PARTITIONING-DRIVEN CONVERGENCE IN THE DESIGN OF RANDOM-LOGIC BLOCKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Hema Kapadia
May 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Balaji Prabhakar

Approved for the University Committee on Graduate Studies:

Abstract

The conventional methodology for computer-aided design of random-logic blocks requires several iterations of logic synthesis and layout tools, and a significant amount of manual intervention, before converging to an implementation that meets the design's constraints. This thesis proposes *Nebula*, a new design flow that uses netlist partitioning to achieve an optimal trade-off between two conflicting requirements for design convergence: accurate wire-load modeling in synthesis and merging incremental netlist optimizations into layout. The flow iterates between low-level synthesis, layout and re-partitioning optimizations. An experimental prototype, which emulates the *Nebula* flow, was developed to explore its viability and limitations.

A partitioning scheme, implemented in the prototype system, accurately models global wires during synthesis, while allowing room for incremental optimizations in local logic. Experimental results show that in spite of overheads of optimizing a partitioned netlist, accurate wire-load modeling narrows the gap between post-synthesis and post-layout timing in the prototype system. This leads to a faster design convergence as opposed to the conventional methodology.

A timing-driven repartitioning heuristic was implemented, which improves post-layout timing by reducing the contribution of global wire-loads along critical paths. Results show that relocating 1% of all gates in a design prunes out 80% of its critical paths. However, the scope of worst-case timing improvements is limited by the ability to merge relocations into the existing layout.

Acknowledgments

My journey to this dissertation was made possible by the guidance, encouragement and support of several people. I would like to take this opportunity to thank them.

I am grateful to my advisor, Prof. Mark Horowitz, who gave me the dream opportunity to join Stanford University as his research assistant. Working with him for the last seven years has been an enriching experience. I hope to have imbibed a small part of his ability to see right through the crux of a problem, and the "let's take a step back" attitude. I would like to thank him for his guidance and patience, for providing generous resources, and for setting high standards.

My associate advisor, Prof. Giovanni De Micheli, enabled my transition into the field of CAD research. I am thankful to him for his guidance and encouragement, for introductions to industry personnel in this field, and for providing unlimited access to his office library. Thanks to Prof. Kunle Olukotun for participating in my oral defense committee. I would like to thank Prof. Balaji Prabhakar who chaired my orals committee, and generously read this thesis for approval while accommodating my deadlines in his busy schedule.

Thanks to Russ Segal and Dwight Hill at Synopsys for helpful discussions, and for providing access to the *Design Compiler* code. Thanks to Prof. Eby Friedman from University of Rochester for introducing me to the joy of reading technical papers, and to Prof. Alexander Albicki for introducing me to the joy of writing technical papers.

I am indebted to members of the Stanford FLASH multiprocessor team for rookie guidance, and for making me a "tough cookie" over the four years on the project. I learnt a lot from their combined technical genius. Specifically, I would like to thank Prof. John Hennessey and Prof. Mendel Rosenblum for their mentorship. Special

thanks to Ricardo Gonzalez, Mark Heinrich, Dave Ofelt, Dave Nakahira and Rich Simoni for their friendship and guidance.

This thesis was made much more readable after Deborah Harber's proof-reading comments. I would like to thank her for the timely help and encouragement. Thanks to Kathleen DiTommaso and Darlene Hadding for crucial administrative support.

I am fortunate to have had very smart and very friendly colleagues. I would like to thank my office-mate Jeff Solomon for the frequent exchange of strong opinions and for his help with LSI tools, .cshrc files, and all things UNIX. Special thanks to Evelina Yeung for her friendship, moral support and motivating talks about life after graduation. Jules Bergmann developed, and yes, supported the *Vex* system, which was crucial for this work. Ron Ho and Ken Mai thoroughly reviewed everything I ever published. Thanks also to Shankar Govindaraju, Luca Benini, and other fellow graduate students who made it a stimulating research environment.

I am thankful to Anju Gupta, Rajiv Kapur, Yashika Deva, Paddy-Gargi Mamtora and all other local friends who helped me keep a balanced outlook, consoled me through stressful times, and persistently asked me the "are you done yet?" question. Special thanks to Raja for never letting me procrastinate. I am thankful to Girish Dahake, Vivek Vohra, Sujeet Kumar, Raka Chakravarty, Victor Adler, Falguni Shah and Kokilaben Tirvedi for the positive influence of their friendships at crucial junctures in my life.

I owe everything I am to the upbringing my family gave me, and to their strong belief in my abilities. Thanks to my parents for letting me go far from home for better education. I am grateful for the unconditional love and nurturing of my sister Hansa who taught me how to read and write, and of my sister-in-law Pushpa. Thanks to my two brothers Hridaynath and Ashutosh, brother-in-law Harshadbhai and sister-in-law Ketkibhabhi, who always ensured of my growth and happiness. I would also like to thank my in-laws for their support.

And last but not least, I would like to thank and congratulate my husband Atul for surviving the last five years of my student life. I could not have done this without his love, encouragement, and creative perspective on technical as well as life matters.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chip design is a complex problem that needs constant innovation to keep up with increasing size, functional complexity and performance of single-chip systems. Computer-aided design tools are used to optimize the logical implementation of a chip as well as to determine the physical locations and interconnections of various components on the chip. Picking the correct logical structure depends on the physical topology, but the physical topology changes with changing logical structures. In early computer-aided design (CAD) tools this coupling between logical and physical design was not handled directly. Instead the designer went through several iterations of logical and physical design, refining the view of one with respect to the latest implementation of the other. These refinements made non-incremental changes to the design description, resulting in a flow that sometimes failed to converge to an implementation that met the design's constraints. To aid convergence, the designer had to manually intervene by making small changes to the logical or physical implementation, or by changing the specification of the design in order to help the chip through the tool flow. As design complexity has increased, this lack of convergence in the conventional design tools has become problematic and new tools that bridge logical and physical design are starting to appear. This thesis explores some of the issues that need to be addressed to fully integrate logical and physical design.

This work in CAD methodology grew out of our experience with a large chip-design project at Stanford University. Our team of six students designed a 2M transistor

ASIC called MAGIC in a $0.5\mu m$ standard-cell technology. Chapter 2 presents both an overview of the computer-aided design flow used on MAGIC and the tools involved at various stages of the chip-design process. Trouble at the lowest level of the conventional CAD tool flow, which consists of time-consuming iterations between logic synthesis and layout of each random-logic block before arriving at its final gate-level implementation, motivated this work. Initial logic synthesis steps in this flow use approximate wire-load predictions before the layout is determined. Inaccuracies in these predictions result in new timing paths after layout that were not foreseen by synthesis. Later synthesis steps optimize the logic along these timing paths by making small local changes to the netlist using accurate wire-load information from the existing layout. However, layout tools have a limited ability to incorporate such netlist changes without perturbing the locations of the unchanged logic. Such perturbations end up nullifying the wire-load assumptions made by local synthesis refinements, resulting in another set of unforeseen timing paths. There is no guarantee that an iteration of synthesis, followed by layout, will lead to incremental improvements in the netlist. Hence, there is no upper bound on the number of iteration required in order to arrive at an implementation that meets the design's performance target. This problem is widely referred to as a lack of design convergence (or closure), or more specifically as a lack of timing convergence (or closure). While ideally the design flow should have no iterations, the industry average today is 8 iterations for a typical design to reach timing closure [1]. Over-design is an expensive workaround often used, where the performance target of the design is significantly lower than what can be achieved with its underlying technology.

This lack of convergence in the conventional CAD flow is expensive due to the amount of design effort and resources required to meet a design's performance and time-to-market targets. Hence, bridging the gap between synthesis and layout tools has been an active area of research in recent years. Several techniques have been proposed for improving convergence in the conventional flow through incremental netlist refinements [2], [3], [4], [74], [6] [7]. Also, new design flows have been proposed that change the nature of interaction between synthesis and layout to make iterations between the two more incremental and productive [8], [9], [10]. Another methodology

is being explored to turn the optimization flow around in such a way that it does not require iterations [11], [12]. Chapter 3 provides an overview of these techniques.

The discussion about related research will lead to the architecture of *Nebula*, our proposal for a CAD flow that targets design convergence by merging logic synthesis and layout in a unified tool. This flow is presented in Chapter 4. Low-level synthesis and layout transforms are iterated in *Nebula*, to keep a consistent view of wire-loads and timing throughout the optimization flow. Timing is checked after each low-level synthesis change is picked and merged into the current layout, before incorporating the change into the netlist. *Nebula* employs netlist partitioning to reach a middle ground between two conflicting requirements: accurate wire-load modeling and incremental optimizations. Netlist partitioning creates regions of placement consisting of small logic clusters. Low-level synthesis decisions are iterated with cluster placement while maintaining soft cluster boundaries that have irregular shapes and allow movement of logic across them. Wire-loads are estimated accurately and updated consistently by using the exact placement of the clusters and the approximate regions of placement of individual gates within clusters. If incremental changes made by synthesis change the sizes of clusters, cluster placement is updated along with the wire-load of inter-cluster nets. Incremental changes made to local logic within each cluster are absorbed by the layout without affecting any wire-loads, because local wire-loads are modeled based on the area of the clusters and not on the exact placement of individual gates.

Several challenging problems need to be solved before a CAD system can be built based on *Nebula*. While netlist partitioning creates accurate wire-load models in this flow, it also reduces the optimization space of synthesis and layout algorithms. A key challenge is to find a partitioning scheme that achieves the best trade-off between predictability and optimization space. Partitioning a netlist early in the design flow pushes the prediction problem from synthesis into partitioning, where wrong logic may cluster together due to lack of accurate physical information. A post-layout repartitioning scheme needs to be added to this flow to improve timing by relocating logic across partition boundaries. A prototype design methodology that emulates the *Nebula* design flow was created using conventional synthesis and layout tools, to serves as an experimentation platform that allows us to study the viability and limitations of

Nebula for improving design convergence. Implementation of the prototype system is presented in the second half of Chapter 4. Chapters 5 and 6 use the prototype system to experiment with our solutions to some of the challenges of the *Nebula* design flow.

Chapter 5 explores a hybrid wire-load model that uses a combination of cluster placement and areas of individual cell-placement to derive accurate estimates for global (inter-cluster) wire-loads and approximate estimates for local (intra-cluster) wire-loads. Wires that are likely to be modeled inaccurately by approximate wire-load models are identified from the netlist structure, and the netlist is then partitioned along those wires. Since synthesis transforms can now use accurate estimates of global wires, such partitioning maximizes the impact of this wire-load model by applying it to the "problematic wires" in the netlist. Next, the maximum cluster size is derived, with the objective to allow incremental layout while keeping approximately modeled local wires predictable. This objective minimizes the impact of potential inaccuracies in wire-load predictions on netlist timing. Comparing these partitioning and wire-load modeling solutions with the conventional methodology, experimental results show that making wire-loads predictable and widening the scope of incremental layout through netlist partitioning enables faster convergence to the desired final design implementation. However, the experiments also show that the lack of soft cluster boundaries in the prototype system results in area and timing overheads due to reduced optimization spaces of synthesis and layout tools.

As a first step towards creating soft partition boundaries, Chapter 6 presents a heuristic for moving logic across partitions based on post-layout timing analysis. The heuristic identifies gates that add significant global wire-length to many timing-critical paths due to the physical locations of their parent clusters, and relocates them to another cluster that appears just before or just after those gates along many critical paths. Experimental results show that while the heuristic gives marginal improvement in the worst-case delay, it significantly reduces the total number of critical paths in a design. This heuristic is also applicable within the context of the conventional methodology as another post-layout optimization technique for helping designs meet timing targets at a late stage in the CAD methodology, without having to resort to manual intervention or iterations with synthesis.

Finally, Chapter 7 concludes this thesis. The key observation is that accurate wire-load modeling and incremental layout capability in the *Nebula* design flow reduces the gap between synthesis and layout views of wire-loads and timing. Adding a timing-driven repartitioning step further helps convergence in this flow by correcting early partitioning decisions based on the latest timing. However, the scope of all post-layout optimization techniques is limited by their ability to merge changes into the existing layout. Future research needs to explore using several different techniques together to get the most benefit from them. An important area of future research towards design convergence is creation of synthesis and layout tools that optimize a partitioned netlist without incurring any overheads, by the treating partition boundaries as soft boundaries.

Chapter 2

Background

Our work in CAD methodology grew out of a computer architecture research program at Stanford called FLASH, which explored building a large-scale distributed-shared-memory multiprocessor. The core of the FLASH machine was MAGIC, a 2M transistor ASIC which acted as the memory controller for each node in the multiprocessor [13]. The design of this chip used a conventional CAD flow which consisted of initial floorplanning, global routing and timing estimation at the chip level, followed by logic synthesis, detailed placement and routing within each random-logic block at the gate level. This chapter first presents the conventional chip-level CAD flow in the context of large designs such as MAGIC, and next focuses on the methodology for gate-level implementation of individual blocks. The optimization flow of the gate-level CAD methodology is described, including a summary of individual CAD tools that make up its building blocks. This discussion will show a lack of convergence in this flow stemming from loose coupling between logic synthesis and layout, and also from non-incremental optimizations that result in an inconsistent view of wire-loads among the various steps of the flow.

