A Combined Clustering and Placement Algorithm for FPGAs

by

Mark Yamashita

B.A.Sc., The University of Toronto, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

The University Of British Columbia

November, 2007

Abstract

One of the major drawbacks of reprogrammable microchips, such as field-programmable gate arrays (FPGAs), is an inherent speed disadvantage over ASIC technologies. To mitigate this speed disadvantage, this thesis presents a novel algorithm to improve timing performance at the possible expense of area and runtime. The algorithm presented leverages node duplication and a depth-optimal initial clustering to provide a starting point for a non-greedy, iterative optimization technique using detailed placement and timing information to develop the final clustering and placement solutions.

For a set of benchmarks commonly used in FPGA research, the proposed algorithm achieves an 11% critical-path delay improvement compared to the VPR academic tool flow. This performance improvement is obtained at the expense of a 44% increase in area usage and a 26x increase in maximum runtime. Techniques have also been implemented to sacrifice performance to moderate the area or runtime increases. For a 1% critical-path delay penalty, the runtime can be improved by a factor of 4. The algorithm also provides facilities to impose area restrictions, in which case timing degradation is proportional to the area saved.

Table of Contents

Appendices

List of Tables

List of Figures

Glossary

Application Specific Integrated Circuit (ASIC):	An integrated circuit that is designed and manufactured for a specific task, as opposed to a reprogammable circuit such as an FPGA.
Basic Logic Element (BLE):	A logic entity in the FPGA consisting of a LUT and a flip-flop.
Computer Aided Design (CAD) Tool:	Software tools that assist designers in building complex systems.
Configurable Logic Block (CLB):	A group of N BLEs
Field Programmable Gate Array (FPGA):	A customizable integrated circuit that can be programmed to perform a given function.
Look Up Table (LUT):	A logic element that can compute any function of up to k inputs.
Microelectronics Corporation of North Carolina (MCNC) Circuits:	A set of FPGA benchmark circuits commonly used in academic research.
T-VPack:	The most commonly used academic clustering tool.
Versatile Place & Route (VPR):	The most commonly used academic place and route tool.

Acknowledgements

First, I would like to thank my supervisor Guy Lemieux for his support and guidance. Without him this work would not have been possible. I would also like to thank other member of the SOC Lab, past and present, for their help and expertise. Particularly Dave Grant, Marvin Tom, Eddie Lee, Karim Allidina, Julien Lamoureux, Scott Chin, Natalie Chan, Andrew Lam, Paul Teehan, Jason Yu and Roberto Rosales.

I would also like to acknowledge Konrad Walus for generously allowing me to use his computer cluster and Nick Geraedts who administers the cluster.

I am grateful to my friends and family for their support. Especially my girlfriend Anya, who has provided constant encouragement over the past two years.

Finally, I would like to thank my parents. To my mother who has always had faith in me, and to my father, who I know would be proud.

Chapter 1

Introduction

1.1 Motivation

A field-programmable gate array (FPGA) is a customizable integrated circuit that can be programmed to perform a given function. The programmability of the device stems from a highly flexible routing architecture and programmable logic elements. The standard approach to using FPGAs is to describe a logic circuit in a hardware description language (HDL) such as Verilog and use an FPGA computer-aided design (CAD) tool to produce a bitstream that will program the routing architecture and logic elements. It is the job of the CAD tool to produce a result that satisfies the requirements of the user, in terms of such metrics as delay, area and power.

The most definitive advantage an FPGA has over a design in a comparable technology, such as an application specific integrated circuit (ASIC) or standard cell design, is a low non-recurring engineering (NRE) cost and fast time to market. Though an ASIC may be faster and have a lower unit cost, the cost to design, verify, and produce a chip can be prohibitively expensive for low to medium volume designs. A modern ASIC design requires considerable man-hours, a mask-set that can cost up to $1.5 million USD for 65-nm technology [31], and possibly several months for the design to be fabricated. An FPGA design is comparatively simpler, has no upfront production costs and requires only that the FPGAs be programmed instead of manufactured at a foundry. As mentioned, though, these advantages are obtained at the expensive of unit cost, power consumption, and speed. According to [28], in a 90nm process technology, an FPGA suffers a 4 times speed disadvantage and 14 times increase in dynamic power when compared to an ASIC.

As process technologies shrink and circuits become larger, the absolute performance gap between ASICs and FPGAs will continue to increase. To remain competitive, FPGA technology must close this performance gap. One of the most direct ways to improve FPGA performance is to create better CAD tools.

In the clustering step of the FPGA CAD flow, the majority of algorithms are based on a greedy approach. While a greedy algorithm is fast and effective, it is limited by an inability to adjust the clustering solution during later stages of the CAD flow and it provides no direct means for node duplication, a method that can reduce interconnect delay by reducing logic depth.

FPGA clustering algorithms that use node duplication, do so either in excess, or as a mechanism for tuning the solution. When node duplication is done solely at the clustering stage, because of a lack of accurate timing information, a large amount of duplication is required to ensure a performance improvement. Using duplication to tune the clustering solution, either after placement ([3], [53]) or after each iteration of the simulated annealer ([9]), may limit the performance gains achievable through node duplication.

Previous research has also tried to combine the clustering and placement steps of the FPGA CAD flow. The approach used by [9] is to integrate clustering changes into a simulated annealing-based placer. While this approach is effective in making incremental improvements to the clustering solution, it still lacks the ability to fully utilize accurate timing information when making fundamental clustering decisions.

Finally, one feature not present in current clustering tools is the ability to trade off area for performance. If the final routed solution does not meet the timing requirements of the design, the designer has no opportunity to expend additional resources to improve the timing of the circuit.

1.2 Objectives

The algorithm presented in this thesis leverages node duplication and a depth-optimal initial clustering to provide a starting point for a non-greedy, iterative optimization technique using detailed

placement and timing information to develop the final clustering and placement solutions. The main goal is to improve timing performance at the possible expense of area and runtime.

To combine the clustering and placement steps in a runtime-efficient manner, the program must first create an initial clustering solution. This solution is comprised of a set of microclusters, which are small groups of connected blocks. The formation of microclusters provides the algorithm with a means to create a preliminary placement from which accurate timing information can be extracted. As the composition of microclusters usually persists through the remainder of the CAD flow, it is the aim of the Microcluster Formation Phase to create microclusters that are small, highly-cohesive entities that facilitate a low critical-path delay.

During the initial clustering phase, a depth-optimal clustering solution is created. This means that the worst-case number of clusters traversed along any path from input to output is minimized. It will be shown that this depth advantage translates into a critical-path delay advantage after routing. Therefore, the clustering and placement algorithm should strive to maintain control over the depth of the circuit.

From an initial placement of microclusters, the program can create an accurate timing model of the circuit. With this comprehensive timing and placement information, the goal of the algorithm is to use this information to make informed decisions about how to alter the clustering and placement solutions. Making a change to the organization or composition of the microclusters constitutes a change in both the clustering and placement solutions. Therefore, the program should use all timing and placement information available to ensure each change maximizes the benefit to timing performance, area and routing resource usage.

Node duplication can be instrumental in helping reduce the critical-path delay of the circuit. Node duplication refers to replicating the same logic function, with the same inputs, in different locations on the chip. Duplication can reduce delay by allowing successor blocks to take inputs from a local copy to improve timing, rather than taking it from a distant original. The objective of the algorithm is to use node duplication as effectively as possible to reduce delay on the critical path,

while not impeding the algorithm from meeting the imposed area restrictions. The approach taken is to create a large amount of duplication during the Microcluster Formation Phase and then reduce the amount of duplication as required throughout the remainder of the algorithm. It is therefore essential that when removing duplication, the algorithm retains those nodes that are the most beneficial to timing performance.

One final objective is to allow the user to sacrifice area for an improvement in timing performance. If the user has already purchased a chip that is larger than required for the circuit, it is the goal of the algorithm to take full advantage of the additional logic to increase performance. This relies on the principle that more node duplication can reduce the critical-path delay of the circuit. The trade-off between area and performance should be a continuous function, where the greater the area increase, the greater the performance gain, up to a certain point.

1.3 Contributions

This thesis introduces a combined clustering and placement algorithm consisting of two unique algorithms, the Node Duplicate Reduction algorithm and the Orchestrator algorithm. The Node Duplicate Reduction (NDR) algorithm provides a proficient means of reducing the node duplication of a label and cluster solution, while maintaining a specified depth. The Orchestrator algorithm introduces a novel strategy for using placement and timing informationto make informed decisions concerning how to reorganize and consolidate the initial clustering from the NDR algorithm to reduce area usage and critical-path delay.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides a concise overview of modern FPGA technology, associated state of the art CAD tools and previous work which is related to the subject of this thesis. Chapter 3 provides a broad overview of the algorithm presented in the

thesis. Phase 1 of the algorithm is described in Chapter 4, and Phase 2 of the algorithm is described in Chapter 5. Chapter 6 compares the results of the proposed algorithm with other academic tools. Finally, Chapter 7 presents conclusions drawn from this research, as well as contributions made and possible future work.

Chapter 2

Background

This chapter first presents an overview of modern field-programmable gate array (FPGA) technology. It then describes the computer-aided design (CAD) flow used to map a circuit on to an FPGA. For each step in the CAD flow, a short description of the function of that step is given, along with a survey of common tools used for the given step, with particular emphasis placed on the clustering and placement steps. Finally, a more thorough review of work most similar to this thesis is presented.

2.1 FPGA Technology Overview

The primary building block of an FPGA is the basic logic element (BLE), consisting of a *k*-input look-up table (LUT) and a flip-flop. A *k*-input LUT can perform any Boolean logic function of up to *k* inputs. The BLE can either be used in sequential mode, where the output is taken from the flip-flop, or in combinational mode, where the output is supplied by the LUT. In this thesis, a BLE is referred to as a *block* or a *node* when representing a netlist entity and as a BLE when representing an architectural entity.

Modern FPGAs such as the Xilinx Virtex-5 [24] and Altera Stratix III [15] group a number of BLEs into a configurable logic block (CLB). A CLB provides fast local interconnect to directly route signals between resident BLEs. This allows the CAD flow to place related BLEs in the same CLB to reduce delay incurred between them. A CLB is characterized by the number of BLEs in the cluster (N) and the total inputs to the cluster (I). It is assumed that each BLE may draw inputs from

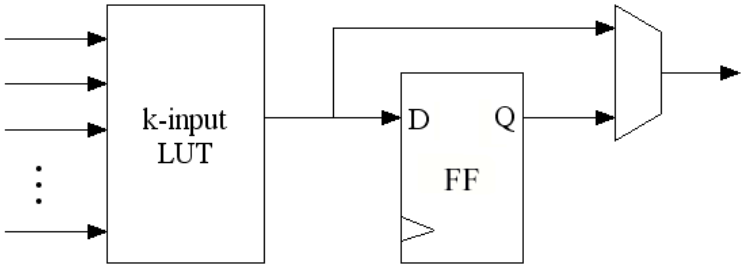

Figure 2.1: Basic Logic Element

any of the *I* cluster inputs or any BLE output within the CLB. In this document CLBs are referred to as *clusters* when representing a netlist entity and as CLBs when representing an architectural entity.

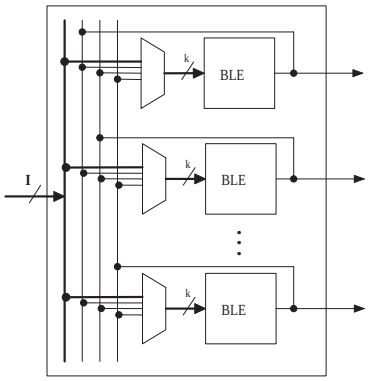

Figure 2.2: Configurable Logic Block

Generally, an FPGA is arranged so that the CLBs form a grid. The routing architecture forms routing channels between each row and column of CLBs. A channel segment is the portion of a channel that spans one CLB. The number of signals that can occupy a given routing channel segment is termed the **channel width** of the FPGA. It is the responsibility of the CAD flow to ensure that the maximum number of signals routed through any channel is less than or equal to the channel width. If the channel width is exceeded, the circuit is said to be unroutable.

During the design process of commercial FPGAs, a single tile is designed consisting of a CLB and its adjacent routing resources. For all devices in that family, the FPGA consists of an array of those tiles in different size grids. So, for a given FPGA family, *k*, *N*, *I* and the channel width are

fixed, and the variant is the number of tiles on the chip. The following table shows the number of BLEs in two popular FPGA families.

Stratix III (Altera) k=6, N=10

EP3SL50	EP3SL70	EP3SL110	EP3SL150	EP3SL200	EP3SE260	EP3SL340
19,000	27,000	42,600	56,800	79,560	101,760	135,200

Virtex-5 (Xilinx) k=6, N=8

XC5VLX30	XC5VLX50	XC5VLX85	XC5VLX110	XC5VLX220	XC5VLX330
19,200	28,800	51,840	69,120	138,240	207,360

Table 2.1: Commercial FPGA Device Sizes in total basic logic elements

For this thesis, unless otherwise noted, the following *de facto* standard FPGA architecture is assumed:

- Lut size: $k = 4$
- Cluster size: $N = 10$
- Inputs per CLB: $I = (k/2) * (N + 1) = 22$, as recommended by [1]
- wirelength = 4 (a single wire spans 4 CLBs)
- I/O ratio = sufficient to ensure circuit is logic limited
- Switch block type = subset
- Connection block input connectivity = 0.4
- Connection block output connectivity = 0.125
- Connection block pad connectivity = 1
- Switch type = buffered

For a further description of these architectural features, please see [5].

2.2 FPGA CAD Flow

The purpose of the FPGA CAD flow is to produce a bitstream used to program the device from a hardware description language (HDL) specification of the circuit. The bitstream designates the function of all BLEs and specifies how the routing architecture should route signals between them. The CAD flow must ensure that no architectural limitations are exceeded and should attempt to optimize a given set of metrics, such as delay, area and power.

The FPGA CAD flow is generally broken into 5 steps: synthesis, technology mapping, clustering, placement and routing. In the standard flow, each step is performed in turn with no backtracking. What follows is a brief description of each step, with particular emphasis on steps that are related to this work.

2.2.1 Synthesis

In the synthesis step, the HDL description of the circuit expressed in a language such as VHDL ((Very-High-Speed Integrated Circuits) Hardware Description Language) or Verilog is translated into a gate-level description of the circuit. Technology-independent logic optimization is also performed during this step.

2.2.2 Technology Mapping

The technology mapping step takes the netlist produced by the synthesis step and maps the circuit into a series of *k*-input LUTs and flip-flops. Figure 2.3 shows an example of technology mapping, where a netlist described as a directed acyclic graph (DAG) is mapped to 4-input LUTs. The goal of technology mapping can be reduce the number of LUTs used ([46], [20], [26], [52]), delay ([11], [10], [21], [47]), or some combination of the two. The technology mapping problem generally equates to finding sets of Boolean gates with a total of no greater than *k* inputs, and implementing these gates within a LUT.

When optimizing for delay, the goal is to minimize the LUT depth of the circuit. The LUT depth

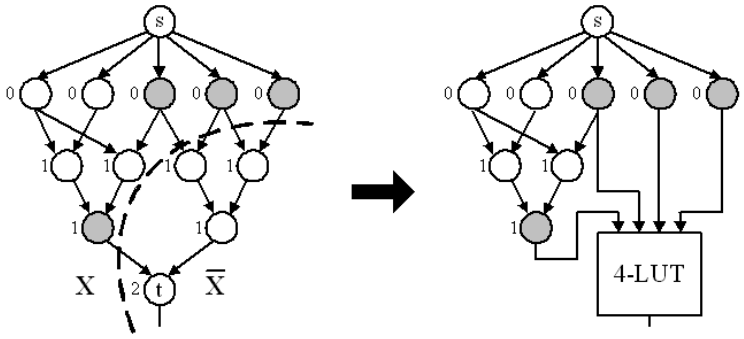

Figure 2.3: Technology Mapping Example (from [30])

is defined as the maximum number of LUTs traversed from circuit input to circuit output. The most well-known academic tool for delay-driven technology mapping is FlowMap [11]. FlowMap will find a solution that has an optimal LUT depth in polynomial time.

For the MCNC benchmark circuits used in this thesis, all technology mapping of circuits was performed by running FlowMap [11] for depth optimality followed by FlowPack [12] for area reduction. Next, SIS [55] was run with the scripts *script.rugged* and *script.algebraic*, and the lower area solution out of the two was chosen.

2.2.3 Clustering

The function of the clustering step is to assign blocks to clusters that will fit in a CLB to reduce the delay and routing resource usage of the circuit. The CLB has its own routing structure that, in general, fully connects all block inputs to: 1. the inputs of the CLB and 2. the outputs of all resident BLEs. This fully-connected intracluster routing is faster than the general routing resources, so any signals that can be routed intracluster will incur a smaller delay. Additionally, using the CLB's routing structure reduces the stress on the general routing architecture, resulting in a lower required channel width. An example of the clustering process is shown in Figure 2.4.

The simplest clustering algorithms are based on a greedy approach, whereby a single block is chosen as a seed, and other blocks are added to the cluster depending on their relation to that seed,

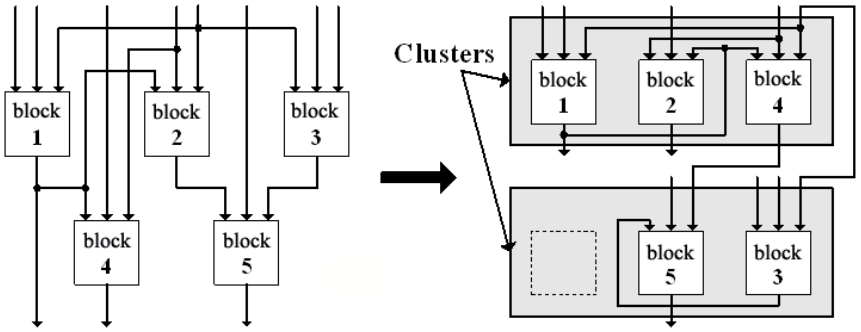

Figure 2.4: Clustering Example (from [30])

until the capacity constraints are met. Being greedy, there is typically no backtracking; once a block is placed in a cluster, it remains in that cluster for the duration of the CAD flow. Many such greedy algorithms exist, all with distinct goals. Such algorithms include VPack [4], which tries to minimize the total number of inputs per cluster; T-VPack [41], which tries to reduce the number of intercluster nets on the critical path; RPack [7], which attempts to reduce the routing effort of the circuit by enclosing intercluster nets in clusters; a timing-aware version of RPack, T-RPack [6], which assigns a criticality to each net to aid in deciding which nets to make intracluster; and finally iRAC [57] which attempts to minimize routing channel width by completely enclosing low-fanout nets within a cluster. Greedy clustering algorithms are fast and area efficient, but due to a lack of backtracking, they can get trapped in local minima.

The next group of algorithms are termed label and cluster, where two passes of the circuit are made. The first assigns a label to each node, relative to the minimum time required before it produces a result; the second pass groups nodes into clusters using the labels previously assigned. The origin of the label and cluster method is the Lawler Levitt Turner (LLT) algorithm [32], which produces the smallest maximum circuit depth, but at the expense of 2-3x area overhead. A more generalized delay model, where gate delays are included, is presented in [45]. Rajaraman and Wong [51] presented a performance-driven optimal clustering algorithm introduced in [45], based on dynamic programming. This algorithm also suffers from a high area overhead through node

duplication.

A number of attempts have been made to build upon the algorithm presented in [51]. The clustering algorithm presented in [63] extended the algorithm by introducing a post-clustering node duplication reduction technique, rooted cluster elimination, which eliminates a cluster when the root of that cluster can be replaced by one of its duplicates without affecting timing. Timing Driven Clustering followed by cluster Packing (TDCP) [16] builds upon [51]'s algorithm by using a slack based node duplication control, whereby any node with a slack greater than some predetermined amount is not duplicated, even if it detrimentally effects circuit delay. In both techniques, node duplication reduction is performed during the clustering step. This limits the amount of timing information available, subsequently restricting the algorithm's ability to make informed clustering decisions.

Multi-level clustering algorithms, though targeted to hierarchical FPGAs, can lend important innovation to clustering as a whole. Two-level clustering (TLC) [13] builds upon the dynamic programming algorithm in [51], and also introduces the node duplication reduction technique of refusing to duplicate nodes exceeding a certain slack. The multi-level clustering tool presented in [60] performs clustering by using the algorithm from [51] at each level. As with the label and cluster techniques, multi-level clustering addresses node duplication reduction as a post-clustering step.

