the *User Constraints File*. Together with the netlists, this is passed to *PlanAhead*, which performs the placement and routing operations according to the constraints.

7) Finally, a complete configuration bitstream and partial bitstreams for each region under different configurations are generated.

IV. PROBLEM FORMULATION

A. Background

Designers generally adopt two simple methods for partitioning PR designs. The first is to assign all reconfigurable modules to a single region and the second is to assign each module to a separate region, allowing each region to be reconfigured to one of the modes at any point in time. Consider two modules, A and B, as an example. Each module has a large (A_2 and B_1) and a small mode (A_1 and B_2), as shown in Fig. 3. Assume then, that three valid system configurations are defined as $A_1 \to B_1$, $A_2 \to B_2$, and $A_1 \to B_2$. The static implementation with all the module modes implemented concurrently requires minimum reconfiguration time, equal to a few clock cycles to switch the multiplexers to change the modes, but the total resource consumption will be the sum of the resource requirements of all modes. This is often infeasible due to restricted resources, and is inefficient since a larger FPGA with higher power consumption must be used, despite the fact that parts of the circuit are never active at the same time.

If each module is assigned to a separate region, each region must be large enough for the largest mode of the corresponding module assigned to it. For the example, in Fig. 3, there would be two regions: one large enough for A_2 and the other large enough for B_1, and hence the total resource requirement is the sum of the resource requirements of these two modes. Any system reconfiguration will require configuring at least one reconfigurable region, and in one case both (when the system switches from $A_1 \to B_1$ to $A_2 \to B_2$). However, if the modules were combined into a single region, that region only needs to be large enough for the largest overall configuration. In this case, the size of
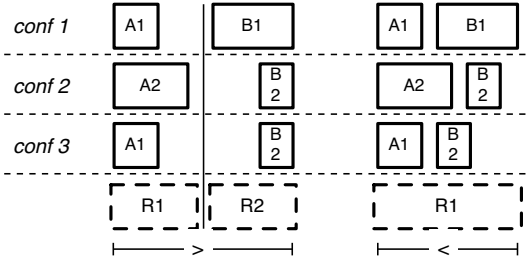
Fig. 3. When assigning modules to separate regions, if some configurations do not exist, combining modules into a single region can save area.

the single region will be that of $\{A_1, B_1\}$. This is because the largest modes, A_2 and B_1, do not co-exist among the possible system configurations. This scheme gives the lowest resource requirement. Here, the drawback is that any system reconfiguration requires reconfiguring the entire region, which increases the reconfiguration time for certain transitions.

Finally consider a hybrid case. Suppose A_2 and B_1 are implemented in a single region and A_1 and B_2 are implemented as static logic. The total resource consumption in this case will be the sum of B_1, A_1 and B_2. A close examination will reveal that the resource consumption in this case is close to that of assigning each module to a separate region, and much less than a complete static implementation. Since A_1 and B_2 are in static logic, several configuration switches do not require reconfiguration, thus reducing overall reconfiguration time.

The example shows that there is no one simple solution, and that configurations are key to determining an efficient allocation. Although adaptive systems are dynamic in nature, we can often determine a restricted set of valid configurations. Hence, the system designer can be aware of valid configurations, although it may be impossible to determine the order in which the system will switch between them, since this depends on environmental conditions.

The important conclusions from this initial analysis are:
- The minimum area required to implement a design is the area required for the largest configuration,
- reconfiguration time is proportional to the area being reconfigured,
- minimising the number of modes implemented in reconfigurable regions reduces reconfiguration time,
- modes that always reconfigure together should be placed in the same region.

The major constraints associated with PR design are resource availability in the target FPGA and reconfiguration time. The target FPGA may already be decided, or alternatively, the designer may like to determine the most suitable FPGA. The tool should be enable us to optimise for both these scenarios. Once an FPGA is determined, it makes sense to make full use of the available resources, since trying to minimise area would then have minimal impact on design cost, while likely impacting reconfiguration time significantly.
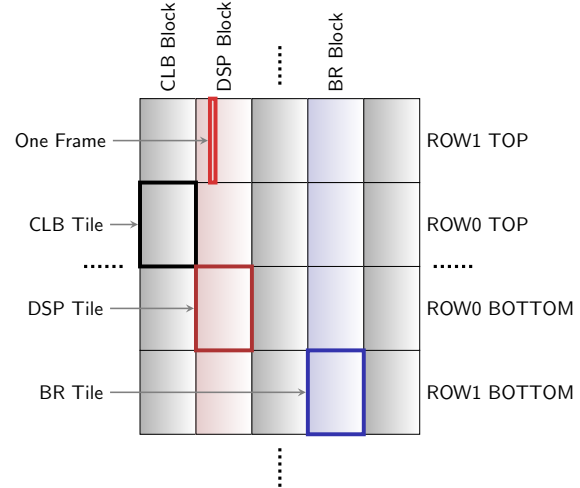
The partitioning problem can be modelled as: Given:

Fig. 4. Xilinx Virtex 5 FPGA architecture.