2.1 Design of the MAGIC Chip

Design of the MAGIC chip started with the system architecture of FLASH, a FLexible Architecture for SHared memory. FLASH is intended to provide a common hard-

ware platform on which the relative merits of different memory sharing protocols are studied [14, 15, 16]. Each node of the FLASH machine contains: a high-performance off-the-shelf microprocessor used as the compute processor, a portion of the machine's global memory, a network port connecting the node to the rest of the machine, I/O devices, and a custom node controller chip - MAGIC. MAGIC, Memory And General Interconnect Controller, handles all communication within the node as well as between the node and the network. It moves data between the components of the node, and manipulates the state of the corresponding memory locations to comply with the memory sharing protocol being executed.

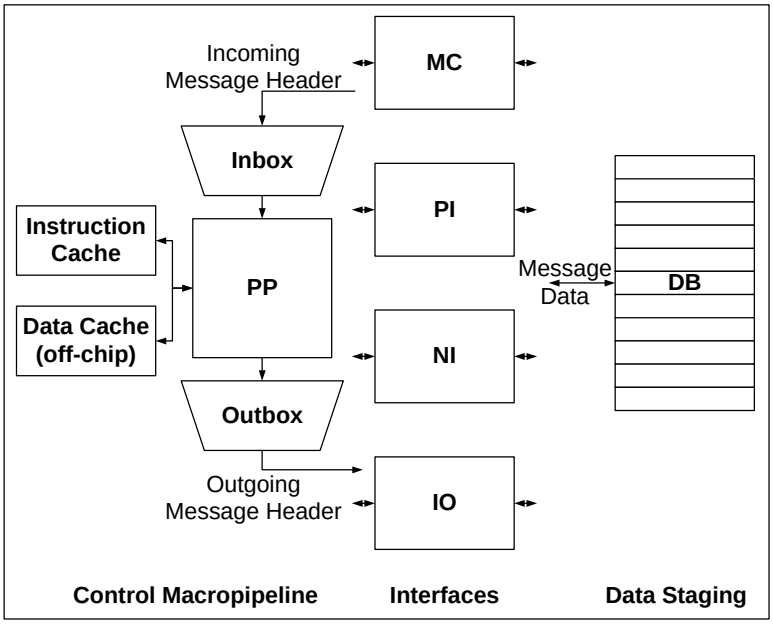


Figure 2.1: Block Diagram of MAGIC

The architectural definition of MAGIC was driven by three factors: its functional definition, the requirement for flexibility of running different memory sharing protocols, and the goal of operating at low latency and high bandwidth to ensure that the flexibility does not come at the cost of performance overhead. Figure 2.1 shows the MAGIC architecture in a block diagram. The functional definition of MAGIC called for three blocks that interface to external components of the node: a processor interface (*PI*) to the compute processor, I/O interface (*IO*) to the PCI bus [17], and a network port interface (*NI*). We added a memory controller (*MC*) block to

interface with the main memory, which is the node's portion of the global memory. Data communication between the interface blocks and manipulation of memory state are controlled by the protocol processor block (*PP*), which is a programmable RISC core giving flexibility to run any memory-sharing protocol. While the protocol code and data reside in a reserved portion of the node's main memory, we added data and instruction caches to enable high-speed access to protocol code and data used frequently by the *PP*. Furthermore, we separated data movement from protocol state manipulation to achieve low latency and high bandwidth in MAGIC. The protocol state manipulation is done by a macropipeline made up of three functional blocks - *Inbox*, *PP*, and *Outbox*. The data is copied once into a staging area called *DB* which contains sixteen data buffers, each large enough to store one cache line. Thus, the architectural definition also defined the functional partitioning of MAGIC, which consists of eight major functional blocks: *NI*, *PI*, *MC*, *IO*, *Inbox*, *PP*, *Outbox* and *DB*.

The functional blocks of MAGIC were assigned to one of six designers for implementation. Hardware description languages such as *Verilog HDL* or *VHDL* [18, 19] provide a platform for describing and simulating hardware in register-transfer logic (*RTL*) format. Alternatively, designers can begin with a higher-level description of a design using a programming language such as C or C++ [20, 21]. CAD tools are available for synthesizing such high-level descriptions into register-transfer logic (*RTL*) format described in *Verilog* or *VHDL* [22, 23, 24, 25, 26]. High-level synthesis tools schedule and bind high-level operations to resources by making trade-offs between the latency in number of clock cycles and the area of well-characterized structures, such as memories and arithmetic operations. These tools are highly effective for designing graphics and DSP systems that require converting data-manipulation algorithms, typically consisting of loops, into datapath-intensive hardware. However, for the functional blocks of MAGIC, mainly consisting of state machines that communicate with well-specified interfaces, the design challenge was not in picking the right way to implement a functionality from several different ways. Rather, the challenge was to interpret the requirements of the different interfaces and to synchronize the exchange of information between them in terms of timing and format. Hence, we designed the functional blocks of MAGIC using *Verilog*.

We described the *RTL* of all the functional blocks of MAGIC in behavioral *Verilog*, with a few structural instantiations for large datapath modules and memories. I was responsible for designing the *IO* interface. The design of my block, as all the remaining blocks, began with understanding the functional specification of all input/output interfaces - namely the PCI bus protocol, communicating headers with *Inbox* and *Outbox*, and exchanging data with *DB*. This understanding determined the datapaths, state-machines and other random-logic I needed to implement to exchange information between MAGIC and *PCI* bus by responding to MAGIC messages and *PCI* commands, while managing common resources in *IO*. Since the designers developed all blocks on the chip concurrently, the implementation of *IO* evolved with changes in the interfaces of *Inbox*, *Outbox* and *DB*.

While *RTL* development was in progress, we did chip-level planning determined the high-level look of the chip layout.

2.1.1 Chip-level Planning

Once the primary input/outputs of the chip as well as the input/outputs of all functional blocks were known, we decided locations of primary input/output pins of MAGIC as well as of each functional block. This led to a chip-level floorplan of the functional blocks. The floorplan was updated each time new implementation details about the functional blocks were made available through implementation of more functionality in the *RTL*.

For chips with many functional blocks each interfacing with many other functional blocks, *RTL* planning tools are available for chip-level planning to determine the relative placement of the blocks that gives the best area and timing. *RTL* planning derives the area, wire-load and timing constraints on the boundaries of functional blocks, and even improves on the functional partitioning to reduce the amount of global wiring on the chip [27, 28, 29]. The logic inside each block is roughly mapped to pre-characterized instances to estimate the area and timing of the block. Chip-level trade-offs are made between the area, timing and locations of different functional blocks to ensure that all blocks fit on the chip and meet the chip-level timing

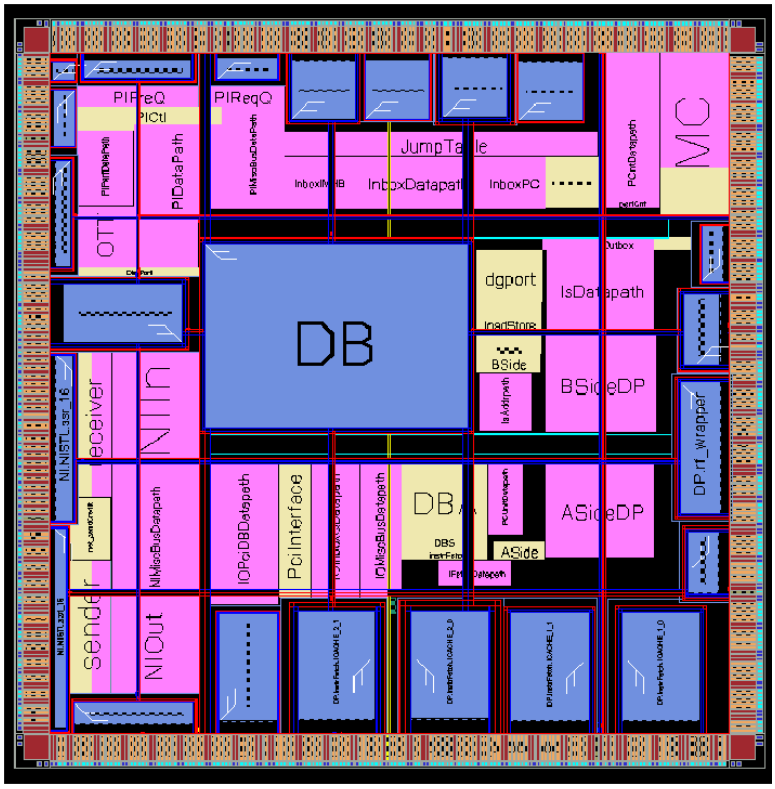


Figure 2.2: Floorplan of MAGIC

constraints. These trade-offs are made by iterating automatic floorplanning, global wire-routing and timing analysis tools [30, 31, 32, 33],

The floorplan of MAGIC is shown in Figure 2.2, where memories are shaded dark and functional blocks are shaded lighter. The memories have pre-determined size and geometries, making them hard macros. The rest of the logic is made up of soft macros with sizes and geometries that were determined by the floorplanner. The fabrication technology put several constraints on the placement of hard macros. Large designs with such constraints in the floorplan are not handled well by the floorplanning tool of the layout system used for MAGIC. Therefore, we manually floorplanned the chip. Each designer periodically synthesized their partially implemented blocks into logic gates to keep track of their latest size and timing characteristics. The timing constraints and capacitive loading of global nets on the boundaries of the major functional blocks were determined by iterating chip-level floorplanning with manual

exchange of block-level timing requirements between the designers.

These area, timing and wire-load characterization on the input/outputs of functional blocks were used to derive gate-level implementation constraints on the boundaries of each block.

2.1.2 Gate-level Implementation

Using the timing constraints and wire-loads of global nets derived by chip-level planning, the mostly behavioral *RTL* description of each functional block was converted to detailed gate-level implementation by each designer. It is common practice to use module compilers for non-critical or well-known datapath and memory modules [34, 35] and to use manual design for datapath or memory modules with aggressive performance targets to take advantage of circuit design techniques [36], [37]. Blocks with large regular structures such as the *PP* instruction cache, internal memories used in *NI*, *PI* and *Inbox*, and two 64-bit wide integer datapaths of the *PP* were designed using LSI's datapath and memory compiler tools. Since the *DB* had many read/write ports and was critical to achieving MAGIC's performance target of 100MHz clock-cycle, its circuit-level implementation was designed manually by two designers through iterations between circuit design, block-level floorplan, and *HSpice* simulations. In Figure 2.2, datapaths designed using the LSI datapath compiler are shaded darker than random-logic blocks. The regularity of logic makes manual design of datapaths and memories feasible. On the other hand random-logic blocks, describing state machines and other control logic, were designed using automatic logic synthesis [38] and layout [39], [40], [41] tools.

While logic synthesis converts the *RTL* description of a random-logic block into gates from a standard-cell library, layout decides the physical placement of these gates and routes nets that connect them. Standard-cell libraries give several implementations of logic gates and regular structures with pre-characterized parameters, such as timing, input loading, and output drive. We used LSI Design's $0.5\mu m$ standard-cell library *lcb500K* along with *Cmde* and *SILO* tool suits [41] for layout, and Synopsys *Design Compiler* [38] for logic synthesis of MAGIC's random-logic blocks. The logic

of each functional block was further partitioned into sub-modules by its designer for design readability and faster run-times during synthesis. Since the actual wire-load of nets was not known during logic synthesis, the post-layout wire-load of nets was predicted using models based on empirical data from previous layouts and the netlist structure. Synthesis was repeated with refined wire-load models derived from the latest layout until the layout of the block met its timing and area constraints.

The design flow of the MAGIC chip shows that each stage of the chip-design process iterates between logical and physical design. This is due to the fact that picking the correct logical structure depends on the physical topology but the physical topology changes with changing logical structures. As the design moves to lower levels of abstraction, the number of iterations increase since there are more components with unknown locations and logical structures. The next section focuses on iterations between gate-level logic synthesis and cell placement done in the conventional design flow at the lowest level of abstraction.

2.2 Conventional Design Flow

The conventional flow for standard-cell based design of random-logic blocks is shown in Figure 2.3. This flow was widely used until 1998, and is being augmented and often replaced by new flows proposed recently [12, 10, 42, 43, 39, 44, 8, 9]. The flow starts with a standard-cell library and corresponding wire-load models, and with the *RTL* description, area constraints and timing constraints of a design. The steps performed by a logic synthesis tool are shaded light and those performed by a layout tool are shaded dark.