Work has also been done on clustering performed in unison with other steps in the FPGA design flow. The algorithm presented in [39] performs simultaneous technology mapping and clustering. Through this, they can make area-aware technology mapping decisions to reduce area usage. The Simultaneous Placement with Clustering and Duplication (SPCD) algorithm, presented in [9] incorporates incremental changes to the clustering solution during annealing and allows duplication insertion after each iteration of the annealer. The algorithm is limited by the fact that duplication and clustering changes are only employed to provide incremental improvements to the solution; the initial clustering is taken directly from T-VPack [41].

A modified clustering and placement algorithm is presented in [53], where certain CLBs are

only partially filled during clustering, and node duplication is performed after placement to reduce the critical-path delay. Duplication is performed after routing in [3] to straighten critical paths, reducing the delay incurred on them. In these last two techniques, placement is performed with no information on where duplication will exist. Therefore, it has no ability to account for duplication during placement to improve performance.

Finally, DPack [17] creates a fast, min-cut partitioning based placement of BLEs to provide placement information during the clustering stage. The placement information is used in a greedy-based clustering tool which integrates the placement information into the algorithm's cost function. While this algorithm demonstrates the merit of including placement information during clustering, the placement information is not accurate enough to provide a useful timing model. Also, it does not consider the benefits of duplication.

A summary of the clustering methods mentioned above is provided in Table 2.2.

Reference	Algorithm Name / Title	Clustering Approach	Duplication
[4]	VPack	Greedy	none
[41]	T-VPack	Greedy	none
[7]	RPack	Greedy	none
[6]	T-RRack	Greedy	none
[57]	iRAC	Greedy	none
[32]	Lawler Levitt Turner (LLT)	Label and Cluster	during clustering
[45]	Generalized LLT	Label and Cluster	during clustering
[51]	Optimum Clustering for Delay Minimization	Label and Cluster	during clustering
[63]	CLUSTER	Label and Cluster	during clustering w/ post-clustering reduction
[16]	Timing Driven Clustering followed by Cluster Packing	Label and Cluster	limited amount during clustering
[13]	Two-Level Clustering	Label and Cluster	limited amount during clustering
[60]	Multilevel Circuit Clustering	Label and Cluster	during clustering w/ post-clustering reduction
[39]	Simultaneous Mapping and Clustering	Label and Cluster with Tech. Mapping	during clustering w/ post-clustering reduction
[9]	Simultaneous Placement with Clustering and Duplication	Greedy with Post-Clustering Modification	during placement
[53]	Using Logic Duplication to Improve Performance in FPGAs	Greedy	post-placement
[3]	Timing Optimization of FPGA Placements by Logic Replication	Greedy	post-routing
[17]	DPack	Greedy w/ placement information	none

Table 2.2: Summary of Different Clustering Techniques

2.2.4 Placement

The job of the placer is to assign a unique location in the FPGA to each block or cluster in the circuit. The placement algorithm may try to optimize for delay, routing resource usage or a combination of the two. Placement algorithms for FPGAs generally fall into two categories: simulated annealing and analytical placement. Figure 2.5 shows an example of placing clusters, labelled **a** through **n**, into a 5x5 grid.

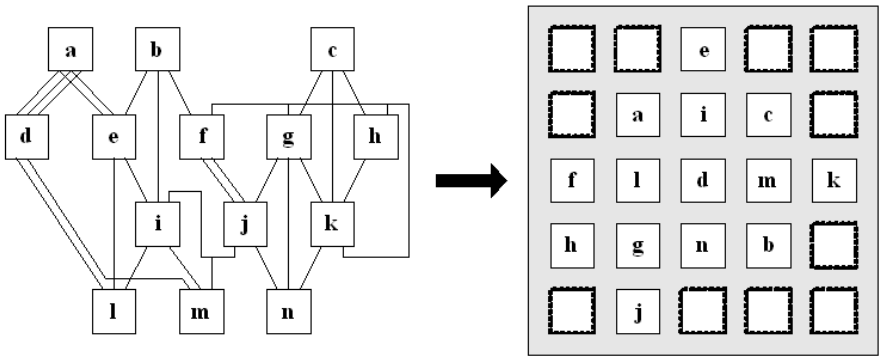

Figure 2.5: Placement Example (from [30])

Simulated annealing was first proposed by [27] as a general optimization algorithm. As it relates to the FPGA placement problem, the idea is to start with a random placement of CLBs, a cost function that acts to assess the metrics of the placement, and some temperature T. For a certain number of iterations, two randomly chosen clusters tentatively swap locations. If the change causes a decrease in the cost function, the change is kept. If the change causes an increase in the cost function, the probability of keeping the change is dependent on the current temperature of the annealer. Detrimental changes may benefit the overall solution by preventing the solution space from getting trapped in local minima. As the algorithm progresses, the temperature is slowly decremented. This causes unfavourable moves to become less likely, thus allowing the solution to find a stable minimum. Examples of simulated annealing tools are TimberWolf [54] for standard cell placements, and Versatile Place & Route (VPR) [5] for FPGAs. The VPR Placement algorithm is further explained

in Section 2.3.1.

Analytical placers represent the placement problem with a system of equations, which are then solved numerically. Force-directed placement [49] is one type of analytical placer that models the system as a series of particles and springs. When applied to the FPGA placement problem, each cluster is analogous to a particle and each net is modelled as a spring with a force constant relative to the criticality of the net. Any given cluster will be under the force of a number of nets. The problem then reduces to solving the system such that each cluster reaches a state of rest, where the forces from all incident nets sum to zero. One significant drawback to these analytical techniques is the overlapping of clusters. A valid solution requires that each cluster have a unique discrete location, therefore the placer must legalize the placement by separating overlapping clusters. Solutions to this problem include repulsive forces for overlapping blocks [49], attractive forces to low density areas [18], and disallowing overlap by forcing one CLB to swap out another [56]. Force-directed placers for VLSI, standard cell and macro-cell designs include [43], [62] and [23], respectively. There has been little work done on force-directed placement for FPGAs. One notable exception is a comparison of placement techniques in [44] where they implemented the force directed placement algorithm described in [56]. The results of [44] show similar performance to the VPR placement tool when allowed similar run times. Force-directed placement is applied to hierarchical FPGAs in [38] by using a force-directed scheme for coarse net-level placement and a similar process for detailed logic cell placement.

Additional placement algorithms exist for FPGAs, such as min-cut partitioning algorithms and other analytical placers. For brevity, a complete description of these algorithms is not included.

2.2.5 Routing

The final step in the FPGA CAD flow is routing all signals that connect to multiple CLBs. This is accomplished by setting programmable switches, buffers, or pass transistors in the switch and connection blocks. In Figure 2.6, CLBs are labelled L, switch blocks are labelled S, and connection

blocks are labelled C. Connections blocks connect routing track signals to CLB inputs and switch blocks connect different routing tracks to each other. The goal of the router may be to either minimize the channel width of the circuit, achieve the lowest critical-path delay or a combination of the two.

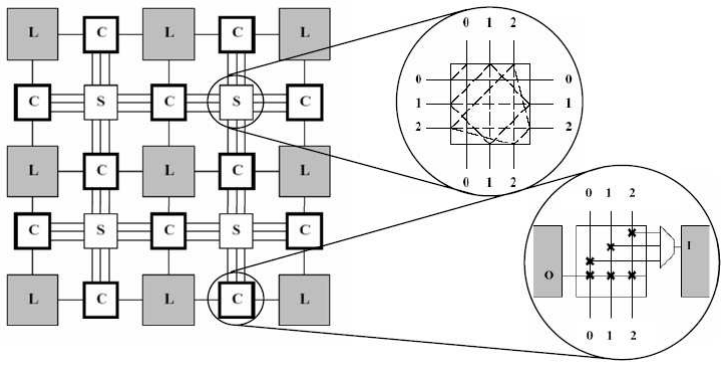

Figure 2.6: Routing Structure (from [22])

The routing problem can potentially be broken into two stages, a global routing stage and a detailed routing stage. The global router chooses the path through the grid for each net and the detailed router assigns the actual wire segments for each net. Examples of algorithms that perform routing in two distinct steps include Course Graph Expansion [8], SEGment Allocator [35], FPR [2] and [36]. Pathfinder [42] and VPR [5] are examples of routers that combine global and detailed routing into single-step routing.

2.3 Previous Work

In the realm of FPGA clustering and placement, there has a been a great deal of research, some of which has been cited in Section 2.2. In this section, a more comprehensive analysis will be presented of algorithms considered to be particularly relevant to the work in this thesis. Considerable attention is paid to the VPR suite, as this not only represents the baseline for comparison, but the simulated annealing engine is employed by the work here.

2.3.1 Versatile Place & Route (VPR)

The 4.30 release of VPR is considered the academic standard for FPGA Clustering, Placement and Routing. It combines a timing-driven clustering tool, T-VPack [41], a simulated annealing placer, T-VPlace [40] and a router based on Pathfinder [42].

T-VPack

As mentioned earlier, T-VPack is a greedy clustering tool with the goal of reducing the number of intercluster nets on the critical path. The first step is to estimate the critical path of the circuit. Assuming logic has a delay of 0.1, intracluster nets have a delay of 0.1 and intercluster nets have a delay of 1.0, T-VPack builds a timing model with all nets initially intercluster. When packing the next cluster *C*, a seed is chosen as the unclustered block driven by the most-critical connection in the circuit. The algorithm continues to add the block with the most-critical connection to cluster *C* until the cluster is full or all inputs are used. In this manner, T-VPack attempts to maximize the number of critical connections made intracluster, thus reducing the expected delay.

T-VPack tends to produce high-quality, efficient clustering results. Depending on the number of inputs per cluster, T-VPack will usually use the fewest possible CLBs ($\lceil total\_BLEs/N \rceil$) and in general has a low critical-path delay. One significant limitation of this approach is the inability to backtrack, or change decisions previously made. Once a block is packed into a particular cluster, it will remain there through the remainder of the clustering phase, and through placement and routing.

Slack Definition

Slack is a measure of how much we may delay the output of a block, before it causes a new critical path to form. Prior to examining the VPR Placement Algorithm, it is important to understand the concept of slack and how it is computed. This is used by both VPR and the timing model of this research to compute a slack estimate.

For a given block *b*, with predecessors $p_1, p_2 \ldots p_n$, each of which produces an output at times

$t_1, t_2 ... t_n$, the completion time of b is defined as:

$$\begin{aligned} t\_completion(b) &= max\{t_1 + delay(p_1, b), t_2 + delay(p_2, b) ... t_m + delay(p_n, b)\} \\ &+ logic\_delay(b) \end{aligned}$$

Where $delay(p_i, b)$ is a function used to describe the delay of routing p_i to b, and $logic\_delay(b)$ is the intrinsic logic delay of node b. Once the completion time of all blocks in the circuit have been computed, we can define the critical-path delay as:

$$t_{crit} = max_{\forall\ j\ \in\ outpads}\{t\_completion(j)\}$$

At this point, the slack for each circuit output, $outpad_i$, can be set as:

$$slack(outpad_i) = t_{crit} - t\_completion(output_i)$$

In terms of connections, slack is defined as the additional amount of delay that can be incurred before the connection becomes critical. Therefore, for a connection from block i to block j:

$$slack(i, j) = (t\_completion(j) - logic\_delay(j) + slack(j)) - t\_completion(i) - delay(i, j)$$

Finally, slack is set for all other blocks in a backwards traversal from outpad to inpad. If block b has successors $s_1, s_2 ... s_m$,

$$\begin{aligned} slack(b) &= min_{\forall successors\_s_k}\{slack(s_k) + t\_completion(s_k) \\ &\quad - logic\_delay(s_k) - delay(b, s_k) - t\_completion(b)\} \end{aligned}$$

If a block has zero slack, it is said to be **critical**.

VPR Placement Algorithm - T-VPlace

The VPR placement algorithm is of particular importance to this work, as the embedded simulated annealing engine is modelled to emulate VPR's simulated annealer. Note that VPR moves entire clusters at a time, not individual blocks. The first step is to randomly place all CLBs and pads. To track the quality of the current placement a cost function is necessary. The cost function takes into account both the aggregate bounding box area and timing cost of all nets. The bounding box cost function is:

$$BB\_cost = \sum_{n=1}^{N_{nets}} q(n) \left[\frac{bb_x(n)}{C_{av,n}(n)} + \frac{bb_y(n)}{C_{av,y}(n)} \right]$$

such that $bb_x(n)$ and $bb_y(n)$ are the bounding box values for net n, $C_{av,n}(n)$ and $C_{av,y}(n)$ are the average channel capacities across n's bounding box (which are both assumed to equal 100) and $q(n)$ is a bounding box correction factor. The correction factor compensates for fact that bounding box estimation underestimates wirelength for high fanout nets.

$q(n) = 2.79 + 0.02616 * (num\_terminals - 50)$

The second cost function attempts to minimize the length of critical nets close together. The timing cost function is a summation over all source-sink pairs (i, j).

$$Timing\_cost = \sum_{(i,j) \subset circuit} delay(i,j) * criticality(i,j)^{criticalityexponent}$$

where the criticality of the connection is defined as:

$$criticality(i,j) = 1 - \frac{slack(i,j)}{t\_crit}$$

The criticality exponent is used to heavily weight critical nets. It is varied from 1.0 to 8.0 through the course of the algorithm.

To evaluate the merit of a move, the total change in cost, ΔC is computed as follows:

$$\Delta C = \lambda \cdot \frac{\Delta Timing\_Cost}{Previous\_Timing\_Cost} + (1-\lambda) \cdot \frac{\Delta BB\_Cost}{Previous\_BB\_Cost}$$

Unless otherwise noted, λ is always 0.5, equally weighting the bounding box and timing cost.

To begin the algorithm, an initial temperature is found by performing a number of moves and setting the temperature high enough that almost all move are accepted. For a given temperature, $10 \cdot (N_{clusters})^{1.33}$ moves are attempted. The temperature is then lowered, with the objective of maintaining a 44% acceptance rate as prescribed in [29] and [59].

The current temperature value, $temp_{current}$ is important when evaluating bad moves. If a swap produces a negative ΔC, the move is considered good, so it is kept. If a swap produces a positive ΔC, the swap is kept only if:

$$e^{\frac{-\Delta C}{temp_{current}}} > rand(0,1)$$

Therefore, the probability of a bad move being accepted is increased if ΔC is small or if $temp_{current}$ is large. Initially, when $temp_{current}$ is large, most swaps are accepted. As the temperature decreases, fewer and fewer bad moves are accepted. This allows the placement to slowly settle to a good solution. The algorithm terminates when $temp_{current} = 0.005 \cdot Cost/N_{nets}$.

While the VPR simulated annealing placer is very proficient at its task, it lacks the ability to adjust the clustering to improve the solution. The pseudocode for the T-VPlace placement algorithm is shown in Figure 2.7.

VPR Routing Algorithm

The VPR Routing algorithm is based on the Pathfinder [42] router, which in turn is modeled after the A* router by [48]. All of these algorithms utilize the Lee maze router algorithm [33], which can find the shortest path between two terminals, provided one exists.

In the Pathfinder algorithm, all nets are routed using the A* router, disregarding the resource usage of other nets. The routing usage is then analyzed, and any resource that exceeds its capacity is assigned a cost. All nets are then re-routed and penalized for using resources that have assigned cost. The cost associated with an overused resources accumulates with successive iterations until it becomes so high a sufficient number of nets avoid it. In this way, overused resources are brought within their capacity limits and the circuit can be routed.

2.3.2 Simultaneous Placement with Clustering and Duplication (SPCD)

SPCD [9] is based on a simulated annealing placement algorithm. The annealing engine is modified to allow incremental changes to the clustering solution and the introduction of duplication after each iteration of the annealer. Starting from a T-VPack clustering solution, the SPCD algorithm permits the annealer to make block-level moves in much the same was as cluster-level moves. After each iteration of the annealer, duplication is performed on nodes residing on the critical path to reduce the critical-path delay. The work cites a 18% critical-path delay improvement over VPR with a cluster size of 4 and wirelength of 1.

2.3.3 Improving Timing-Driven FPGA Packing with Physical Information (DPack)

DPack [17] is a greedy algorithm which incorporates placement information into the clustering step. Using a min-cut partitioning-based placer, a fast placement of blocks is created. This placement information is used to create another term in the clustering tool's cost function. The work cites an 8% reduction in critical-path delay and a 19% reduction in channel width.

2.3.4 iRAC

iRAC [57] is greedy-based clustering tool that is specifically concerned with reducing the routing resource usage of the circuit. In FPGA CAD, iRAC is considered state of the art in terms of achieving the lowest channel width for a circuit. The goal of the algorithm is to make as many low-fanout

nets intracluster as possible. It accomplishes this by choosing seeds with a high number of incident nets and a low number of associated terminals. It then packs clusters in such a way as to encompass as many nets as possible into the cluster. The iRAC tool can reduce the routing channel width by an average of 34% compared to T-VPack, without substantially altering the timing performance.

2.3.5 Using Logic Duplication to Improve Performance in FPGAs

Work by Schabas et al. [53] describes an algorithm that inserts logic duplication after placement to improve the critical-path delay of the circuit. During the clustering phase, an approach similar to T-VPack is employed, but some clusters are left under-full. In the paper, the clustering algorithm leaves a minimum of 4 blocks empty in a cluster of 10, for clusters deemed to be on the critical path. After a simulated-annealing placement of the circuit, the algorithm inserts duplicates along timing-critical paths to reduce the overall circuit delay. The paper cites a 14.1% decrease in critical-path delay for a 20% increase in area. This area increase roughly equates to increasing the grid size by 2 rows and 2 columns.

```
S = RandomPlacement ();
T = InitialTemperature ();
R_limit = InitialR_limit ();
Criticality_Exponent = ComputeNewExponent();

ComputeDelayMatrix();

while (ExitCriterion () == False) {        /* "Outer loop" */

        TimingAnalyze();          /* Perform a timing-analysis and update each connections criticality */
        Previous_Wiring_Cost =Wiring_Cost(S);    /* wire-length minimization normalization term */
        Previous_Timing_Cost = Timing_Cost(S);              /* delay minimization normalization term */

        while (InnerLoopCriterion () == False) {    /*  "Inner loop" */

                S_new = GenerateViaMove (S, R_limit);
                ΔTiming_Cost = Timing_Cost(S_new) - Timing_Cost(S);
                ΔWiring_Cost =Wiring_Cost(S_new) -Wiring_Cost(S);
                ΔC =  λ·(ΔTiming_Cost/Prev_Timing_Cost) +
                                                (1-λ)·(ΔWiring_Cost/Previous_Wiring_Cost); /* new cost fcn */
                if (ΔC < 0) {
                        S = S_new /* Move is good, accept */
                }
                else {
                        r = random (0,1);
                        if (r < e^(-ΔC/T)) {
                                S = S_new;  /* Move is bad, accept anyway */
                        }
                }
        }           /* End "inner loop" */

        T = UpdateTemp ();
        R_limit = UpdateR_limit ();
        Criticality_Exponent = ComputeNewExponent();

}    /* End "outer loop" */
```

Figure 2.7: T-VPlace Pseudocode (from [40])

Chapter 3

Combined Clustering and Placement Algorithm Overview

The algorithm presented in this thesis leverages node duplication and a depth-optimal initial clustering to provide a starting point for a non-greedy, iterative optimization technique using detailed placement and timing information to develop the clustering and placement solutions. The main goal is to improve timing performance at the possible expense of area and runtime. This chapter describes the algorithm at a high level and presents rationale for the approach.

3.1 Algorithm

The clustering and placement steps of the FPGA CAD flow require assigning each pad, LUT or flip-flop to a CLB, and assigning each CLB to a location on the FPGA. The inputs to the combined clustering and placement algorithm are:

- A description of the logic circuit defined by a set of *k*-input LUTs, flip-flops and pads.
- An architectural description of the target FPGA.

The outputs are:

- A netlist describing the clustering solution.
- A file describing the placement solution.