1) an FPGA with specific resource availability (optional),
2) a set of design modules and their modes,
3) a set of valid configurations,

partition the design in such a way that:
1) the design fits in the FPGA,
2) all configurations are implemented,
3) all configuration transitions are possible,
4) total reconfiguration time is minimised.

B. Architecture Analysis

This section discusses how area is calculated for Xilinx FPGAs. Xilinx Virtex FPGAs are divided into rows[1] and columns. The number of rows in a device depends upon the size of the device. Resources such as CLBs, Block RAMs etc. are arranged in a columnar fashion extending the full height of the device and these columns are referred to as *blocks*. A tile is one row high and one block wide, and contains a single type of resource, as shown in Fig. 4, and is the smallest unit that can be partially reconfigured using the supported flow. One CLB tile contains 20 CLBs, one DSP tile contains 8 DSP Slices, and one BRAM tile contains 4 Block RAMs. To properly determine the area of a region, we must consider it in terms of these basic tiles since configuration must occur on a per tile basis. It is possible to allocate partial tiles to a region, but this creates other constraints: the remainder of the tile must not be allotted to any other region, and additional circuitry is required to read the contents of the configuration memory, add the reconfiguration information, then re-write, hence, we avoid this. Xilinx imposes additional constraints such as PR regions being rectangular in shape and not overlapping.

In Virtex FPGAs, the smallest addressable unit of configuration memory is called a *configuration frame* (frame). In Virtex-5 FPGAs, the height of a frame spans an entire device

---

[1]Note "rows" here refers to a Xilinx term used in their architecture descriptions.

row [12]. One CLB tile contains 36 frames, a DSP tile 28 frames and a BRAM tile 30 frames. The area requirement of a region can hence be calculated in terms of frames using the formula

$$Area_r = \sum_i W_i * R_{ri} \quad (1)$$

where i is the tile type, $i \, \epsilon$ (CLBs, DSP blocks, BlockRAMs), R_{ri} is the number of tiles of type i present in region r, W_i is the number of frames in tile type i.
A frame contains 41 words or 1312 bits of data, mapped to an equivalent space in the configuration memory.
Since reconfiguration time is proportional to the amount written to the configuration memory, it is proportional to the total number of frames being reconfigured. The actual reconfiguration time also depends upon additional factors such as the delay in fetching partial bitstreams from external memory and transfer speed through the internal configuration interface (ICAP).

C. Algorithm

The proposed algorithm tries to determine the best partitioning scheme for a given PR system. This algorithm can be used to find the best partition for a given FPGA or can suggest the smallest FPGA suitable to implement the given design.

The minimum possible area required for system implementation will be the area of the largest configuration (when all the modes are implemented in a single reconfigurable region). Hence, the algorithm first checks implementation feasibility by comparing this area with the resource availability of the given FPGA. If the resource availability is insufficient, the device choice is rejected and another device must be chosen. If a solution is feasible, a connectivity matrix is generated with each row representing a configuration and each column representing a reconfigurable module. An element (i, j) in the matrix with value 1 represents mode j being present in configuration i. For the example design in Section-III, for the 5 configurations, the matrix representation is given below.

$$
\begin{bmatrix}
 & A_1 & A_2 & A_3 & B_1 & B_2 & C_1 & C_2 & C_3 \\
Conf.1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
Conf.2 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
Conf.3 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
Conf.4 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
Conf.5 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
$$

This matrix is used to determine weights for use in the optimisation. The *node weight* of a mode is the number of times that mode appears in the possible configurations. It is the columnar sum of corresponding modes from the matrix. For mode A_1 in the example, the node weight is 2 and for B_2, it is 4. The *edge weight*, W_{ij} between any two modes i and j is the number of times these modes occur concurrently in the possible configurations. For modes A_1,B_1, the edge weight is 1 and for B_2,C_3, it is 2.

Once the weights are calculated, a modified hierarchical clustering algorithm [13] with agglomerative strategy is used for partitioning. The metric used for clustering is the edge

weight, W_{ij}. The agglomerative strategy is a bottom-up clustering method, which iterates by adding new edges between the nodes in a network. Here, the nodes are the different modes present in the system. Initially all nodes are disconnected. Now the algorithm checks for complete sub-graphs in the network. A complete sub-graph is a sub-graph, where every pair of distinct vertices is connected by a unique edge. Since initially none of the nodes are connected, each node can be considered as a sub-graph with number of edges, $k = 0$.

The algorithm iterates and in each iteration, it links the two nodes with the highest edge weight. The rationale for this is that a larger edge weight indicates that two modes occur concurrently more frequently for the given configurations, and hence these modes should be grouped in the same region. Once two nodes are connected, the algorithm checks for new complete sub-graphs. This is shown in Fig. 5(a). The edge value between A_3 and B_2 is 2, which is the highest. In this case A_3 and B_2 are linked together. A search for new complete sub-graphs finds $\{A_3,B_2\}$ with number of edges, $k = 1$.