The two tools communicate information through data files that follow standard formats. The input to the first logic synthesis step is a hardware-description language (*HDL*) file, describing the behavioral *RTL* of the design. Synthesis optimizes the design netlist and generates another *HDL* file with a gate-level *RTL* description. The layout tool determines rough initial placements of these gates. If the design does not meet timing constraints after placement, full synthesis is repeated using physical information from layout which is back-annotated on the netlist through *set_load* and

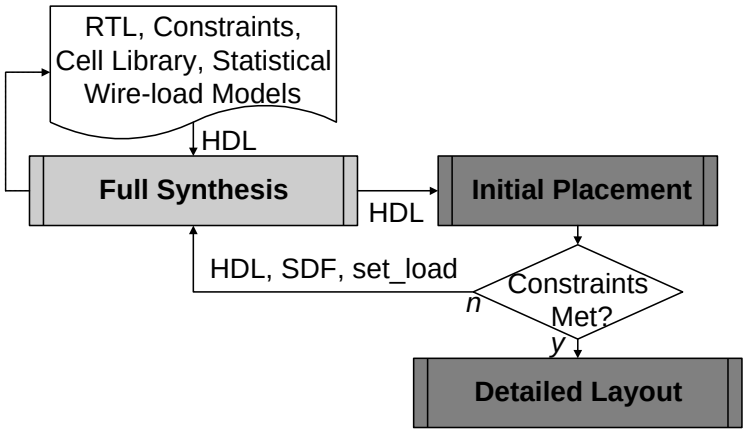


Figure 2.3: Conventional Methodology

SDF files. The *set_load* file gives the capacitive load of each net and the standard delay format (*SDF*) file gives delays through each gate and net segment. Iterations between full synthesis and initial placement continue in this manner until timing constraints are met after the initial placement, when the flow proceeds forward to detailed layout.

Both synthesis and layout tools use static-timing analysis to calculate delays in the design, both for generating *SDF* reports at the end of each step, and for identifying timing-critical portions of the design during optimizations within the tool.

2.2.1 Static Timing Analysis

Static timing analysis [45, 46] traverses the netlist graph twice in reverse topological order [47, 48] to calculate the latest arrival time and the earliest required time of each node in the netlist graph. A netlist graph consists of nodes representing gate pins and primary input/output pins of the design, and of edges representing interconnections of those nodes. Timing constraints on the primary input/outputs of the design and the desired clock speed are used to calculate the timing slack of each node, which is the difference between the required signal arrival time and the worst-case actual arrival time of the node. Nodes with negative timing slack are critical, because they violate the timing constraints of the design. Primary inputs and data outputs of synchronous logic elements such as flip-flops and latches are timing sources, whereas

primary outputs and data inputs of synchronous logic elements are timing sinks. Timing paths go from timing sources to sinks, and have timing slack equal to the slack of the sink node. Timing paths with negative timing slack are called critical paths. The most critical path decides the maximum speed at which the design can be clocked. Both synthesis and layout tools give higher priority to optimizing the timing of critical paths even at the expense of some area increase, and recover area from other non-critical parts of the netlist. Thus, it is very important for synthesis to have an accurate notion, not only of the worst critical paths of the netlist, but also of the relative criticality of paths in order to pick the right logic for timing optimizations.

During static timing analysis, gate delays are derived using pre-characterized timing information from the standard-cell library, which depends on the parasitic loading at the output of each gate. Wire-delays, as well as the output loading of each gate, are derived from the parasitic loading of wires in the design. Determining wire-loads in a layout is a matter of interpreting the physical information already available. However, wire-loads need to be estimated for timing-analysis during synthesis since the gate locations and the wire-routes are unknown. Thus, the accuracy of timing analysis during synthesis depends on the accuracy of its wire-load estimates. The next section describes wire-load estimation techniques for logic synthesis.

2.2.2 Wire-load Estimation

Before any layout is done, statistical wire-load models, provided by the standard-cell library, are used to estimate net-lengths during synthesis. Statistical wire-load models characterize all nets with the same fanout count by the same wire-load capacitance. The model provided by standard-cell library vendors is derived from the statistical distribution of post-layout wire-loads of nets in previous designs that have similar area and are based on the same library and technology. This model usually represents the median of the distribution for each fanout.

After initial layout is done, the actual wire-loads of nets are back-annotated to logic synthesis, allowing synthesis to re-optimize the netlist with more accurate wire-load models. Custom statistical wire-load models are derived from the statistical

distribution of wire-loads back-annotated from the latest layout of the design. Custom models are more accurate than library-provided statistical models, since the data used to derive the model represents the size, aspect ratio, and type of netlist connectivity of the design. Such models are refined further by trimming a small percentage of all nets from the top and bottom of the wire-load distribution in order to ignore outliers. Another refinement makes the model more conservative by picking a wire-load number that represents higher than the 50 percentile point in the distribution for each fanout. However, the inherent limitation of all statistical models is that all nets with the same fanout are represented by one number.

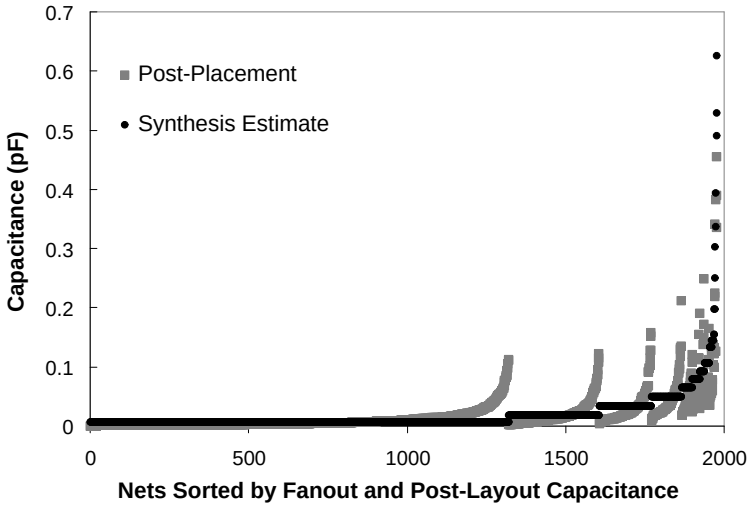


Figure 2.4: Limitation of Statistical Wire-Load Models

Figure 2.4 illustrates the error in wire-load estimation when a custom statistical model was used for all internal nets of a 7K-gate design in a $0.5\mu m$ technology. The figure compares the post-placement wire-load of nets, shaded light, with the corresponding custom wire-load estimate used by full synthesis, shaded dark. Points on the x-axis represent nets sorted by increasing fanout and then by post-placement wire-load capacitance. It is evident here that the post-placement wire-load capacitance has a wide distribution for each fanout, which is modeled by one number. Even though the wire-load of 80% short and medium-length nets is estimated either conservatively or accurately, the longest 20% nets are highly underestimated.

Two techniques have been proposed for estimating wire-loads using parameters

derived from the netlist structure [49, 50]. These techniques analyze the fanin and fanout cones of nets, to sort them by expected wire-load capacitances. Nets are separated into short, medium and long categories. This enables piece-wise linear modeling of the wire-load distribution in a design using a different statistical model for each of the three categories. These models are more accurate than fanout-based models, because each category is modeled separately within a given fanout. However, due to the steep tail of wire-load distribution for each fanout seen in Figure 2.4, these models are also inaccurate for long nets even though the estimation errors are lower than fanout-based models.

Statistical wire-load models are used by the full synthesis step in the flow of Figure 2.3. The next section describes this step.

2.2.3 Full Synthesis

A logic synthesis tool parses the *RTL* of a netlist and applies Boolean and algebraic transformations to minimize its logic [51], [52]. The netlist is usually represented in an internal graph format, where primary input/output pins as well as Boolean logic expressions make up the nodes of this graph, whereas interconnections between nodes make up the edges. Technology mapping maps this graph to gates from the standard-cell library being used, while meeting the area and timing constraints of the design.

Technology mapping partitions the netlist graph into smaller cones of logic and matches these cones against graphs representing library cells. Dynamic programming is used to pick the best set of graph matches that meet the area constraints while minimizing the delay through the design. The timing analysis uses statistical wire-load models given by the library or custom models derived from earlier layout to estimate wire-lengths of nets that connect the logic cones. The minimum unit of optimization here is a small logic cone, typically similar to the size of a few basic logic gates. Since this is the technology-dependent portion of synthesis optimizations, it is most susceptible to inaccuracies in wire-load estimates. Later in this thesis, we will look at peer research combining technology mapping with layout, and our work will

increase the size of the smallest optimization unit of this step to a larger cluster of gates in a new design flow.

Technology mapping creates gate-level representation of the design, which is sent to the layout tool for initial cell-placement in Figure 2.3.

2.2.4 Initial Cell-Placement

The gate-level description generated by logic synthesis is converted to the lowest level description in terms of silicon layers by an automatic layout tool [31, 53]. Initial cell-placement decides the relative placement of all gates. In order to meet the total area constraint and the timing constraints, the total length of interconnections between gates is minimized. Interconnection lengths between timing-critical gates are minimized while non-critical gates are placed to minimize the area. Constructive cell placement algorithms are used to give the initial locations of gates. Such algorithms determine relative locations of all interconnected gates close to one another [54, 55, 56], although these locations could be overlapping. Since the nets are not routed at this stage, wire-lengths are estimated using the placement of their pins. These estimates are used to derive placement cost functions, such as total interconnect length, and to perform static timing analysis.

The wire-length of a net can be roughly estimated as the half-perimeter of the bounding box enclosing all pins of the net [57]. While this estimate is fairly accurate for nets with three or fewer pins, a more accurate estimate is needed for higher fanouts. Higher fanout nets can be estimated accurately by drawing a spanning tree that connects all of its pins once [58]. A spanning tree estimates a realistic route of the net, while ignoring the effect of other nets and of the potential area congestion that could alter the actual routing.

After initial placement, static timing analysis gives more accurate timing information about the design compared to the timing estimated by full synthesis. Due to the inaccuracy of statistical wire-load models used by full synthesis 2.4, a design that meets timing constraints after full synthesis does not always meet timing constraints after initial placement. In that case a custom statistical wire-load model is derived

by back-annotating net-lengths extracted from initial placement. Full synthesis is repeated with the custom wire-load model, followed by another pass through the initial placer. Neither full synthesis nor initial placement use the output of the previous pass of the tool as an initial solution, rather they start from scratch each time. Hence, both are unable to preserve the wire-load model used by earlier passes. Such non-incremental iterations between full synthesis and initial placement are repeated until the the design meets timing constraints after initial placement, when it is taken to the detailed cell-placement step.

2.2.5 Detailed Cell-Placement

Detailed placement takes the rough initial placement as a seed solution and optimizes it further by using a corrective placement tool. Corrective placement algorithms use random cell movements to improve this rough initial placement for a given objective function such as area or total length of interconnections or both [54, 59], while ensuring legal (non-overlapping) locations for all gates. Cells in a standard-cell library usually have the same height but different widths, so legal cell placement consists of rows of uniform height cells separated by channels of space kept open for routing. Significant changes are made to the initial placement, changing the wire-lengths of several nets. Since constructive placers use random algorithms, the fanout-based custom wire-load model used by full synthesis can not predict these changes in wire-lengths.

The longest nets with underestimated wire-loads, which appear in the tail of the wire-load distribution for each fanout in Figure 2.4, affect design convergence in several ways. Synthesis picks the wrong logic structure to drive such a net, resulting in insufficient drive after placement. Since timing is a worst-case metric, such nets end up on critical paths after placement. Also, synthesis has the wrong notion of relative criticality of paths. Hence, placement may reveal new critical paths that are not well optimized for timing, whereas other relatively non-critical paths may be unnecessarily optimized by synthesis. The post-placement timing of the design may have higher worst-case delay, more critical paths in violation of the timing constraint,

and some non-critical logic that has been over-optimized by synthesis. Thus, even if the design met timing after initial placement, there is no guarantee that the timing would be met after detailed placement.

Sylvester and Keutzer show that the delay of medium-length wires is not likely to increase in comparison to gate-delay as technology scales, hence the impact of such wires on design timing is not likely to increase [60]. However, Ho *et al.* [61] predict an increase in the number of long wires on a chip as well as in each module as higher integration will motivate synthesizing larger design blocks to manage chip complexity. They also show that as technology scales, long wires will have increasing impact on design timing since an average library gate will be able to drive shorter and shorter wires before the wire-delay becomes a significant fraction of the gate delay. This will make the tails of the wire-load distribution, such as the one shown in Figure 2.4, even longer; escalating the problems created by the inaccuracy of statistical wire-load estimation.

Timing-driven placement algorithms use static-timing analysis to sort gates and nets in the design by the relative criticality of timing paths going through them. Higher preference is given to optimizing the layout of more critical gates and nets in this list [62], [63]. Such algorithms refine the cost function of placement algorithms by introducing critical-path timing information in it, but they do not help design convergence in this flow. This is due to the fact that the relative criticality of gates and nets is not updated while changes are begin made to the layout. Hence, timing-driven layout algorithms also suffer from the problem of estimating the timing too early and not keeping it consistent with the latest layout.

Thus, more logic synthesis is required to fix timing violations of new critical paths that appear after detailed placement.