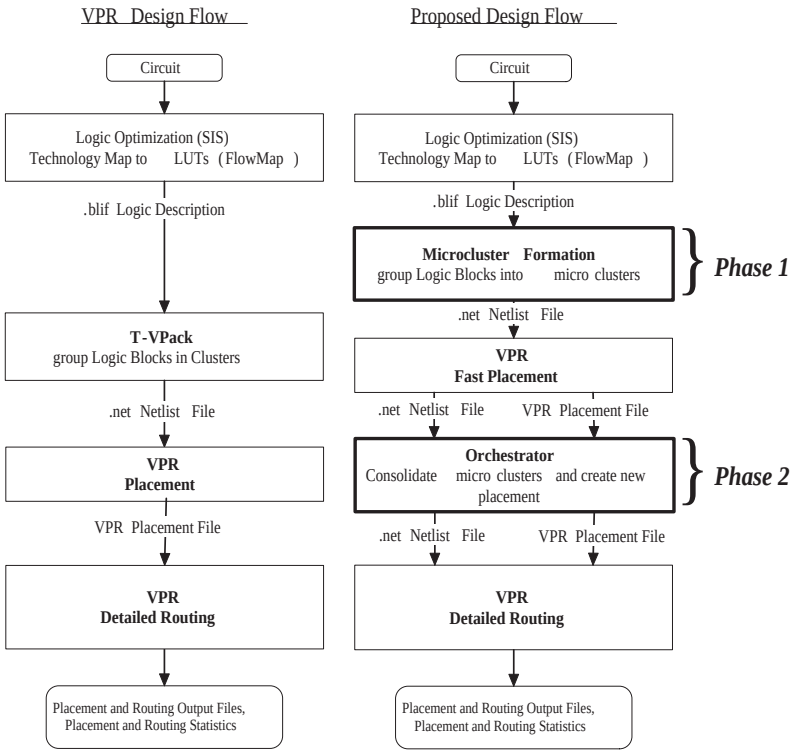

Figure 3.1: VPR and Proposed Design Flows

As shown in Figure 3.1, the proposed design flow is broken into two phases, the Microcluster Formation phase and the Microcluster Compaction with Orchestrator phase. The term microcluster is used to describe a group of logic blocks that will be put into the same CLB, but may still leave room in the CLB for additional microclusters.

3.2 Microcluster Formation

The purpose of the Microcluster Formation stage is to construct microclusters by grouping connected blocks in a depth-controlled manner. Creating a solution with a minimum possible depth requires introducing node duplication into the circuit. This phase is also responsible for managing the amount of duplication in the intermediate solution that is passed to the Microcluster Compaction phase.

To create microclusters, the Lawler Levitt Turner algorithm [32] is used to create a clustering solution with a minimum circuit depth. To create a minimum-depth clustering, the LLT algorithm requires a great deal of duplication. Some amount of duplication is desirable, as it will be used to improve performance. Conversely, too much duplication may limit the ability of the algorithm to meet the area restrictions imposed by the user. Therefore, a Node Duplicate Reduction (NDR) technique is presented to reduce the amount of duplication in the initial clustering solution. Using these two techniques in tandem, an initial clustering solution is created where the amount of duplication is customized to the area constraints of the circuit.

For subsequent steps in the CAD flow, grouping blocks into microclusters provides certain advantages over using individual blocks as basic entities. These advantages include:

- reduced runtime in creating an initial placement as there are fewer entities to place
- a more useful placement model, as microclusters are more representative of a CLB than individual blocks
- a controlled depth that the Microcluster Compaction Phase can use to deliver a critical-path delay improvement over a clustering approach with no depth control

The Microcluster Formation phase produces a set of microclusters, where each microcluster is a group of connected blocks of size $1 \ldots N$, with up to I external inputs. A microcluster is essentially a precursor to a cluster, but conforms to the same restrictions as a cluster. The Microcluster Formation phase is described in detail in Chapter 4.

3.3 Placement

After Microcluster Formation, but before Microcluster Compaction, a placement of the microclusters is created with the VPR simulated annealer using the -fast flag. A fast placement is sufficient as the Orchestrator tool produces similar results with either a full placement or fast placement. Dur-

ing this stage, each CLB is limited to hold only 1 microcluster, regardless of how many blocks it contains. The VPR Placement algorithm is described in Section 2.3.1.

The placement of microclusters provides the Microcluster Compaction phase with a detailed estimate of circuit timing, which is then used to produce the final clustering and placement solution.

3.4 Microcluster Compaction with Orchestrator

The goal of the Microcluster Compaction phase is to reorganize and consolidate the microclusters created in the Microcluster Formation phase to better match CLB capacity. This will reduce the number of CLBs required, improving area efficiency, wirelength and delay. Using the fast placement result, the algorithm builds an accurate timing model of the circuit. Using this timing information, the algorithm iteratively moves microclusters to locations that improve delay and CLB usage.

To move a microcluster, the algorithm analyzes all predecessor and successor microclusters to build a set of source and sink positions. From this, a set of possible move locations are generated that would place the microclusters in a position that is more advantageous from a timing perspective. If multiple move locations are found, they are ordered by how they affect the bounding box cost of the circuit. The algorithm then attempts each move location, in order, until a legal move is found, or until the algorithm reaches the microcluster's current location. Through this technique, the algorithm can compact the placement and iteratively reduce delay on critical or near critical nets and gradually lower the overall critical-path delay.

The Orchestrator algorithm can also try to force a circuit to conform to a given area restriction. If the user provides a maximum grid size, the algorithm will reduce superfluous duplication and CLB usage while moving microclusters. This allows the designer to trade-off area for performance in a controlled fashion.

The final output of the Orchestrator algorithm is a valid clustering and placement solution that can be routed by VPR.

Chapter 4

Phase 1: Microcluster Formation

The Microcluster Formation routine groups individual blocks into microclusters. Using the Lawler Levitt Turner (LLT) algorithm, a depth-optimal clustering solution is formed. To reduce the total amount of node duplication in the clustering solution, a Node Duplicate Reduction (NDR) algorithm is used. The NDR algorithm takes advantage of depth slack to reduce duplicates. Finally, if further duplication reduction is required, the NDR algorithm can increase the depth to facilitate a greater reduction in duplication.

4.1 Introduction and Motivation

To goal of the Microcluster Formation phase is to create a set of highly cohesive microclusters that provide good opportunity for subsequent steps in the algorithm to achieve low delay. In forming microclusters, the algorithm does remove some freedom in altering the clustering solution during later steps. As microclusters usually persist for the duration of the CAD flow, there is little opportunity to undo the initial clustering. Therefore, it is important to create microclusters that promote low delay throughout the circuit.

When forming microclusters, to facilitate lower delay in the final routed solution, the concept of cluster depth (or microcluster depth, depending on the context) is important. Microcluster depth (or hereafter referred to simply as depth) is the worst-case number of microclusters traversed along any combinational path through the circuit. Circuit depth directly influences critical-path delay as the depth indicates the number of intercluster nets that must be traversed in the routed solution. As

intercluster nets have larger delay compared to intracluster nets and logic delay, they tend to dictate the delay on the critical path. Therefore, it is in the best interest of the algorithm to minimize circuit depth during the Microcluster Formation phase.

Another important consideration when forming microclusters is node duplication. Node duplication refers to creating a block with the same inputs and same functionality of another block, but in a different microcluster. As will be shown, node duplication is used to reduce depth and critical-path delay, but this comes at the expense of increased area. The objective for the duplication is to provide more opportunity for critical-path delay reduction in subsequent phases by strategically placing duplicates.

For a combined clustering and placement approach, a placement of individual blocks would require significant runtime for large circuits. Even if such a placement were possible, without any clustering information, the timing information available would be of little merit. To allow the program to create a meaningful initial placement, the Microcluster Formation phase must create a set of microclusters. During Phase 2 of the algorithm, groups of microclusters are consolidated and placed to form the final clusters. Since microclusters will not be broken apart later, they provide the opportunity to reduce the amount of effort required during placement. If left as individual blocks, even more runtime would be needed due to the increased amount of flexibility.

This chapter begins with a complete description of the Microcluster Formation algorithm, broken into 3 steps labelled 1A, 1B and 1C. Step 1C is described in Section 4.5.1 as it is quite long. Section 4.6 explains how NDR algorithm is used to further reduce duplication by relaxing circuit depth. Finally, the results of the Microcluster Formation Phase are presented and analyzed in Section 4.7.

4.2 Algorithm Description

As shown in Figure 4.1, the Microcluster Formation phase is broken into 3 steps: 1A - Handling of Sequential Circuits, 1B - LLT algorithm, and 1C - NDR algorithm. In this section, each of the these

steps is explained in detail.

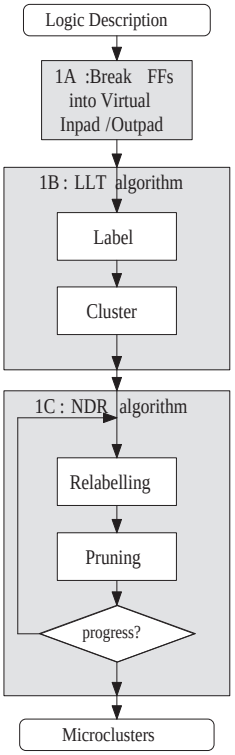

Figure 4.1: Microcluster Formation Flow

4.3 Step 1A - Handling of Sequential Circuits

When performing clustering and placement, special consideration must be given to sequential circuit elements, namely blocks which use a flip-flop. As a flip-flop is a delimiter for the critical path, any output of a flip-flop starts a new path through the circuit, and any input to a flip-flop finishes a path. In this work, a flip-flop is treated as two elements, a virtual input and a virtual output. After Microcluster Formation is completed, the flip-flops are re-inserted to the circuit. Depending on the positioning of the flip-flops, Phase 2 will either assign each flip-flop its own BLE, or match it with a LUT and combine the two elements into a single BLE.

4.4 Step 1B - Lawler Levitt Turner Algorithm

The Lawler Levitt Turner (LLT) algorithm [32] was originally devised as a packing algorithm for digital networks, but has found relevance with FPGAs. This relevance is first due to the unit delay model, which to a first order approximation, can be applied to FPGAs. This delay model assumes:

1. all BLEs have zero delay,
2. intracluster connections have zero delay, and
3. intercluster connections have one unit delay.

This algorithm produces a clustering solution with the lowest possible maximum depth, which generally translates into a low critical-path delay. However, the algorithm has a drawback, which we will try to fix later in Step 1C: an excessive amount of duplication is required.

The LLT algorithm is comprised of two phases, the labelling phase and the clustering phase. In the labelling phase, each node n is assigned a label, $l(n)$ representing the worst-case delay to that node, under the unit delay model. The labelling phase proceeds as follows:

1. Label all primary/virtual inputs 0.
2. Find an unlabelled node n, such that all predecessors of n have been labelled. Let k be the largest label of any predecessor node, and let the set P_k be all ancestors of n with label k.

$$if\ |P_k|+1 > N,\ l(n) = k+1,\ otherwise,\ l(n) = k$$

3. Continue until all nodes are labelled. In this manner, for any node n of label k, the number of predecessors of n with label k, plus n itself, is always less than or equal to N (maximum blocks per cluster).

The clustering phase uses the labels to create a depth-optimal clustering solution. During the clustering phase, inpads and outpads are ignored (as they are not assigned to clusters) and latches are treated as virtual outputs. The clustering stage proceeds as follows:

1. Find a node *r* with label *k*, such that *r* is a virtual output or all successors of *r* have labels greater than *k*. This node *r* is referred to as a **microcluster root**.
2. Form a microcluster which includes *r* and all predecessors of *r* with label *k*.
3. If unclustered nodes still exist, return to step 1.

At the conclusion of the clustering phase, all nodes (except inpads and outpads) should be assigned a microcluster. Instances may also exist where a single block exists in a number of microclusters. This is referred to as **node duplication**. In Figure 4.2, two microclusters exist, one rooted at node D, the other rooted at node E. As node B is a predecessor of both D and E, and shares the same label, it must exist in both microclusters. Therefore, a block will exist in two microclusters, rooted and D and E, with equivalent logic and the same set of inputs.

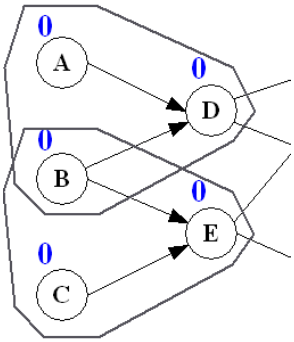

Figure 4.2: Node Duplication Example

Node duplication is not only essential in producing a depth-optimal clustering solution, it can also have a large effect on delay, area and routing resource usage. If, in the previous example, microclusters D and E were on different sides of the chip after placement, without duplication the

signal originating at B would have to be routed to both microclusters. With duplication, we are no longer required to route the signal originating at B to two different microclusters, as a copy of B will exist in both microclusters. This can reduce delay, as D and E no longer incur the large delay of an intercluster wire, but the comparatively smaller delay of an intracluster wire.

The effect of node duplication on routing resource usage is difficult to quantify. On one hand, the output of the duplicated block now requires no general routing resources. On the other hand, by duplicating a node, each input net to that node may have to route an additional terminal.

Finally, node duplication has an affect on the area usage of the circuit. Excessive duplication can cause a substantial increase in the logic usage of the circuit. Compounding this problem is the fact that the LLT algorithm makes no attempt to completely fill microclusters. This fragmentation also contributes to area overhead. The problem of under-utilized CLBs will be addressed in Chapter 5.

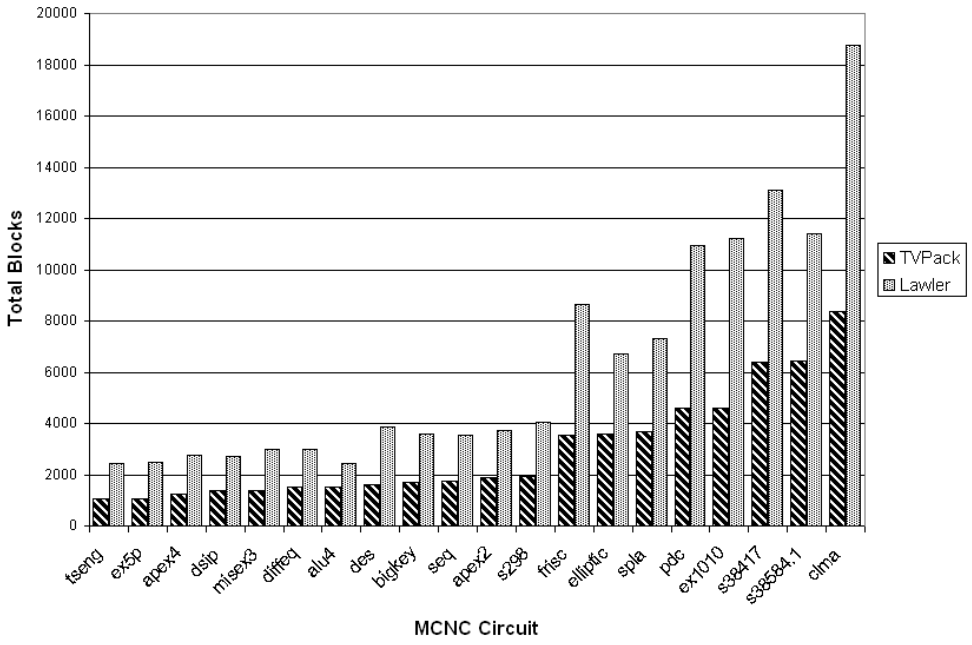

Figure 4.3: Motivation for NDR

4.5 Step 1C - Node Duplicate Reduction Algorithm

The LLT algorithm in the previous section produces a minimum-depth clustering. Figure 4.3 shows there is a 112% increase in block usage after LLT due to node duplication. LLT minimizes depth for *all* paths in the circuit, so a certain amount of slack exists on all non-critical paths. The Node Duplicate Reduction algorithm attempts to convert this slack into a reduction in node duplication and an improvement in the number of nets enclosed within a microcluster.

The driving observation here is that non-critical paths may be delayed, provided it does not cause that particular path to become critical. This allows the algorithm to delay a block's result (i.e., its label), forcing it into a different microcluster(s). Through this technique, NDR attempts to find the position for all nodes which will minimize node duplication, maximize net absorption and retain a predetermined maximum depth. A complete description of the NDR algorithm is presented in Section 4.5.1.

To further reduce duplication, we may increase the maximum depth of the circuit to provide all nodes with some degree of slack. This allows the algorithm a greater range of possible relabels. This technique is further explored in Section 4.6.

4.5.1 NDR Algorithm Description

The NDR algorithm proceeds by iteratively applying to following steps:

1. Determine the set of labels to which a block may graduate.
2. Score each of these new labels in terms of how it will affect node duplication, intercluster nets and input sharing.
3. Update the labelling and re-clustering the circuit.
4. Prune microclusters which exceed cluster size or input limits.

After each iteration, the algorithm assesses if any progress has been made. Small oscillations in the number of blocks used can occur between iterations, therefore progress is made only if the algorithm has reduced the overall block usage to a new low. If no progress is made after a defined number of iterations (default=10), the program terminates successfully.

Using this process, the NDR algorithm can achieve a predetermined maximum depth with significantly less node duplication, and hence area overhead, than the LLT algorithm. Each of these step is now described in detail.

4.5.2 Relabelling

The first step is to determine the applicable label increases for all clusterable nodes in the circuit. If the current node under consideration is node a, with label $l(a)$, having successors $s_1, s_2 \ldots s_m$, then the maximum label possible for a is $l_{max}(a) = min\{l(s_1), l(s_2) \ldots l(s_m)\}$. In this manner, no node may graduate to a higher label than any of its successors. Intuitively, $l_{max}(a) - l(a)$ is representative of the slack of node a. Once an l_{max} has been associated with all nodes, the next step is to score each possible move in terms of how it will affect the clustering solution. For each node, an influence array is created with indices from $l(a)$ to $l_{max}(a)$: $I_{l(a)} \ldots I_{lmax(a)}$. The largest influence value will be used to determine the new label. The influence value for a given relabelling has two components, a weighted score which depends on how the change will affect duplication, net usage and input sharing, and an element of randomness.

The calculated portion for the influence value is a weighted sum of the change in node duplication ($\Delta duplication$), intercluster nets ($\Delta intercluster$), and shared inputs ($\Delta shared\_inputs$). To determine $\Delta duplication$ for relabelling block a to $l(a)+i$ we must examine all of a's predecessors and successors at the same label.

$$\Delta duplication = max\{0, |succ\_clusters\ at\ l(a)+i| - 1\} - |duplicated\_preds\ at\ l(a)|$$

Intuitively, $\Delta duplication$ counts the net change in the number of duplicates. It is larger when re-

labelling a to $l(a)+i$ forces a to be duplicated for existence in multiple successor microclusters, and smaller when relabelling a to $l(a)+i$ causes duplicated predecessors in the current microcluster rooted at a to be dissolved. These dissolved duplicates can be replaced by the output of some other microcluster at a lower label than $l(a)+i$. For example, if block b_1 existed in microcluster A and A was dissolved by relabelling a, a different copy of b_1 in another cluster, b_2, could now supply the successors of b_1.

Similarly, $\Delta intercluster$ counts how many intercluster nets are made intracluster during a relabelling. If a is relabelled to $l(a)+i$, it is increased by the number of intracluster nets to a predecessor that are made intercluster, and it is decreased for each intercluster net to a successor made intracluster.

$$\Delta intercluster = |predecessors\ at\ l(a)| - |succ\_clusters\ at\ l(a)+i|$$

Next, $\Delta shared\_inputs$ is a count of how many blocks share the same input to a cluster. $\Delta shared\_inputs$ is found by comparing the distinct inputs to each successor microcluster $C_{succ\_a}(j)$ at $l(a)+i$ with the inputs to a itself. Any net n which exists in both sets will reduce its total number of sinks by one if a is promoted to $l(a)+i$.

$$\Delta shared\_inputs = \sum_{C_{succ\_a}(j)\ at\ l(a)+i} |distinct\ inputs\ to\ C_{succ\_a}(j)| \cap |inputs\ to\ a|$$

The final element to the influence value calculation is the weighting factors $\gamma_1, \gamma_2, \gamma_3$, which can be adjusted by the user. The final influence value is calculated as:

$$I_{l(a)+i} = -\gamma_1 \cdot \Delta duplication - \gamma_2 \cdot \Delta intercluster + \gamma_3 \cdot \Delta shared\_inputs + \rho$$

Note that negative values for $\Delta duplication$ or $\Delta intercluster$ are good, since we want to promote a reduction in node duplication and intercluster nets. Through empirical tests, weighting factors

are selected as $[\gamma_1, \gamma_2, \gamma_3] = [100, 4, 8]$. These values can be changed to reflect a higher priority for certain metrics.