The sub-graphs found in each iteration are called *base partitions*. Base partitions represent the set of mode clusters which can be used to determine the final partitioning. The frequency of occurrence of a base partition in the configurations is represented by a term called *frequency weight*. For sub-graphs with $k = 0$, *frequency weight* is equal to the node weight and for sub-graphs with k=1, the *frequency weight* is equal to the edge weight. For sub-graphs with a higher number of edges, the *frequency weight* is the smallest edge weight present in the sub-graph. For example, in Fig. 5(b), the *frequency weight* of sub-graph $\{A_3,B_2,C_3\}$ is 1, which is the edge weight between A_3 and C_3. The algorithm iterates until all the possible links are added to the graph. The final sub-graphs detected will be the full configurations, with *frequency weight* 1. The *base partitions* for the example design are listed in Table I

Once *base partitions* are generated, a covering algorithm is used to select those used for partitioning. For this purpose, the *base partitions* are arranged in a list in ascending order of the number of modes included. As the number of modes in a region increases, the frequency of reconfiguring that region increases, since modifying even a single mode in the region requires the complete reconfiguration of the whole region. Since our objective is to minimise reconfiguration time,
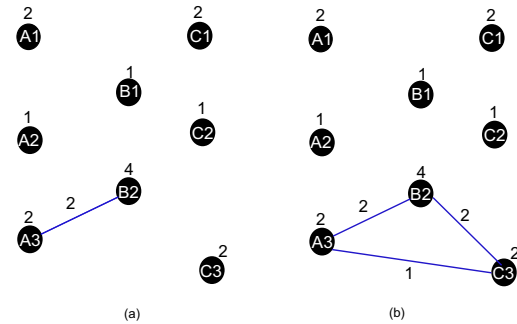

Fig. 5. (a) a sub-graph with $k = 1$. (b) a sub-graph with $k = 3$.

TABLE I
BASE PARTITIONS WITH THEIR FREQUENCY WEIGHT.

Base Part'n	Freq wt	Base Part'n	Freq wt
$\{A_2\}$	1	$\{A_1, C_2\}$	1
$\{C_2\}$	1	$\{A_1, B_1\}$	1
$\{B_1\}$	1	$\{B_1, C_1\}$	1
$\{A_1\}$	2	$\{A_2, C_3\}$	1
$\{C_1\}$	2	$\{A_3, C_1\}$	1
$\{C_3\}$	2	$\{A_3, C_3\}$	1
$\{A_3\}$	2	$\{B_2, C_3\}$	2
$\{B_2\}$	4	$\{A_3, B_2\}$	2
$\{A_1, B_2\}$	1	$\{A_3, B_2, C_3\}$	1
$\{B_2, C_1\}$	1	$\{A_1, B_1, C_1\}$	1
$\{A_1, C_1\}$	1	$\{A_3, B_2, C_1\}$	1
$\{B_2, C_2\}$	1	$\{A_1, B_2, C_2\}$	1
$\{A_2, B_2\}$	1	$\{A_2, B_2, C_3\}$	1

regions are prioritised based on the number of modes. If two *base partitions* have the same number of modes, they are arranged in ascending order of *frequency weight*. Subsequent steps of the algorithm show that this prioritisation keeps the high frequency *base partitions* as candidates when the algorithm iterates. *Base partitions* with the same frequency weight are arranged in ascending order of their area.

Now *base partitions* are selected from the list in sequence order and compared with the connectivity matrix. For each configuration, i.e. for each row in the connectivity matrix, the corresponding modes present in the selected *base partition* are set to zero. For example, the first *base partition* selected from the list is $\{A_2\}$. For the fifth configuration, A_2 is active. The corresponding element A_2 is set to zero and the fifth row of the connectivity matrix becomes

$$[0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1]$$

Subsequently, *base partitions* $\{C_2\}$, $\{B_1\}$ etc. are used to cover more configurations. *Base partitions* are selected and compared from the list until all elements in the matrix become zero. If a *base partition* does not cover any new mode, it is not considered as a candidate. The set of *base partitions* used to cover all configurations becomes a *candidate partition set*. In other words, a *candidate partition set* is a set of *base partitions*, whose modes can cover all the possible configurations. For the example design, the first *candidate partition set* is $\{\{A_2\}, \{B_1\}, \{C_2\}, \{A_1\}, \{C_1\}, \{C_3\}, \{A_3\}, and \{B_2\}\}$. A closer examination shows that these are actually all the modes present in the design.

As the next step, the tool finds the compatible set of partitions for each *base partition* from the *candidate partition set*. Two partitions are compatible, if the modes present in them do not co-occur in any of the configurations. For example $\{A_1\}$ and $\{A_2\}$ are compatible partitions since they do not co-exist in any of the possible configurations, while $\{A_1\}$ and $\{B_1\}$ are not compatible, since there is a configuration $S \rightarrow A_1 \rightarrow B_1 \rightarrow C_1$. This step is necessary to make sure that all configuration transitions are possible. If two base partitions required for a single configuration are allocated to the same region, that configuration cannot be implemented since at a

given instance, only one *base partition* will be active in a configurable region.