2.2.6 Reoptimization and Layout-Merge

The design flow of Figure 2.3 needs to be extended in order to fix critical paths appearing after detailed placement due to inaccurate wire-load estimation during earlier synthesis. This extended flow is shown in Figure 2.5. The reoptimization

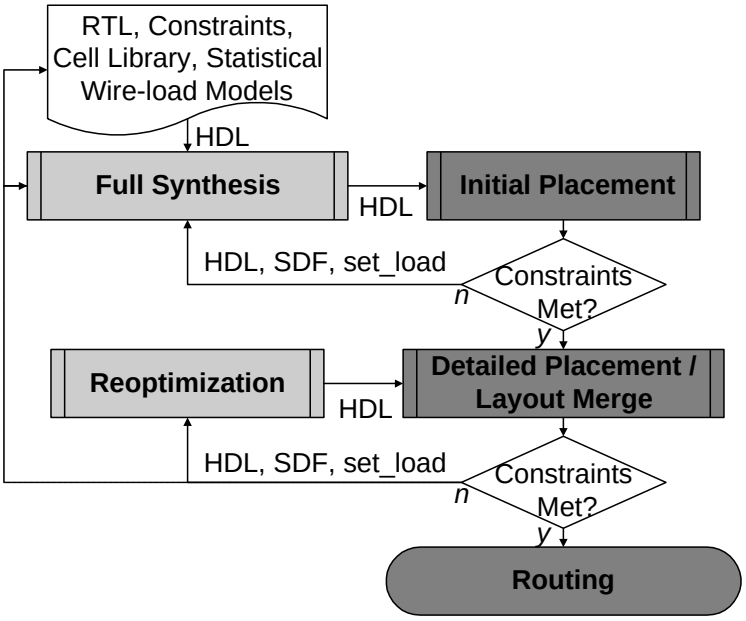


Figure 2.5: Conventional Methodology with Post-Layout Optimizations

step performs limited synthesis transforms to fix timing violations while preserving the layout that was committed thus far. Such transforms consist of driver sizing, buffer/inverter insertion and re-structuring of logic along timing-critical paths. Recent research on these and other post-layout timing-optimization techniques will be discussed in Chapter 3. Accurate wire-loads derived from estimated routes in detailed placement are back-annotated to synthesis, hence the initial timing analysis during reoptimization gives accurate timing information corresponding to the current layout. Back-annotated wire-loads are maintained on all nets with no changes or in-place changes in their fanin or fanout logic. In-place changes typically consist of exchanging gates in synthesis that can be similarly exchanged in layout because the two gates have the same physical footprint. This is usually done when a gate is replaced by a different drive-strength implementation of the same logic gate. However, nets with changed logic on their fanin or fanout are represented by a custom wire-load model derived from the initial back-annotation, since synthesis does not know where the new gates will be placed.

Netlist changes made by reoptimization are merged with the existing placement in

the layout-merge step, using an engineering change order (ECO) placement tool that performs incremental layout [64], [65]. ECO placers nudge cells around their original locations to accommodate new or upsized cells. The successful implementation of an ECO depends on the number and amount of changes attempted, and also depends on the area-utilization of the original placement. Often the addition of many new gates and fragmentation of layout area requires random placement of previously placed gates, which nullifies the wire-load back-annotation used by reoptimization to pick its transforms. Hence, maintaining incremental layout changes limits the amount of optimizations that can be allowed during reoptimization.

Iterations between reoptimization and layout-merge are repeated until the layout meets timing constraints, at which point it is sent for wire-routing. Timing problems discovered after routing are usually fixed by in-place sizing and manual buffer insertions, since the amount of detail committed in layout does not allow for significant changes in the netlist.

This design flow has an unfortunate property; as more accurate information becomes available through increasing detail in layout, less freedom is available in synthesis to use it. Reoptimization is very effective in pruning out timing problems by making small changes to the netlist at a late stage in the design cycle. However, in tightly constrained designs such limited optimizations are not sufficient to fix timing problems that are created due to wrong logic structures picked by earlier full synthesis. Netlist-level granularity of tool iterations, inaccuracy of statistical wire-load models and non-incremental nature of layout tools result in an inconsistent view of wire-loads between synthesis and placement throughout this flow. Hence, there is no guarantee that one iteration in Figure 2.5 will give better timing than the previous one. When these iterations do not converge, either the design is taken to a previous stage of the flow, or the designer manually makes changes to the layout or the netlist in order to arrive at the final design implementation. Either of the two solutions costs a large amount of designer effort and time.

Addressing the limitations of this methodology has been an active area of research in recent years. The next chapter covers related research in the area of bridging the gap in the wire-loads and timing viewed by synthesis and layout.

Chapter 3

Bridging the Gap Between Synthesis & Layout

Deep-submicron fabrication technologies have led to the integration of entire system-on-chip (SoC), with several millions of gates and aggressive performance targets. Shrinking device sizes and increasing chip complexity has led to new design challenges such as controlling power dissipation, maintaining signal integrity along long wires, and achieving high performance. All of these tasks must be completed in short time-to-market schedules in order to be competitive. Increasing design complexity directly translates into increasing requirements on design automation tools. CAD techniques, for predictable convergence to an implementation that meets the design's constraints, have been a topic of active research in recent years.

A CAD flow is convergent if the design meets its constraints, such as area, timing, power dissipation, signal integrity rules, etc., within a known maximum number of iterations. If the design constraints are too tight, an implementation that meets all constraints may not exist, or may not be achievable in a given CAD flow. In such cases, the CAD flow is convergent if, within a known maximum number of iterations, it arrives at a best-effort (not necessarily optimal) design implementation with some constraint violations. Such designs would require relaxing some of the design constraints, in order to meet time-to-market constraints. Our work focuses on timing convergence, which is the problem of design convergence restricted to two

conflicting constraints: timing and area. Chapter 2 showed that the conventional CAD flow lacks timing convergence since there is no upper bound on the number of netlist-level design iterations needed before design constraints are met, or before it is known that future iterations would not improve constraint violations. This chapter presents related research which addresses this problem, and the next chapter presents our approach.

Keutzer *et al.* [66] described the impact of deep-submicron technology on large scale designs, and on the future of the design automation methodology. While their paper does not describe implementation details, it does propose various high-level alternatives for tighter interaction between synthesis and physical design that fall into three categories: synthesis-driven layout, layout-driven synthesis, and simultaneous synthesis and layout. Research work in these three areas is described in Sections 3.1-3.3. Synthesis-driven layout takes a radical approach where synthesis drives the layout instead of estimating its outcome. Layout-driven synthesis techniques make incremental optimizations to the netlist topology and layout, thus improving the implementation generated by earlier passes of synthesis and layout tools. Simultaneous synthesis and layout techniques iterate low-level synthesis and layout transforms in an incremental fashion, while keeping a consistent view of wire-loads and timing between the two and creating a convergent optimization flow. The discussion involving related research leads to our view of the ideal optimization flow that merges synthesis and layout in one integrated tool.

3.1 Synthesis-driven Layout

Synthesis-driven layout techniques budget the timing of the design during logic synthesis. These timing budgets serve as constraints that the layout tool is required to meet through placement and sizing. Instead of predicting post-layout wire-loads during synthesis, synthesis-driven layout makes the layout obey cell-delay and wire-length assumptions made by synthesis. This approach has the potential for creating a forward-only methodology that does not require iterations between synthesis and layout.

A synthesis-driven layout methodology, based on wire-planning during logic synthesis, is being developed by Brayton *et al.* [67], [68], [11]. In wire-planning, logic synthesis distributes delay among global wires and functional blocks at the chip-level and among gates and local wires at the block-level. Block-level synthesis produces a netlist with region-based gate-placement constraints. Synthesis optimizes the netlist, using timing information derived from the assumption that the layout will meet its gate-placement constraints. The placement constraints ensure that every path from a primary input to a primary output of the block are no longer than the Manhattan distance between its endpoints, which enables simple but accurate wire-load modeling during synthesis. The placement engine has to meet the gate-placement constraints, but is free to pick gate sizes in order to meet corresponding gate-delay budgets assumed by synthesis. This is enabled by a cell-library that has many different sizes for each logic gate.

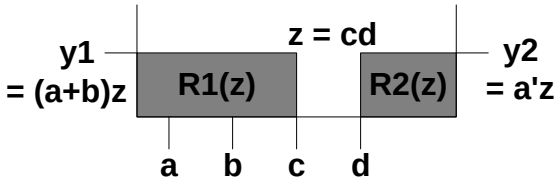


Figure 3.1: Example of Duplications Required in Wireplanning

However, if a node is in the fanin cone of two paths going in opposite directions, it has to be duplicated to ensure Manhattan lengths for both paths. This is shown in the example of Figure 3.1. Here, node z belongs to the fanin cones of two primary outputs y1 and y2. In order to ensure that the path from c to $y1$ is not longer than the Manhattan distance from c to $y1$, z should to be placed within region $R1(z)$. Similarly, for the path from d to $y2$, z should be placed in region $R2(z)$. Since $R1$ and $R2$ are non-overlapping, z needs to be duplicated. Due to such duplications, wire-planning suffers from exponential area overheads even in small designs. Its applicability will depend on future work done to address this limitation.

Magma Design Automation [12] is a startup company that has taken a similar approach. This approach is based on the work by Brand *et al.* [69] which proposed that synthesis should be done to guarantee a performance target, and that layout can

meet the same target if continuous gate-sizes are available. Magma's *FixedTiming* methodology uses a common netlist database and static timing analyzer for both synthesis and layout algorithms. Their methodology freezes the timing of the design during synthesis using logical effort [70] to estimate and budget post-layout timing. The netlist is mapped to "*superCells*" that represent the functional implementation of the logic with constant delay but variable area. Next, their placement tool performs sizing, buffer insertion, and logic restructuring; along with placement; in order to maintain gate-delay budgets by matching gate sizes to their output load. The sizes of non-critical *superCells* are reduced to make room for increase in the sizes of critical *superCells* during placement. However, logical effort does not model wire-delays that are significant in large designs and that also change with wire-length changes caused by resizing *superCells* in the fanout of long wires. Due to this, it is unclear whether this methodology scales effectively to large designs. In our opinion, the effectiveness of this methodology depends on the placement engine's ability to find avenues for making sizing trade-offs that keep the design area reasonably low, while incorporating changes in delay budgets due to logic restructuring and buffering.

3.2 Layout-driven Synthesis

As seen in Section 2.2.6, reoptimization and layout-merge are important steps for narrowing the gap between synthesis and layout timing in the conventional methodology. Research in layout-driven synthesis (also referred to as post-layout optimization) techniques focuses on improving the effectiveness of these steps. These techniques make local changes to the netlist or the layout using transforms that target a few of the worst critical paths; based on accurate wire-load and timing information available after layout. As discussed in Section 2.2.6, the scope of such transforms is limited by the ability of the layout tool to incrementally merge netlist changes without significantly perturbing unchanged portions. Moreover, converging to timing improvements through the application of these transforms also depends on the accuracy of its timing analysis engine. Section 3.2.1 describes the impact of timing analysis on the effectiveness of layout-driven synthesis transforms. Sections 3.2.2-3.2.3 describe the

research in layout-driven synthesis heuristics, divided in three categories based on their target optimization space: buffering and sizing transforms improve the drive on heavily-loaded nets; relocation and re-wiring transforms shorten either net-lengths or logic depth along critical paths; and logic re-structuring transforms resynthesize critical logic to improve timing. The techniques presented here fit very well into the conventional CAD flow, and are incorporated in most commercial CAD tools today.

3.2.1 Timing Analysis for Layout-driven Synthesis

Timing analysis is used to choose parts of the netlist for applying a layout-driven synthesis transform. Furthermore, it is used after applying such a transform to accept or reject the changes. More accurate timing analysis is computationally expensive but gives a more convergent flow of post-layout timing optimizations.

The timing analysis engine needs to estimate the capacitive loading of nets on driving gates and the propagation delay of nets. If routing is not done, the length of a net is estimated by less accurate measures, such as half-perimeter of its bounding box [57]; or by more accurate but time consuming measures, such as spanning-tree approximation of its Steiner route [58]. The propagation delay (or RC-delay) is roughly estimated using the product of lumped resistance and capacitance for a given net-length. A more accurate calculation of RC-delay is done by viewing the net as a distributed RC tree, and deriving its Elmore delay [71], [72]. The Elmore delay of a RC network represents the first moment of its impulse response.

Either the timing of the most critical path or several long paths can be considered when choosing a layout-driven synthesis transform. Tightly constrained designs have several timing paths with similar timing, and sharing a lot of logic. Hence, fixing only the worst path may introduce new paths that are also affected by the transforms. A more conservative, but computationally more expensive, approach would consider several of the longest paths when choosing a transform. The complexity of enumerating all paths in a netlist with N nodes is $O(e^N)$ [47]. Alternatively, only the top K critical paths can be enumerated using the algorithm proposed by Ju and Saleh [73] which also allows for the enumeration to be stopped when a desired max-

imum CPU time is spent. Enumerating a higher number of paths before picking a transform improves the chances of getting timing improvements, without introducing new critical paths.

The granularity of interaction and the consistency of wire-load data, between actual layout and the layout-driven synthesis transform, determines whether the transform converges to timing improvements. Some physical information is lost when the topology or placement of logic on a net is changed. If several netlist or layout changes are made before analyzing post-layout timing, the interaction of different transforms may have a negative impact on the overall timing. A safe, but computationally more expensive, approach would be to accepts only those transforms that improve timing after the layout has been updated for each low-level transform.

Research in layout-driven synthesis techniques, presented below, differ in the type of transforms performed, in the accuracy of timing and wire-load analysis, and in the granularity of netlist modifications made before layout and timing are updated.

3.2.2 Buffering and Sizing

Inserting buffers as repeaters on very long nets is a very effective way of improving the RC delay of a net, while reducing the load seen by the net's driving gate. The technique by Sato *et al.* [2] derives segmented wire capacitances and resistances from the layout tool, and calculates the Elmore delay of nets for timing analysis [71]. Their technique chooses terminals on the most critical path for buffer insertion on high-fanout nets and for gate sizing. Kannan et al. [3] ignore the RC delay of wires in their timing analysis but derive accurate wire-load capacitances through detailed analysis of the fanout tree of each net. Their technique also chooses nets on the worst critical paths for buffer insertion. The layout has to be able to absorb new buffers in specific locations along a net's route in order to get the benefit of buffer insertion.

