TIMING AWARE PARTITIONING FOR MULTI-FPGA BASED LOGIC SIMULATION USING TOP-DOWN SELECTIVE FLATTENING

A Thesis

by

SUBRAMANIAN POOTHAMKURISSI SWAMINATHAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Computer Engineering

TIMING AWARE PARTITIONING FOR MULTI-FPGA BASED LOGIC SIMULATION USING TOP-DOWN SELECTIVE FLATTENING

A Thesis

by

SUBRAMANIAN POOTHAMKURISSI SWAMINATHAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Sunil P. Khatri
Committee Members,	Peng Li
	Andrew (Anxiao) Jiang
Head of Department,	Costas N. Georghiades

August 2012

Major Subject: Computer Engineering

ABSTRACT

Timing Aware Partitioning for Multi-FPGA Based Logic Simulation Using Top-down Selective Flattening. (August 2012)

Subramanian Poothamkurissi Swaminathan, B.Tech., National Institute of Technology, Trichy, India

Chair of Advisory Committee: Dr. Sunil P. Khatri

In order to accelerate logic simulation, it is highly beneficial to simulate the circuit design on FPGA hardware. However, limited hardware resources on FPGAs prevent large designs from being implemented on a single FPGA. Hence there is a need to partition the design and simulate it on a multi-FPGA platform. In contrast to existing FPGA-based post-synthesis partitioning approaches which first completely flatten the circuit and then possibly perform bottom-up clustering, we perform a selective top-down flattening and thereby avoid the potential netlist blowup. This also allows us to preserve the design hierarchy to guide the partitioning and to make subsequent debugging easier. Our approach analyzes the hierarchical design and selectively flattens instances using two metrics based on slack. The resulting partially flattened netlist is converted to a hypergraph, partitioned using a public domain partitioner (hMetis), and reconverted back to a plurality of FPGA netlists, one for each FPGA of the FPGA-based accelerated logic simulation platform. We compare our approach with a partitioning approach that operates on a completely flattened netlist. Static timing analysis was performed for both approaches, and over 15 examples from the OpenCores project, our approach yields a 52% logic simulation speedup and about 0.74× runtime for the entire flow, compared to the completely flat approach. The entire tool chain of our approach is automated in an end-to-end flow from hierarchy extraction, selective flattening, partitioning, and netlist reconstruction. Compared to an existing method which also performs slack-based partitioning of a hierarchical netlist, we obtain a 35% simulation speedup.

To my family and friends

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. Sunil P. Khatri for his guidance, encouragement and support during the course of my masters program. I would also like to thank my research group members for their valuable suggestions in various aspects of my research.

I would like to thank my family for their moral support when I needed it. A final note of thanks to Texas A&M University and Department of Electrical and Computer Engineering for giving me the opportunity to pursue my master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Logic verification takes up the majority of computation and engineering resources in VLSI design. Logic simulation is used to determine the functional correctness of the implemented circuit with respect to a high level specification. The functionality of the circuit can be verified by logically simulating a series of test vectors, and testing that the outputs are correct by comparing them with a golden output. Since logic verification takes up a lot of time, it is crucial to accelerate this step. Logic simulation techniques can be broadly classified as software based methods and hardware based methods.

Event-driven simulation, levelized code simulation and compiled code simulation are examples of software based techniques. In the event-driven method, simulation events are stored in a queue of temporally sorted pending events. New events are scheduled as existing events are processed. This is continued until no events are left in the event queue. Levelized code simulators simulate gates strictly in level order from primary inputs to primary outputs. However, circuits that contain loops (sequential circuits) cannot be levelized. In the compiled code method, the circuit description is compiled into a series of machine language instructions and then simulated. In all these methods, *n* vectors can be simulated in parallel, where *n* is the instruction width of the computer being used.

Hardware emulators are hardware assisted techniques wherein the behavior of the circuit is emulated in hardware. Software techniques serialize the simulation of gates in the circuit which the hardware emulators avoid. Hence hardware emulators are several orders of magnitude faster than software solutions. Reconfigurable hardware platforms like FP-

The journal model is *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

GAs (Field Programmable Gate Arrays) are used for hardware acceleration of logic simulation. FPGAs provide the flexibility to perform quick prototyping and fast logic simulation during engineering development, due to the parallelism that is inherent in the FPGA.

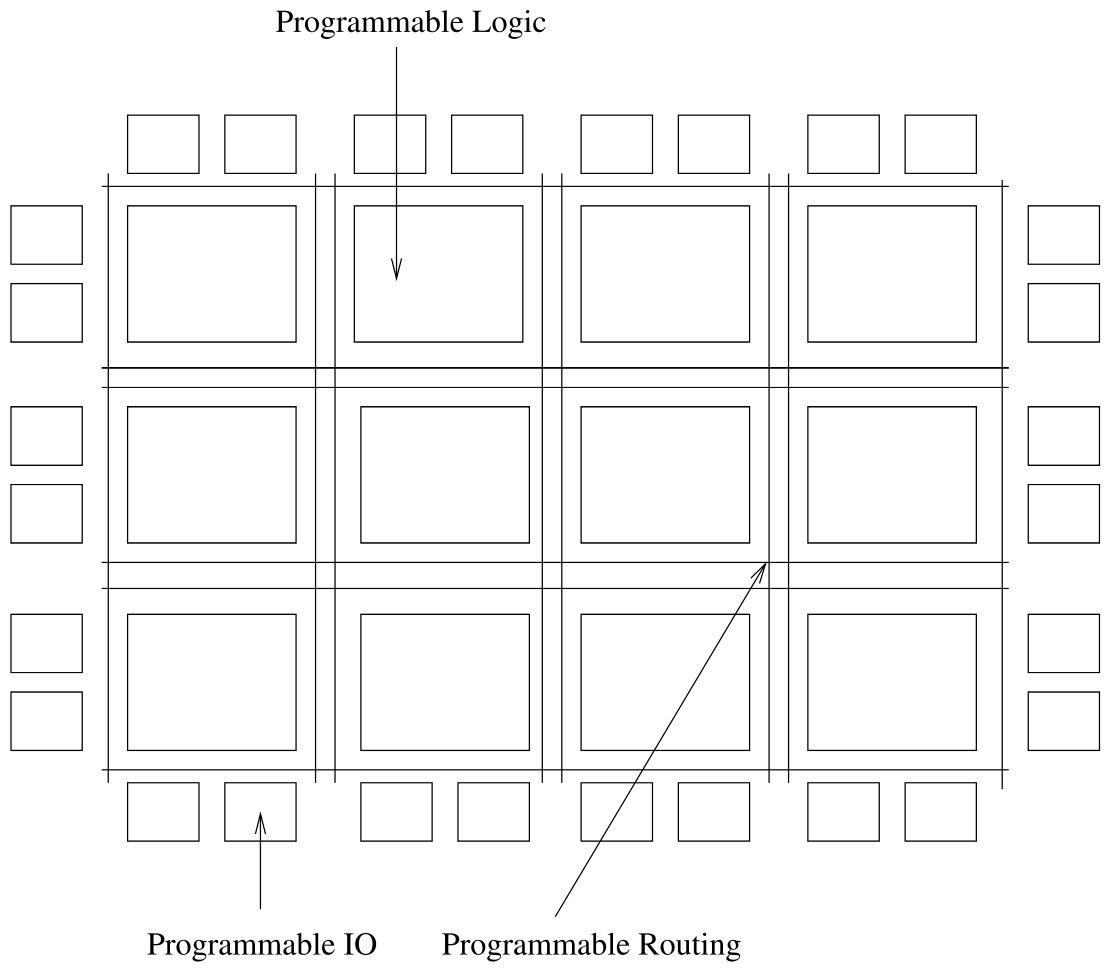

Fig. I.1. A Generic FPGA Architecture

FPGAs implement circuits through programmable logic, programmable interconnect and programmable IO. Figure I.1 depicts a typical FPGA. Each programmable logic block (Configurable Logic Block (CLB) in the case of Xilinx or Logic Array Block (LAB) in the case of Altera) are made up of one or more logical cells (Binary Logic Elements/BLEs in the case of Xilinx or Arithmetic and Logic Modules/ALMs in the case of Altera). Each logic cell consists of a Look Up Table (LUT) and a flip flop. Each $n-bit$ LUT can implement any Boolean function of up to n inputs. An $n-bit$ LUT is implemented using $2^n \times 1$ multiplexor, where the n select lines of the multiplexor are the n inputs of the LUT and the 2^n inputs of the multiplexor are connected to the constants obtained via recursive Shannon expansion. Typically, a value of n between 4 and 6 is chosen. A 2-input LUT implemented using a 4×1 multiplexor, is shown in Figure I.2, where a and b are inputs of the LUT and $S0-S3$ are constants stored in SRAM cell S. Programmable routing is implemented using programmable transistor switches at each intersection of vertical and horizontal wires. A transistor based programmable switch is shown in Figure I.3, which is turned on based on the value stored in the SRAM cells that drives it's gate. By selectively turning on programmable switches, any two intersecting wires can be connected.