Region allocation starts by allocating each element of the *candidate partition set* to a separate region, since this is equivalent to the static implementation which requires minimum reconfiguration time. The total resource requirement and reconfiguration time for this partitioning is calculated. To find a new solution, two compatible *base partitions* are assigned to the same region. The cost function for assigning two *base partitions* to a single region is calculated in terms of the total number of frames being reconfigured considering all the configuration transitions. When two *base partitions* with area P_1 and P_2 (in terms of frames) are assigned to the same region r, the area of the region is calculated as,

$$P_r = \max(P_1, P_2) \tag{2}$$

The area of the region will be the area of the largest *base partition* assigned to it. To find the exact number of frames present in the region, the region is considered in terms of CLB, DSP, and BlockRAM tiles. Depending upon the number of resources present in each tile as described in section IV-B, the number of tiles required for each resource type for region r is calculated as.

$$R_{rclb} = \lceil \max(P_{1clb}, P_{2clb})/20 \rceil, \tag{3}$$

where R_{rclb} is the total number of CLB tiles required.

$$R_{rdsp} = \lceil \max(P_{1dsp}, P_{2dsp})/8 \rceil, \tag{4}$$

where R_{rdsp} is the total number of DSP tiles required.

$$R_{rbr} = \lceil \max(P_{1br}, P_{2br})/4 \rceil, \tag{5}$$

where R_{rbr} is the total number of BlockRAM tiles required.

If the total resource requirement of the partition for each resource type is less than or equal to the resources available in the FPGA, the reconfiguration time is calculated.

The total number of frames required for the new region is calculated as

$$P_r = \sum_t W_t * R_{rt} \tag{6}$$

where t is the tile type, $t \; \epsilon$ (CLBs, DSP blocks, BlockRAMs), $W_{clb} = 36$, $W_{dsp} = 28$, $W_{br} = 30$.

System performance can be measured in terms of total reconfiguration time and worst-case reconfiguration time. Total reconfiguration time gives a measure of overall system performance, and is a useful proxy when we do not know the specific configuration transitions up front, as is the case for adaptive systems. Total reconfiguration time is measured as the sum of all possible configuration transitions, i.e. by considering transitions from all configurations to all other configurations. If some statistical information about the probabilities of different configurations occurring is known, this could be factored into the measure.

In some applications, such as real time systems and safety critical systems, the system cannot tolerate reconfiguration time beyond a certain limit. Here it is important that no
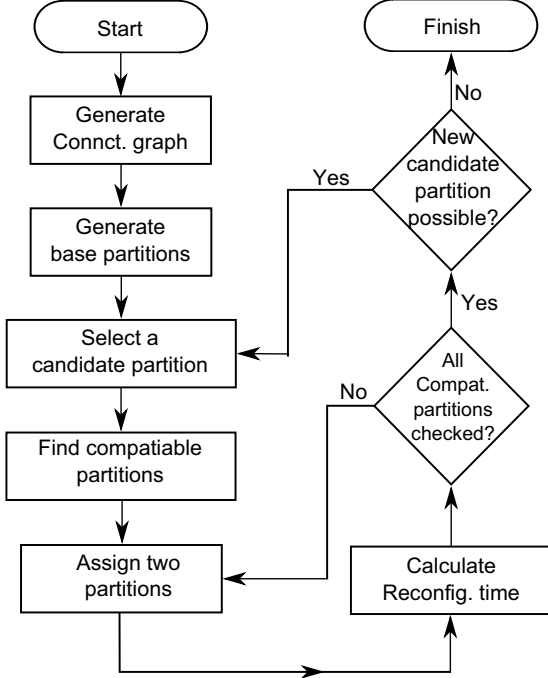
Fig. 6. Flow chart for the proposed algorithm.

configuration transitions take longer than this stipulated time. Worst case reconfiguration time is a useful measure in this situation. It is the largest configuration transition time among all the possible configuration transitions.

Mathematically, the total reconfiguration time is given by

$$t_{total} = \sum_{i=1}^{c-1} \sum_{j=i+1}^{c} tcon_{i,j} \qquad j > i \qquad (7)$$

where, c is the total number of configurations, and $tcon_{i,j}$ is the time required to change the system configuration from i to j, and is calculated as

$$tcon_{i,j} = \sum_{r=1}^{N} d_{i,j} \times tcon_r \qquad (8)$$

Where $d_{i,j}$ is a decision variable which is equal to 1 if region r contains different *base partitions* in configuration i and configuration j. $tcon_r$ is the time to configure region r and N is the total number of regions.
The configuration time for a region is proportional to the area of the region.

$$tcon_r \propto P_r \qquad (9)$$

Hence total reconfiguration time in terms of frames is:

$$t_{total} = \sum_{i=1}^{c-1} \sum_{j=i+1}^{c} \sum_{r=1}^{N} d_{i,j} \times \sum_{t} W_t * R_{rt} \qquad (10)$$

t is the tile type, $t \; \epsilon$ (CLBs, DSP blocks, BlockRAMs),

The worst case reconfiguration time is calculated as

$$t_{worst} = \max(tcon_{i,j}) \qquad (11)$$

If the total reconfiguration time for the partition scheme is less than the present lowest time, the scheme is stored as the present best partition scheme. Once the total number of frames is calculated, *base partitions* assigned to the region are removed from the list and the new region is added to the list as a new *base partition* and compatible partitions are recalculated.