Driver sizing improves timing by upsizing those gates that significantly contribute to critical path delay due to excessive capacitive loading on their outputs. Chuang and Hajj [4] calculate the timing slack for all gates using lumped RC models for wires; and they solve a linear program to minimize the slack of critical gates and those directly

connected to them through gate-sizing and relocation. Sizing makes small changes to the layout to accommodate for differences in area of gates with different drive strengths. Usually these are absorbed in the existing layout by nudging neighboring gates away.

Buffering and sizing are inexpensive transforms that improve the drive on heavily-loaded nets, but require small amount of layout merge. Buffering is very effective in pruning out timing problems due to the parasitic loading of long nets. Sizing effectively fixes gate drives on underestimated short to medium length nets. However, their scope is limited. Inserting a buffer on local nets of a block takes up delay and area. This can be avoided if a stronger logic structure is synthesized to drive the net. Sizing is not always possible since only a limited amount of upsizing can be done on a gate before other gates in its fanin cone are overloaded.

3.2.3 Relocation and Re-wiring

Gate relocation moves critical-path gates closer in order to improve timing, by reducing the wire delay along critical paths. Chuang and Hajj [4] use linear programming to derive optimal netlist-level relocations based on timing analysis of the current layout. However, making many relocations at once before analyzing their impact on post-layout timing results in non-incremental layout changes that end up with worse timing. Instead, it would be safer to use a heuristic that interleaves low-level relocation decisions with incremental layout and timing updates.

Marek-Sadowska *et al.* [74, 5] find equivalent cones of logic in a netlist, and re-wire gates along critical paths to bypass unnecessary logic stages. Their technique does slack calculation on all gates and chooses nets on and in the vicinity of the critical path for re-wiring and other transforms. In order to consider the effect on other similar critical paths, they also consider the timing of 5 of the next longest paths going through a net as a secondary objective when choosing the net. This is a convergent technique, since every transform chosen is accepted only if it gives timing improvements after layout merge.

Their paper also shows that post-layout optimizations should be applied before

detailed routing, since it is more difficult to absorb netlist changes through ECO when all the details are already committed in the layout. They show that applying re-wiring and buffer insertion together, followed by gate sizing, gives better timing improvements rather than applying any of the three techniques separately. So far, the different layout-driven synthesis techniques exist as individual tools in commercial layout and synthesis systems. There is no one post-layout optimization tool that takes the advantage of simultaneously applying buffering, sizing, relocation and re-wiring.

3.2.4 Logic Restructuring

Logic re-structuring is more effective for timing improvements than other post-layout optimization techniques due to its wider scope. Post-layout logic restructuring of gates on and close to the critical path is shown to improve timing by Lee *et al.* [6] and Stenz *et al.* [7]. Both of these techniques choose logic on and around the most critical path in a design, and re-synthesizes it to improve timing. Computational cost limits the number of critical paths being considered for restructuring. Also, the lack of incremental layout tools limit the amount of logic that can be restructured.

Layout-driven synthesis techniques are very effective in pruning out timing problems at a late stage in the design by making local changes to a design based on accurate physical information. However, these techniques improve timing by making corrective changes at a late stage in the design flow. Tightly constrained designs, with many critical paths that share a lot of logic and that have similar timing, hit the scope limitations of these techniques. Hence, post-layout timing optimizations are not sufficient in fixing the lack of convergence in the conventional methodology.

There is a need for more pro-active approaches earlier in the methodology that can achieve timing convergence by picking the right logic structures to drive the right loads.

3.3 Simultaneous Synthesis and Layout

Simultaneous synthesis and layout techniques improve the accuracy of information exchanged and the granularity of optimizations performed by synthesis and layout iterations, creating a more convergent flow starting from first synthesis. Identifying a design flow for simultaneous synthesis and layout, and finding solutions to the challenges involved in developing a CAD system along such a flow, has been the focus of our work as well as of other concurrent efforts including one company and two academic research projects. Monterey Design Systems [10] is a startup company developing a tool for simultaneous synthesis and layout. Their tool is described in more detail in Chapter 4. The two research projects are described in the following subsections.

3.3.1 Simultaneous Technology Mapping and Linear Placement

One such approach was developed by Pedram *et al.* [8], which targets small designs. In this technique, technology mapping choices are made simultaneously with placement choices; to avoid problems of wire-load estimation. In order to achieve this, a netlist is hierarchically partitioned into trees with a given maximum number of nodes per tree. For each tree, different linear placement choices are considered and two-dimensional trade-off curves for gate-area vs. required number of routing tracks are plotted. The netlist is floorplanned by selecting globally best trade-off points of all trees; the minimum area is the objective.

Next, this implementation needs to be optimized for timing. Timing analysis uses accurate wire-loads of global (inter-tree) nets derived from the initial floorplan. For each tree on a critical path, three-dimensional trade-off curves are plotted for the gate-area, number of routing tracks, and signal arrival time at the root of the tree. Critical paths are fixed one at a time by solving these three-dimensional trade-offs for new technology map, floorplan, and linear placement choices within each tree.

This flow uses partitioning to reduce errors due to wire-load estimation, by fixing the wire-lengths of global nets in the floorplanning steps and using them to do accurate

timing analysis during the timing-optimization step. However, partitioning reduces the optimization space of technology mapping and linear placement transforms from the entire netlist to the size of each tree.

Iterations between synthesis and layout are eliminated in the floorplanning step by enumerating all technology map and linear placement choices up front, before choosing the best implementation for each tree. The timing optimization step has to iterate between fixing critical paths and re-calculating the 3-dimensional trade-off curves based on new timing. Such iterations do not assure timing convergence, since fixing one critical path at a time has the inherent limitation that some fixes can introduce new critical paths.

3.3.2 Iterating Constraint Generation, Synthesis, and Floorplanning

Su *et al.* [9] presented a methodology for updating timing-constraints on cluster boundaries to enable convergence in iterations between timing-driven resynthesis and floorplanning.

This methodology uses partitioning to create timing boundaries between tightly connected logic clusters. First, a synthesized netlist is coarsely partitioned by clustering all logic that is clocked by the same source. Large clusters are broken down to a desired maximum size using a traditional partitioning scheme [75, 76]. Very small clusters are merged together using a clustering value that encourages merging those clusters that are connected strongly together through signals as well as shared critical paths [77].

Clusters in the partitioned netlist are treated as soft macros that have irregularly shaped boundaries. These clusters are floorplanned using a timing-driven soft-macro placer which determines their locations and shapes. Detailed placement is done within each macro, followed by accurate timing analysis using wire-lengths extracted from the layout. Timing constraints are tightened on the boundaries of critical macros and relaxed on the boundaries of non-critical macros. The soft macro with the worst timing slack on its boundaries is re-synthesized, and the timing-driven soft-macro

placer is run again to recompute the shapes and locations of the soft macros.

The main advantage of this methodology is its ability to update the floorplan, and also the ability to budget timing constraints on cluster boundaries based on accurate timing derived from detailed placement. Hence, it is ideally suited for accurately characterizing the floorplan and sub-module level timing constraints in very large designs that usually can not be handled flat by a synthesis tool. However, the timing constraints of a cluster are not dynamically updated based on synthesis choices made in other clusters sharing timing paths. Hence, similar to the flow proposed by Pedram *et al.* [8], the optimization space of logic synthesis and layout is restricted to individual clusters and would give suboptimal results if small designs are partitioned into even smaller clusters. Moreover, critical paths in a small design would go through several different clusters, requiring several iterations between cluster-level logic synthesis and soft-macro placement before converging on the right timing budgets on cluster boundaries. Hence, this methodology is not suitable for gate-level design of individual sub-modules of large designs.

Netlist partitiong used in the works of Pedram *et al.* [8] and Su *et al.* [9] enables accurate timing analysis based on partial layout of partitions. Furthermore, partitioning isolates parts of the netlist within each cluster; these can be changed locally without affecting the top-level timing information. This gives a powerful combination of predictable global wire-loads and the ability to make incremental changes to the design, which should be the key characteristic of an ideal CAD flow. Instead of restricting the partitions to be trees, such a flow should use a partitioning scheme similar to the one of [9] that uses the netlist topology and timing to reduce overheads in synthesis optimization space by ensuring that strongly connected logic can be optimized together. In order to assure timing convergence, the ideal flow should pick the maximum cluster size to ensure that changes in wire-loads of local nets within a cluster do not significantly affect the overall timing of the netlist.

Arbitrarily shaped cluster boundaries of [9] reduce layout-area overheads due to partitiong. The ideal CAD tool should make these boundaries even softer by allowing movement of logic across them. This would enable updates in early partitioning decisions based on the latest timing. The granularity of timing optimizations in [9] is

too coarse since an entire cluster is resynthesized before its floorplan or timing constraints are re-evaluated. On the other hand, the granularity of timing optimizations in [8] is too local since it looks at one critical path at a time. The ideal CAD tool should do global timing optimizations, while updating the layout and wire-loads after each low-level netlist change in order to ensure timing convergence. The next chapter presents our proposal for the architecture of an ideal CAD flow and describes the challenging problems that need to be solved before it is implemented as a tool.

Chapter 4

A CAD Flow Targeting Design Convergence

For predictable convergence to the final implementation of a design, a CAD flow needs to merge synthesis and layout optimizations in one integrated tool that interleaves low-level synthesis and layout transforms. The view of both wire-loads and timing should be kept consistent between synthesis and layout steps through the flow. In order to achieve this consistency, synthesis should do away with statistical wire-load modeling. Instead, synthesis should estimate wire-loads based on current and potential future co-ordinates of gates in the layout. Layout should take hints from synthesis when updating these co-ordinates and should make incremental changes that keep previous synthesis and layout choices valid throughout the methodology. After each low-level netlist change is incorporated in the layout, wire-loads as well as timing should be updated to keep a consistent view of the design. Making synthesis aware of gate locations would give highly accurate wire-load estimates, however it would make it even more difficult to do incremental layout modifications. The discussion in Chapter 3 showed that netlist partitioning can be used to address this trade-off between accuracy of wire-load estimates and flexibility of netlist optimizations.

We propose *Nebula*, a design flow that uses partitioning for timing convergence by targeting the optimal trade-off between accuracy of wire-load estimates and flexibility of incremental optimizations. *Nebula* iterates between low-level synthesis, layout, and

partitioning transforms. Each iteration is followed by an update of wire-loads and timing, in order to reject non-convergent iterations and to synchronizes the design parameters viewed by synthesis and layout subsystems. *Nebula* creates clusters of strongly connected logic, and loosely fixes the relative placement of these clusters early in the flow. It uses a new wire-load model that roughly estimates local wires within each cluster, but accurately calculates the wire-load of timing-critical global wires between the clusters. This creates nebulous regions of local areas in clusters, within which the netlist can be incrementally optimized without nullifying previous wire-load estimates that motivated those optimizations. However, the accuracy of wire-load estimation and the flexibility of incremental layout changes comes at the cost of reduced optimization space due to partitioning. We propose changing the view of partition boundaries in the flow, to make them appear as "soft" boundaries. Soft boundaries are enforced in the low-level iterations of *Nebula* by selectively relocating parts of a logic cluster to another partition. This allows the flow to correct early partitioning choices rendered sub-optimal by the timing of the latest netlist implementation.

A commercial CAD tool based on a similar flow is being developed by Monterey Design Systems [10], which is described in Section 4.1. However, implementation details of Monterey's flow are unknown and several questions remain unanswered about the challenging problems that need to be solved before such a flow is made into a CAD tool. Section 4.2 describes the *Nebula* flow we propose, and identifies the challenging problems that need to be solved in order to make each step of the flow a reality. We have developed an experimental prototype of *Nebula* using commercially available synthesis and layout tools to validate different approaches to solving these challenging problems. This prototype CAD system is presented in Section 4.3.

4.1 An Industry Tool for Design Convergence

Monterey Design Systems [10] is a startup company developing a tool for simultaneous synthesis and layout. Their *Dolphin* physical design system uses common timing engine and data structures for synthesis and layout, essentially converting the

methodology into a unified tool. This tool is implemented with parallel algorithms that take advantage of the memory capacity and processing power of commercially available multiprocessor hardware; improving the speed and scalability of complex optimizations for large designs. The tool starts with coarse wire-load estimates at the top-most level of hierarchy to make high-level design choices. These estimates are refined further as the tool partitions the netlist into smaller clusters for local optimizations. Instead of netlist-level iterations between synthesis and layout, their tool does iterations at all levels of hierarchy down to very small clusters, with increasing accuracy in wire-load estimates derived from the layout. Cluster boundaries are treated as soft boundaries, having irregular shapes and allowing logic to be moved across clusters. This gives added freedom to synthesis and layout transforms, of optimizing logic across cluster boundaries at various levels in the design hierarchy,

While the design flow in this architecture follows closely along the lines of *Nebula*, key implementation details, determining its viability, remain unpublished. For example, the criteria for partition at each level of hierarchy needs to be determined based on two factors: how the partitions are used for wire-load modeling at that level, and how flexible the boundaries are at lower levels of hierarchy. The wire-load model at each level needs to cater to the amount of detail committed in synthesis and layout at that level while taking advantage of the physical information available thus far. To decide when logic should be moved across cluster boundaries, a post-layout timing optimization technique needs to isolate the right logic for movement while minimizing unnecessary perturbations in layout. This chapter discusses these implementation challenges in the framework of *Nebula*.