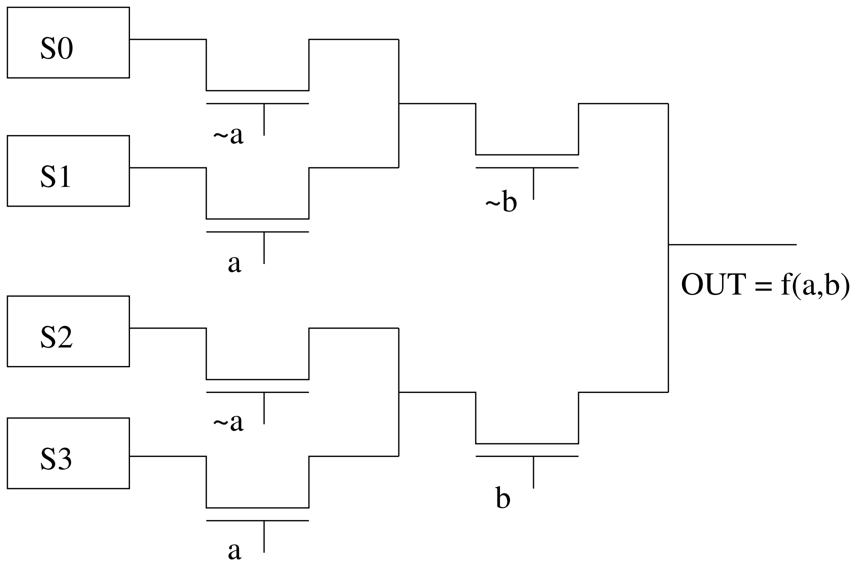

Fig. I.2. A 2-input LUT

FPGA based logic simulation for large circuits is hampered by the limited hardware

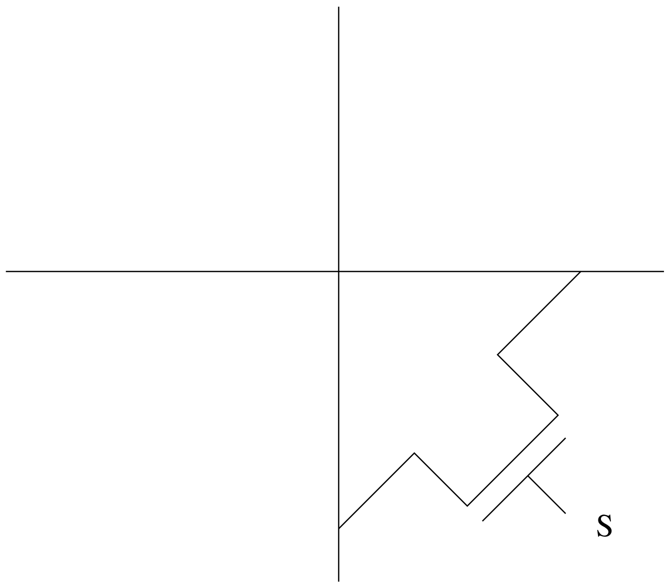

Fig. I.3. A Programmable Switch

resources on the FPGA. As a result, there is a move to perform logic simulation using a multi-FPGA platform. To simulate a large design on such a multi-FPGA platform, the design must be efficiently partitioned, with each partition residing on one of the FPGAs in the platform. Figure I.4 shows the typical flow for logic simulation using a multi-FPGA platform.

In our approach, we *strictly preserve the hierarchy of the design* by performing a *top-down selective hierarchy flattening*. This guarantees that we do not incur the blowup which is possible due to full netlist flattening. Also, tightly related logic in the unflattened instances of the design is not partitioned, yielding good results, while not resulting in a large graph for the partitioner to handle. Since we retain the original design hierarchy and signal names, subsequent debugging and analysis becomes easier.

This thesis presents a selective hierarchy flattening algorithm that exploits the design hierarchy. In addition, selective flattening can be guided by user-provided constraints (al-

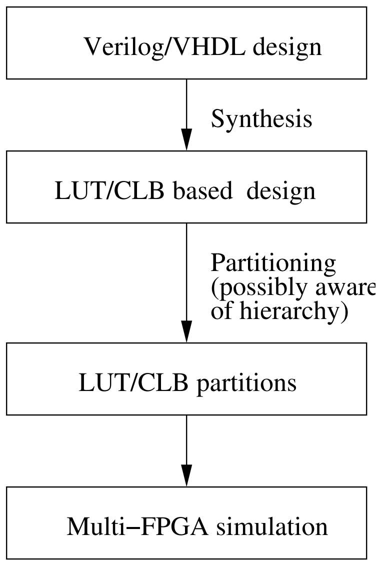

Fig. I.4. Existing State-of-the-art Post-Synthesis Partitioning Flow for Logic Simulation on a Multi-FPGA Platform

though in our experimental results, no user input is assumed). Our approach is targeted to a commercial multi-FPGA logic simulator platform offering. This platform uses 2 (or 4 in another variant) Altera Stratix III FPGAs on a single board. With such a choice of FPGA platform, it was determined that a majority of today's ASIC designs can be targeted. Our primary goal is to prioritize the logic simulation speed while retaining design hierarchy.

The key contributions of this thesis are:

- We present a partitioning approach for multi-FPGA based logic simulation using selective flattening to preserve design hierarchy. Our goal is to maximize the logic simulation speed of the resulting partitioned design.
- Since we perform a top-down selective flattening, we never incur the potential netlist blowup resulting from full flattening.
- Compared to designs partitioned using a flat algorithm, the resulting design simulates

1.52× faster using our approach. Compared to a competing approach [1], our logic speedup is 35%.

- We provide a user selectable flattening threshold per design. Also, our approach can allow for user to manually specify which instances to flatten or not to flatten.
- Our approach can be easily generalized to *k*-way partitioning (where there are *k* FPGAs in the hardware platform).
- Our approach is implemented using an end-to-end scripted tool flow to perform hierarchy analysis, selective flattening, partitioning, and netlist reconstruction.

The remainder of this thesis is organized as follows. Chapter II discusses previous work in partitioning for FPGA-based fast logic simulation. In Chapter III we describe our approach for selective hierarchy flattening and partitioning. In Chapter IV, we apply our tool flow on several design examples and present experimental results. Conclusions are drawn in Chapter V.

CHAPTER II

PREVIOUS WORK

In the past, many partitioning approaches have been proposed for Multi-FPGA based logic simulation. Table II.1 summarizes the previous work in this field, broken down by approach, objective and decision cost.

Most of the proposed algorithms focus on total dollar cost minimization via higher logic utilization [2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]. A natural objective of many partitioning algorithms is to reduce the cut-size [2, 3, 4, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42]. In contrast, our work focuses on maximizing simulation speed of the final partitioned design when it is embedded into a multi-FPGA simulation accelerator.

There exist a few performance-driven algorithms [5, 6, 7, 8, 9, 10, 1, 29, 30, 43, 44, 45, 46, 47] targeting delay minimization. High-level estimators [5, 6], Maximum matching with node ordering [7], cone partitioning [8, 43, 47], clustering of CLBs [9, 10, 1, 45], rectilinear partitioning with signal path delay estimation [44], replication [29, 46] and re-timing [30] are some of the methods used for performance optimization in the literature.

Based on the approach used, partitioning algorithms can be broadly classified as iterative improvement algorithms (also referred to as top down or refinement algorithms) [3, 4, 5, 6, 9, 10, 29, 30, 32, 33, 34, 35, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 48, 11, 14, 15, 16, 18, 21, 25, 28] and clustering-based (also known as bottom-up or constructive) algorithms [2, 7, 8, 9, 1, 30, 31, 32, 36, 37, 41, 42, 43, 45, 47, 48, 11, 12, 13, 16, 17, 19, 20, 21, 23, 24, 25, 26, 28]. Iterative algorithm improve an initial partition by moving elements between partitions based on an objective function. Clustering-based algorithms collapse elements together controlled by an appropriate objective function. Clustering algorithms can be combined with iterative methods to improve partitioning and runtime.