The random element is an integer, $\rho \in [0, TEMP]$, where $TEMP$ is a value that is decremented by $TEMP\_STEP$ after each iteration of the algorithm. As the chief factor in the influence value is the duplication term, the initial temperature value is chosen to be proportional to post-LLT node duplication:

$$initial\ TEMP = (avg\_duplicates\_per\_node)^2$$

$$TEMP\_STEP = avg\_duplicates\_per\_node$$

The ρ term causes nodes to promote to labels which may not be the most effective during the initial iterations, but can prevent the solution from settling to a local minima. In the next section it is shown that a node relabelling may be undone. Therefore, a degree of uncertainty can combat cycles from forming in the relabelling process by having nodes attempt to relabel to different values when multiple options exist.

4.5.3 Pruning to Correct CLB Violations

After the relabelling phase, the possibility exists that some microclusters have greater than N blocks or I distinct inputs. Therefore, NDR traverses through each microcluster in a backwards breadth first order starting at the outputs and prunes blocks from illegal microclusters until all constraints are met. Pruning begins at the outputs, so that if a pruned node causes a predecessor microcluster to overfill, that microcluster will be reached later and pruned accordingly.

For a given microcluster, each valid candidate c is assigned a score, $P(c)$, where a greater $P(c)$ translates into a higher probability of the node being pruned from the current microcluster. A valid candidate is any leaf node whose label may be reduced by one without becoming less than

its original label assigned by the LLT algorithm. We restrict all nodes to remain at or above their original LLT label to avoid instances where the algorithm would be forced to demote a node below label zero.

Pruning scores are determined in a similar manner as influence scores: $\Delta intercluster$ and $\Delta duplication$ are found for relabelling c to $l(c)-1$, and the pruning score is calculated as:

$$P(c) = -\gamma_1 \cdot \Delta duplication - \gamma_2 \cdot \Delta intercluster$$

Note that pruning ignores input sharing and declines the random element for simplicity. The largest $P(c)_i$ will decide which node reduces its label by one.

4.5.4 NDR Algorithm Results

Using this technique, node duplication can be reduced by an average of 40%, compared to the original LLT algorithm. Figure 4.4 shows the block usage results for T-VPack, the LLT algorithm and the NDR algorithm. While the post-duplication reduction results still have a 39% block overhead from T-VPack, it is much less than the LLT algorithm.

4.6 Additional Duplicate Reduction Through Depth Relaxation

The NDR algorithm trades off slack on non-critical paths for a reduction in block usage. A way to further decrease duplication is to increase the maximum depth of the circuit, thereby creating some degree of slack along all paths. This technique can be especially useful when a tight area restriction is imposed, since reducing area usage during the initial clustering phase may be less detrimental than reducing area during a subsequent phase.

Duplication limiting is implemented as follows:

- The user defines a duplication limit, $duplication\_limit$, relative to the total number of original blocks in the circuit. Therefore, if $duplication\_limit = 20\%$, the algorithm is permitted, at

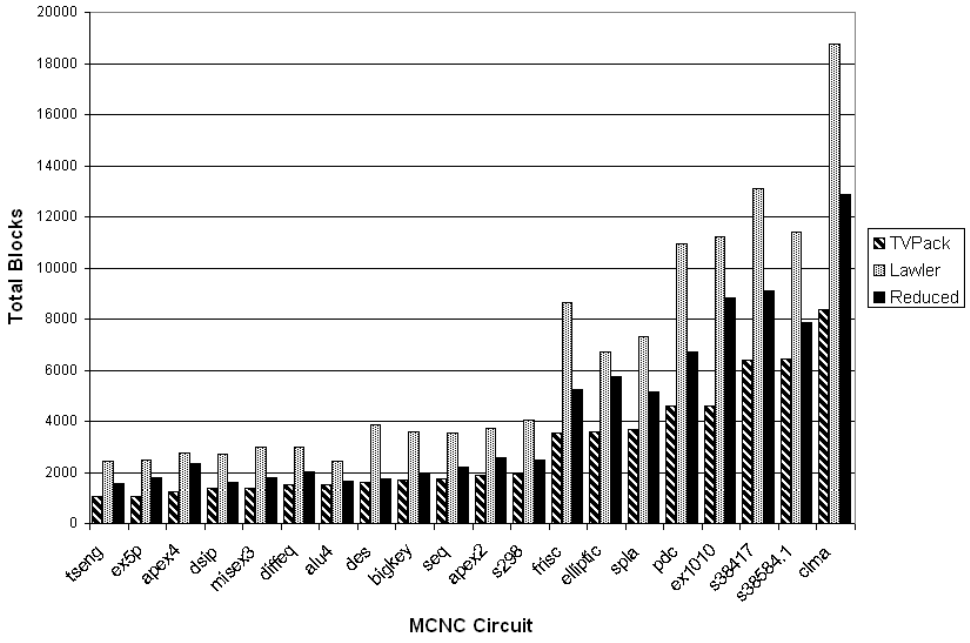

Figure 4.4: Reduction of Blocks After Using NDR

most, a 20% increase in block count.

- After each iteration of the program, the number of blocks used in the circuit is counted. If the block usage is within the predefined limit, the program terminates successfully.

- If the algorithm's progress count expires without reaching the pre-defined duplication limit, the depth of the circuit is increased by 1. The algorithm then resets the progress count and continues running with the new maximum depth. A limit (default=5) is set on the maximum depth increase that is allowed. When this limit is reached without meeting the duplication limit, the algorithm terminates and produces a netlist for the current clustering solution.

To determine the effect this technique has on timing performance and area usage, a comparison was performed with a number of different settings. The results are shown in Figure 4.5. CLB usage and timing results are post-routing values, where T-VPack uses the VPR placement engine and all other tests use the Orchestrator tool presented in Chapter 5. All results are geometric means across

the largest 20 MCNC benchmarks. The different clustering settings shown in the figure are:

- T-VPack, with VPR placement
- LLT clustering with Orchestrator placement and compaction
- LLT with a single pass NDR (Section 4.5.1) and Orchestrator placement and compaction (no duplication limiting)
- LLT with multi-pass NDR using a duplication limit of: 5%, 10%, 20%, 30%, 50%, and 70%

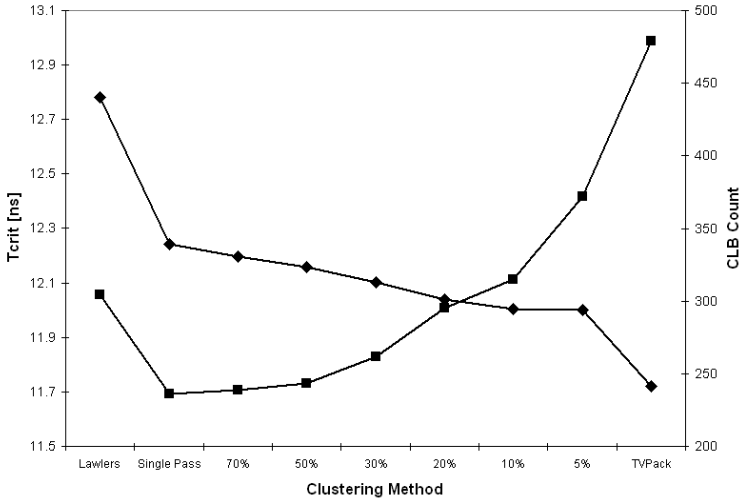

Figure 4.5: Duplicate Reduction Results

Figure 4.5 shows the timing results as squares and the CLB usage as diamonds. These results show that the Additional Duplicate Reduction technique is effective at reducing the CLB count, but at the expense of critical-path delay. Timing results also show that, with the exception of the LLT clustering, more duplication results in better timing performance.

Since the multi-pass NDR technique sacrifices the depth advantage established by the LLT algorithm to reduce node duplication, it is important to understand how much of the depth advantage is exhausted. Table 4.6 shows the depth increase for different duplication limits, averaged across

the 20 largest MCNC benchmarks. Note that in some cases the depth increase limit of 5 is reached and the duplication limit is ultimately not met.

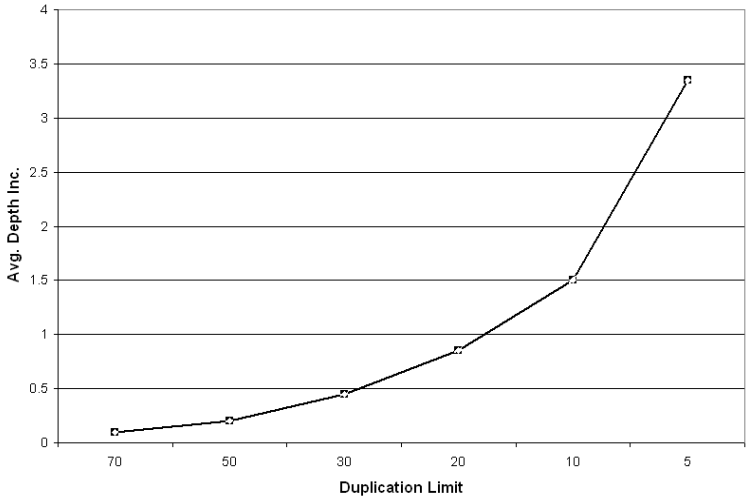

Figure 4.6: Depth Increase vs. Duplication Limit

4.7 Analysis and Results

The analysis and results presented in this section are meant to demonstrate to the reader that the Microcluster Formation phase provides a valuable contribution to the clustering and placement algorithm, but that further work is required.

4.7.1 Node Duplicate Reduction Results

In this section, results are provided for the Lawler Levitt Turner Algorithm described in Section 4.4 and the Node Duplicate Reduction Algorithm described in Section 4.5. The purpose of presenting these results is to demonstrate the merit of the Microcluster Formation phase.

To provide a fair comparison of each algorithm, the clustering solutions from each algorithm are placed and routed with VPR, in the same manner baseline results are produced with T-VPack. For all tests the standard architecture, as described in Section 2.1 is used, with critical-path delay found

using a fixed channel width of 100. All results are a geometric mean across the 20 largest MCNC benchmarks. Table 4.1 presents results for T-VPack, the LLT algorithm and LLT with a single pass of the Node Duplicate Reduction algorithm.

	Tcrit	**# Blocks**	**# CLBs**	**Channel**
TVPack	12.99 ns	2388.1 blocks	241.6 CLBs	43.6 tracks
Lawler	1.06	2.11	5.22	0.40
NDR	0.99	1.39	3.24	0.52

Table 4.1: LLT/NDR Results Normalized to T-VPack

The results show that the LLT algorithm achieves an average critical-path delay which is 6% worse than T-VPack. This is likely because of large amounts of node duplication and high fragmentation. The NDR results illustrate how effective the NDR algorithm is at reducing duplication, and that this results in a noticeable performance improvement. Even with a 3x increase in CLB count, the NDR timing performance is at par with T-VPack.

The final column shows that, in general, the channel width is reduced in the Lawler and NDR solutions. This is primarily due to spreading smaller clusters outs across a larger grid size. A larger grid has more routing channels overall, therefore the routing usage is spread out over more resources.

4.7.2 Analysis of Microcluster Formation Results

To understand the LLT/NDR results better, it will be shown that there are two competing factors influencing the performance of the routed solution. The first is the depth of the circuit, the second is the grid size.

The depth of the circuit is important because it directly affects the number of intercluster nets on the critical path. As the delay of intercluster nets is more substantial than logic delay or intracluster net delay, in most circumstances it the primary factor in the delay of the critical path. Table 4.2 provides depth information for each of the 20 largest MCNC benchmarks. Lawler Depth specifies the lowest achievable clustered depth of the circuit under the architecture constraints. Actual Critical

Path Depth is calculated by the VPR Router and it specifies the depth of the critical path. It should be noted that the final cluster depth may be less than the Lawler Depth if the critical path is along a route with larger interconnect delays.

	Lawler Depth/	**Actual Critical Path Depth (Post-Routing)**	
File	**NDR Single Pass Depth**	*TVPack*	*NDR Single Pass*
alu4	5	5	5
apex2	5	6	5
apex4	4	6	4
bigkey	3	3	2
clma	7	10	6
des	4	7	4
diffeq	6	5	5
dsip	3	3	2
elliptic	5	8	4
ex1010	5	7	5
ex5p	4	7	3
frisc	8	16	7
misex3	4	5	4
pdc	5	9	5
s298	8	14	7
s38417	6	8	4
s38584.1	5	9	4
seq	4	6	4
spla	5	7	5
tseng	5	8	4
Geomean:	4.88	6.01	4.25

Table 4.2: Depth Analysis of Benchmarks

Overall, the critical-path depth of the T-VPack solution is 30% larger than the NDR clustering solution. It is interesting to note that for some circuits the routed depth of T-VPack is particularly high. For benchmarks elliptic, frisc, s298, s38417, and tseng, the T-VPack clustering solution has a depth at least twice that of optimal. Conversely, T-VPack does well in terms of depth for a number of circuits, such as alu4, bigkey and diffeq. The correlation between depth reduction and critical-path delay reduction is shown later in Section 6.2.

If depth were the only consideration for performance, the LLT algorithm would be substantially

better compared to T-VPack than what is presented in Table 4.1. The major downside of the LLT algorithm is the amount of area required due to node duplication and fragmentation. This affects the critical-path delay of the circuit by increasing the average net length.

For a larger grid, it is reasonable to assume that a net, on average, must travel farther to connect two terminals. This will require more delay, and therefore, the delay through the circuit will increase. Figure 4.7 shows a plot of grid size versus average net length across the 20 largest MCNC benchmarks. The average net length is produced by the VPR Router and specifies the average number of wire segments (of length 4) used across all nets.

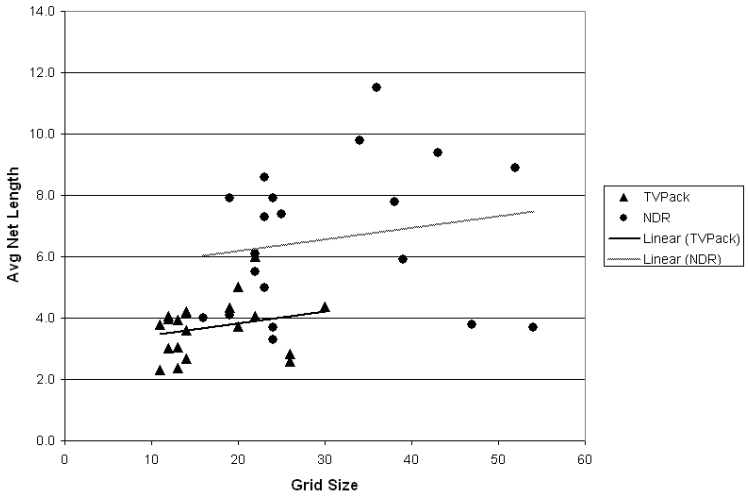

Figure 4.7: Grid Size - Avg Net Length Relationship

The trendlines in Figure 4.7 show a weak correlation between grid size and average net length for T-VPack and the NDR algorithm. Although the correlation is low, it does show that small circuits have low average net lengths, and most large circuits have high average net lengths.

The previous analysis shows that the NDR algorithm has an advantage over the greedy approach in terms of depth. This advantage is counteracted by the increase in area usage of the NDR algorithm which leads to an increase in the average net length. This makes it extremely important to compact the microclusters into a smaller grid.

4.8 Summary

This chapter has described how the Microcluster Formation Phase creates an intermediate clustering solution that:

- group blocks into entities (microclusters) that are large enough to ease subsequent placement steps
- introduces a controlled amount of duplication to improve critical-path delay
- has a controlled depth, typically being the minimum possible depth, unless additional node duplication is required

An analysis of the NDR algorithm results show that a performance advantage is gained by having a lower circuit depth, which results in fewer intracluster nets in the final solution. This advantage is mitigated by the area increase required by the solution, which results in an increase in average net length. This implies that subsequent phases of the algorithm should attempt to reduce the area usage of the circuit without altering the depth. This provides the motivation for the next phase of the algorithm, Microcluster Compaction with Orchestrator. In Appendix A, we present a breakdown of microcluster sizes for the interested reader.

Chapter 5

Phase 2: Microcluster Compaction with Orchestrator

After the Microcluster Formation Phase is complete, the next portion of the algorithm further consolidates the microclusters into CLB-sized clusters to reduce the number of CLBs. This should result in a grid and net length reduction to improve critical-path delay further. For simplicity, this portion of the tool is called Orchestrator. Figure 3.1 highlights where in the CAD flow the Orchestrator tool resides.

5.1 Introduction and Motivation

As was shown in Section 4.7, Phase 1 produces a set of microclusters with an optimal depth, but high fragmentation (under-filled CLBs) and duplication. The goal of Phase 2 is to reorganize microclusters to reduce fragmentation and CLB count. It will also attempt to improve critical-path delay by considering detailed placement and timing information while finishing the clustering.

The primary activity of the Orchestrator tool is to move microclusters to improve the timing and area efficiency. With clustering and placement information available, the algorithm can build an accurate timing model of the circuit. With this timing information, the algorithm iteratively moves microclusters to reduce the critical-path delay and consolidate microclusters. The final result of the Orchestrator algorithm is a valid clustering and placement solution that can be routed by VPR.

When an area restriction is given, the Orchestrator tool strives to meet this restriction in a way

that is least detrimental to timing performance. The Orchestrator uses techniques to reduce node duplication and vacate CLBs to gradually reduce area usage until the restriction is met. The area restriction imposed by the user carries a higher precedence than critical-path delay, so increases in the critical path, though undesirable, are tolerated.

The main focus of this chapter is Section 5.2, which explains the Orchestrator algorithm. After the algorithm description, an analysis of the results is presented, and a summary of the Orchestrator algorithm concludes the chapter.

5.2 Algorithm Description

The Orchestrator tool proceeds in an iterative fashion, with two distinct stages. The first stage relocates microclusters with the sole objective of reducing the critical path of the circuit. The second stage is invoked if the area constraints are not met. This stage works to reduce area usage by removing duplication and relocating the set of microclusters to fit into fewer CLBs.

During each iteration of the algorithm, Orchestrator attempts to move each microcluster. A microcluster may move to a location which is already occupied, provided the aggregate of all microclusters at that location still meet the CLB constraints of the architecture ($\leq N$ blocks, $\leq I$ inputs). In this chapter, location and CLB are used interchangeably to represent an (x,y) location that coincides with a CLB in the final solution. At the completion of the Orchestrator algorithm, all microclusters that reside at the same location are combined in a single cluster. Through this consolidation process, the Orchestrator tool has the ability to alter the clustering solution.

To improve the timing of the initial solution, the Orchestrator tool relocates microclusters to reduce the critical-path delay of the circuit. In relocating a microcluster, the algorithm analyzes predecessors and successors of the microcluster to create a set of possible move locations. The set of possible move locations is chosen such that moving the microcluster will not increase the critical-path delay of the circuit, and may even reduce it. Once a set of possible move locations has been established, the algorithm will either move or swap the microcluster, provided a gain in

performance or bounding box cost is achieved.

5.2.1 Description of Inputs

The Orchestrator tool takes a number of inputs:

1. A BLIF description of the circuit, required to build a list of all blocks and nets in the circuit.
2. A VPR-style Netlist (.net) which describes the initial clustering solution found in the previous stage.
3. A VPR-style Placement (.p) showing the location of all microclusters and pads in the circuit. It should be noted that if no placement file is specified, the Orchestrator tool will use its embedded simulated annealing engine to perform a fast placement equivalent to VPR -fast.
4. A VPR-style Architecture file (.arch) that is used to build a timing model of the FPGA.
5. A grid size which the algorithm attempts to fit the circuit to. If no grid size is given, the algorithm assumes that no area restrictions are imposed.

5.2.2 Orchestrator Preliminary Operations

Once the circuit description has been read in, a number of preliminary tasks must be performed. These include:

1. Remove all unused elements in the circuit (blocks, flip-flops or inpads without any successors).
2. Pack blocks and flip-flops together where possible. This requires finding all blocks that have a single flip-flop successor, and combining the two elements into a single BLE. The resultant BLE will reside where the block originally existed. We will continue to refer to these BLEs as blocks when we are discussing the netlist, and as BLEs when discussing the FPGA architecture.