4.2 Architecture of *Nebula*

Figure 4.1 shows the architecture of *Nebula*, the CAD flow we propose for targeting timing convergence during gate-level optimizations. The RTL description of the design is mapped to structural logic by initial synthesis using statistical wire-load models provided by the standard-cell library. The structural netlist is partitioned into clusters of logic gates. Initial cluster placement first estimates the layout area

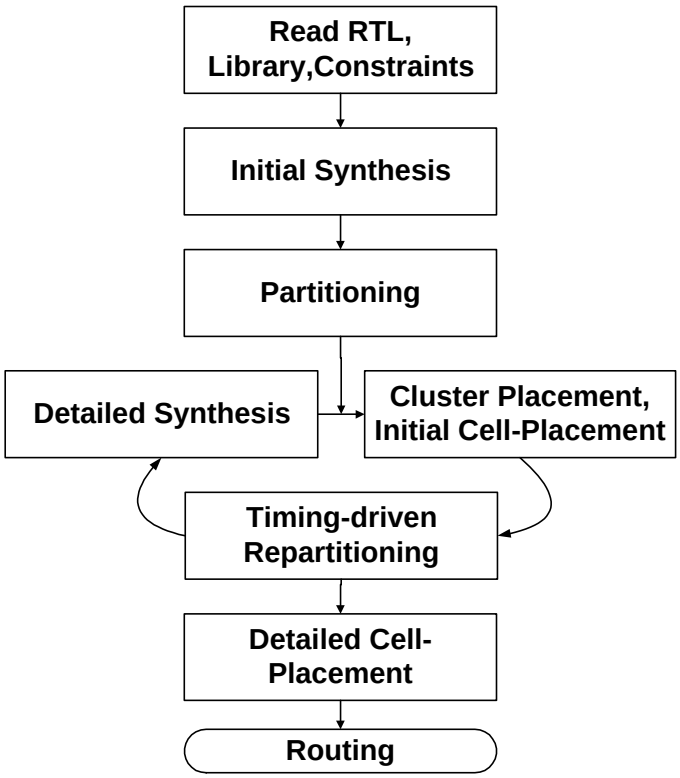


Figure 4.1: *Nebula* - A CAD Flow Targeting Design Convergence

required by each cluster. Using these area estimates, the cluster-placement step determines the relative placement of these clusters based on their interconnections and the design's timing and area constraints. Initial cell-placement determines rough initial locations of the logic within each cluster, providing data for wire-load modeling during later synthesis steps. Wire-lengths are estimated from this layout to derive a wire-load model for detailed synthesis. This model accurately calculates the wire-load of global (inter-cluster) nets based on the current cluster placement, but roughly estimates the wire-load of local (intra-cluster) nets. This keeps local wire-loads independent of detailed cell-placement within the clusters while giving accurate global wire-loads. Detailed synthesis uses this wire-load model to optimize the logic within each cluster; where the goal is to meet the design's timing constraints.

Initial partitioning is done early in *Nebula*, before accurate timing information is available. While the logic and the layout are being optimized during this flow,

some partitioning decisions may need to be revisited in light of the latest timing information. Unnecessary global wire-length may be added to critical paths due to the location of the partition containing a cluster of logic. If such logic is relocated to another partition containing gates that shares many critical paths with it, global wire-loads can be reduced along critical paths to improve timing. Such relocations would also give synthesis better control over timing optimizations, since it can optimize several timing-critical logic gates simultaneously if they are all within one partition. The timing-driven repartitioning step makes such decisions for relocating logic across partition boundaries based on the latest timing and layout.

At the core of the design flow in Figure 4.1, low-level synthesis, cluster placement and re-partitioning transforms are iterated. In order to keep wire-loads consistent between synthesis and layout, each low-level synthesis transform is followed by an update in the placement, incorporating the latest netlist changes into the layout. Such updates usually make small changes to the relative placement of clusters when the area taken by the logic inside a cluster changes due to the synthesis transform. Neighboring clusters are nudged around their original locations to accommodate for the new area or shape of such clusters. Global wire-loads are updated to reflect these changes in the cluster placement. Initial cell-placement is repeated in clusters with changed logic, area, or shape; and local wire-load estimates are updated for those clusters. Since wire-load estimations use coarse statistical models, any local changes in the layout of the clusters are automatically incremental since they do not significantly affect the wire-load assumptions made by earlier synthesis transforms. Thus, detailed synthesis in this flow has accurate global wire-loads that are consistent with the latest layout and timing of the netlist.

The last synthesis transform is thrown away and a new transform is attempted if the design does not converge closer to meeting its implementation constraints after the layout is updated and timing-driven repartitioning is attempted. Otherwise, the synthesis transform is accepted and detailed synthesis continues with the next transform. In order to avoid getting stuck in local minima, the criteria for rejecting one or a number of previous transforms needs to be determined. Initially, several synthesis transforms are allowed to occur in each iteration, before deciding to reject

them if the design does not converge after layout. As the optimization flow progresses, the number of transforms allowed before evaluating their impact on convergence is gradually lowered. When the design meets its constraints during these low-level iterations, detailed cell-placement is run within the floorplan created by the cluster-placement.

Three properties of the iterations in *Nebula* aid timing convergence:

1. Synthesis uses accurate wire-load models that are constantly updated to reflect every small layout change caused by netlist changes.

2. Any local changes in the layout within the clusters have insignificant effect on the overall timing of the design, since such changes do not affect the wire-load assumptions made by earlier synthesis transforms.

3. Non-incremental changes in the netlist or the layout do not find their way into the design, since low-level iterations that increase constraint violations are rejected.

Thus, the timing impact of each iteration is bounded by the low-level granularity of iterations, and by the hybrid wire-load model. Hence, *Nebula* promises faster convergence than the conventional CAD flow. Implementation of such a CAD system requires solving challenging problems in the underlying tools to enable such convergence through low-level iterations without incurring significant overheads. These implementation challenges are described in the following subsections.

4.2.1 Partitioning and Wire-load Estimation

A partitioning scheme and corresponding wire-load model are needed to enable accurate wire-load prediction in synthesis, while allowing room for incremental layout optimizations throughout the steps of the *Nebula* design flow.

As we saw in Chapter 2, accurate wire-load estimation is crucial to ensuring timing convergence in a CAD flow. Conventional synthesis tools estimate the wire-load of a net using either a statistical model or a back-annotated capacitance from earlier

layout. However, as seen in Section 2.2.6, gates connected to terminals of back-annotated nets can not be changed during synthesis optimizations. The wire-load estimation scheme in *Nebula* should use the relative placement of clusters to accurately model global nets that are unlikely to change significantly through the flow. Moreover, to allow incremental optimizations of local logic structures and of the layout of each cluster, local nets and local segments of global nets should be roughly estimated using custom statistical wire-load models. This calls for the synthesis engine to create a hybrid wire-load model for global nets that uses a combination of back-annotation and statistical modeling.

Conventional partitioning schemes cluster logic structures with the objective of reducing the area and timing contributed by long inter-cluster wires. On the other hand, the objective of the partitioning scheme in *Nebula* is twofold: to enable accurate global wire-load modeling and to create regions of grey area for incremental local optimizations. Hence, its should identify nets that are likely to be unpredictable and partition along those. It should also keep strongly connected logic together so incremental synthesis steps can optimize it further with fewer changes in cluster boundaries and hence fewer changes in global wire-lengths.

There are two conflicting requirements on the sizes of clusters created by the partitioning scheme in *Nebula*. Partitioning a design into very small clusters would reduce the optimization space of cluster-level synthesis and cell-layout algorithms, calling for large clusters. Moreover, the clusters need to be large enough to be able to absorb changes in their local logic structures within their layout geometry. However, since local wire-loads are approximately modeled, the worst-case local parasitics should be small enough to ensure that estimation errors on local wire-loads do not significantly affect the post-layout timing of the design. This requires the partitioning scheme to make smaller clusters.

Depending on the size of the clusters, partitioning can result in sub-optimal implementations due to restricted optimization space of detailed synthesis and cell-placement algorithms. These algorithms need to be modified to maintain a reasonable local optimization space, while preserving partition boundaries used to create accurate models of global wire-loads. The following three subsections address these

requirements of treating partition boundaries as "soft" boundaries in conventional synthesis and layout algorithms.

4.2.2 Cluster Placement

The first cluster placement decides the initial size and shape of clusters, while subsequent cluster-placement steps update those to reflect netlist and timing changes made by synthesis transforms. If the design has a few large clusters, the relative placement or boundaries of clusters are unlikely to change significantly due to low-level synthesis optimizations within the clusters. However, if the design has several small clusters, low-level synthesis transforms may require significant changes to the sizes or relative placement of the clusters. This may result in a fragmented layout area if the clusters are required to have rectangular geometries. In order to avoid overheads due to fragmented layout area, the cluster-placement engine needs to create soft cluster boundaries with irregular shapes that can be modified without fragmenting the layout.

4.2.3 Synthesizing Partitioned Logic

Conventional synthesis tools optimize each cluster of a partitioned design separately, and perform minimal boundary-optimizations at the top level. Boundary optimizations are restricted to removing redundant nets and propagating inversions. Depending on the size of the clusters in *Nebula*, this restriction can result in significant overhead in the quality of the design implementation produced by detailed synthesis, since it reduces the optimization space of synthesis algorithms. Hence for small cluster sizes, synthesis needs to treat clusters as soft boundaries across which critical logic can be optimized. However, if the synthesis engine in *Nebula* is allowed to synthesize complex gates by collapsing logic from two or more clusters, it would also need to estimate the wire-loads of new global nets created in that process.

This is shown in the example of Figure 4.2, where gates *U1*, *U2* and *U3* belong to partitions *S1*, *S2* and *S3* respectively. If synthesis with soft boundaries collapses the logic of gates *U1-3* in one gate *U4*, global nets *N4* and *N5* go away and four

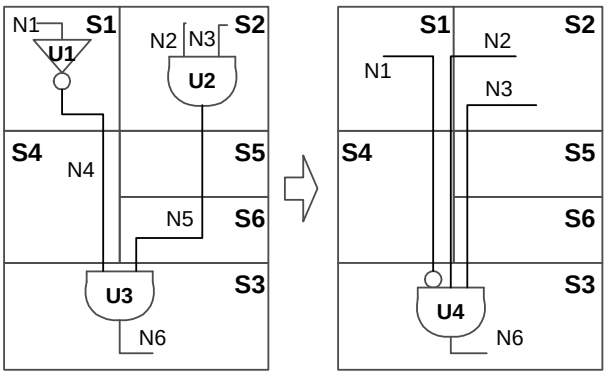


Figure 4.2: Example of Technology Mapping Across Clusters

new global nets *N1-3,N6* appear. To calculate the wire-load of these nets, synthesis would have to determine whether *U4* should be placed in cluster *S1, S2* or *S3*; and also determine the approximate pin-locations of the new global nets. This requires the addition of layout knowhow to synthesis algorithms, along with the enabling of layout algorithms to follow placement directives from synthesis. If the granularity of iterations in *Nebula* is small, the timing-driven repartitioning step could create soft boundaries by re-clustering critical logic together so future synthesis transforms can optimize it.

4.2.4 Detailed Cell-placement in a Partitioned Layout

Similar to detailed synthesis, detailed cell-placement should preserve the top-level interconnection structure of global nets to allow accurate modeling of global wire-loads. With a conventional tool, cell-placement would be done separately within each cluster. If the clusters are small, this would result in a significant loss of optimization space, giving a layout with suboptimal area and timing. For such designs, the cell-placement engine should optimize the design as a flat netlist but with region constraints on individual cells confining them within the area of their parent cluster's location.

4.2.5 Timing-driven Repartitioning

A post-layout timing optimization scheme is needed for moving logic across cluster boundaries to improve early partitioning choices in light of the current logical and physical implementation of the netlist. The scheme for timing-driven repartitioning should relocate those gates that would give the most timing improvements. It is important for such a scheme to make incremental changes to the layout and netlist in order to maintain convergence. Before accepting a relocation, it should be verified that the relocation improves timing after the relocated logic is merged into the existing layout. As discussed in Section 3.2.1, accuracy of the timing-analysis engine is a key factor in such post-layout timing optimization techniques. Hence, the timing analysis engine should accurately estimate parasitics and timing from the layout. Also, the timing of several critical paths should be considered together when making a relocation decision, to ensure that fixing one path does not worsen the delay of several other paths that share logic with it and have similar delay. Since such detailed timing analysis is exponential in the size of the design (3.2.1), the accuracy of timing analysis engine used should be chosen based on the size of the design.