Table II.1. Literature Summary

Paper	Approach					Objective			Decision cost	
	Design Hierarchy Guided	Initial Netlist not Flat	Top Down / Refinement / Iterative Improvement	Bottom-Up / Clustering / Constructive	Pre-Synthesis Partitioning	Logic Utilization (Dollar Cost)	Delay (Sim. Speed)	Cut Size	Slack / Timing Aware	Size and IO
Our method	√	√	√	√			√		√	
[2]				√		√		√		√
[3, 4]			√			√		√		√
[5, 6]	√	√	√		√	√	√		√	√
[7]				√		√	√		√	√
[8]				√		√	√			√
[9]			√	√		√	√		√	√
[10]			√			√	√		√	√
[1]	√	√		√		√	√		√	√
[29]			√				√	√		√
[30]			√	√			√	√	√	√
[31]				√				√	√	
[32]	√		√	√				√		√
[33]			√					√		√
[34]			√					√		√
[35]			√					√		√
[36]				√				√		√
[37]			√	√				√		√
[38, 39, 40]			√					√		√
[41]			√	√				√	√	√
[42]			√	√				√		√
[43]				√			√		√	√
[44]			√				√		√	
[45]			√	√			√		√	
[46]			√				√			√
[47]			√	√			√		√	√
[48]			√	√						√
[11, 21]	√	√	√	√		√				√
[12, 13, 22, 17, 24, 26, 19]				√		√				√
[14, 15, 27, 18]			√			√				√
[16, 25, 28]			√	√		√				√
[20, 23]	√	√		√		√				√

Most of the above algorithms work on flat netlists, therefore incurring a blowup in the netlist size, along with large runtimes due to initial flattening step. Just like our approach (which avoids initial flattening), there exists some papers that perform partitioning while considering the hierarchy of the circuit [5, 6, 1, 32, 11, 20, 21, 23]. Design hierarchy guided clustering is used in [32], where the cut-size reduction is the main objective. A hierarchical netlist is partitioned with the goal of optimizing logic utilization in [11, 20, 21, 23]. In contrast, our work seeks to maximize simulation speedup. Pre-synthesis or behavioral partitioning is performed in [5, 6], guided by estimates of size, I/O and performance, made by high-level estimators. Since these estimations are to be performed for a large number of partition options, quick yet accurate estimates need to be obtained, which can be hard. However, with post-synthesis partitioning (our approach), size and delay can be easily and accurately estimated, since technology mapped netlists are used.

Based on our extensive literature survey, the only effort that operates on a post-synthesis hierarchical netlist while targeting performance optimization is [1]. In [1], an integrated synthesis and partitioning method is described. The hierarchy considered in [1] consists of a single level of module instances, the processes contained in the instances and the functions contained in these processes (which form the leaves). First, an HDL-netlist is synthesized in a fine-grained manner, forming CLB-based clusters according to the structure of the design. Then a hierarchical set covering algorithm with functional replication is used to perform partitioning. The cost function used in the set covering algorithm is a combination of slack, size/IO and connectivity metrics. The time complexity of this approach is $O(n^2 * m)$, where n is the number of functional nodes and m is the number of FPGAs used. In contrast, our method has an $O(n)$ complexity since it uses hMetis [49] which is a partitioner based on multi-level FM [34]. Partitioning is performed at the granularity of bit-level decomposed functions or modules, in contrast to our approach where the granularity uses any combination of LUTs or module instances. Also, the netlist hierarchy is only one level deep, in contrast to our work where it is arbitrary. In [1], replication results in an increase in the number of CLBs in the partitioned design, unlike our work. We strictly preserve the design hierarchy, yielding a significantly improved debuggability of the design unlike [1] wherein replication and functional decomposition may modify the hierarchy. Additionally, [1] reports results on 6 relatively small designs, while we test our algorithm on 15 designs from the OpenCores [50] project. Finally, our results show that our technique achieves a 35% speedup over [1].

By keeping related critical logic in unflattened instances, our approach can provide an overall speedup of the logic simulation, while yielding a good result quality. We achieve a 26% faster runtime for the total tool flow including flattening, partitioning, and netlist reconstruction compared to a reference approach which is based on a complete flattening of the netlist.

CHAPTER III

OUR APPROACH

This chapter describes our selective hierarchy flattening method as well as the entire logic simulation flow targeted to multi-FPGA platforms. Our approach performs selective flattening of the hierarchical netlist before partitioning and contains the following steps: hierarchy analysis, selective flattening, netlist translation to hMetis hypergraph format, running hMetis, and hypergraph to netlist reconstruction. The overall toolflow is shown in Figure III.1 and each step is explained in detail in the following sections.

In the rest of this thesis, the term *cells* refers to base cells in the Altera FPGA netlist (such as Lcells, DFFs etc).

III-A. Hierarchy Analysis

The base design provided to our algorithm is a hierarchical netlist in the Verilog Quartus Mapping (.vqm) format of Altera. The .vqm netlist is recursively parsed in Perl from the top module down, to infer the hierarchy tree structure. The output of hierarchy analysis is an ASCII file (.hier) containing the instance hierarchy of the design.

III-B. Selective Flattening

In the .hier file generated by the previous step, each instance has an attribute which can be either ’F’ (flatten), ’D’ (don’t flatten) or ’X’ (to be decided). Initially, before the selective flattening algorithm has been invoked, all instances are marked ’X’, unless the user modifies this file and marks specific instances as ’D’ (in which case the algorithm will not flatten the specific instance) or ’F’ (in which case the algorithm will flatten the specific instance). At this point, we invoke the selective flattening algorithm on the .hier file and hierarchical .vqm

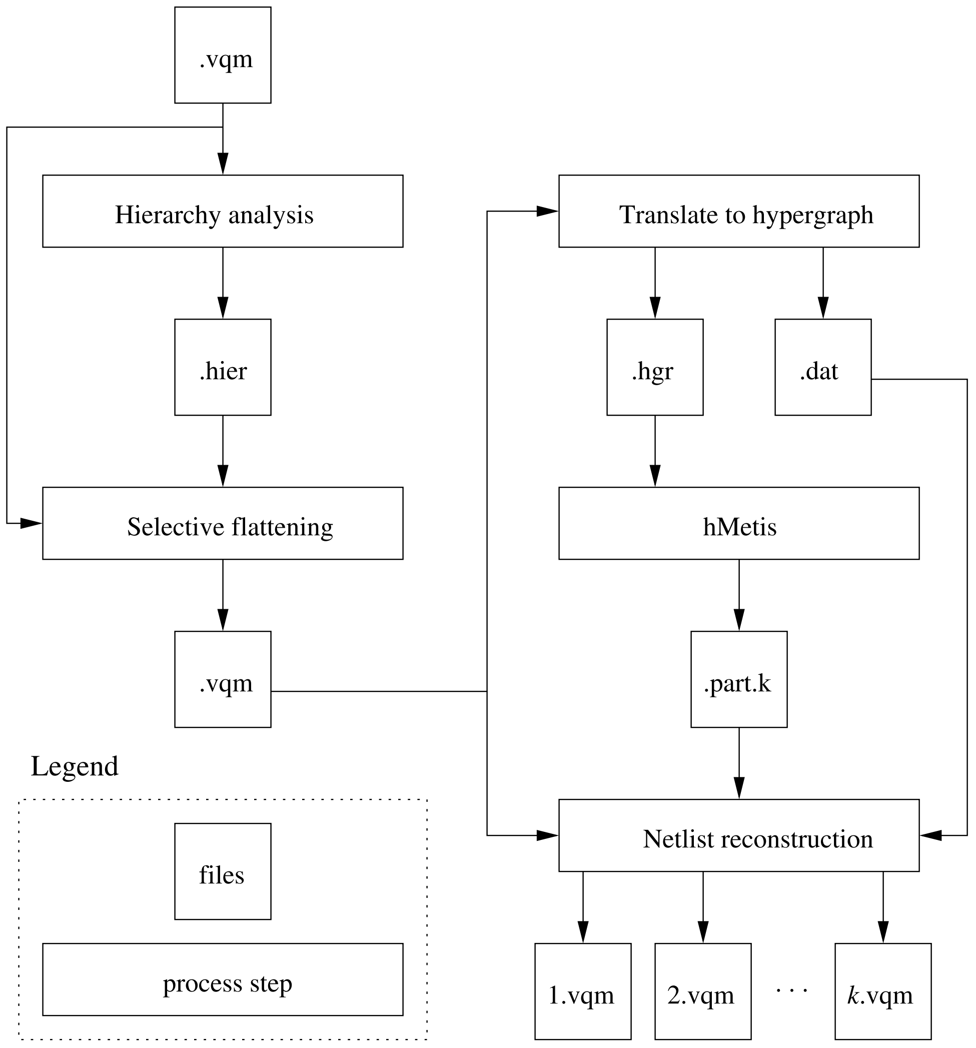

Fig. III.1. Our Selective Hierarchy Flattening Based multi-FPGA Logic Simulation Flow

files to determine which instances to flatten. The output of the selective flattening step is a .vqm file, in which nodes are selectively flattened as determined by the selective flattening algorithm. The selective flattening algorithm produces a modified .hier file, in which each 'X' instance is assigned an 'F' or 'D' label, based on whether selective flattening should be performed.

III-B.1. Selective Flattening Algorithm

Our selective flattening algorithm is guided by a combination of two metrics. The first metric is based on average slack (*AS*) of the instance and the second metric is based on minimum slack (*MS*) of the instance. Each of these metrics helps to determine whether a particular instance should be flattened or not. In all the metrics described below, top level nets such as reset, clock or supply pins are not included in the slack computation.