The algorithm iterates by assigning two new compatible partitions to a region. If all possible compatible *base partition* assignments are done, the algorithm restarts from the initial *candidate partition set*, and assigns two compatible *base partitions* to the same region, which are distinct from those used to begin the previous iterations. Once all combinations of compatible base partitions are considered for initial assignment, a new set of *base partitions* are selected from the list to generate a new *candidate partition set*. For this purpose, the top most *base partition* is removed from the list, and the covering algorithm is re-applied. Due to the arrangement of the *base partitions*, the one with the lowest frequency weight is removed from the list. For the example design, after the first set of iterations, $\{A_2\}$ is removed from the list and $\{A_2, B_2\}$ is added to the new *candidate partition set*.

The algorithm iterates until no more *candidate partition sets* are possible. When the algorithm terminates, it selects the scheme with the lowest reconfiguration time as the final partitioning. Considering the valid configuration information in the partitioning step makes it a tractable problem, whereas if all possible combinations of modes were considered, the problem would become NP-hard and we would only be able to find sub-optimal solutions. One key difference in our new approach is that we focus on making use of all available resources in the target FPGA. Rather than only minimising resource usage, likely at a cost of increasing reconfiguration time, this approach will optimise reconfiguration time, using all the resources available on the FPGA specified, and hence, may implement multiple modes at the same time.

D. Special Conditions

One scenario we have worked to include in this formulation is where the system does not consist of a number of distinct design modules that have different modes. For example consider the design example used in [7]. The system has only two configurations.

1) CAN controller (C) → FIR filter(F)
2) Ethernet controller (E) → Floating point unit (P) → CRC (R)

Here, there are no definite mode relations. In our algorithm this is dealt with by specifying each reconfigurable module with a single mode. While specifying the configurations, the modules which are not present in a configuration are marked as having *mode* 0. For this example, the configurations are specified in our algorithm as

1) $C_1 \rightarrow F_1 \rightarrow E_0 \rightarrow P_0 \rightarrow R_0$
2) $E_1 \rightarrow P_1 \rightarrow R_1 \rightarrow C_0 \rightarrow F_0$

The algorithm treats *mode* 0 as the absence of the corresponding module, and no column is allocated for zero modes in the connectivity matrix. This allows us to mix multi-mode modules and one-off modules.

V. CASE STUDY

The proposed algorithm was implemented using the Python programming language [14].

For a realistic evaluation, we apply our region allocation approach to an example design implemented on a Virtex-5 FX70T FPGA. The design is a wireless video receiver chain with blocks used from existing designs and vendor IP. It has one static region and five reconfigurable modules. The system can operate in various modes, and adapts to channel conditions and user requirements at runtime. Modules communicate with each other using a simple streaming bus interface, which is registered to ensure timing is not affected by partitioning. The resource utilisation for each reconfigurable module and mode is shown in Table II.

TABLE II
RESOURCE UTILISATION FOR RECONFIGURABLE MODULES.

Module	Mode	Slices	BR	DSP
Matched Filt (F)	1. Filter1	818	0	28
	2. Filter2	500	0	34
Recovery (R)	1. Fine	318	1	13
	2. Coarse1	195	1	5
	3. Coarse2	123	0	8
	4. None	0	0	0
Demodulator (M)	1. BPSK	50	0	2
	2. QPSK	97	0	4
Decoder (D)	1. Viterbi	630	2	0
	2. Turbo	748	15	4
	3. DPC	234	2	0
Decoder (V)	1. MPEG4	4700	40	65
	2. MPEG2	4558	16	32
	3. JPEG	2780	6	9

The different configurations used by the system are the following:

$S \rightarrow F_1 \rightarrow R_3 \rightarrow M_1 \rightarrow D_1 \rightarrow V_1$
$S \rightarrow F_1 \rightarrow R_3 \rightarrow M_1 \rightarrow D_1 \rightarrow V_2$
$S \rightarrow F_1 \rightarrow R_3 \rightarrow M_1 \rightarrow D_1 \rightarrow V_3$
$S \rightarrow F_2 \rightarrow R_1 \rightarrow M_2 \rightarrow D_3 \rightarrow V_1$
$S \rightarrow F_2 \rightarrow R_2 \rightarrow M_1 \rightarrow D_1 \rightarrow V_1$
$S \rightarrow F_2 \rightarrow R_2 \rightarrow M_1 \rightarrow D_1 \rightarrow V_2$
$S \rightarrow F_2 \rightarrow R_2 \rightarrow M_1 \rightarrow D_1 \rightarrow V_3$
$S \rightarrow F_1 \rightarrow R_2 \rightarrow M_1 \rightarrow D_2 \rightarrow V_2$