As described in the previous subsections, implementation of the *Nebula* flow for gate-level optimizations requires several enhancements to the conventional tools. In this work we explored implementations of two such enhancements: a partitioning scheme and corresponding wire-load model for creating accurate wire-load estimates while enabling incremental layout optimizations, and a scheme for timing-driven repartitioning. We implemented these enhancements in an experimental prototype of *Nebula*, using commercially available CAD tools. The prototype allows us to study the viability of *Nebula* in achieving convergence through iterations of synthesis and layout optimizations. It also gives some insight into the limitations of *Nebula* that need to be addressed before it is used to make gate-level timing convergence a reality. The next section describes this prototype system.

4.3 An Experimental Prototype

In the prototype system we implemented those steps of the *Nebula* design flow that play a key role in enabling timing convergence, and used conventional synthesis and layout tools for the remaining steps. Hence, we implemented the partitioning scheme and the corresponding wire-load model that create accurate wire-load prediction while allowing incremental layout optimizations. Since it was beyond the scope of this work to develop new synthesis and layout tools that optimize a partitioned design without incurring overheads, we used conventional synthesis and layout tools for netlist optimizations in the prototype system. This prevents us from breaking down the netlist-level iterations between synthesis and layout, supported by conventional tools, into lower level iterations called for in *Nebula*. Thus, the prototype system only does one pass through all synthesis and layout step of Figure 4.1. We implemented a timing-drive repartitioning scheme that takes the final placement generated in the prototype and performs incremental timing optimizations by relocating logic across partition boundaries.

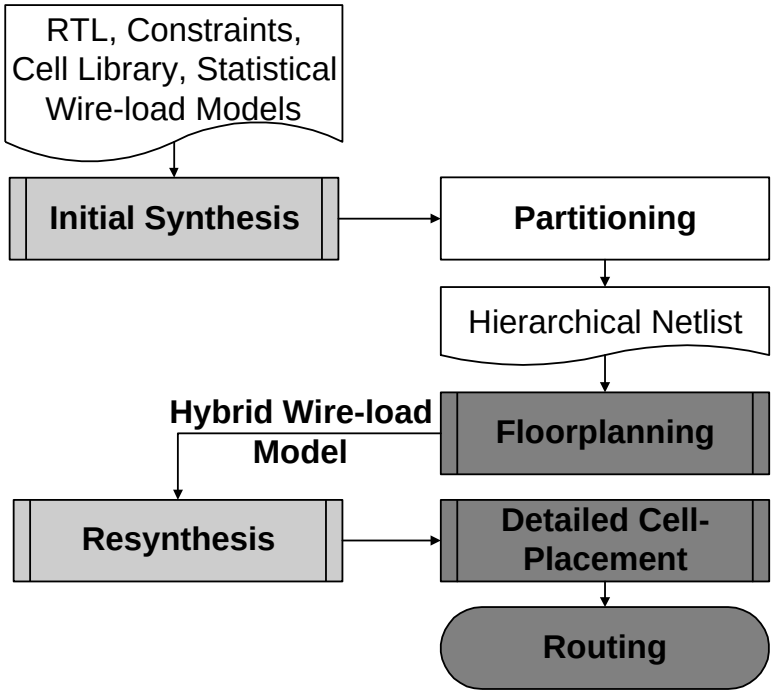


Figure 4.3: Prototype Design System

The prototype design flow is shown in Figure 4.3. We used Synopsys *Design Compiler* [38] for logic synthesis and LSI Logic's *CMDE* and *SILO* tool suits [41] for layout steps in the prototype. The technology was *lcb500k*, LSI's $0.5\mu m$ standard-cell library. The *Design Compiler* was enhanced to use our hybrid wire-load model derived from the initial floorplan of the partitioned netlist. Netlist manipulations, required for the partitioning and timing-driven repartitioning steps, were implemented in the Vex netlist database system [78], [79]. The following sections describe the steps of the prototype system and its limitations.

4.3.1 Initial Synthesis and Partitioning

The initial synthesis step optimizes the RTL description of a design with the objective to meet its design constraints, using statistical wire-load models from the standard-cell library. Partitioning creates clusters of strongly connected logic where the maximum size of a cluster is small enough to allow statistical wire-load modeling on local wires. This partitioning scheme will be discussed in detail in Chapter 5. Thus, this step creates a new netlist with two levels of hierarchy, where the top level has the design with instantiations of all clusters and global wires, and the next lower level has all the clusters containing logic gates and local wires.

4.3.2 Floorplanning

A conventional floorplanning tool from the LSI tool suit determines the shapes and locations of the clusters. In order to fix the wire-load of the top-level segments of global nets, the floorplanning step also determines pin locations on the boundaries of individual clusters.

Since the LSI tool suite does not have an automatic pin-assignment tool, we implemented a simple pin-assignment algorithm in *Vex*. The inputs to this algorithm are: floorplanned locations of clusters in the LSI floorplan format (.cfun), and gate-level *Verilog* description of the partitioned netlist. For each cluster, its boundary and the entire floorplan are divided into four quadrants with respect to the center of the cluster. For each input/output net of the cluster, its pin location is assigned

to the quadrant containing the geometric center of all of its fanin/fanout locations. Here, the location of a gate is represented by the center of the cluster containing it; whereas the location of a primary input/output pin of the block is represented by its pre-defined pin location. Pins within a quadrant of a cluster are arbitrarily sorted and assigned equi-distant locations along the two edges of the quadrant.

Neither the LSI floorplanner nor the pin-assignment algorithm are timing-driven. This makes the timing-driven repartition step quite important for converging to a design's timing constraints in the prototype system.

Next, LSI's rough initial placement tool determines the initial cell-placement within each floorplanned cluster. The placement tool is run with a area-utilization target of 55-65%. This low area-utilization allows room for making changes in the area taken by the logic local to the clusters without perturbing the floorplan.

4.3.3 Resynthesis with a Hybrid Wire-load Model

Using accurate wire-load information derived from the floorplan, one way to resynthesize the netlist would be to characterize timing and wire-loads on the boundaries of each cluster separately and to optimize the clusters independently. However, this would restrict the optimization space of synthesis transforms to each cluster, giving suboptimal synthesis output. Instead, the partitioned netlist is resynthesized as a whole to allow the synthesis tool to make trade-offs in the boundary conditions of each cluster while making local synthesis choices within each cluster. This also allows the synthesis tool to optimize logic across partition boundaries as much as possible in the tool. While the Synopsys *Design Compiler* does not modify boundaries of logical partitions, it does limited optimizations across boundaries through inversion propagation and removal of redundant signals. Physical information from the partial layout created by the floorplanning step is used to generate an accurate wire-load model for resynthesis. We call this a hybrid wire-load model, since it uses a combination of back-annotation and statistical modeling. This wire-load model is described in detail in Chapter 5.

4.3.4 Detailed Cell-placement

Within the floorplanned geometry of each cluster, LSI's detailed cell-placement tool is run with the target of achieving maximum area-utilization. Since the cluster sizes, determined by floorplanning, change after resynthesis; a significant amount of compaction or expansion of the floorplan may be required after the detailed cell-placement step. Compaction of the floorplan makes global nets shorter than the back-annotation used for resynthesis, resulting in some nets being over-driven at the expense of unnecessary area. However, this is more desirable than expansion, which makes global nets longer than the earlier back-annotation; resulting in new timing paths due to insufficient drive on some nets. A low area-utilization target of 55-60% in the floorplanning and initial cell-placement step guarantees that resynthesized clusters would either fit or be smaller than their floorplanned geometries. The floorplan is revisited after detailed cell-placement is done in all clusters to remove any empty spaces and overlaps. Since there was no tool for floorplan compaction in the LSI tool suit, we manually compacted the floorplan.

The correlation between wire-loads estimated by resynthesis and generated after detailed placement is expected to be high since the top-level connectivity of global nets is fixed by floorplanning, and the wire-load of local nets varies within small geometries of clusters. Thus, if timing constraints were met after resynthesis, they also would be met after detailed placement.

4.3.5 Timing-driven Repartitioning

This step improves post-layout timing by selectively relocating logic across partition boundaries. This enables the prototype flow to re-visit early partitioning and floorplanning decisions based on post-layout timing. Chapter 6 presents a heuristic for timing-driven repartitioning implemented in the prototype system, and shows its potential for improving convergence in the *Nebula* design flow through experimental results. This heuristic improves post-layout timing by reducing the contribution of global wire RC to critical-path delays. In *Nebula*, such a heuristic would also enable future synthesis steps to optimize critical logic by re-grouping critical gates within a

cluster.

4.4 Summary

This chapter presented the architecture of *Nebula*, a CAD flow for bridging the gap between logic synthesis and layout. The proposed flow aims to improve convergence to the final design implementation, by interleaving small-scale synthesis and layout decisions and by keeping wire-loads consistent between synthesis and layout. Netlist partitioning is used at the core of this approach, which separates unpredictable nets from predictable ones. Unpredictable nets are modeled accurately using partial layout information, and predictable nets are confined to small geometries for reasonable statistical modeling. The prototype design flow was developed based on conventional CAD tools to emulate key steps of *Nebula* in order to understand its viability and limitations. This prototype will be used in the next two chapters to derive experimental results that explore the effectiveness of better wire-length estimation on synthesis results, identify the overheads of partitioning a design before detailed logical and physical optimizations are done, and evaluate the effectiveness of timing-driven repartitioning for timing convergence.

Next, Chapter 5 presents a partitioning scheme and a hybrid wire-load model for the *Nebula* design flow.

Chapter 5

Partitioning for Better Wire-Load Models

Partitioning is a key step for enabling convergence in the CAD flow proposed in Figure 4.1. It enables accurate wire-load modeling during synthesis, which creates a consistent view of wire-lengths and timing between synthesis and layout subsystems. Partitioning also creates regions of grey area in the netlist that allow incremental local optimizations during low-level iterations between synthesis and layout. However, partitioning a design can reduce the optimization space of synthesis and layout algorithms by confining them to individual partitions.

This chapter first presents a hybrid wire-load model, consisting of both back-annotation from layout and statistical estimates, for accurately predicting global wire-loads while allowing incremental changes in local logic. Section 5.2 presents a partitioning scheme that enables such wire-load models. It also derives the size of individual partitions for achieving the optimal trade-off between the advantages and overheads of partitioning. The partitioning scheme and wire-load model were implemented as part of the prototype tool flow of Figure 4.3. Experimental results in Section 5.3 show the overheads of partitioning, especially when it is used with conventional tools in the prototype flow. In spite of these overheads, results comparing the prototype with the conventional methodology show that using this partitioning scheme and the corresponding hybrid wire-load model reduces the gap between logic

synthesis and layout.

5.1 A Hybrid Wire-load Model for Logic Synthesis

The hybrid wire-load model uses a combination of back-annotation from layout and statistical modeling to derive accurate wire-load estimates that also allow room for future netlist optimizations. It consists of back-annotating wire-loads from layout that separate local wires and local segments of global wires from global segments of global wires. Local wires and local segments of global wires are used to derive statistical models within each partition; whereas global segments of global wires are are back-annotated. Implementation of this model within the context of the prototype design flow is described in the next three subsections.

5.1.1 Back-annotation from Layout

Net-lengths are estimated from layout using LSI's *net-length estimator* and *delay calculator* tools. The *net-length estimator* uses Steiner-route estimates based on the floorplan and cell-placement to derive net-lengths from the layout. The *delay calculator* converts this estimate into wire-load back-annotation using the per micron wire capacitance of the *lcb500k* technology. This back-annotation is generated in the standard format accepted by the Synopsys *Design Compiler*. Global nets are broken down into segments contained within each hierarchical boundary, namely the top-level segment and several cluster-level segments. The wire-load of each segment is back-annotated separately. Local nets are treated in the conventional way, by back-annotating their total wire-load.

For example, Figure 5.1 shows a floorplan with two local nets $N1$ and $N2$, and one global net $N3$. For the purpose of back-annotation, $N3$ is broken down into a top level segment ST_{N3}, and four cluster-level segments $S1_{N3} - S4_{N3}$ corresponding to clusters $U1 - U4$. The back-annotation data shown in Figure 5.1 contains separate entries for each of these segments, and one entry for every local net.

Most commercial layout systems including the LSI tool suit do not generate seg-

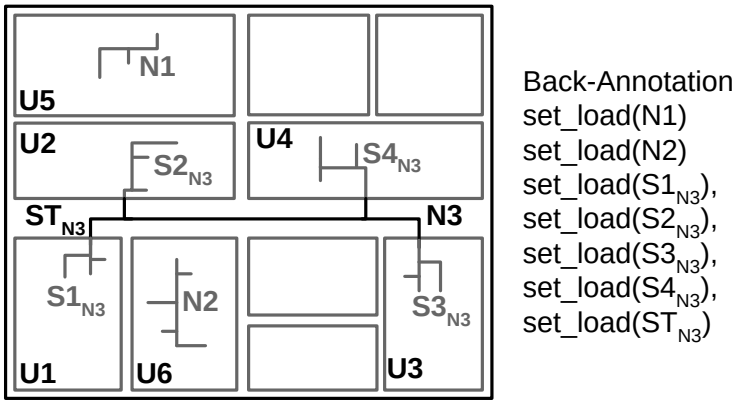


Figure 5.1: Example Floorplan for Hybrid Wire-Load Modeling

mented back-annotation for nets spanning different hierarchical levels. In order to get around this, we first extract the total lengths of global nets by running the *net-length estimator* at the top-level. Next, we extract the lengths of local segments of global nets and the lengths of local nets, by running the *net-length estimator* again within each cluster. However, the *delay calculator* generates back-annotations for only those nets that are enclosed within the given hierarchical level. Since the local segments of global nets are not enclosed within individual clusters, we create a pseudo boundary around each cluster before running the *delay calculator*. We developed a tool in Vex to automate this process. This tool creates a new design for each cluster's netlist and inserts an inverter ring around it, enclosing local segments of global nets within the cluster. Running the *delay calculator* on each of these netlists gives local back-annotation for each cluster, consisting of local nets and local segments of global nets. Next, the back-annotation for each local segment of a global net is subtracted from its total back-annotation to generate the back-annotation for the top-level segment of that net.