The final step before the main operation of the algorithm is to build the timing model of the circuit. This is described in detail in Section 5.2.4.

5.2.3 Orchestrator Operational Overview

Figure 5.1 shows a high-level flow chart of the Orchestrator algorithm.

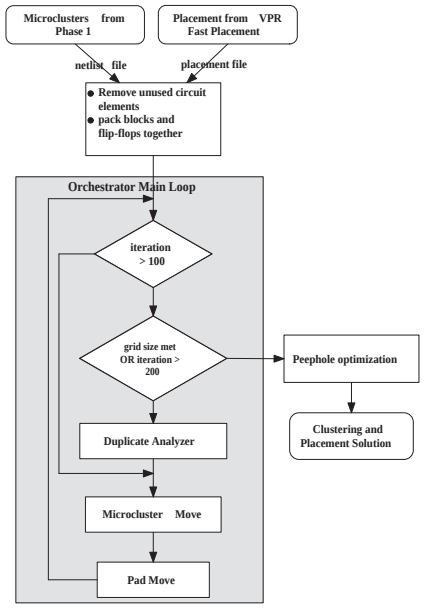

Figure 5.1: Orchestrator Flow Chart

The main loop of the Orchestrator algorithm is responsible for moving microclusters and pads, removing duplicates and readjusting the grid size. The objective of the main loop is to reduce the grid size to that specified by the user while maintaining a low critical-path delay. The main loop is described in Section 5.2.5.

Once the main loop of the program has completed, one final ”peephole optimization” is performed on the duplicates to improve routability. Using the Duplicate Analyzer (see Section 5.2.6), each sink selects the closest duplicate as its source, provided this does not increase its arrival time. The program is forced to keep all duplicates. However, each sink chooses the duplicate that both meets its timing requirements and has the smallest Manhattan distance. While this may increase de-

lay on non-critical paths, it does not affect the critical-path delay because sinks continue to enforce their timing requirements. This improves the routing resource usage of the circuit. The average channel width across the 20 largest MCNC benchmarks is 43.74 without the final adjustment, and 41.46 with it. The maximum channel width is 100 without the final adjustment, compared to only 78 with it.

After the peephole optimization, the program merges all microclusters at a given (x,y) location into a single cluster. The resulting cluster will obey all constraints assigned to a CLB (LUTs, inputs, etc.).

To allow the circuit to be routed, the clustering description (.net) and a placement file (.p) of the solution are output. If requested, the program will also produce a BLIF file representing the final logic description of the circuit. This can be useful when performing formal verification to ensure the final circuit is logically equivalent to the original. Finally, the program completes by reporting statistics on the clustering/placement solution.

5.2.4 Orchestrator Timing Model and Timing Graph

The Orchestrator timing model is similar to the VPR timing model in most respects, except for a simplified wire delay estimation routine. When a net delay is requested, the timing model determines the minimum number of wire segments required, multiplies the number of wire segments by the wire segment delay, and finally, adds the delays associated with connection blocks and CLB internal routing. The wire segment delay is calculated using an Elmore delay model [19].

Using this model, a timing graph is constructed for a circuit in order to compute:

- the arrival time of a signal at a node,
- the completion time of a node (when the node produces an output) or the arrival time of a node (time at which all inputs to the node have arrived),
- the delay incurred along a net between any two terminals, and

- the slack of any node or net as described in Section 2.3.1.

Generally, the timing graph keeps an up-to-date value for all block completion times and slack values, and computes net delays and net slack estimates as required. Figure 5.2 shows a flow chart explaining the process by which the timing model is updated after a block is moved. When multiple blocks are moved, all arrival times are updated as shown in the left of the figure, then all slack estimates are updated as shown in the right of the figure.

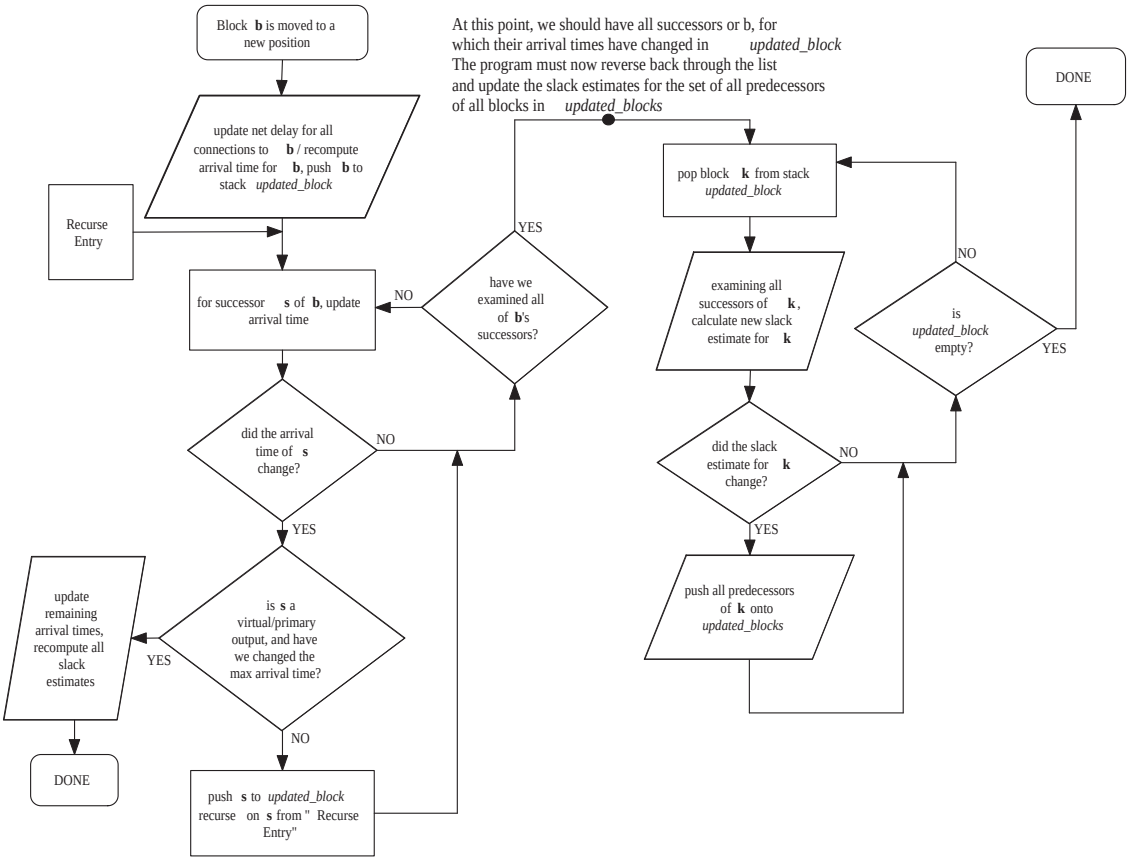

Figure 5.2: Flow Chart of Timing Graph Update for a Block Move

5.2.5 Orchestrator Main Operation

This portion of the thesis describes the operation of the main loop in the Orchestrator program. The main loop has two distinct stages: *Reorganize* and *Reduction*. The Reorganize stage re-orders the microclusters with no emphasis paid to reducing the area/CLB usage; its primary focus is to reduce the critical-path delay of the circuit. The Reduction stage is executed after the circuit has reached a steady-state during the Reorganize stage, but has still not met the grid size requirement. The Reduction stage actively removes duplicates and attempts to reduce the number of CLBs used, so that the circuit will fit within the specified grid.

The default behaviour is for the Reorganize stage to run for 100 iterations, and if required, the Reduction stage to run for up to 100 further iterations. In most circuits, a steady-state is reached in the Reorganize stage prior to 100 iterations, but as run-time is not a major concern at this time, we use 100 iterations as a safe margin to ensure the Reorganize stage has completed. The default number of iterations can be changed by providing the program with a different value at run-time, but for simplicity, this document will assume default values. If run-time was a concern, the Reorganize stage could conclude after a number of iterations without the algorithm making any progress in terms of area and delay.

1. If the main loop is in the Reduction phase ($iterations \geq 100$), and the grid size is still not met, do the following two substeps:

   (a) Perform Duplicate Reduction as described in Section 5.2.6. The slack steal parameter is computed as $slack\_steal = \frac{iterations-100}{100}$. The duplicate reduction step will remove all duplicates that it deems superfluous, allowing the program to *use* slack_steal percent of the slack of any given sink block to aid in duplication reduction.

   (b) The most aggressive technique the algorithm uses to reduce cluster count is to force a given location to expel all of its microclusters. This is done through an $eject$ flag associated with each CLB location. The $eject$ flag will be set for a percentage of the

least-critical CLBs and all empty CLBs. The algorithm strives to gradually reduce the cluster count of the circuit while minimizing the impact this has on critical-path delay. The goal is to finish the Reduction stage at iteration 150. Define a variable $eject\_pct$, where

$$\begin{aligned} eject\_pct &= \frac{(used\_clbs - desired\_grid\_size^2)}{used\_clbs} && \text{if iteration} < 150 \\ eject\_pct &= 0.02 && \text{if iteration} \geq 150 \end{aligned}$$

Notice that if the area is not met by iteration 150, a fixed 2% of locations are expelled until the area requirement is met.

The $eject$ flag is set for all empty CLBs, and the $eject\_pct$% least-critical of all occupied locations. A location's criticality is the inverse of the minimum slack of all its blocks. If the $eject$ variable is set for a cluster, the algorithm will not move any microclusters into it; it will only move microclusters away from that location even if this means an increase in critical-path delay. The usage of the $eject$ flag is further explained in Section 5.2.8.

2. Order all microclusters according to criticality:
   - First priority is the minimum slack of all blocks in the microcluster.
   - Second priority is the average slack of all blocks in the microcluster. If multiple microclusters have the same minimum slack, the average slack across all blocks in the microcluster is used as a tie-breaker.
3. For each microcluster, in descending order of criticality, attempt to move each microcluster. See Section 5.2.7 for details on the moving process.
4. For each pad, in no particular order, attempt to move. See Section 5.2.9.

5. Perform compaction if enough free space exists to fit the circuit into a smaller grid size. Assuming $G_{current}$ is the length/width of the current grid (excluding pads), a grid compaction is possible if the following equation is satisfied:

$$(G_{current} - 1)^2 \geq total\ clusters$$

   Compaction is performed by an incremental placer described in Section 5.2.10.

6. The main loop of the program exits when all of the following conditions have been met:

   - The program has completed at least 25 iterations since the last compaction (Step 5). This criteria is in place to give the algorithm sufficient opportunity to recover any detrimental effects the compacting/annealing may have had on the critical path.
   - The program has run the predefined minimum number of iterations (default 100) AND met the desired grid size
   - The program has not yet reached a hard limit on iterations (default 200).

   If any of these conditions have not been met, return to Step 1.

5.2.6 Duplicate Analyzer

input: `float` slack_steal $\in [0,1]$

The job of the Duplicate Analyzer is to remove node duplication in an effort to meet the area specification of the design. Under no circumstances should the Duplicate Analyzer remove a duplicate which causes the critical-path delay to increase. The Duplicate Analyzer may remove a duplicate which causes the completion time of a non-critical block to increase. The *slack_steal* variable specifies, as a percentage, the amount of slack that can be sacrificed in order to further reduce the number of duplicates. An initial step the Duplicate Analyzer performs is to merge all duplicates that exist at the same (x,y) location but in different microclusters. This requires migrating

all successors to one of the duplicates and removing the unused ones. This not only simplifies the task of the Duplicate Analyzer, it prevents certain errors from occurring.

The main portion of the Duplicate Analyzer works by traversing all blocks in a breadth-first order starting at the primary outputs. Starting from the outputs will prevent the Duplicate Analyzer from accommodating a sink that will later be removed. For each set of duplicates that has not yet been analyzed:

1. Assemble lists of 1. all clones and 2. all sinks

2. Create a set of sets, the **clone usage set**, where each internal set represents all clones which satisfy the timing requirements a given sink. A clone is said to satisfy a given sink, provided:

   $$completion\_time(clone) + net\_delay(clone, sink) < arrival\_time(sink) + [slack(sink) * slack\_steal]$$

   The *slack_steal* variable is used to trade off slack for node duplicate reduction. It allows the algorithm to exhaust the slack of some nodes to find a smaller set of duplicates which satisfy the requirements of all sinks.

3. From the clone usage set, find a hitting set. The hitting set is a set of clones such that at least one element from the hitting set exists for each internal set in the clone usage set. Intuitively, this creates a set of clones (possibly smaller than the original set) that are able to satisfy the timing requirements of all sink blocks. The hitting set is an NP-Complete problem [25], so a simple greedy heuristic is required. The algorithm gives precedence to clones with a higher cardinality.

   To find the hitting set:

   - Start by adding all blocks that are essential to a given sink. If we conclude that $sink_A$ can only be satisfied by $clone_B$, automatically include $clone_B$ in the hitting set.
   - Remove internal sets of clone usage set that are satisfied.

- Provided internal sets remain in the clone usage set, find the clone that can satisfy the most remaining sinks. Include that clone.
- Remove internal sets of clone usage set that are satisfied.
- If sinks remain to be satisfied, return to first step. Otherwise, return hitting set.

4. For each sink, update its input to the clone in the hitting set that will produce the earliest arrival time of that signal.
5. Remove all duplicates that are not in the hitting set.
6. Update timing graph.

5.2.7 Microcluster Relocation

input: `microcluster` *MC*

The microcluster relocation routine will attempt to move *MC* to a new location, with respect to the following objectives, in order of priority:

- the validity of the solution is maintained (no cluster constraints are broken),
- the critical-path delay of the circuit is not increased,
- the arrival time for successors of *MC* is minimized, and
- the overall bounding box cost is minimized.

If possible, the algorithm will move *MC* to a location which reduces the delay on a critical path; this may occur at the expense of slack on other non-critical paths.

The microcluster relocation routine works by maintaining a two dimensional array of possible move locations. By analyzing the predecessor nodes and successor nodes of *MC*, the algorithm constructs a set valid move locations and attempts to relocate *MC* to one of these locations.

The first step of the microcluster move routine is to check that *MC* can be moved from its current location. A microcluster cannot be moved out if its removal would increase the number of inputs to the location and cause a violation of the input constraints. This may happen if the output of *MC* is used within the location by another microcluster.

Provided *MC* is movable, the microcluster move routine is performed as follows:

1. Create two arrays, equal in size to the current grid. The first, VALID_LOCATIONS, is an array of `bools`, where each element specifies whether a given (x,y) location meets the timing requirements for microcluster *MC*. Initialize the VALID_LOCATIONS so that all pads are false and all CLBs are true. The second array, BB_OVERLAP, is an array of `integers`, where each element specifies how many bounding boxes of nets incident to *MC* overlap a given (x,y) location. Initialize all elements to 0.

2. Compute the BB_OVERLAP array. The BB_OVERLAP array is used to assess how a move will affect the bounding box cost of the circuit. For all input nets ni_i to the microcluster:

   - Compute the bounding box of ni_i ignoring any blocks in microcluster *MC*. Ignoring *MC*'s current position prevents a biasing towards *MC* remaining where it is.
   - For all locations that fall outside of the bounding box of ni_i, increment the corresponding BB_OVERLAP element by 1.

   For all output nets no_i of the cluster:

   - Compute the bounding box of no_i, ignoring the source location and any sinks that are intracluster.
   - For all locations that fall outside of the bounding box of no_i, increment the corresponding BB_OVERLAP element by 1.

   The BB_OVERLAP array now represents how many bounding boxes will increase in size if microcluster *MC* is moved to any location on the grid.

3. Next, the VALID_LOCATIONS array is constructed. From the input nets to the microcluster, form a set of valid move locations. A move location is valid provided that performing the move does not cause the completion time of any block in *MC* to increase beyond its slack margin. To create the set of valid move locations, iterate through all input nets ni_i and do the following:

   - Determine the time at which the source of ni_i produces a result, $source\_finish$. The value is calculated to the point when the signal leaves the CLB of the source block.
   - Find the sink of ni_i in *MC* with the earliest arrival time, $sinkMC_i$. For $sinkMC_i$, compute

$$\begin{aligned} required\_arrival(sinkMC_i) = \quad & arrival\_time(sinkMC_i) + slack(sinkMC_i) \\ - \quad & t_{CLB\_input} - t_{LUT\_input} \end{aligned}$$

     t_{CLB_input} is the delay through the connection block to the CLB input pin, t_{LUT_input} is the delay from a CLB input pin to a BLE input pin.
   - The time allowance of net ni_i is defined as the difference between the required arrival of $sinkMC_i$ and $source\_finish$.

$$input\_allowance(ni_i) = required\_arrival(sinkMC_i) - source\_finish$$

     Intuitively, *MC* must be positioned such that the signal from ni_i can reach it in the time allowance, or risk creating a critical path.
   - As we have sufficient placement and timing information, we can determine exactly which locations can meet the constraint of the timing allowance. From the timing model, the delay of a wire segment is known to be $Tdel\_wire$. Therefore,

$$hops = \lfloor input\_allowance(ni_i)/Tdel\_wire \rfloor$$

is the number of wire segments that net ni_i has to reach *MC*. So, for all locations not within *hops* of the source of ni_i, mark corresponding element of VALID_LOCATIONS as false.

Figure 5.3 presents an example of the VALID_LOCATIONS masking procedure. In the example, *MC* is shown in blue and predecessors of *MC* are shown in red. Examining the predecessor to the bottom-right of *MC*, assume *hops* is found to be 3. In Figure 5.3b, the VALID_LOCATIONS array has been masked such that all invalid move locations farther than 3 hops are shown in grey. By definition, *MC*'s current location will always be a valid move location, as it is known to meet the timing requirements of *MC*.

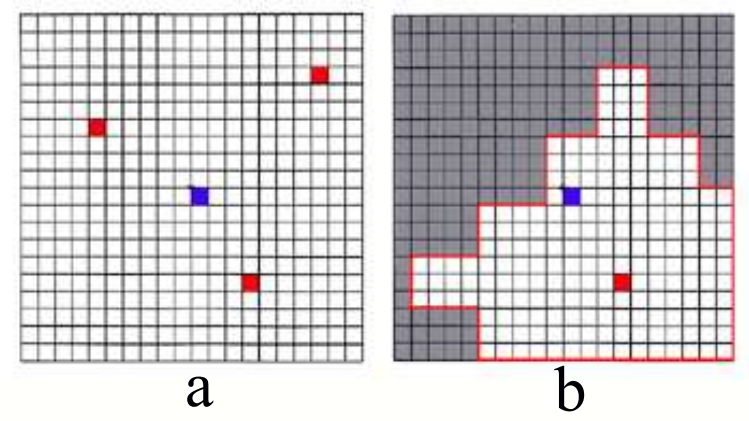

Figure 5.3: VALID_LOCATIONS Masking Example

After all input nets have been examined, the VALID_LOCATIONS should have a number of elements marked true (at the very least, only *MC*'s current location is marked true). The elements marked true represent (x,y) locations that *MC* can move to and still meet the timing requirements of all blocks in *MC*. If the only valid location is *MC*'s current location, return from the microcluster move routine, leaving *MC* where it is.

4. The next step requires the program to impose some unobtainable timing restrictions on the output nets of the microcluster, and slowly relax those restrictions until a suitable move loca-

tion is found. The idea is to find a location to move *MC* to that satisfies the tightest restrictions. The hope is that this will in turn cause the arrival time of successor blocks to decrease.

To proceed, the algorithm must find a starting point that is guaranteed to be unobtainable. Find successor s_{min}, such that $arrival\_time(s_{min}) + slack(s_{min})$ is the lowest value for all successor blocks. Define a variable *margin*, and set it to:

$$margin = arrival\_time(s_{min}) + slack(s_{min}) - t_{CLB\_input} - t_{LUT\_input}$$

The margin variable is used to adjust the restrictions on the output nets. This initial value is shown to be unobtainable later in this section.

5. The process of finding a valid move location for MC is as follows:

   - For each output net MC_out_i, provided it has at least one intercluster connection, mask the VALID_LOCATIONS for each sink. First compute the time at which the signal MC_out_i leaves microcluster *MC*.

     $$completion\_time(MC\_out_i) = completion\_time(source\ block\ of\ MC\_out_i) + t_{LUT\_output}$$

     where t_{LUT_output} is the delay from the output of a LUT to the output pin of the CLB.