The total FPGA resources available for the implementation of the PR design has been set to 6800 CLBs, 50 BRAMs, and 150 DSP slices in this example, saving the rest of the FPGA for the static region. Implementing the design completely statically, using multiplexers to select between modes, requires 15053 CLBs, 68 BRAMs, and 202 DSP slices, which exceeds the capacity of the target device. Implementing each module in a separate region requires 6580 CLBs, 48 BRAMs and 144 DSP slices. Although this scheme can fit in the FPGA, the total reconfiguration time is 244872 frames. The proposed

algorithm finds a solution that requires 6600 CLBs 48 BRAMs and 140 DSP slices, with a total reconfiguration time of 235266 frames, 4% less than the one module per region implementation. The low percentage improvement is due to the large size of the decoder module modes compared to other modules and also since in the final solution all decoder modes are assigned to the same region. The final scheme determined by the algorithm is as shown in Table III. The resource requirements for each scheme are shown in Table IV.

TABLE III
PARTITIONS DETERMINED BY ALGORITHM.

Region	Base Partitions
PRR_1	$M_2, \{M_1, D_2\}$
PRR_2	D_3, R_2, R_3
PRR_3	D_1, R_1
PRR_4	F_1, F_2
PRR_5	V_1, V_2, V_3

TABLE IV
PROPERTIES FOR DIFFERENT PARTITIONING SCHEMES.

Scheme	CLBs	BRAMs	DSPs	Total Recon. time
Static	15053	68	202	0
Modular	6580	48	144	244872
Proposed	6600	48	140	235266

Now consider the system configurations are changed to:
$S \rightarrow F_1 \rightarrow R_3 \rightarrow M_1 \rightarrow D_1 \rightarrow V_1$
$S \rightarrow F_1 \rightarrow R_2 \rightarrow M_1 \rightarrow D_1 \rightarrow V_3$
$S \rightarrow F_2 \rightarrow R_3 \rightarrow M_1 \rightarrow D_1 \rightarrow V_3$
$S \rightarrow F_1 \rightarrow R_1 \rightarrow M_2 \rightarrow D_3 \rightarrow V_1$
$S \rightarrow F_2 \rightarrow R_1 \rightarrow M_2 \rightarrow D_3 \rightarrow V_2$

The solution found by the proposed algorithm is given in Table V. This scheme requires 6500 CLBs, 48 BRAMs, and 144 DSP slices, with a total reconfiguration time of 92120 frames. This is 6% less that the one module per region scheme. From the explored schemes, the scheme with the smallest reconfiguration time that can fit in the FPGA is selected as the final solution. These results show that for optimal performance, partitioning needs to be a function of the system configurations and resource availability.

TABLE V
PARTITIONS DETERMINED BY ALGORITHM FOR MODIFIED CONFIGURATIONS.

Region	Base Partitions
static	$M1, D2$
PRR_1	$D1, R1$
PRR_2	$R2, R3, M2, D3$
PRR_3	F_1, F_2
PRR_4	V_1, V_2, V_3

For a more thorough investigation of the proposed algorithm, we require more PR designs. Unfortunately, there are very few such designs in the literature, and many of those available are very simple. Hence, we use synthetic designs for a more thorough evaluation. We generated 1000 synthetic
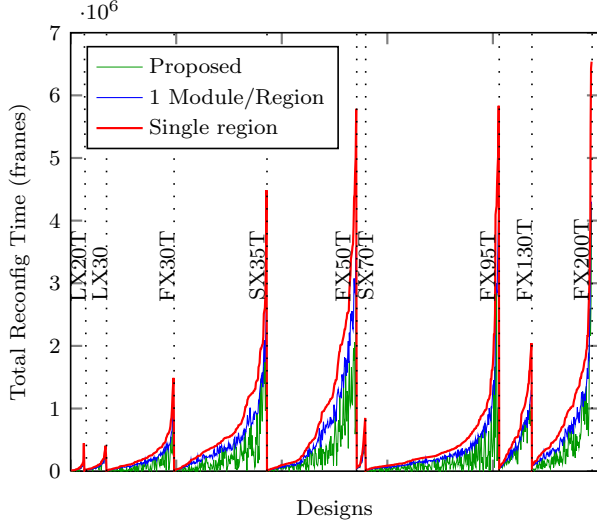
Fig. 7. Total reconfiguration time for proposed method vs one module per region implementation and single PR region method sorted according to target FPGA size.
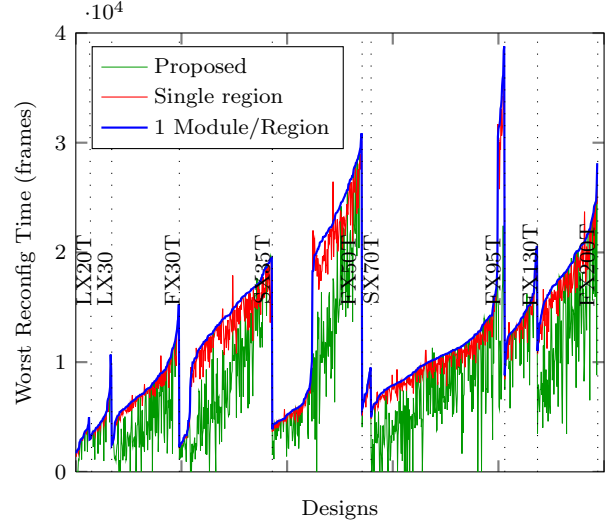