III-B.1.a. Average Slack Based Flattening

The *AS* metric for an instance X is defined as the average slack at each pin inside X. The *AS* value of X is calculated as the ratio of sum of slack values of all pins in X, to the number of pins in X. For example, in Figure III.2, the *AS* of instance R is 4.125 $((5+4+4+5+5+4+3+3)/8)$. For an instance X under consideration for flattening, if it's *AS* value is greater than a threshold τ_1, then instance X is flattened.

The motivation behind the effectiveness of the *AS* heuristic metric is as follows. If an instance X has a high value of *AS*, then it indicates that most cells and instances in X are not on critical paths. Similarly, a low value of *AS* indicates that a large number of cells and instances in X are on critical paths. Hence the *AS* metric is a measure of how timing-critical the cells and instances of X are.

To prevent the partitioner (hMetis in our case) from cutting critical nets, instances with low *AS* should not be flattened. Keeping these logic elements as a single node during

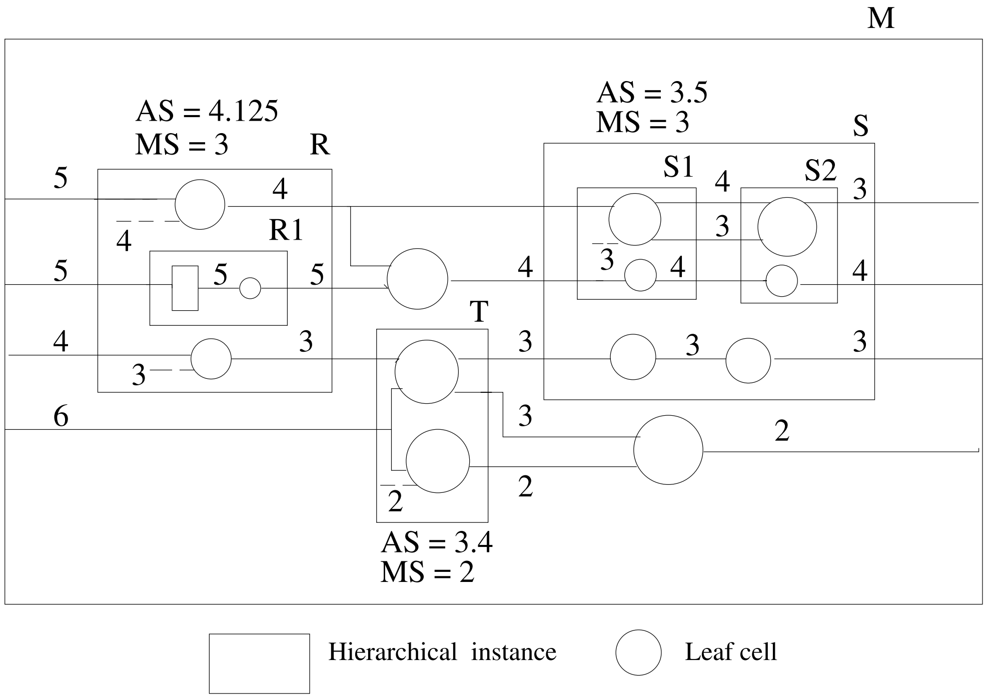

Fig. III.2. A Hierarchical Instance with Pin Slack Values Annotated

partitioning prevents hMetis from potentially performing bad cuts on critical paths, while reducing the size of the hypergraph. On the other hand, to improve node balancing and to reduce cut-size, non-critical nets can be safely exposed to hMetis by flattening instances with a high *AS*. A cut made inside an instance with a high *AS* will not significantly worsen the post-partition simulation speed of the design, since hMetis will typically cut nets with high slack.

Threshold τ_1 is calculated for each level in the design hierarchy tree according to Equation 3.1. The threshold is modified dynamically since the *AS* values vary as we descend into the hierarchy and make flattening decisions.

$$\tau_1 = p_1 * (max\{AS\} + min\{AS\})/2 \tag{3.1}$$

The set $\{AS\}$ consists of AS values of all candidate instances at the same level as the instance X in the hierarchy tree. From Figure III.2, the τ_1 value used as threshold for instance R is $p_1 * (4.125 + 3.4)/2$. A user-defined threshold multiplier p_1 is used in conjunction with the AS metric.

III-B.1.b. Minimum Slack Based Flattening

The MS metric of an instance X is defined as the minimum slack of all the inputs and outputs of the instance X. For example, in Figure III.2, the MS of instance R is 3. An instance X under consideration is flattened if it's MS value is greater than a threshold τ_2.

Instances with a high (low) value of MS indicate that the instance X is less timing-critical (more timing-critical). Instances with a low value of MS should not be flattened, otherwise hMetis might cut critical nets. On the other hand, instances with a high value of MS can be safely flattened, thereby exposing non-critical nets to hMetis.

Similar to τ_1, the threshold τ_2 is calculated dynamically for each level using Equation 3.2.

$$\tau_2 = p_2 * (max\{MS\} + min\{MS\})/2 \tag{3.2}$$

The set $\{MS\}$ consists of MS values of all instances at the same level as instance X in the hierarchy tree. From Figure III.2, the τ_2 value used as threshold for instance R is $p_2 * (3 + 2)/2$. Similar to the AS, a user-defined threshold multiplier p_2 is used in conjunction with the MS metric.

III-B.1.c. Hybrid Algorithm

The *hybrid* metric is a combination of the AS and MS metrics. An instance X under consideration is flattened, if it's AS is greater than a threshold τ_1. If not, X is flattened if it's MS is greater than a threshold τ_2.

In the *hybrid* approach we use both the average value metric (*AS*) and the minimum value metric (*MS*) of an instance in deciding whether to flatten an instance *X* or not. User-defined threshold multipliers p_1 and p_2 are used in conjunction with the *hybrid* metric.

The rationale behind using our *hybrid* method is as follows. In practice, we may have a set of instances $\{C_1\}$, with a large AS and possibly a small MS value. Such instances typically have a broad slack histogram, and are good candidates for selective flattening. Similarly there may be a set of instances $\{C_2\}$, with a small AS but a large MS value. Such instances typically have a narrow slack histogram, and are also good candidates for flattening. By itself, the AS metric targets instances in $\{C_1\}$, while the MS metric targets instances in $\{C_2\}$. No single method can target instances in both $\{C_1\}$ and $\{C_2\}$, hence the *hybrid* method is chosen.

Starting from the top module, our selective flattening algorithm recursively calculates the AS and MS metrics of all instances. If our algorithm determines that an instance should be flattened, then that instance is marked as 'F'. Otherwise, the instance is marked 'D' and all its children are marked 'D' as well. If the user had modified the initial .hier file described in Section III-A by marking specific instances as 'D', then all its children are also marked 'D'.

Algorithm 1 describes the procedure.

Algorithm 1 Pseudocode of Selective Flattening Algorithm

```
1: $selective\_flattening(x)$
2: if $(AS(x) > \tau_1)$ and $(x \neq D)$ then
3:     $x \leftarrow F$
4: else if $(MS(x) > \tau_2)$ and $(x \neq D)$ then
5:     $x \leftarrow F$
6: else
7:     $x \leftarrow D$
8:     $children(x) \leftarrow D$
9: end if
10: $selective\_flattening(children(x))$
```

III-B.2. Z-Frontier

The above algorithm yields a hierarchy where all instances are marked ’D’ or ’F’. Furthermore, any sub-tree rooted at an instance marked ’D’ will have all of its (recursive) children instances marked ’D’ as well. This allows us to define a *Z-Frontier*, where all the instances on or below this frontier are marked ’D’, and all instances above this frontier are marked ’F’. The selective flattening algorithm produces an initial Z-Frontier which we call $Z0$. Figure III.3 shows an example hierarchy with instances marked as ’F’ or ’D’, as well as the corresponding $Z0$ frontier (solid line). In addition to the $Z0$ frontier, we allow our algorithm to explore additional frontiers Zi by expanding the $Z0$ frontier by i levels towards the leaves of the hierarchy. Figure III.3 shows the $Z1$ frontier as well (dotted line).

In addition to the *AS*, *MS* and *hybrid* metrics, we also explored other metrics such as instance weight imbalance, internal connectivity of an instance etc. For dynamic threshold calculation, the median value of the slack metrics of the set of candidate instances was also considered in place of the average of maximum and minimum values. We also perturbed the $Z0$ frontier based on the cut nets information obtained, after analyzing the post-partitioned circuit. However, these methods resulted in selectively flattened hierarchy with worse simulation speed than the proposed method (described earlier in this section), and as such are not further discussed in this thesis.

After the $Z0$ frontier is computed by the selective flattening algorithm, we take the original .vqm netlist, and flatten all instances that are marked ’F’ in the .hier file (produced by the selective flattening algorithm) to obtain the $Z0$ frontier. The result is a modified, selectively flattened .vqm file which is utilized for partitioning. In case we are considering the Zi frontier, we appropriately flatten i more levels of instances (towards the leaves) from the $Z0$ frontier to generate the .vqm file for the Zi frontier.