     Next, for each sink $sink_k$ of MC_out_i:

     – Calculate the arrival time of $sink_k$ minus the time required to route a signal from the CLB input to BLE input:

     $$\begin{aligned} CLB\_arrival(sink_k) = \quad & arrival\_time(sink_k) \\ - \quad & t_{CLB\_input} \\ - \quad & t_{LUT\_input} \end{aligned}$$

– For net MC_out_i, $completion\_time(MC\_out_i)$ is the time at which the signal leaves microcluster *MC*. $CLB\_arrival(sink_k)$ is the time at which net MC_out_i must reach $sink_k$ to prevent the arrival time of $sink_k$ from increasing. The $output\_allowance$ value is calculated as:

$$\begin{aligned} output\_allowance = \quad & CLB\_arrival(sink_k) + slack(sink_k) \\ - \quad & completion\_time(MC\_out_i) \\ - \quad & margin \end{aligned}$$

The $output\_allowance$ value is the delay permitted for signal MC_out_i to reach $sink_k$. If the program can successfully find a position for microcluster *MC* where the $output\_allowance$ value is less than current net delay, it may cause a reduction in arrival time for $sink_k$. The $slack(sink_k)$ term increases $output\_allowance$ by depleting the slack of non-critical blocks. The *margin* variable is used to adjust the timing restrictions for block $sink_k$.

Note that with an initial margin value of $arrival(s_{min}) + slack(s_{min}) - t_{CLB\_input} - t_{LUT\_input}$, if $sink_k == s_{min}$, terms one and two will be cancelled out by the *margin* term, leaving $output\_allowance$ less than or equal to zero. Therefore, we guarantee our initial value is unobtainable.

– In order to meet the allowance delay, microcluster *MC* must be within $hops = \lfloor allowance/Tdel\_wire \rfloor$ of $sink_k$. Therefore, mask all elements of VALID_LOCATIONS, which are not within *hops* of $sink_k$, as false.

6. Once the VALID_LOCATIONS has been masked for all sinks of all output nets, there may or may not be a some elements which are still marked as valid locations. If there are no valid locations (implying the timing restriction is too tight), do the following:

   • restore the VALID_LOCATIONS array computed after masking the input net sources

- decrement the margin variable by $Tdel\_wire/4$
- return to Step 5

Decrementing margin by $Tdel\_wire/4$ was determined by experimentation described in Appendix B.

If valid locations are still present, the algorithm attempts to find the best candidate for relocation. The algorithm has four possible outcomes at this stage, here listed in descending order of desirability:

(a) move microcluster *MC* to a new location

(b) swap microcluster *MC* with a less critical microcluster, provided it results in microcluster *MC* adhering to a tighter timing restriction

(c) retain microcluster *MC* at this current position

(d) if no valid moves or swaps can be found, and *MC*'s current position does not meet this timing restriction, decrement margin and perform another iteration

7. The first step in attempting to relocate *MC* is to sort the valid move locations. The metric used is the number of bounding boxes that are not increased by relocating the microcluster. Using the BB_OVERLAP array computed earlier, order all valid relocation candidates.

8. For each valid relocation candidate (x_j, y_j), in ascending order of maintained bounding boxes:

   - If this is *MC*'s current position, concede the fact that no timing improvement can be made from moving *MC* during this iteration. Any move that can be made will result in an overall increase in bounding box cost; if a swap exists, no timing improvement will be gained. Return from the microcluster move routine.
   - If microcluster *MC* can not be legally moved to (x_j, y_j) (violates a cluster constraints - inputs/cluster, BLEs/cluster), advance to the next valid relocation candidate.

- If microcluster *MC* can be legally moved to (x_j, y_j), perform the move. Update the timing graph to reflect the change. Return from the microcluster move routine.

9. If no legal moves can be found, and *MC*'s current position is not a valid relocation candidate, swapping *MC* with a less critical microcluster may result in a decrease in timing cost. So, for each valid relocation candidate (x_j, y_j), in ascending order of maintained bounding boxes:

   - For all microclusters in (x_j, y_j), try to find the least critical, $MC_{\neg critical}$ such that moving *MC* to (x_j, y_j) while moving $MC_{\neg critical}$ to *MC*'s current position is legal for both. If no swap partner can be found, proceed to the next valid relocation candidate.
   - If a valid $MC_{\neg critical}$ is found, perform the swap. Update the timing graph to reflect the change. In the event that the critical-path delay of the circuit has increased, undo the move. This can occur if moving $MC_{\neg critical}$ is so detrimental that it causes $MC_{\neg critical}$ to become highly critical and create a new critical path. If the swap does not increase the critical path, return from the microcluster move routine.

   If no valid swaps can be found:

   - restore the VALID_LOCATIONS array computed after masking the input net sources
   - decrement the margin variable by $Tdel\_wire/4$
   - return to step 5

5.2.8 Microcluster Relocation During the Reduction Stage

During the reduction stage, the microcluster relocation routine is modified to aid in area reduction. It breaks non-critical microclusters into individual blocks and uses them to fill up other clusters. In Step 1b of the main loop body, the *eject* flag is set true for a number of locations. The *eject* variable indicates to the microcluster relocation routine:

1. not move any microclusters into this location

2. not swap any microclusters into this location

3. move all resident microclusters away from this location if possible, even if it means a critical-path delay penalty

This technique allows the algorithm to reduce the CLB usage to the prescribed limit.

The microcluster relocation routine during the Reduction stage is very similar to the normal routine, with a few distinct modifications:

- If the current position of microcluster *MC* has its *eject* flag set, do not attempt any swaps.
- When ordering the valid move locations in Step 8, ignore any locations with a set *eject* flag.
- When relocating a microcluster *MC*, in a location with the *eject* flag set, break *MC* into its constituent blocks, with each block now existing in its own microcluster. This technique is especially useful when the circuit is highly utilized as a single element microcluster has more freedom to move than a near-full microcluster.
- Provided the *eject* flag of *MC*'s current location is set, leaving *MC* (or its derivative microclusters) in its current position is not a preferable option. Therefore, a valid move location may only be found by moving *MC* to a location which causes a critical-path delay increase. In this instance the algorithm is forced to sacrifice performance to obtain the required area. In rare circumstances, no valid move location exists; if *margin* ever becomes negative, the algorithm gives up and leaves *MC* in its current position.

5.2.9 Pad Relocation

input: `pad` *p*

The pad relocation routine is used to move inpads and outpads to improve the circuit performance.

Outpads

In the architecture model assumed here, outpads may only be driven by a single source. When relocating outpads the algorithm attempts to reduce the Manhattan distance between the pad and its predecessor. Consider an outpad p, with predecessor p_{pred} located at (x_{pred}, y_{pred}). The process for moving p is as follows:

- Determine the current Manhattan distance from p to p_{pred}, $manh_{current}$.
- Create a list of possible move locations *attempt_locations*, organized in ascending order of Manhattan distance to (x_{pred}, y_{pred}). For all I/O locations, (x_i, y_j), compute the Manhattan distance to (x_{pred}, y_{pred}), $manh_{i,j}$. If $manh_{i,j} < manh_{current}$, insert (x_i, y_j) into *attempt_locations*.
- If *attempt_locations* is empty, the pad relocation routine terminates, otherwise, traverse the *attempt_locations* list in order.
- For each location $(x_k, y_l) \in$ *attempt_locations*, provided (x_k, y_l) has space for an additional pad, and moving p does not increase the critical-path delay of the circuit, move p to (x_k, y_l). If the move is successful, the pad relocation routine updates the timing graph and terminates. If the move is not successful, advance to the next location.
- If the *attempt_locations* list is exhausted, terminate without moving p.

Inpads

An inpad may supply a number of different successors, therefore it requires a more complicated algorithm which is based on the same principles used in the microcluster relocation routine. The process to move an inpad p proceeds as follows:

1. Create an array of `bools` equal in size to the current grid, VALID_LOCATIONS. Each element specifies whether a given (x,y) location meets the timing requirements for pad p. Initialize the VALID_LOCATIONS so that all I/Os are true and all CLBs are false.

2. Next, set the initial value of the *margin* variable. For all successors s_i of p, set:

$$margin = min\{arrival(s_i) + slack(s_i)\} \forall s_i$$

3. For each successor s_k, compute the signal time allowance as:

$$allowance = arrival(s_k) + slack(s_k) - margin$$

4. Therefore, p must be within $hops = \lfloor allowance/Tdel\_wire \rfloor$ wire segments of s_k. Mask the VALID_LOCATIONS grid as shown in Figure 5.3, where all locations farther than *hops* are set to false.

5. If no locations of VALID_LOCATIONS remain true after masking for all successors, reset the VALID_LOCATIONS array, decrement the margin by $Tdel\_wire/4$ and start again.

6. If legal move locations do exist, randomly choose one location (x_i, y_j).

   - If (x_i, y_j) is p's current location, the routine terminates without moving p.
   - If (x_i, y_j) has sufficient space for an additional pad, move p to (x_i, y_j), update the timing model and return.
   - If (x_i, y_j) does not have sufficient space, mark (x_i, y_j) as false and randomly choose another location.

While the microcluster relocation routine is more deterministic, the pad relocation routine is allowed to have a degree of uncertainty. This allows inpads to explore different locations, which may allow the microcluster relocation routine to achieve a lower critical-path delay, while guaranteeing that the critical path will never increase by moving an inpad. This method was found to be more effective than trying to use the bounding box cost as a tie-breaker, as is done in the microcluster relocation routine.

5.2.10 Compaction

The compaction routine uses an incremental placement algorithm designed by David Leong [37]. It is used to reorganize all CLBs into a smaller grid whenever:

$$(G_{current} - x)^2 \geq total\ used\ CLBs$$

x is chosen as the largest integer value that will satisfy the inequality. This means that the grid is compacted as much as possible after each iteration of the program.

The compaction algorithm works by defining a sub-grid of size $G_{current} - x$ within the current grid. All CLBs that exist outside this sub-grid are said to be illegal. The illegal area of the original grid is split into 8 regions: 4 sides adjacent to the sub-grid and 4 corners diagonally connected to the sub-grid. For each region which contains illegal CLBs, all free space in the sub-grid is move adjacent to the side or corner in question. The algorithm then moves the stray CLBs into the sub-grid. This continues until all CLBs are contained within the sub-grid.

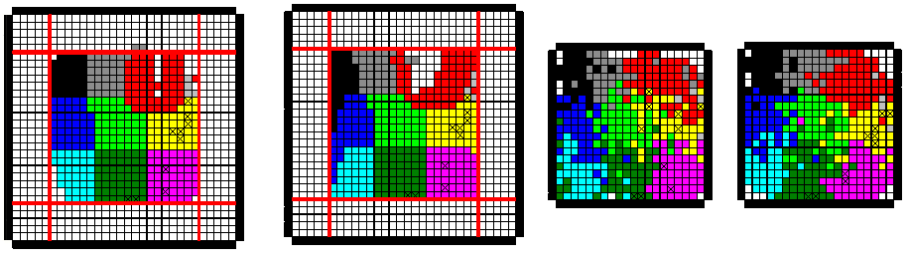

Figure 5.4: Compaction Routine Example (from [37])

Figure 5.4 shows an example of the compaction process. The far left image shows a circuit which is to be compacted into the sub-grid outlined in red. Notice the two illegal blocks north of the sub-grid. The second image shows the result of the compaction routine. All free space in the sub-grid is moved adjacent to the illegal blocks, and the illegal blocks are moved into the sub-grid. The third image shows the circuit after the refinement anneal has been performed. The fourth

image is the result of a complete anneal of the circuit into the smaller grid. Notice from the block colouring that the compaction/refinement anneal provides similar results to a full anneal, but with less computation time.

After a compaction, a fast refinement anneal is performed to recover any quality degradation caused by the compaction. The refinement anneal uses the programs embedded simulated annealer, based on the VPR annealer described in Section 2.3.1. The refinement anneal differs from a complete anneal in that it assumes the current solution is already close to a good solution. Provided the compaction step does not drastically alter the circuit placement, a low temperature anneal will recover the bound box and timing cost increases caused by the compaction.

Specifically, the refinement anneal differs from a complete anneal, described in Section 2.3.1, in the following ways:

1. The initial temperature is set to achieve a 44% acceptance rate from the start of the refinement. Assuming the compaction routine has not changed the placement too severely, the local minimum should coincide with the global minimum.

2. The move range for a CLB is set to 12.5% of the grid width. This limits how much the anneal may perturb the current placement.

3. The temperature reduction factor is set to 0.8. This causes the refinement annealer to reduce the temperature more quickly than the VPR annealer, and therefore allow the refinement annealer to run faster.

The refinement anneal will decrease the timing and bounding box cost of the circuit to a level similar to the pre-compaction state of the circuit.

5.3 Analysis and Results

This section provides initial timing results to compare Orchestrator with T-VPack/VPR. A more extensive comparison of results appears in Chapter 6.

5.3.1 Timing Results

The combined clustering and placement algorithm presented here achieves, on average, an 11% timing improvement over T-VPack. The maximum timing improvement obtained was 20.3% (s38417). Orchestrator failed to improve timing for only one of the twenty circuits (diffeq), which suffered a 3.5% performance degradation. All results are for no area restrictions and a fixed channel width of 100. Figure 5.5 shows a graph of timing performance for T-VPack and Orchestrator in ascending order of T-VPack critical-path delay. Full numeric results can be found later in Table 6.1.

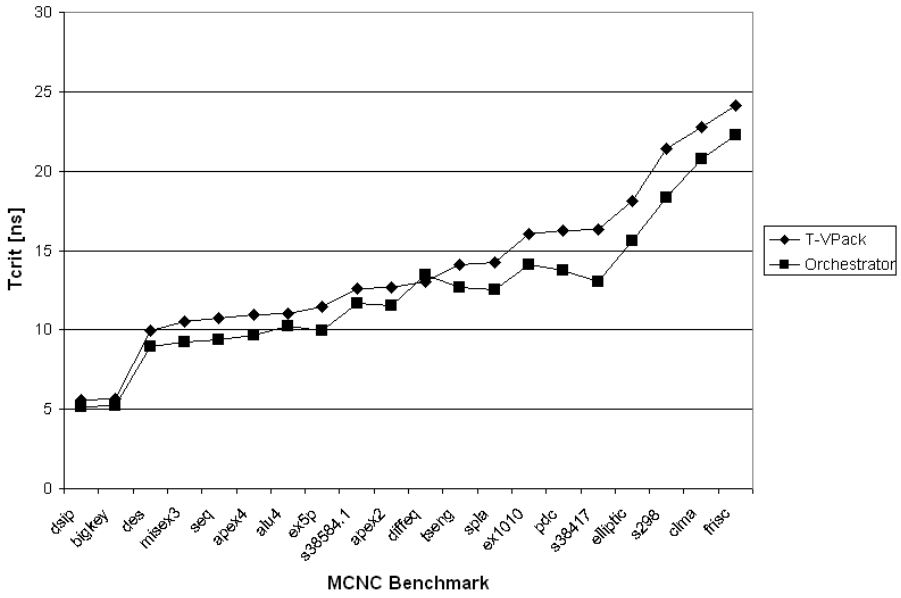

Figure 5.5: Critical-Path Delay Results

The overall results show a noticeable improvement over T-VPack. The smaller circuits, bigkey and dsip, have minimal improvements as each has such a short minimum depth, providing little room for improvement. The circuit diffeq is the only MCNC benchmark where T-VPack performs better than Orchestrator. This may be attributed to T-VPack's ability to cluster this circuit in a depth-optimal manner. Further exploration of the depth/timing correlation is investigated in Chapter 6.

5.3.2 Orchestrator with Area Restrictions

When area restrictions are imposed on the circuit, the Orchestrator tool may increase the critical-path delay to meet the area restriction. To avoid this as much as possible, the Additional Duplicate Reduction Technique described in Section 4.6 can be employed to reduce duplication prior to instantiating the Orchestrator tool. To determine the best initial clustering solution for different area restrictions, a series of tests were performed. The results shown in Figure 5.6 represent final timing results produced by the Orchestrator tool. The initial clustering solutions explored were:

- LLT clustering
- single-pass NDR clustering (no duplication limiting)
- NDR clustering with a duplication limit of: 5%, 10%, 20%, 30%, 50%, and 70%

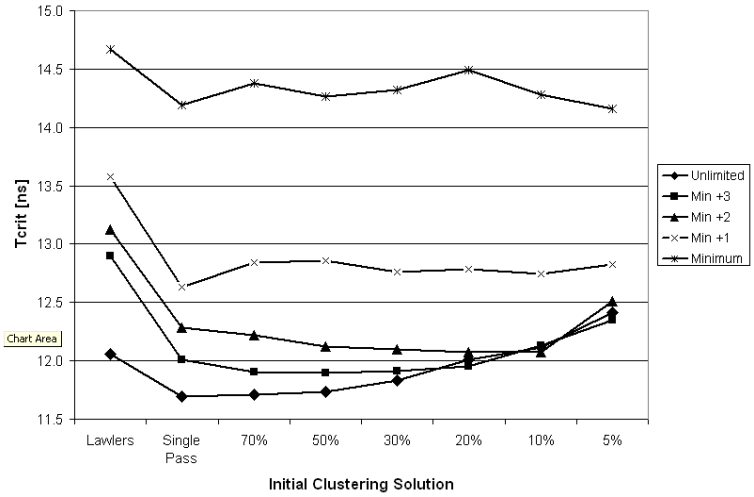

Figure 5.6: Duplication Limiting Test Results

From these results, the settings for Microcluster Formation were chosen for various grid size setting to reduce delay. These settings, shown in Table 5.1, are used.

Table 5.2 shows timing results, normalized to T-VPack, for the Orchestrator tool under various area restrictions. **Min** refers to the minimum grid size required as determined using T-VPack. The

Orchestrator Grid Setting	**Initial Clustering Settings**
Unlimited	Single Pass
Min +3	50% duplication limit
Min +2	20% duplication limit
Min +1	Single Pass
Minimum	5% duplication limit

Table 5.1: Duplication Limiting Final Settings

results are for the standard architecture, as described in Section 2.1, with low-stress routing, where the channel width is set to 30% more than the minimum required. Results are presented in ascending order of T-VPack critical-path delay.

The results presented in Table 5.2 show how area allowance is directly related to performance. Even with a grid increase of 1 unit, a performance improvement over T-VPack is possible. With the exception of a few circuits, minimum grid size results are on par with T-VPack.

5.3.3 Timing vs. Area Performance

The main goal of the Orchestrator tool is to improve timing performance within the area restrictions imposed. To show how effectively Orchestrator accomplishes this, Figure 5.7 plots the geometric mean critical-path delay (squares) and CLB usage (triangles). Clustering solutions include the single pass NDR algorithm described in Section 4.5.1 with placement by VPR, the Orchestrator tool with various area restrictions, and T-VPack/VPR. It should be noted that for two circuits (ex1010 and frisc), the minimum grid size was not met.

The results show that Orchestrator provides a significant improvement over using just single-pass NDR from Chapter 4. Orchestrator achieves a 9% delay improvement while reducing the CLB usage by 56%. When area restrictions are imposed, the Orchestrator tool provides a continuous trade-off between area and performance improvement.