Fig. 8. Worst reconfiguration time for proposed method vs one module per region implementation and single PR region method sorted sorted according to target FPGA size.

designs, with an equal number of logic-intensive, memory-intensive, DSP-intensive and DSP-and-memory-intensive circuits. Each design is also augmented with a static region requiring 90 CLBs and 8 BRAMs, based on our custom ICAP controller and associated logic [15]. Designs are generated containing 2-6 modules, each with a number of modes varying from 2 to 4.Each mode can use 25 to 4000 CLBs, and the number of other resources is chosen from a range determined by the number of CLBs and the type of the circuit (logic-intensive, memory-intensive etc.). Configurations are randomly generated, until every mode present in the design is utilised at least once. This results in a wide range of design types, that we expect to give us a better idea of how well the proposed algorithm performs.

For each design, the minimum resources required for implementation are determined by considering a design using a single PR region. This is used to determine the smallest FPGA that can accommodate the design. The FPGAs used are from the Xilinx Virtex-5 family [16]. If at the end of an iteration of the algorithm, no partitioning scheme other than a single region is feasible, we select the next largest FPGA and the design is partitioned again. The program takes between a few seconds and one minute to determine the best solution for a design depending upon its size and the number of configurations.

201 of the 1000 designs could not be alternatively arranged on the smallest FPGA, so they were re-iterated using larger FPGAs. In 13 cases, the proposed algorithm was able to fit the design in a smaller FPGA than is required for the one module per region scheme.

A comparison of total reconfiguration time for the one-module-per-region scheme, a single-region scheme, and the scheme determined by the proposed algorithm is shown in Fig. 7. The results have been sorted based on the target FPGA.

The total reconfiguration time for the single-region scheme is high since for each reconfiguration, the complete PR region needs to be reconfigured. In most of the cases, the proposed algorithm finds a better solution than the one-module-per-region scheme.

A comparison for worst-case reconfiguration time is shown in Fig. 8. In almost all cases, the proposed algorithm has a lower worst-case reconfiguration time compared to the one-module-per-region scheme. The plot shows that in several scenarios, the worst-case reconfiguration time for a single-region scheme is lower than the one-module-per-region scheme and the solution of the proposed algorithm. This occurs because the single-region implementation scheme has the minimum resource requirement when all modes are implemented in PR regions (i.e. no modes are moved to the static region). For this scheme, the worst-case reconfiguration time is independent of configuration transitions, since each transition requires the entire region to be reconfigured and hence it is the same for all transitions. Meanwhile the worst-case for the other schemes will typically be where all modules switch mode, and hence, the increased sum area of PR regions causes this to be longer. But the impact of this scheme on the overall system performance is evident from Fig. 7, since for all configuration transitions the whole region needs to be reconfigured.

Profiles of the percentage improvement of the proposed algorithm compared to the one-module-per-region and single-region schemes are shown in Fig. 9. The proposed scheme performs better than the one-module-per-region scheme in terms of total reconfiguration time in 73% of cases and performs better than the single-region scheme in all cases.

In terms of worst-case reconfiguration time, the proposed algorithm finds a better solution than the one-module-per-region scheme in 70% of cases. For 3 designs, the output of the algorithm performs worse. Compared to the single-region

scheme, the proposed method improves or matches worst-case reconfiguration time in 87.5% of cases. In the remaining 12.5% of cases, the *total* reconfiguration time, which is what we optimise for, is very high and hence, this is not relevant.
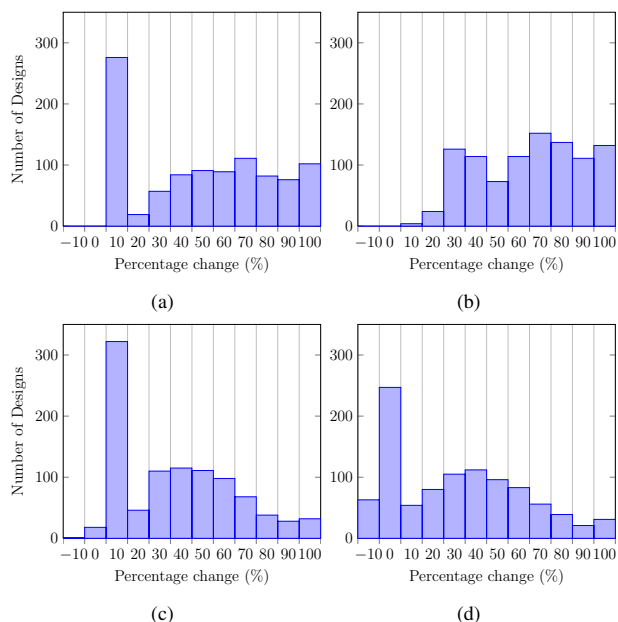