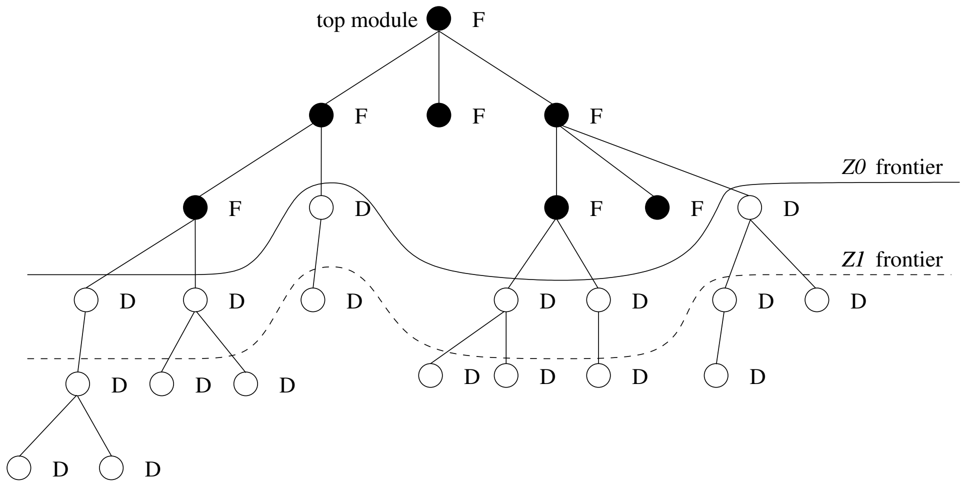

Fig. III.3. Hierarchy Tree with F/D Instances and $Z0$ and $Z1$ Frontier

III-C. Netlist Translation to hMetis Hypergraph

The output of the selective flattening step is a modified .vqm of the original design containing the selectively flattened netlist. From this modified .vqm, our method creates two files, an hMetis hypergraph file (.hgr), and a connectivity lookup file (.dat). The .dat file is used during the netlist reconstruction step after partitioning. In the hypergraph file, each node is either a cell or an instance on the $Z0$ frontier. Each hyperedge corresponds to a net that connects cells and/or instances. The hypergraph is constructed using the following node and edge weight definitions:

- **Node weight :** A cell (Lcell or DFF) has a weight of 1. An instance X on the Z frontier has a weight equal to the sum of cells in instance X and its children recursively to the leaves.

- **Edge weight :** Each edge is assigned a slack based weight $slack\_edge\_wt_i$ calculated according to Equation 3.3.

$$slack\_edge\_wt_i = \lceil max\_slack\_ckt - edge\_slack_i + 1 \rceil . \tag{3.3}$$

  max_slack_ckt is the maximum slack of the circuit. $edge\_slack_i$ is the slack of the $edge_i$. In the reference flat algorithm, a unit edge weight is assumed. For any hyperedge, it's $edge\_slack_i$ is defined as the minimum slack among all the edges that make up that hyperedge.

III-D. hMetis

The .hgr hypergraph is then partitioned using hMetis [49], a software package used for partitioning large hypergraphs, particularly those encountered in VLSI design. In k-way hypergraph partitioning, the nodes are assigned to k different partitions, such that the number of edges between partitions are minimized. The partitioning algorithm used in hMetis consists of a series of successive coarsening phases to reduce the size of the netlist, followed by an initial partitioning phase using the Fiduccia-Mattheyses (FM) algorithm [34]. This is followed by a series of uncoarsening (refinement) phases which each expose a finer hypergraph on the boundary of the partition. In each refinement step, the partition is iteratively refined around the boundary in order to improve partition quality. For hMetis, the package accepts the .hgr file and outputs a .part.k file which stores the results of the k-way partitioning.

III-E. Hypergraph to Netlist Reconstruction

After invoking hMetis, we obtain a k-way partition file (.part.k) which lists every hypergraph node and which partition (0 to $k-1$) that the node is assigned to. To reconstruct the *k* partitioned .vqm netlists, we use the selectively flattened (.vqm) design, the connectivity lookup (.dat) file, and the hMetis (.part.*k*) file to generate *k* .vqm files, each of which is programmed on to one of the *k* FPGAs. We create new primary IOs in each .vqm for any nets cut, when the hyperedge is not fully contained in a single partition. We use $k = 2$ for our experiments.

CHAPTER IV

RESULTS

We implement our partitioned FPGA-based logic simulation approach as an end-to-end tool flow which is completely automated using scripts. The hierarchy parser, selective flattening, hypergraph construction, slack annotation and netlist reconstruction were implemented in Perl [51]. Synthesis and timing analysis were performed using Altera Quartus II 10.1 EDA tools in Tcl [52]. All scripts were implemented and run on a Windows machine with a 2 GHz Core2 Duo processor with 2GB RAM.

We use hMetis 1.5.3 for partitioning process. As mentioned in Chapter III, we use a hypergraph representation of the selectively flattened netlist, with weights on the nodes and edges. The hMetis algorithm uses a random initial placement, and as a result, gives different results each time it is run. As a result, we make three calls to hMetis, and the best result among these is selected (both for our algorithm and the reference algorithm which we use for comparison (this algorithm partitions the completely flattened netlist)). We invoke hMetis with the options shown in Table IV.1.

Both our algorithm and the completely flattened reference were targeted to a logic simulation board which hosts 2 Altera Stratix III (EP3SL340F1517) [53] FPGAs. These FPGA parts contain 135200 ALMs, 13520 LABs, 112 LVDS IOs. The speed of the LVDS IOs is 1.25 GHz. To calculate the interconnect delay T for every signal that traverses the 2 FPGAs, we use the formula in Equation 4.1.

$$T = \lceil \#edges\_cut / \#avail\_IOpins \rceil * 0.8 + (max\_wire\_delay * distance) * 2 \tag{4.1}$$

The first term of Equation 4.1 is explained as follows. We assume that 50% of the 112 high speed LVDS IO pins are available for connecting the two Stratix FPGAs. As a result, Table IV.1. hMetis Settings Used

Option	Value	Description
Nparts	2	Number of desired partitions
UBfactor	1	Allowed imbalance between partitions (1%)
Nruns	10	Number of different bisections
Ctype	1	Vertex grouping scheme in coarsening phase
Rtype	1	Refinement policy in uncoarsening phase (FM)
Vcycle	1	Type of V-cycle refinement on bisection step
Reconst	0	Removes cut hyperedges in recursive bisection

$\#avail\_IOpins = 56$. The rest of the pins are used for control signals in the multi-FPGA logic simulation platform. Hence the number of available IO pins is fixed before partitioning. If the number of hyperedges cut exceeds the number of available IO pins, we time-division multiplex the signals. Hence the $\lceil \#edges\_cut / \#avail\_IOpins \rceil$ term represents the required depth of the time division multiplexing. Since the LVDS IO pins operate at 1.25 GHz (0.8 ns clock period), the first term of Equation 4.1 is $\lceil \#edges\_cut / \#avail\_IOpins \rceil *$ 0.8.

In addition to the inter-FPGA communication delay, we also incur delay in driving the cut signals from their driving LUTs to the periphery of the FPGA. This delay is reflected in the second term of Equation 4.1. To determine *max_wire_delay*, which is the maximum delay for a signal to traverse the FPGA, we assume a square-shaped FPGA, which implies each side will have $\sqrt{13520}$ rows of LABs. Given an average LAB-to-LAB delay of approximately 125ps [53], *max_wire_delay* will be 14.53ns. We also assume that the distance a signal travels from a LAB to reach an IO pin is $1/4$ of the chip dimension, hence *distance* is 1/4. The delay is doubled since the signal is driven from one FPGA and received by the second FPGA.

We validated our approach using multiple benchmark examples from the OpenCores [50] project. Table IV.2 lists the examples used in our experiments, along with a brief description and some statistics of each benchmark. To verify the correctness of the partitioned designs, our benchmark examples were processed through our entire tool flow. For the partitioned and unpartitioned versions of the design, we verified that both gave identical responses to a testbench for that design. In order to do this, we had to create a top-level module which instantiated the 2 partitions generated by the output of the netlist reconstruction step.

In practice, if the design being simulated can fit entirely in one FPGA of our platform, partitioning is not invoked. Since the focus of this thesis is the partitioning algorithm, we validate our approach by partitioning various designs, and estimating the speed of the partitioned design, when it is embedded into the target FPGA platform. In all examples tested, both partitioned designs fit in the target FPGA used in the platform.