	Unlimited	**Min +3**	**Min +2**	**Min +1**	**Minimum**	**TVPack**
dsip	0.95	0.94	1.06	1.01	1.04	5.57 ns
bigkey	0.95	0.95	0.95	0.97	1.03	5.60 ns
des	0.94	0.93	0.91	1.04	1.02	9.83 ns
misex3	0.91	0.97	0.97	0.98	1.03	10.46 ns
seq	0.89	0.87	0.87	0.96	0.95	10.79 ns
apex4	0.89	0.96	1.00	0.95	0.98	11.11 ns
alu4	0.92	0.97	0.97	0.93	0.97	11.44 ns
ex5p	0.89	0.91	0.96	1.04	1.00	11.58 ns
s38584.1	0.92	0.97	0.97	1.00	1.10	12.61 ns
apex2	0.93	0.91	0.96	0.95	0.99	12.77 ns
diffeq	1.02	1.06	1.05	1.02	1.18	13.04 ns
tseng	0.89	0.93	0.93	0.87	1.00	14.14 ns
spla	0.93	1.13	1.01	0.97	1.00	14.51 ns
s38417	0.83	0.82	0.82	0.92	0.99	16.22 ns
ex1010	0.86	0.99	0.87	0.88	1.12[a]	16.30 ns
elliptic	0.88	0.87	0.87	0.95	1.32	18.04 ns
pdc	0.73	0.79	0.78	0.82	0.91	18.93 ns
s298	0.94	0.88	0.84	0.94	1.06	22.44 ns
clma	0.94	1.03	1.06	1.07	1.37	22.78 ns
frisc	0.92	0.92	0.92	0.99	1.10[b]	24.22 ns
Mean:	0.91	0.94	0.94	0.96	1.05	14.12 ns

a - ex1010, used 499 CLBs, missing the target of 484 by 15 CLBs
b - frisc, used 369 CLBs, missing the target of 361 by 8 CLBs

Table 5.2: Orchestrator Results for Various Area Restrictions, Normalized to T-VPack

5.4 Orchestrator Summary

The Orchestrator tool has been show to be effective at consolidating microclusters and reducing the critical-path delay of the intermediate solution. With no area restrictions and a fixed channel width, the Orchestrator solution has an 11% average delay improvement over T-VPack (9% when a fixed channel width is replaced with +30% minimum channel width). Using different techniques to remove duplicates and reduce CLB usage, the Orchestrator tool can meet most area restrictions while still providing competitive performance compared to a greedy approach.

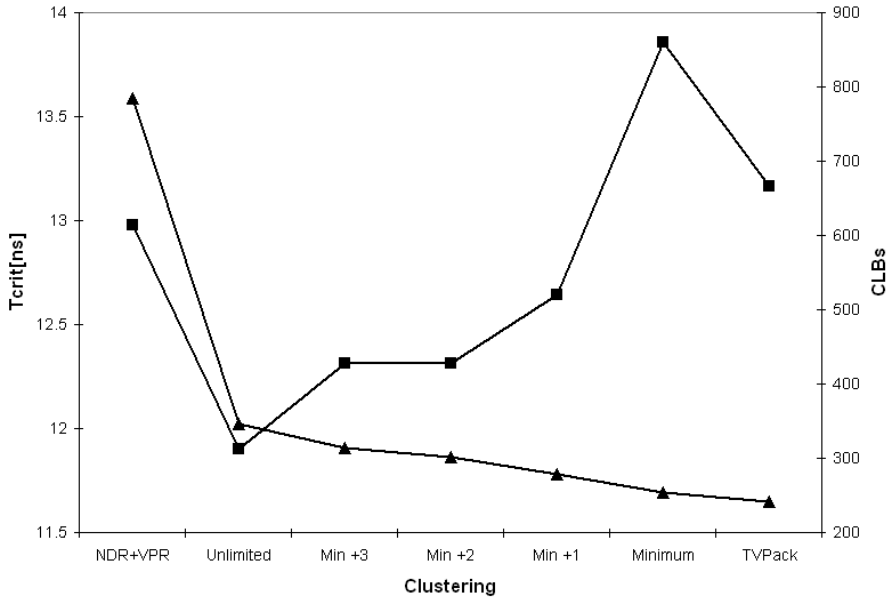

Figure 5.7: Orchestrator Area vs. Timing Performance

Chapter 6

Final Results

This chapter examines the performance of the Orchestrator tool in a number of relevant areas, include: Timing Performance, Circuit Depth, Routing Resource Usage, Area Usage and Runtime Performance. Comparisons are provided against a number of available greedy algorithms: T-VPack, T-RPack and iRAC, and other clustering/placement tools when applicable.

Unless otherwise noted, the final results use the standard architecture described in Section 2.1, with a fixed channel width of 100. To reduce noise in results, T-VPack and Orchestrator (no area restriction) results are average values across 5 independent runs.

For full VPR placement and routing, the following options are used:

$-inner\_num\ 20\ -max\_router\_iterations\ 300\ -pres\_fac\_mult\ 1.1$

All tests were run on an Intel Xeon 2.66GHz CPU with 2 Gb of RAM, running Linux kernel 2.6.18-8.1.15.el5. All source code was compiled using g++ 4.1.1-52 with -O2 optimization, except for the Microcluster Formation Phase code which was compiled using gcc 3.4.6-4 using -O2 optimization.

6.1 Timing Performance

As improving timing performance is the main goal of this research, an effort is made to compare the results to as many other tools as possible.

Table 6.1 presents final timing results for different clustering algorithms using the standard architecture with a fixed channel width of 100. T-RPack was obtained from the author of [7] and the iRAC tool was reproduced for [61]. Circuits are in ascending order of T-VPack critical-path delay.

	T-VPack	**T-RPack**	**iRAC**	**Orchestrator**
dsip	5.58	5.60	5.77	5.19
bigkey	5.69	5.58	5.54	5.23
des	9.93	10.77	10.50	8.98
misex3	10.53	10.06	10.42	9.26
seq	10.77	10.58	11.27	9.41
apex4	10.92	10.94	10.85	9.66
alu4	11.02	11.28	11.49	10.27
ex5p	11.44	11.49	12.52	9.97
s38584.1	12.60	11.95	13.35	11.67
apex2	12.70	12.69	13.24	11.51
diffeq	13.00	14.98	17.61	13.45
tseng	14.08	14.79	15.80	12.65
spla	14.28	14.23	14.54	12.56
ex1010	16.06	15.58	17.16	14.08
pdc	16.25	16.42	16.41	13.71
s38417	16.32	14.96	15.71	13.00
elliptic	18.10	16.85	23.68	15.62
s298	21.38	20.58	21.39	18.31
clma	22.76	22.07	23.10	20.80
frisc	24.14	24.22	26.06	22.24
Geomean:	12.99	12.94	13.76	11.62
vs T-Vpack:	1.00	1.00	1.06	0.89

Table 6.1: Timing Results for Different Clustering Algorithms

The results of Table 6.1 show an 11% timing improvement over T-VPack and T-RPack, and a 19% delay improvement over iRAC. As is shown in subsequent sections, this is at the expense of area, and in the case of iRAC, minimum channel width.

In an effort to identify and quantify the sources of this performance improvement, the clustering solution produced by Orchestrator was placed and routed by VPR. A full placement by VPR was performed using the final netlist produced by Orchestrator. The result was a 3% degradation in timing performance and a 9% improvement in minimum channel width. This indicates that node duplication and depth-optimality are the main source of timing improvement, but the placement routine outperforms the T-VPlace simulated annealer to contribute an additional 3% timing improvement.

Also presented in the Previous Work section is the SPCD algorithm [9], which states a 18% de-

lay improvement compared to T-VPack. Although not explicitly stated, the presumed architecture in that paper uses $k = 4$-input LUTs, $N = 4$ LUTs/CLB and wire segments of length 1. When the Orchestrator tool is run with a similar architecture, under low-stress routing, a 15% delay improvement is achieved over T-VPack. However, one final difference is that SPCD uses $0.35\mu m$ technology delays and Orchestrator uses $0.18\mu m$ delays.

A number of techniques native to the SPCD algorithm could be modified to improve timing in Orchestrator results. The most prominent of these techniques is the BLE level moves during simulated annealing. During refinement anneals, or as a post-placement adjustment, the SPCD-modified simulated annealer could be used to adjust clustering and placement. As Orchestrator does not have the same ability to modify the clustering solution, noticeable performance improvements may be possible.

The DPack [17] algorithm uses a $k = 4$, $N = 4$ and $I = 18$ architecture with single length wires. Under similar conditions, the Orchestrator achieves a 15% performance improvement over T-VPack, versus an 8% improvement cited by DPack.

The work by Schabas et al. [53] obtains slightly better timing results than Orchestrator by performing logic duplication after placement. With a 20% area increase, Schabas' new placement algorithm achieves a 14.1% delay improvement. Orchestrator achieves an 11% delay improvement with a 44% increase in area. The work by Schabas et al. may have an advantage over Orchestrator because their architecture uses 50% pass transistors and 50% buffered switches, whereas the architecture in this thesis uses fully buffered switches. Performing logic duplication on circuits with pass transistors can reduce the delay through a pass transistor by reducing fanout. A pass transistor with a smaller fanout will have a decreased output capacitance, and therefore have a smaller delay.

The post-placement logic duplication of [53] could also be used after the Orchestrator flow to increase timing performance. The most affective way to utilize the technique proposed by Schabas et al. would be to:

- Remove superfluous duplication after the Orchestrator tool has finished, as described in Sec-

tion 5.2.6.

- Use the Schabas et al. technique to create duplication in vacant logic elements to reduce delay.

This process would not increase the critical-path delay of the circuit, and according to [53], could provide a maximum of critical-path delay improvement 7.7%.

6.2 Depth

In the Microcluster Formation phase, node duplication was used to obtain a depth-optimal clustering. It was proposed that a depth-optimal initial clustering solution resulted in better timing performance in the final solution. This section examines the Orchestrator tool's ability to maintain a depth advantage over T-VPack and establishes the relationship between depth and final critical-path delay.

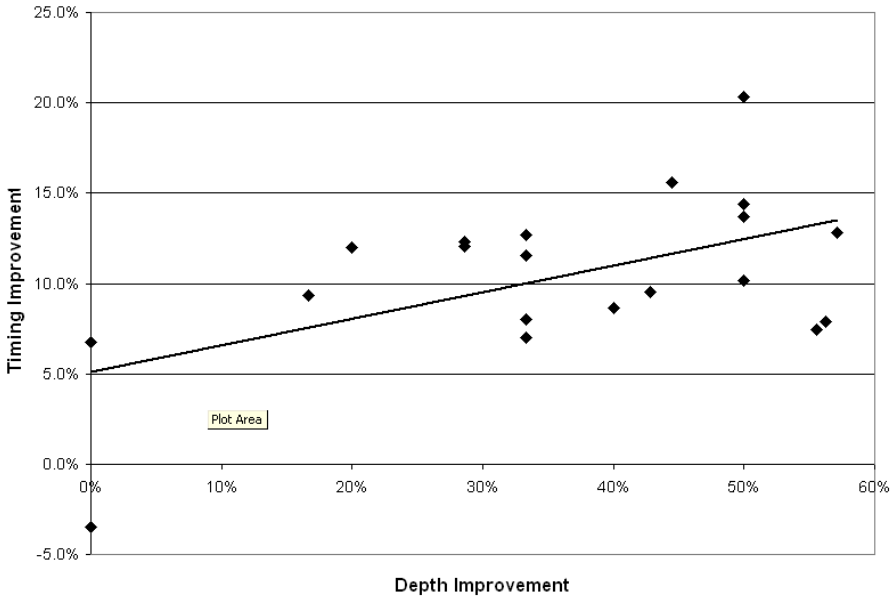

Figure 6.1: Depth Improvement vs. Timing Improvement

File	Actual Critical Path Depth TVPack	Orchestrator	Δ Depth	Timing Improvement
alu4	5	5	0	6.8%
apex2	6	5	1	9.4%
apex4	6	4	2	11.5%
bigkey	3	2	1	8.0%
clma	10	6	4	8.6%
des	7	4	3	9.5%
diffeq	5	5	0	-3.5%
dsip	3	2	1	7.0%
elliptic	8	4	4	13.7%
ex1010	7	5	2	12.3%
ex5p	7	3	4	12.8%
frisc	16	7	9	7.9%
misex3	5	4	1	12.0%
pdc	9	5	4	15.6%
s298	14	7	7	14.4%
s38417	8	4	4	20.3%
s38584.1	9	4	5	7.4%
seq	6	4	2	12.7%
spla	7	5	2	12.1%
tseng	8	4	4	10.1%

Table 6.2: Depth - Timing Improvement Comparison

Figure 6.1 presents a scatter plot of depth improvement versus timing improvement for Orchestrator, relative to T-VPack/VPR. The plot shows a modest ($R^2 = 0.3$) positive correlation between depth and timing improvement. Also, it is interesting to note that circuits alu4 and diffeq have equal depth in T-VPack and Orchestrator, and Orchestrator achieves little or no timing improvement. These results show that depth improvement has a direct influence on critical-path delay.

6.3 Routing Resource Usage

This section examines the performance of Orchestrator in terms of routing resource usage. Minimum channel width is used as the primary metric to assess how effectively each tool utilizes the routing architecture. Table 6.3 presents the minimum channel width for the 20 largest MCNC cir-

	T-VPack	**T-RPack**	**iRAC**	**Orchestrator**
alu4	33.5	32	31	32.6
apex2	46.3	46	41	41.2
apex4	47.5	46	43	36.8
bigkey	41.5	36	37	32.6
clma	61.8	59	47	70.8
des	40.8	39	39	42.4
diffeq	29.5	30	21	28.4
dsip	36.3	36	35	31.2
elliptic	49.5	44	36	42.2
ex1010	53.5	52	45	78.2
ex5p	48.5	49	42	43.8
frisc	52.0	52	44	59.2
misex3	42.3	41	37	36.2
pdc	65.5	63	60	60.6
s298	26.5	26	25	26.2
s38417	40.5	36	29	59.4
s38584.1	43.5	38	34	48.2
seq	45.5	42	38	41.6
spla	53.0	54	49	51.2
tseng	36.3	26	20	26.2
Geomean:	43.6	41.1	36.4	42.3
vs T-Vpack:	1.00	0.94	0.83	0.97

Table 6.3: Minimum Channel Width Comparison

cuits for three greedy approaches and for Orchestrator with no area restrictions.

As expected, T-RPack and iRAC consistently outperform T-VPack in terms of minimum channel width. Orchestrator shows a general improvement compared to T-VPack, but individual results are mixed. A maximum channel width decrease of 18% is achieved on tseng, while a maximum increase of 46% is suffered on ex1010. No specific relationship can be established between the minimum channel width required by Orchestrator and a greedy approach. This value is highly influence by the amount of duplication in the circuit, It should generally be assumed that Orchestrator requires a greater channel width than T-VPack.

As mentioned in Section 6.1, a complete VPR placement was performed on the clustering solution produced by Orchestrator. The result was a 9% decrease in the average minimum channel

	T-VPack	T-RPack	iRAC	Orchestrator
tseng	107	105	105	142.6
ex5p	109	110	110	204.6
apex4	132	130	131	246.4
dsip	137	137	137	226
misex3	142	142	141	193
diffeq	151	150	150	181.2
alu4	153	154	153	184.2
des	160	160	160	225.2
bigkey	171	171	171	228.6
seq	176	177	176	247.6
apex2	190	190	191	277
s298	194	194	194	264.4
frisc	356	357	359	500.2
elliptic	363	361	363	470.4
spla	374	373	372	551.8
pdc	463	462	461	718.8
ex1010	480	477	472	868.2
s38417	642	641	641	812.2
s38584.1	645	645	645	772.8
clma	842	841	841	1272.8
Geomean:	241.6	241.2	241.1	346.8
vs T-Vpack:	1.00	1.00	1.00	1.44

Table 6.4: CLB Usage Comparison

width, and a decrease from 46% to 35% in the worst case minimum channel width penalty. These results indicate that while the Orchestrator placement routine is effective at reducing delay, it is inferior to T-VPlace in terms of channel width.

6.4 Area Usage

The most direct measurement of area usage is the number of CLBs required. Table 6.4 shows the CLB usage for T-VPack, T-RPack, iRAC and Orchestrator (with no area restrictions). The three greedy approaches have similar CLB counts, with Orchestrator requiring and average increase of 44%.

This CLB increase represents the area increase required to achieve the 11% critical-path delay

	T-VPack	T-RPack	iRAC	Orchestrator
Geomean:	5.69E+06	5.46E+06	5.08E+06	7.82E+06
vs T-Vpack:	1.00	0.96	0.89	1.38

Table 6.5: Total Area Comparison [min. sized transistors]

improvement cited in Section 6.1. This area penalty can be reduced by imposing an area restriction on the circuit and suffering a decrease in timing performance, as described in Section 5.3.2.

A more comprehensive analysis of area usage is presented in Table 6.5, where area is the total routing and logic area, calculated in total equivalent minimum sized transistors. Logic usage was calculated by Transcount [5] and routing usage was provided by the VPR router.

The results of Table 6.5 reinforce the result that Orchestrator requires approximately 40% more area to achieve the 11% delay improvement cited earlier.

6.5 Runtime Performance

The timing improvements over a greedy approach are had at the expense of area and runtime. This section aims to quantify the runtime penalty when using the combined clustering and placement algorithm instead of T-VPack. As the majority of runtime for the T-VPack flow is spent in the placement stage, it can be presumed that iRAC and T-RPack have similar runtimes as T-VPack.

All runtime results represent the time required for clustering, placement and a binary-search routing. Routing is included because routing runtime may vary for T-VPack and Orchestrator clustering solutions because of different grid sizes and routing resource usage. Therefore, to provide a thorough evaluation of runtime, the effect on routing runtime should be included.

To demonstrate the potential for runtime improvement to the Orchestrator tool, a modified version of the tool is created with more emphasis on runtime performance. The following changes have been made to the Orchestrator program described in Chapter 5:

1. The minimum number of iterations since the last compaction/anneal is reduced from 25 to 10.

2. The *margin* variable is decremented by $wire\_Tdel/1$ after each iteration instead of $wire\_Tdel/4$.

3. If no progress (reduction in cluster count, timing improvement) is made for 10 iterations, the Reorganize stage is terminated.

4. The following refinement anneal parameters are adjusted:

    - The temperature adjustment factor is changed from 0.8 to 0.7.
    - The number of moves per temperature is reduced by a factor of $2/3$.

The results for the Orchestrator Fast achieve a 10% critical-path delay improvement over T-VPack instead of an 11% improvement with the standard Orchestrator algorithm. All other metrics for Orchestrator Fast are within 2% of the original Orchestrator results.

The results presented in Table 6.6 show that the Orchestrator tool is significantly slower than T-VPack, but runtimes are still feasible for large circuits. The results also demonstrated that significant runtime improvements can be made with little code modification and only a small performance degradation.

File	T-VPack	Orchestrator	Orchestrator Fast
alu4	1	4	4
apex2	3	14	7
apex4	1	13	4
bigkey	1	5	3
clma	33	783	256
des	1	6	4
diffeq	1	5	3
dsip	1	3	3
elliptic	7	32	19
ex1010	9	656	106
ex5p	1	5	4
frisc	7	87	36
misex3	1	4	2
pdc	20	106	35
s298	2	14	6
s38417	5	270	72
s38584.1	7	48	29
seq	2	6	05
spla	8	49	23
tseng	1	2	1
Arithmetic Mean	5	102	28

Table 6.6: Run Time Results [min]

Chapter 7

Conclusion, Contributions and Future Work

7.1 Conclusions

This thesis presents a novel approach to the FPGA Clustering and Placement problem. Through the use of node duplication, depth-optimal clustering and a combined clustering and placement approach, an 11% performance improvement has been demonstrated over T-VPack. This timing improvement is obtained at the expense of area, runtime, and in some cases, routing resource usage. It has also been shown that the proposed algorithm can gradually reduce area usage at the expense of timing performance to fit the area restrictions imposed by the user.

In Phase 1: Microcluster Formation, microclusters are created in a depth-optimal manner by the LLT algorithm. It was shown in Section 6.2 that a depth advantage in the initial clustering stage results in a critical-path delay advantage in the final routed solution. The results show that a depth advantage is instrumental in Orchestrator achieving a performance increase over a greedy approach such as T-VPack.

To create a depth-optimal clustering solution, the LLT algorithm requires significant amounts of node duplication. As can be seen in Figure 5.6, this excessive duplication limits the ability of Phase 2: Microcluster Compaction to reduce the critical-path delay. To reduce the amount of duplication to an acceptable level, the NDR algorithm is presented, which trades off slack on non-critical paths for a reduction in duplication. When no area restrictions are imposed, a solution with

reduced duplication and optimal depth results in the best post-routing timing performance. When area restrictions are imposed, it was shown that sacrificing depth in a controlled manner to reduce duplication ultimately resulted in better performance by reducing the performance penalty incurred in Phase 2.

In Phase 2: Microcluster Compaction, the Orchestrator algorithm is described, which iteratively reorganizes and consolidates microclusters. Provided sufficient node duplication exists and some depth advantage is present in the intermediate solution, the Orchestrator algorithm can produce better timing performance than a greedy approach. Compared to T-VPack, the Orchestrator tool produces an 11% critical-path delay improvement with a 44% increase in CLBs.