Fig. 9. Percentage changes for total reconfiguration time found by the proposed algorithm compared to (a) one module per region and (b) single region schemes and for worst reconfiguration time compared to (c) one module per region and (d) single region schemes.

We can see that the proposed algorithm offers tangible improvements over both traditional partitioning approaches, especially in cases where it determines that modules can be moved to the static region. At the same time, it is clear that using general measures of total reconfiguration time and worst-case reconfiguration time may not tell the whole story. A more detailed analysis would require knowledge of the specific transition probabilities; something that we plan to consider in future work.

VI. Conclusion and Future Work

In this paper we have proposed a new technique for automating the partitioning of designs for partial reconfiguration. A hierarchical graph clustering approach is used to determine an optimal allocation based on the set of valid configurations, optimising for reconfiguration time. The only input the designer needs to provide is the design description and the target device. The approach is the first to make use of all available resources in the given device to bring reconfiguration time down to the minimum, by moving modes into the static region when possible, in addition to intelligent partitioning. It can also accommodate designs that contain optional single-mode modules. Through our ongoing research, we aim to raise the level of abstraction and automation in design for partial reconfiguration, so that adaptive systems design using PR can become easier.

We plan to integrate our floorplanning algorithm [11] in future work. Even though we find a partitioning scheme can theoretically fit into an FPGA based on resource availability, at the time of floorplanning we may find that other issues such as device layout, the presence of hard-macros, and PRR shape constraints prevent this from being feasible. Feedback from the floorplanner will be added to the partitioning tool to overcome this when a selected scheme cannot be floorplanned.

We also hope to formalise an approach for describing an adaptive system at a high level of abstraction, including both the composition and adaptation behaviour, extending the approach in [17].

Combining a formalised adaptive system description model with automated partitioning and floorplanning will enable adaptive system designers to leverage PR without the need for expert knowledge.

References

[1] J. Lotze, S. A. Fahmy, J. Noguera, B. Ozgül, and L. Doyle, "Spectrum sensing on LTE femtocells for GSM spectrum re-farming using Xilinx FPGAs," in *Proceedings of the Software-Defined Radio Forum Technical Conference (SDR Forum)*, 2009.

[2] C. Conger, R. Hymel, M. Rewak, A. D. George, and H. Lam, "FPGA design framework for dynamic partial reconfiguration," in *Proceedings of Reconfigurable Architectures Workshop (RAW)*, 2008.

[3] S. Ganesan and R. Vemuri, "An integrated temporal partioning and partial reconfiguration technique for design latency improvement," in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2000.

[4] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of ACM/SIGDA tenth international Symposium on Field Programmable Gate Arrays (FPGA)*, 2002.

[5] V. Rana, S. Murali, D. Atienza, M. D. Santambrogio, L. Benini, and D. Sciuto, "Minimization of the reconfiguration latency for the mapping of applications on fpga-based systems," in *Proceedings of IEEE/ACM International Conference on Hardware/Software Codesign and Systtem Synthesis (CODES+ISSS)*, 2009.

[6] A. Jara-Berrocal and A. Gordon-Ross, "Runtime temporal partitioning assembly to reduce FPGA reconfiguration time," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2009.

[7] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and floorplanning in dynamically reconfigurable FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, pp. 24:1–24:34, November 2010.

[8] K. Vipin and S. A. Fahmy, "Efficient region allocation for adaptive partial reconfiguration," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2011.

[9] *WP-01137: Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs*, Altera, 2010.

[10] M. Bourgeault. (2011, June) Altera's partial reconfiguration flow. Altera.

[11] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Reconfigurable Computing: Architectures, Tools and Applications – Proc. Int. Symposium on Applied Reconfigurable Computing (ARC)*, 2012, pp. 13–25.

[12] *UG191: Virtex-5 FPGA Configuration User Guide*, Xilinx Inc., 2010.

[13] M. Srinivas and C. K. Mohan, "Efficient clustering approach using incremental and hierarchical clustering methods," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, 2010.

[14] Python programming language - official website. [Online]. Available: http://www.python.org/

[15] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA partial reconfiguration," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2012.

[16] *DS100: Virtex-5 Family Overview*, Xilinx Inc., Feb. 2009.

[17] J. Lotze, S. Fahmy, J. Noguera, and L. Doyle, "A model-based approach to cognitive radio design," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 29, no. 2, pp. 455–468, Feb. 2011.