Our approach was compared with a reference algorithm which partitioned the completely flattened netlist. In the reference algorithm, the netlist is completely flattened upfront and unit edge weights are used for all hyperedges, but otherwise follows the same steps as our approach. To compare the logic simulation speedup, both designs were subjected to static timing analysis using the Quartus II TimeQuest Timing Analyzer. For both approaches, dummy cells with the delay of Equation 4.1 were introduced on each of the cut nets, to model the delay of inter-FPGA communication. For each example, we compute the speedup of our approach by computing the ratio of the final clock period (reported by the timing analyzer) of reference algorithm to that of our selective flattening based partitioning algorithm, after both designs are ported to the 2-FPGA simulation platform. All runtimes reported include the time incurred by the entire flow of Figure III.1.

We calculate the geometric mean of the speedup for both the *AS* and *MS* based selective flattening algorithms over 5 examples and present this speedup against the values of p_1

Table IV.2. Details of Benchmark Examples

No	Name	Hierarchy depth	Modules	File Size	LUTs
1	openMPS430	4	18	1.4Mb	2215
2	ac97_top	3	46	1.4Mb	3398
3	can_top	3	42	1.6Mb	3114
4	mips_core	4	74	2.4Mb	4311
5	oc8051_top	4	24	2.6Mb	5837
6	m1_core	2	6	2.7Mb	5042
7	openfire_cpu	3	10	2.7Mb	5630
8	aes_cipher_top	3	43	2.8Mb	3676
9	pci_bridge32	5	139	3.3Mb	6397
10	usbDevice	5	65	5.1Mb	10799
11	spiMaster	4	17	5.9Mb	13606
12	eth_top	4	67	6.8Mb	16255
13	vga_enh_top	4	16	9.2Mb	24384
14	wb_conmax_top	5	266	11.9Mb	18251
15	xge_mac	4	28	12.4Mb	31738

and p_2 (the threshold multipliers) respectively. We use the geometric mean (GM) instead of the arithmetic mean, since individual speedup ratios for the different designs were distributed over a relatively wide range. Figure IV.1 (IV.2) reports the geometric mean of the speedup (y axis) for the *AS* (*MS*) algorithms respectively against the p_1 (p_2) value used (x axis). From Figures IV.1 and IV.2, we note that the speedup is poor for both low and high values of p_1 as wells as p_2. Good speedup values are obtained for p_1 values close to 0.9. From Figure IV.2, we note that good speedup values are obtained for p_2 values near 1.1. When the threshold multiplier values are too low, the hierarchy of the design is almost flat. On the other hand, when the threshold multiplier values are high, not enough hierarchy is exposed to the hMetis. In both these cases, sub-optimal partitioning results are obtained.

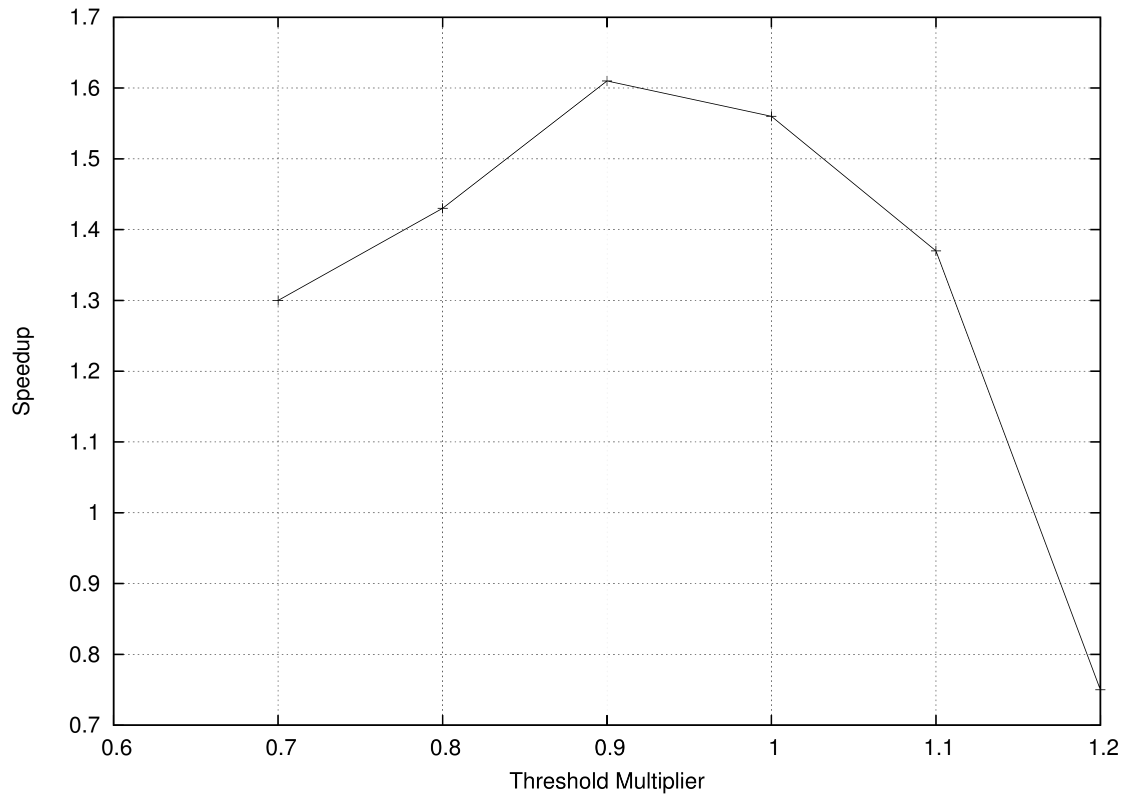

Fig. IV.1. Simulation Speedup of *AS* for 5 Examples over Flat Algorithm

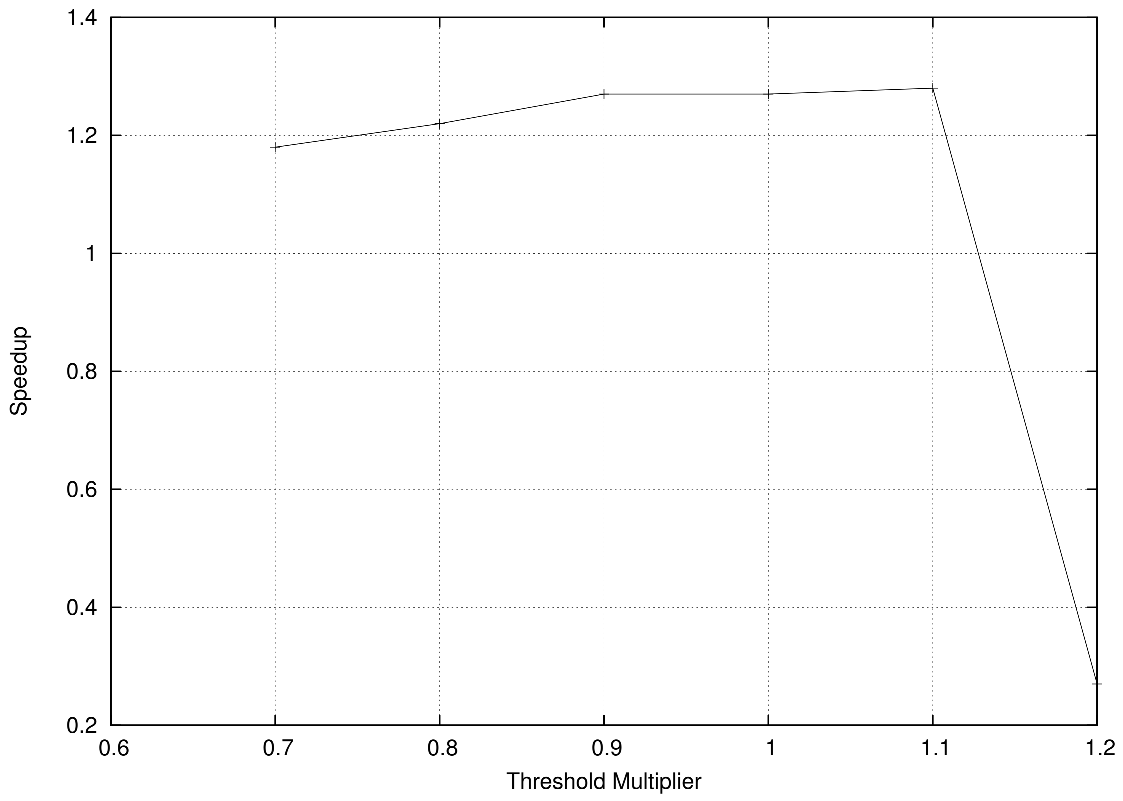

Fig. IV.2. Simulation Speedup of *MS* for 5 Examples over Flat Algorithm

For the *hybrid* algorithm, we choose p_1 and p_2 values that independently yield good performance for the *AS* and *MS* algorithms. The chosen values of p_1 is thus 0.9 and the value of p_2 is 1.1, based on the results of the 5 examples. We run the *hybrid* algorithm for 15 examples and report the speedup and runtime for each example. We also computed the speedup for $Z1$ and $Z2$ frontiers by perturbing the $Z0$ frontier as described in Section III-B.2.