Finally, a modification to the Orchestrator tool was presented where the solution was forced to conform to certain area limit. Through this, it was shown that more area allowed a greater amount of duplication in the final solution, which translated into a lower critical path. The proposed algorithm demonstrated the ability to outperform T-VPack in terms of timing with as little as a single unit grid size increase over T-VPack.

7.2 Contributions

Listed below are a number of contributions that have come out of this research.

Orchestrator Framework

The Orchestrator tool is an independent program written in object-oriented C++. The framework has the ability to model the circuit with logic, clustering and placement information simultaneously; contrary to the T-VPack/VPR framework, where T-VPack records logic and clustering information, and VPR is concerned with placement. The Orchestrator framework also has the ability to track timing properties of the circuit directly to the block level. While not as accurate as VPR, the timing model is more accessible because of a simplified interface. Other algorithms have also been integrated into Orchestrator, such as the VPR simulated annealer, Dave Leong's incremental placer and

the Lawler Levitt Turner algorithm.

As the program is written in object-oriented C++, it has a number of advantages over the VPR environment, written in C: intuitive hierarchical structure, simple manipulation of the circuit through class function calls and low coupling which allows altering a specific area of the program. These properties have already prompted other students to use the Orchestrator framework to do further research.

Combined Clustering and Placement Approach

In Phase 2: Microcluster Compaction, one of the major advantages over FPGA CAD flows that separate the clustering and placement steps is the availability of placement and timing information. By understanding how a change to the clustering solution will affect placement, delay, and area, the Orchestrator tool can make more informed decisions. Though combining clustering and placement does increase the overall runtime of the CAD flow, the approach presented here, whereby microclusters are formed first, provides a reasonable trade-off between runtime and clustering flexibility.

Node Duplicate Reduction Strategy

The NDR algorithm presented in Section 4.5.1 provides a proficient means of reducing node duplication while maintaining a predetermined depth. In other research on label and cluster techniques ([32], [51], [63]), duplicate reduction is treated as an afterthought. The NDR algorithm should be applicable to other label and cluster algorithms based on the LLT algorithm. With minor modifications, the NDR algorithm should also be applicable to algorithms which use the general delay model [45].

Orchestrator Algorithm

The Orchestrator algorithm described in Chapter 5 presents a novel approach to the clustering and placement problem. Of particular innovation is the manner in which microclusters are moved. To

the best knowledge of the author, such a technique that uses placement and timing information to incrementally move clusters to improve timing has not been used in FPGA research.

7.3 Future Work

The combined clustering and placement algorithm presents a new approach to the clustering and placement steps of the FPGA CAD flow. A number of improvements for future work are described below.

7.3.1 Microcluster Formation Phase

The Microcluster Formation Phase may be improved to produce a better clustering solution for the Orchestrator tool through the following techniques.

- A number of algorithms ([51], [16], [13]) have improved on the LLT algorithm by using the general delay model presented in [45]. In this work, Lawler's original algorithm was chosen as the base of the initial clustering algorithm as it was conducive to the Node Duplicate Reduction technique described in Section 4.5. Additional performance improvements may be gained by transitioning the Microcluster Formation Phase to a general delay model, but at the cost of significant modification to the NDR algorithm.

- Figure 4.5 demonstrates the significant effect node duplication has on the final performance of a circuit. While a great deal of attention has been paid to node duplication, no ubiquitous formula has been devised to specify the most advantageous level of duplication for any given circuit. If such a formula could be determined, it would allow the algorithm to better leverage node duplication to improve performance.

7.3.2 Orchestrator

The Orchestrator algorithm presents a novel approach to FPGA clustering and placement. The algorithm was written from scratch, with the exception of the Compaction routine [37]. The program therefore introduces a great many avenues of exploration, as opposed to an established algorithm such as VPR which has been thoroughly explored over the past decade. A selection of possible areas for future research are presented below.

- In general, microclusters formed during the Initial Clustering Phase persist throughout the Orchestrator algorithm. While this simplifies the Orchestrator tool and reduces computational complexity, it limits the ability of Orchestrator to alter the clustering solution. A more robust solution would be to allow the Orchestrator to reorganize the grouping of blocks for a specific location as required. This technique could be beneficial by:

  1. placing blocks connected by a critical connection in the same microcluster and forcing the connection to remain intracluster
  2. breaking microclusters with low cohesion, allowing greater freedom of movement and reducing fragmentation
  3. consolidating related microclusters, which ultimately will reduce runtime

  While this technique holds promise, it will require a considerable amount of effort to determine the best method to reorganize microclusters.

- Retiming [34] refers to adjusting the location of flip-flops to improve critical-path delay. A great deal of research ([58], [50], [14]) has been done on the performance gains possible through FPGA retiming. Retiming during the clustering stage will result in a possible depth reduction and retiming during placement will results in a possible critical-path delay reduction. Considering how the Orchestrator tool uses a depth advantage to produce a timing performance gain over greedy approaches, retiming could result in substantial performance gains.

- Logic duplication is an integral part of this research and an effort has been made to utilize it to its full potential. When duplication reduction is performed in the Orchestrator tool, the algorithm removes duplication provided it does not adversely affect performance for the current clustering and placement solution. Unfortunately, the possibility exists that subsequent clustering and placement solutions may benefit from removed duplicates. One solution, employed by other clustering tools ([53], [9]), allows duplication during or after placement. This might be implemented by adding a duplicate insertion step after each iteration of the Orchestrator tool. The inserted duplicates would still be subject to removal, so any duplicates that are not actually needed would be reduced.

Bibliography

[1] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on field programmable gate arrays*, pages 3–12, New York, NY, USA, 2000. ACM Press.

[2] Michael J. Alexander, James P. Cohoon, Joseph L. Ganley, and Gabriel Robins. Performance-oriented placement and routing for field-programmable gate arrays. In *EURO-DAC '95/EURO-VHDL '95: Proceedings of the conference on European design automation*, pages 80–85, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[3] Giancarlo Beraudo and John Lillis. Timing optimization of FPGA placements by logic replication. In *Proceedings of the 40rd Design Automation Conference*, pages 196–201, 2003.

[4] V. Betz and J. Rose. Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size. In *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, pages 551–554, 5-8 May 1997.

[5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[6] Elaheh Bozorgzadeh, Seda Ogrenci Memik, Xiaojian Yang, and Majid Sarrafzadeh. Routability-driven packing: Metrics and algorithms for cluster-based FPGAs. *Journal of Circuits, Systems, and Computers*, 13(1):77–100, 2004.

[7] Elaheh Bozorgzadeh, Seda Ogrenci-Memik, and Majid Sarrafzadeh. Rpack: routability-driven packing for cluster-based FPGAs. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 629–634, New York, NY, USA, 2001. ACM Press.

[8] Stephen Dean Brown, Jonathan Rose, and Zvonko G. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(5):620–628, 1992.

[9] Gang Chen and Jason Cong. Simultaneous placement with clustering and duplication. *ACM Trans. Design Autom. Electr. Syst.*, 11(3):740–772, 2006.

[10] Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar. Dag-map: Graph-based FPGA technology mapping for delay optimization. *IEEE Design & Test of Computers*, 9(3):7–20, 1992.

[11] Jason Cong and Yuzheng Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *ICCAD*, pages 48–53, 1992.

[12] Jason Cong and Yuzheng Ding. On area/depth trade-off in LUT-based FPGA technology mapping. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 213–218, New York, NY, USA, 1993. ACM Press.

[13] Jason Cong and Michail Romesis. Performance-driven multi-level clustering with application to hierarchical FPGA mapping. In *Proceedings of the 38th Design Automation Conference*, pages 389–394, 2001.

[14] Jason Cong and Chang Wu. FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 644–649, New York, NY, USA, 1997. ACM.

[15] Altera Corporation. Stratix iii device handbook.

[16] Mehrdad Eslami Dehkordi and Stephen Dean Brown. The effect of cluster packing and node duplication control in delay driven clustering. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology*, pages 227–233. IEEE, 2002.

[17] Kristofer Vorwerk Doris Chen and Andrew Kennings. Improving timing-driven FPGA packing with physical information. In *Field Programmable Logic and Application*, pages 117–123, 2007.

[18] Hans Eisenmann and Frank M. Johannes. Generic global placement and floorplanning. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 269–274, New York, NY, USA, 1998. ACM Press.

[19] W.C. Elmore. The transient analysis of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19(1):55–63, 1948.

[20] Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *DAC*, pages 227–233, 1991.

[21] Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. Technology mapping on lookup table-based FPGAs for performance. In *ICCAD*, pages 568–571, 1991.

[22] Daniel Gomez-Prado and Maciej Ciesielski. *A Tutorial on FPGA Routing*. Department of Electrical and Computer Engineering, University of Massachusetts, Amherst.

[23] Brent Goplen and Sachin Sapatnekar. Efficient thermal placement of standard cells in 3d ics using a force directed approach. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 86, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Xilinx Inc. Virtex-5 user guide.

[25] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[26] Kevin Karplus. Xmap: A technology mapper for table-lookup field-programmable gate arrays. In *DAC*, pages 240–243, 1991.

[27] S. Kirkpatrick, Gelatt Cd, and Vecchi Mp. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[28] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and asics. In *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, pages 21–30, 2006.

[29] Jimmy Lam and Jean-Marc Delosme. Performance of a new annealing schedule. In *DAC*, pages 306–311, 1988.

[30] Julien Lamoureux. On the interaction between power-aware computer-aided design algorithms for field-programmable gate arrays. Master's thesis, University of British Columbia, Vancouver, BC, 2003.

[31] Mark LaPedus. Mask prices flatten but tool costs soar. *EE Times*, March 2006.

[32] E.L. Lawler, K.N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Trans. Computers*, pages 47–57, 1969.

[33] C.Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, 10:346–365, 1961.

[34] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[35] Guy G. Lemieux and Stephen Dean Brown. A detailed router for allocating wire segments in FPGAs. In *ACM/SIGDA Physical Design Workshop*, pages 215–226, 1993.

[36] Guy G. Lemieux, Stephen Dean Brown, and Daniel Vranesic. On two-step routing for FPGAs. In *ISPD*, pages 60–66, 1997.

[37] David Leong. Incremental placement for field-programmable gate arrays. Master's thesis, University of British Columbia, Vancouver, BC, November 2006.

[38] Hao Li, Wai-Kei Mak, and Srinivas Katkoori. Force-directed performance-driven placement algorithm for FPGAs. In *2004 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004), Emerging Trends in VLSI Systems Design*, pages 193–198, 2004.

[39] Joey Y. Lin, Deming Chen, and Jason Cong. Optimal simultaneous mapping and clustering for FPGA delay optimization. In *Proceedings of the 43rd Design Automation Conference*, pages 472–477, 2006.

[40] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for FPGAs. In *FPGA*, pages 203–213, 2000.

[41] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 37–46, New York, NY, USA, 1999. ACM Press.

[42] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117, New York, NY, USA, 1995. ACM Press.

[43] Fan Mo, Abdallah Tabbara, and Robert K. Brayton. A force-directed macro-cell placer. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 177–181, Piscataway, NJ, USA, 2000. IEEE Press.

[44] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *FPGA*, pages 29–36, 2001.

[45] Rajeev Murgai, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. On clustering for minimum delay/area. In *ICCAD*, pages 6–9, 1991.

[46] Rajeev Murgai, Yoshihito Nishizaki, Narendra V. Shenoy, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *DAC*, pages 620–625, 1990.

[47] Rajeev Murgai, Narendra V. Shenoy, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *ICCAD*, pages 572–575, 1991.

[48] Ravi Nair. A simple yet effective technique for global wiring. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(2):165–172, 1987.

[49] Jr Neil R. Quinn and Melvin A. Breuer. A forced directed component placement procedure for printed circuit boards. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(6):377–388, 1979.

[50] Peichen Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. *ACM Trans. Des. Autom. Electron. Syst.*, 3(3):437–462, 1998.

[51] Rajmohan Rajaraman and D. F. Wong. Optimal clustering for delay minimization. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 309–314, New York, NY, USA, 1993. ACM Press.

[52] Prashant Sawkar and Donald E. Thomas. Area and delay mapping for table-look-up based field programmable gate arrays. In *DAC*, pages 368–373, 1992.

[53] Karl Schabas and Stephen D. Brown. Using logic duplication to improve performance in FPGAs. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 136–142, New York, NY, USA, 2003. ACM Press.

[54] Carl Sechen and Alberto Sangiovanni-Vincentelli. Timberwolf3.2: a new standard cell placement and global routing package. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 432–439, Piscataway, NJ, USA, 1986. IEEE Press.

[55] Ellen Sentovich, Kanwar Jit Singh, Cho W. Moon, Hamid Savoj, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 328–333, Washington, DC, USA, 1992. IEEE Computer Society.

[56] Khushro Shahookar and Pinaki Mazumder. Vlsi cell placement techniques. *ACM Comput. Surv.*, 23(2):143–220, 1991.

[57] Amit Singh and Malgorzata Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 59–66, New York, NY, USA, 2002. ACM Press.

[58] Deshanand P. Singh and Stephen Dean Brown. Integrated retiming and placement for field programmable gate arrays. In *FPGA*, pages 67–76, 2002.

[59] William Swartz and Carl Sechen. New algorithms for the placement and routing of macro cells. In *ICCAD*, pages 336–339, 1990.

[60] Cliff C. N. Sze, Ting-Chi Wang, and Li-C. Wang. Multilevel circuit clustering for delay minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7):1073–1085, 2004.

[61] Marvin Tom. Channel width reduction techniques for system-on-chip circuits in field-programmable gate arrays. Master's thesis, University of British Columbia, Vancouver, BC, March 2006.

[62] K. Vorwerk, A. Kennings, and A. Vannelli. Engineering details of a stable force-directed placer. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 573–580, Washington, DC, USA, 2004. IEEE Computer Society.

[63] Hannah Honghua Yang and Martin D. F. Wong. Circuit clustering for delay minimization under area and pin constraints. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(9):976–986, 1997.

Appendix A

Microcluster Statistics

This appendix provides statistics on the microclusters produced by the Microcluster Formation Phase with a single pass of the NDR algorithm. The statistics are presented to give the reader a concise idea of what is produced in the Microcluster Formation Phase.

In later stages of the algorithm, microclusters are moved and consolidated, but always such that no location contains more than *N* blocks. Therefore, smaller microclusters have greater mobility. It is not necessarily advantageous to have all microclusters contain few blocks, but at least some portion should to allow the placement and clustering solution to evolve. Figure A.1 shows a histogram of blocks per cluster averaged across the 20 largest MCNC benchmarks. Figure A.1 shows that in general, most microclusters are less than half utilized. The average block usage per microcluster is 4.08.

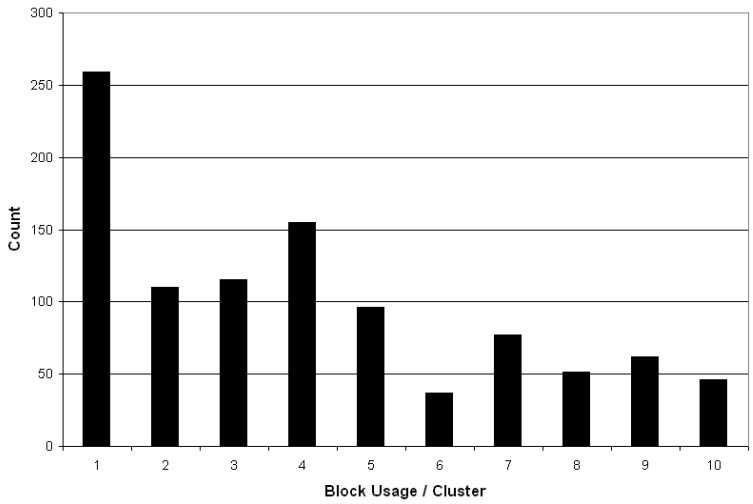

Figure A.1: Average Blocks Per Microcluster

It should be noted that characterizing blocks per microcluster is difficult because it is highly dependent on the nature of the circuit. Figure A.2 shows the distribution of microcluster sizes by file. While no discernible pattern can be formed, the graph does show that most circuits have a good distribution of large and small microclusters.

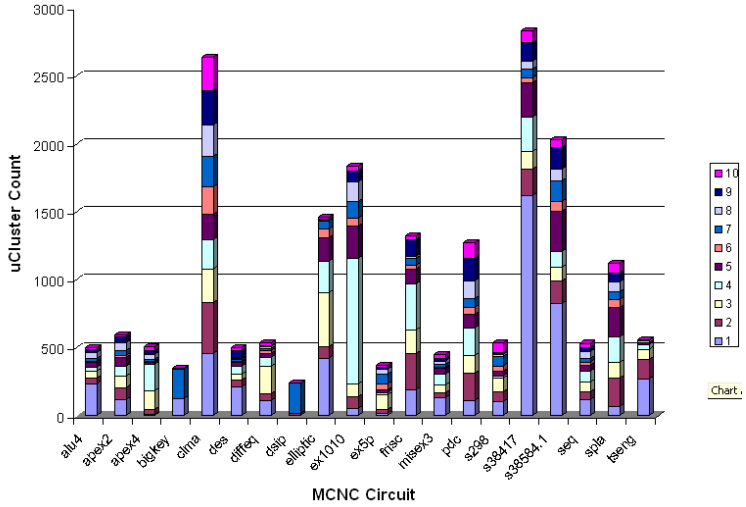

Figure A.2: Microcluster Size by Circuit

Appendix B

Margin Interval Test

When moving pads and microclusters, the *margin* variable controls how quickly the timing restrictions are relaxed. As the relocation routine is the most computationally intensive portion of the Orchestrator tool, the *margin* variable has a large influence on the run-time of the program.

Figure B.1 presents the performance and run-time results for a number of different margin intervals. Margin intervals range from $wire\_Tdel * 8$ to $wire\_Tdel / 10$. All performance results (shown as squares) are geometric means across the 20 largest MCNC benchmarks; all run-time results (shown as triangles) are arithmetic means across the 20 largest MCNC benchmarks.

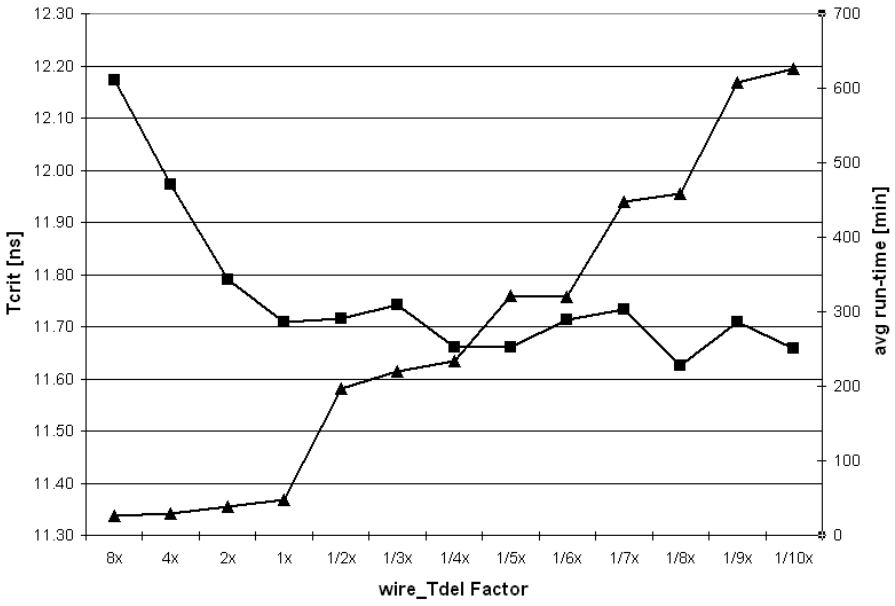

Figure B.1: Margin Interval Test Results

Performance results show a tendency to level off for intervals $\leq wire\_Tdel$. $wire\_Tdel/4$ is chosen as the final margin interval as it provides slightly better overall results than $wire\_Tdel$ and still has a reasonable worst-case run-time (clma = 15h44m). $wire\_Tdel/8$ was also considered, but it has a substantially higher worst-case run-time (clma = 32h23m) and after a second independent run, the timing advantage compared to $wire\_Tdel/4$ in Figure B.1 proved to by anomalous.