Table IV.3 provides a detailed reporting of the relative simulation speedup and runtime of each of the 15 examples, for the hybrid approach (compared to the reference algorithm). In this table, the relative simulation speedup and the relative runtime are reported for $Z0$,

Table IV.3. Speedup and Runtime of *hybrid* Algorithm Compared with Fully Flat Algorithm

Example No	Reference Sim. Speed in ns	Reference Runtime in seconds	Relative Speedup *Z*0	Relative Runtime *Z*0	Relative Speedup *Z*1	Relative Runtime *Z*1	Relative Speedup *Z*2	Relative Runtime *Z*2
1	62.84	27	1.03	1.59	0.85	1.67	1.00	1.82
2	26.28	54	0.88	1.45	0.88	1.45	0.88	1.45
3	29.96	48	1.00	1.10	1.00	1.47	1.00	1.47
4	37.09	105	1.11	0.59	2.13	0.70	2.13	1.05
5	38.71	78	0.89	0.93	0.89	1.02	1.67	1.08
6	34.84	41	1.05	1.67	1.00	2.19	1.00	2.19
7	37.59	48	0.65	1.59	0.71	1.63	0.76	1.90
8	58.13	58	1.56	1.55	1.52	1.47	1.52	1.47
9	50.55	314	1.79	0.26	1.54	0.38	1.45	0.96
10	28.53	300	0.94	0.29	0.96	0.30	0.93	0.40
11	27.29	218	1.22	0.57	0.79	0.61	0.79	0.93
12	70.80	443	5.56	0.62	2.86	0.67	1.92	1.15
13	39.45	684	4.76	0.28	1.15	1.07	1.15	1.19
14	33.74	1574	7.14	0.29	1.01	0.31	1.00	0.55
15	24.70	675	1.47	0.96	1.47	0.83	1.47	0.86
GM	37.99	150	1.52	0.74	1.15	0.89	1.00	1.13

*Z*1 and *Z*2 frontiers. Column 1 presents the example number. Column 2 reports the clock period on the 2-FPGA platform and Column 3 reports the runtime of the completely flattened (reference) algorithm respectively. Columns 4, 6 and 8 report the relative simulation speedup and Columns 5, 7 and 9 report the relative algorithmic (for the hybrid algorithm) runtime for *Z*0, *Z*1 and *Z*2 frontiers respectively.

The runtime is calculated as the sum of hierarchy analysis, selective flattening, partitioning and netlist reconstruction runtimes. Selective flattening includes the slack calculation and flattening algorithm. Partitioning includes .hgr generation and hMetis runtimes. For the completely flat reference approach, slack calculation runtime is not included.

We note that our *hybrid* algorithm gives a healthy simulation speedup for most examples, with an average speedup of 1.52$\times$ for *Z*0 frontier. A few small examples exhibit a slowdown, attributed to the fact that in some instances, our algorithm does not expose enough of the hierarchy to hMetis, resulting in a sub-optimal partition. We also observe that average runtime for our *hybrid* *Z*0 algorithm is 0.74$\times$ compared to completely flat algorithm.

We also note that among the three frontiers explored, $Z0$ yields best simulation speedup and runtime. $Z1$ and $Z2$ exhibit worse simulation speedup compared to $Z0$ with more runtime. $Z1$ achieves an average speedup of 1.15×, with 0.89× runtime compared to the reference approach. Since the $Z2$ frontier is yet further flattened, average simulation speedup achieved is 1, with 1.13× relative runtime compared to the completely flat approach. As a result, we recommend the *hybrid* method with $Z0$ as the frontier of choice for the selective flattening algorithm.

If we perform *no* flattening (except the top level module) with unit edge weights, we mimic the MRFM (module synthesis followed by recursive Fiduccia-Mattheyses) method, which was the reference method used in [1]. Table IV.4 reports the relative speedup MRFM method compared to our completely flat reference algorithm. We note that the average simulation speedup obtained by MRFM (over 15 examples) is 0.90 compared to our completely flat reference algorithm. Hence, compared to MRFM, our approach gives a speedup of 1.52/0.90 = 1.69× over the 15 examples, whereas [1] reports a speedup of 1.25× over 6 examples. Hence our method achieves a 35% simulation speedup over [1].

Table IV.5 demonstrates the simulation speedup improvement of the *hybrid* algorithm compared to the stand-alone AS and MS algorithms, for the $Z0$ frontier. The p_1 and p_2 used for this table are 0.9 and 1.1 respectively. As shown by Table IV.5, the *hybrid* algorithm on average achieves a 1.52/1.39 = 1.09× improvement over the AS algorithm, and a 1.52/1.19 = 1.28× improvement over the MS algorithm. The reason for this, as indicated in Section III-B.1.c, is that the AS algorithm does not flatten instances of type $\{C_2\}$ (with a small AS and large MS values). Similarly the MS algorithm does not flatten instances of type $\{C_1\}$ (with a large AS and a small MS values). However, the *hybrid* algorithm can flatten instances of both types.

Table IV.4. Speedup and Runtime of MRFM Algorithm Compared with Fully Flat Algorithm

Example No	Reference Sim. Speed in ns	Relative Speedup of MRFM used in [1]
1	62.84	1.03
2	26.28	0.93
3	29.96	0.84
4	37.09	1.57
5	38.71	0.89
6	34.84	0.74
7	37.59	0.91
8	58.13	0.69
9	50.55	1.54
10	28.53	0.94
11	27.29	1.22
12	70.80	1.21
13	39.45	1.26
14	33.74	0.15
15	24.70	1.00
GM	37.99	0.90

Table IV.5. Relative Speedup of *hybrid*, *AS* and *MS* Algorithms Compared with Fully Flat Algorithm

Example No	Relative Speedup for *Z*0 frontier		
	hybrid	*AS*	*MS*
1	1.03	1.03	1.03
2	0.88	0.88	0.91
3	1.00	1.00	1.00
4	1.11	1.11	1.11
5	0.89	0.89	0.89
6	1.05	1.06	0.74
7	0.65	0.66	0.80
8	1.56	1.52	0.96
9	1.79	1.79	1.32
10	0.94	0.94	0.94
11	1.22	1.22	1.22
12	5.56	5.56	5.56
13	4.76	1.27	4.76
14	7.14	7.14	0.61
15	1.47	1.47	1.00
GM	1.52	1.39	1.19

CHAPTER V

CONCLUSIONS

In this thesis we have presented a partitioning method for a multi-FPGA based logic simulator platform. This approach employs a top-down selective flattening of the design hierarchy, thus avoiding the potential blowup in netlist size. Also, our approach preserves design hierarchy, thereby resulting in better debuggability. A hierarchical netlist is selectively flattened using a hybrid metric based on the average slack and the minimum slack of any instance in the hierarchy. Experimental results on a set of examples of various sizes from the OpenCores benchmark suite demonstrate that our approach obtains an average speedup of $1.52\times$, with an average runtime of $0.74\times$ compared to a full flattening based partitioning approach.

REFERENCES

[1] W.-J. Fang and A.-H. Wu, “Performance-driven multi-FPGA partitioning using functional clustering and replication,” in *Proc. 35th Design Automation Conference*, June 1998, pp. 283 –286.

[2] Y.-C. Lin, S.-F. Tseng, Y.-S. Hung, and T.-M. Hsieh, “Cost minimization of partitioning circuits with complex resource constraints in FPGAs,” in *IEEE Asia-Pacific Conference on Circuits and Systems*, 2000, pp. 556 –559.

[3] R. Kuznar, F. Brglez, and B. Zajc, “Multi-way netlist partitioning into heterogeneous FPGAs and minimization of total device cost and interconnect,” in *31st Conference on Design Automation*, June 1994, pp. 238 – 243.

[4] D. Bhatia and V. Narasimhan, “Simple yet effective replication for FPGA partitioning,” in *Proc. 7th IEEE International ASIC Conference and Exhibit*, Sep. 1994, pp. 152 –155.

[5] F. Vahid, T. D. Le, and Y.-C. Hsu, “A comparison of functional and structural partitioning,” in *Proc. 9th International Symposium on System Synthesis*, Nov. 1996, pp. 121 –126.

[6] V. Srinivasan, S. Govindarajan, and R. Vemuri, “Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 140 –158, Feb. 2001.

[7] Y.-S. Hung, C.-H. Lee, S.-F. Tseng, and T.-M. Hsieh, “Minimum cost complex resource FPGA partition with performance refining,” in *45th Midwest Symposium on Circuits and Systems*, vol. 2, Aug.. 2002, pp. II–107 – II–110 vol.2.

[8] D. Pashley, D. Brasen, and G. Saucier, “New partitioning technology permits FPGA prototypes,” in *IEE Colloquium on Fast Prototyping of IC Designs*, June 1994, pp. 2/1 –2/6.

[9] C. Kim, H. Shin, and Y. Yu, “Performance-driven circuit partitioning for prototyping by using multiple FPGA chips,” in *Proc. Design Automation Conference, IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration, Asian and South Pacific*, Aug.-1 Sep. 1995, pp. 113 –118.

[10] C. Kim and H. Shin, “A performance-driven logic emulation system: FPGA network design and performance-driven partitioning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 5, pp. 560 –568, May 1996.

[11] D. Behrens, K. Harbich, and E. Barke, “Hierarchical partitioning,” in *Digest of Technical Papers., IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1996, pp. 470 –477.

[12] W. McDermith, “A bottom-up approach to FPGA partitioning,” in *Proc. IEEE Custom Integrated Circuits Conference*, May 1992, pp. 5.4.1 –5.4.4.

[13] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai, and R. Lindelof, “Circuit partitioning for huge logic emulation systems,” in *31st Conference on Design Automation*, June 1994, pp. 244 – 249.

[14] R. Kuznar, F. Brglez, and K. Krzysztof, “Cost minimization of partitions into multiple devices,” in *30th Conference on Design Automation*, June 1993, pp. 315 – 320.

[15] N.-S. Woo and J. Kim, “An efficient method of partitioning circuits for multiple-FPGA implementation.” in *30th Conference on Design Automation*, June 1993, pp.

202 – 207.

[16] Z. Marrakchi, H. Mrabet, and H. Mehrez, "Evaluation of hierarchical FPGA partitioning methodologies based on architecture Rent parameter," in *Research in Microelectronics and Electronics 2006, Ph. D.*, 0-0 2006, pp. 85 –88.

[17] D. Brasen, J. Hiol, and G. Saucier, "Finding best cones from random clusters for FPGA package partitioning," in *Proc. Design Automation Conference, IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration, Asian and South Pacific*, Aug.-1 Sep. 1995, pp. 799 –804.

[18] Z. Rongzheng, T. Jiarong, and T. Pushan, "Fpart: a multi-way FPGA partitioning procedure based on the improved fm algorithm," in *Proc. Design Automation Conference Asia and South Pacific*, Feb. 1998, pp. 513 –518.

[19] G.-M. Wu, J.-M. Lin, and Y.-W. Chang, "Generic ILP-based approaches for time-multiplexed FPGA partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1266 –1274, Oct. 2001.

[20] W.-J. Fang and A.-H. Wu, "A hierarchical functional structuring and partitioning approach for multiple-FPGA implementations," in *Digest of Technical Papers., 1996 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1996, pp. 638 –643.

[21] H. Krupnova, A. Abbara, and G. Saucier, "A hierarchy-driven FPGA partitioning method," in *Proc. 34th Design Automation Conference*, June 1997, pp. 522 –525.

[22] S. Hauck and G. Borriello, "Logic partition orderings for multi-FPGA systems," in *Proc. 3rd International ACM Symposium on Field-Programmable Gate Arrays*, 1995,

pp. 32 – 38.

[23] W.-J. Fang and A.-H. Wu, "Multi-way FPGA partitioning by fully exploiting design hierarchy," in *Proc. 34th Design Automation Conference*, June 1997, pp. 518 –521.

[24] D.-H. Huang and A. Kahng, "Multi-way system partitioning into a single type or multiple types of FPGAs," in *Proc. 3rd International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 140 – 145.

[25] J. Hidalgo, R. Baraglia, R. Perego, J. Lanchares, and F. Tirado, "A parallel compact genetic algorithm for multi-FPGA partitioning," in *Proc. 9th Euromicro Workshop on Parallel and Distributed Processing*, 2001, pp. 113 –120.

[26] G. Saucier, D. Brasen, and J. Hiol, "Partitioning with cone structures," in *Digest of Technical Papers., IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1993, pp. 236 –239.

[27] P. Chan, M. Schlag, and J. Zien, "Spectral-based multiway FPGA partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 5, pp. 554 –560, May 1996.

[28] Y.-C. Wei and C.-K. Cheng, "Towards efficient hierarchical designs by ratio cut partitioning," in *Digest of Technical Papers., IEEE International Conference on Computer-Aided Design*, Nov. 1989, pp. 298 –301.

[29] "Performance-driven partitioning using a replication graph approach," in *32nd Conference on Design Automation*, 1995, pp. 206 –210.

[30] R. Burra and D. Bhatia, "Timing driven multi-FPGA board partitioning," in *Proc. 11th International Conference on VLSI Design*, Jan. 1998, pp. 234 –237.

[31] A. Singla and T. Conte, “Bipartitioning for hybrid FPGA-software simulation,” in *Proc. 9th International Conference on VLSI Design*, Jan. 1996, pp. 211 –214.

[32] Y. Cheon and M. Wong, “Design hierarchy-guided multilevel circuit partitioning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 4, pp. 420 – 427, Apr. 2003.

[33] L. Hagen and A. Kahng, “Fast spectral methods for ratio cut partitioning and clustering,” in *Digest of Technical Papers., 1991 IEEE International Conference on Computer-Aided Design*, Nov. 1991, pp. 10 –13.

[34] C. Fiduccia and R. Mattheyses, “A linear-time heuristic for improving network partitions,” in *19th Conference on Design Automation*, June 1982, pp. 175 –181.

[35] L. Hwang and A. El Gamal, “Min-cut replication in partitioned networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 1, pp. 96 –106, Jan. 1995.

[36] D. T. Prashant Sawkar, “Multi-way partitioning for minimum delay for look-up table based FPGAs,” in *32nd Conference on Design Automation*, 1995, pp. 201 –205.

[37] S. Harikumer and S. Kumar, “Multiobjective search based algorithms for circuit partitioning problem for acceleration of logic simulation,” in *Proc. 10th International Conference on VLSI Design*, Jan. 1997, pp. 239 –242.

[38] V. Sankarasubramanian and D. Bhatia, “Multiway partitioner for high performance FPGA based board architectures,” in *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1996, pp. 579 –585.

[39] J. Hwang and A. El Gamal, “Optimal replication for min-cut partitioning,” in *Proc. IEEE/ACM international conference on Computer-aided design*, ser. ICCAD

’92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 432–435. [Online]. Available: http://dl.acm.org/citation.cfm?id=304032.304148

[40] L.-T. Liu, M.-T. Kuo, C.-K. Cheng, and T. Hu, “A replication cut for two-way partitioning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 5, pp. 623 –630, May 1995.

[41] K. Roy and C. Sechen, “A timing driven n-way chip and multi-chip partitioner,” in *Digest of Technical Papers., IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1993, pp. 240 –247.

[42] S. Dutt and W. Deng, “VLSI circuit partitioning by cluster-removal using iterative improvement techniques,” in *Digest of Technical Papers., IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1996, pp. 194 –200.

[43] D. Brasen and G. Saucier, “FPGA partitioning for critical paths,” in *Proc. European Event in ASIC Design, European Design and Test Conference, The European Conference on Design Automation, European Test Conference*, Feb.-3 Mar. 1994, pp. 99 –103.

[44] K. Roy-Neogi and C. Sechen, “Multiple FPGA partitioning with performance optimization,” in *Proc. 3rd International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 146 – 152.

[45] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli, “On clustering for minimum delay/area,” in *Digest of Technical Papers., IEEE International Conference on Computer-Aided Design*, Nov. 1991, pp. 6 –9.

[46] L.-T. Liu, M. Shih, N.-C. Chou, C.-K. Cheng, and W. Ku, “Performance-driven partitioning using retiming and replication,” in *Digest of Technical Papers., IEEE/ACM*

International Conference on Computer-Aided Design, Nov. 1993, pp. 296 –299.

[47] D. Brasen and G. Saucier, “Using cone structures for circuit partitioning into FPGA packages,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 7, pp. 592 –600, July 1998.

[48] Y. Choi, Y. S. Jeong, and C. Rim, “A topology-based multi-way circuit partition for ASIC prototyping,” in *IEEE 39th Midwest symposium on Circuits and Systems*, vol. 1, Aug. 1996, pp. 357 –360 vol.1.

[49] G. Karypis and V. Kumar, *A Software package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices*, http://www-users.cs.umn.edu/∼karypis/metis, Sep. 1998.

[50] “The OpenCores project,” http://www.opencores.org.

[51] “The Perl programming language,” http://www.perl.org.

[52] “The Tcl developer site,” http://www.tcl.tk.

[53] *Stratix III Device Handbook*, http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf, Mar. 2011.

VITA

Subramanian Poothamkurissi Swaminathan received his B. Tech degree in electrical engineering from the National Institute of Technology (NIT) Trichy, India in 2010. His undergraduate research focused on control strategies for power conversion in wind turbine systems. He joined Texas A&M University in August 2010 to pursue his master's degree in computer engineering. He worked as a Design Engineer Intern at Intersil for 6 months in 2011. His current research interests include FPGA partitioning and algorithms.

Subramanian Poothamkurissi Swaminathan may be reached at:
102 WERC,
Texas A&M University,
College Station, TX-77843
e-mail: subramanianps@neo.tamu.edu