This back-annotation data is used to create separate wire-load models for local and global nets.

5.1.2 Wire-load Models for Local Nets

Custom statistical models are generated for each cluster using the back-annotation data of local nets and local segments of global nets present in the cluster. We use the Synopsys *Design Compiler*'s *create_wire_load* command that generates a statistical wire-load model for a module based on the statistical distribution of back-annotated wire-loads. During resynthesis the wire-load of a local net is looked up from this statistical model of the cluster enclosing the net. For the example of Figure 5.1, net $N1$ is modeled by the fanout-based custom statistical wire-load model of its parent cluster $U5$, and net $N2$ is modeled by that of its parent cluster $U6$ as shown in Equation 5.1.

$$\begin{aligned} wireload(N1) &= custom\_model\_lookup(fanout(N1), U5) \\ wireload(N2) &= custom\_model\_lookup(fanout(N2), U6) \end{aligned} \tag{5.1}$$

Custom statistical wire-load models are influenced by the size, aspect ratio and type of netlist structure within each cluster. Therefore, they estimate the wire-loads of local nets more accurately than the area-based statistical model used for initial synthesis. Moreover, the inaccuracies of statistical modeling has a smaller impact on the overall timing of the design since each cluster is made small enough to have a narrow distribution of local wire-loads. Even the wires in the tail of the post-layout wire-load distribution within each cluster are short enough to be sufficiently driven by an average library gate; hence underestimation does not result in critical timing paths otherwise caused by over-loaded gates. The maximum size of a cluster, for ensuring that underestimated wire-loads do not affect timing, is derived in Section 5.2.3.

5.1.3 Wire-load Models for Global Nets

In order to maintain consistency with layout, conventional synthesis tools do not modify any gates connected to a back-annotated net. However, this reduces the optimization space of synthesis transforms and defeats the purpose of using an accurate

wire-load model to pick the right logic structure to drive global nets. Hence, we requested Synopsys Inc. for access to the *Design Compiler* source code, and modified it as part of this work, to model global nets in a segmented fashion during synthesis optimizations. The total wire-load of a global net is calculated, by adding of the back-annotated wire-load of its top-level segment with the statistical wire-load of each of its local segments. For the example of Figure 5.1, the wire-load of global net $N3$ is modeled by Equation 5.2.

$$\begin{aligned} wireload(N3) &= back\_annotation(ST_{N3}) + \\ & custom\_model\_lookup(fanout(S1_{N3}), U1) + \\ & custom\_model\_lookup(fanout(S2_{N3}), U2) + \\ & custom\_model\_lookup(fanout(S3_{N3}), U3) + \\ & custom\_model\_lookup(fanout(S4_{N3}), U4) \end{aligned} \tag{5.2}$$

This model maintains the top-level back-annotation which remains unchanged, while incorporating wire-load changes due to arbitrary changes in the fanout of local segments of global nets during resynthesis. The partitioning scheme used in the *Nebula* flow should take advantage of the accuracy of global wire-loads provided by this model. The next section describes our implementation of such a scheme.

5.2 Partitioning Scheme

The goal of the partitioning scheme is twofold:

1. To take advantage of the accuracy of global wire-load models; by identifying unpredictable nets from the structure of the netlist and partitioning along those nets.

2. To maintain reasonable accuracy in statistical estimates of local nets; by determining a maximum budget for the size of a cluster which ensure that the post-layout measurements of local wire-loads have a narrow distribution around

their statistical median.

5.2.1 Identifying Cluster Boundaries

After initial logic synthesis, a flat design netlist is partitioned to separate nets into two categories: *global nets* interconnecting the clusters and *local nets* enclosed within the clusters. During hierarchical resynthesis, global nets are modeled accurately using partial layout whereas local nets are modeled statistically.

The partitioning scheme identifies unpredictable nets from the netlist structure and partitions along them so that they can be modeled accurately. To identify unpredictable nets, we look for nets that go to several loosely connected gates in the netlist. Since the cost functions of all placement algorithms attempt to reduce the distance between interconnected gates, loosely connected gates would be placed at arbitrary locations with respect to each other. Hence, the connecting net is more likely to fall into the tail of the post-layout wire-load distribution, such as the one shown in Figure 2.4. On the other hand, nets that go to strongly connected gates are more likely to be short, since strongly connected gates would be placed in close vicinity of each other by any placement algorithm. Such nets are made local since they are likely to be close to the statistical median of the fanout-based wire-load distribution, and hence can be modeled with reasonable accuracy using a statistical model.

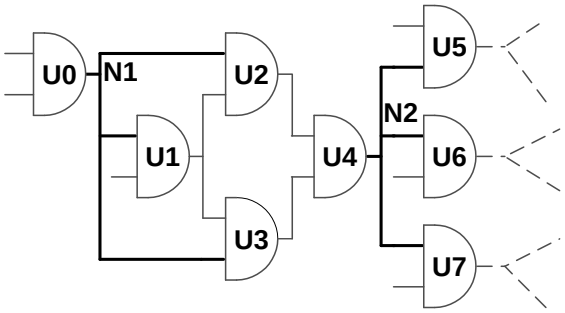


Figure 5.2: Example Netlist for *MFFC* Clustering

Figure 5.2 shows an example netlist, where statistical wire-load modeling would estimate equal wire-loads for nets *N1* and *N2*, since both nets have 3 fanouts. However, fanouts of *N1* go to gates *U1, U2* and *U3* that are strongly connected to each

other since their fanout cones converge into gate *U4*. Fanouts of *N2* go to gates *U5, U6* and *U7* that have no other inter-connection. Using any placement engine with reasonable cost functions, *U1-U4* would to be placed closer to each other, making *N1* a relatively short net. Gates *U4-U7* have stronger connections to other parts of the netlist than each other, so they are likely to be placed arbitrarily with respect to each other. This would make *N2* a relatively long net with unpredictable wire-load. Thus, the discrepancy in wire-loads between synthesis and placement would be higher for *N2* as compared to *N1*. Hence, *N2* is classified as a global net, whereas *N1* is classified as a local net.

This initial partitioning is done within the *Vex* framework using the clustering algorithm proposed by Cong *et al.* in [80] and [81], which creates maximal fanout-free cones (*MFFC*) of logic in a netlist. An *MFFC* contains gates and nets with fanouts that all end up in one sink, which is the root of the cone. Our implementation of this algorithm starts with the primary outputs as the roots of *MFFC*s and traverses the fanin cone of each primary output. It creates an *MFFC* out of the netlist in this fanin cone until it finds a gate that has outputs going to fanin cones of multiple *MFFC* roots. Such gates are made roots of new *MFFC*s and their fanin cones are further searched, until the entire netlist is covered by *MFFC*s. The complexity of this search is $O(N + E)$; where N represents the number of gates and primary outputs in the netlist, and E represents the number of nets and their fanouts. This algorithm would put net *N1* in Figure 5.2 inside a logic cone that is rooted at net *N2*. Due to many multiple fanout nets, we found that the average size of a *MFFC* was only 3-4 gates and the maximum was 20 gates. Hence, these *MFFC*s are merged to form larger clusters using the heuristics discussed in the next subsection.

5.2.2 Merging Small Clusters

*MFFC*s are merged to form larger clusters that are smaller than a desired maximum size, as shown in the pseudo code in Figure 5.3. The minimum cluster size is set to a number very close to the maximum size, since it is desirable to keep the size of each cluster large for high degrees of freedom in local synthesis and placement

optimizations. The two *while* loops merge pairs of clusters if their combined size is smaller than the maximum size. The *while* loops exit either when there are no more clusters smaller than the minimum size, or when an iteration does not give any further reduction in the number of clusters.

```
PartitionNetlist() {

  ClusterList = MFFC_Cluster(Netlist);
  old_smallClusters = 0;
  smallClusters = Num_Small_Clusters(ClusterList);

  while ( ( smallClusters > 0) && (old_smallClusters != smallClusters) ) {
    ClusterValues = Compute_ClusterValues(ClusterList);
    Sort_In_Decreasing_Order(ClusterValues);
    Merge_Clusters(ClusterValues, ClusterList);
    Collapse_Single_Fanout_Nets(ClusterList);
    old_smallClusters = smallClusters;
    smallClusters = Num_Small_Clusters(ClusterList);
  }

  while ( ( smallClusters > 0) && (old_smallClusters != smallClusters) ) {
    Blind_Merge(ClusterList);
    old_smallClusters = smallClusters;
    smallClusters = Num_Small_Clusters(ClusterList);
  }
}
```

Figure 5.3: Pseudo code of Partitioning Procedure

In the first *while* loop, `Compute_ClusterValues` computes a clustering value for all cluster pairs that share at least one input net, giving up to N^2 clustering values for N clusters. Next, pairs of clusters are sorted by decreasing clustering values. `Merge_Clusters` merges the pair of clusters with the highest clustering value, if neither cluster has merged with another cluster of higher clustering value. The clustering value for clusters i and j is given in Equation 5.3, where IO_k is the pin-count and $Size_k$ is the size (in number of gates) of a cluster k. IN_{ij} is the number of inputs shared by clusters i and j, and $FanOut_{ij}$ is the total fanout of shared inputs.

$$Clustering\_Value = \sum_{k=i,j} \frac{IN_{ij}}{(IO_k - IN_{ij}) \times FanOut_{ij} \times Size_k} \tag{5.3}$$

This clustering value encourages merging of clusters sharing many inputs with small fanout, essentially converting global nets with small fanout counts into local nets. This is motivated by the fact that a net with a large fanout count is more unpredictable before layout than one with a small fanout, because higher degrees of freedom are available in placing all the members of the net's fanout. The clustering value also discourages merging of clusters with many input/outputs that are not shared with each other, which reduces congestion on cluster boundaries. Finally, the clustering value encourages merging of smaller clusters over larger ones, resulting in evenly sized final clusters.

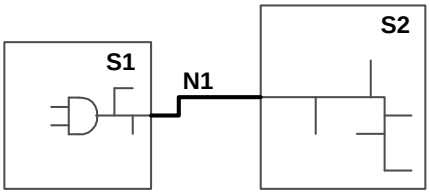


Figure 5.4: A Case For Collapsing Single-Fanout Nets

Next, `Collapse_Single_Fanout_Nets` localizes nets that have all fanouts going to one cluster by merging their source and destination clusters. Since the majority of the wire-load of such a net is going to be modeled statistically in the destination cluster, the back-annotation at the top level does not help in accurate modeling. Hence, such nets are localized. Figure 5.4 shows an example, where net *N1* in and has fanouts in clusters *S1* and *S2* only. `Collapse_Single_Fanout_Nets` localizes *N1* by merging clusters *S1* and *S2*.

If small clusters remain at the end of the first *while* loop, they are blindly merged in the second loop by `Blind_Merge`. Clusters are sorted by increasing size, and adjacent pairs are merged. This case occurs commonly for a primary input of the design that gets buffered and goes to many loosely connected gates in several different clusters. `Blind_Merge` merges such a buffer with the smallest cluster in its fanout. This is shown in the example of Figure 5.5. Here a primary input of the design gets buffered by gate *U1*, which drives several fanouts in clusters *S1, S2* and *S3*. Since there is no one obvious cluster that *U1* belongs to, `Blind_Merge` merges it with *S2*, the smallest cluster in its fanout.

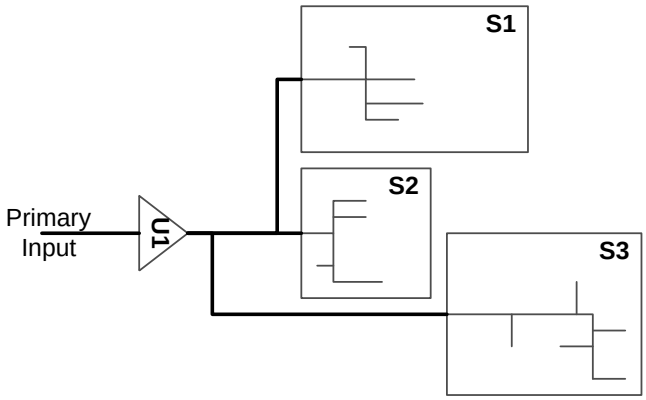


Figure 5.5: A Case For Blind Merge Of Clusters

The minimum cluster size should be large enough to reduce overheads in synthesis and layout optimizations space. On the other hand, the maximum cluster size should be small enough to allow statistical modeling of local wire-loads without having an impact on timing convergence. The next section derives this maximum cluster size.

5.2.3 Maximum Cluster Size

If the worst case wire-load of a net in a cluster does not over-load the average library gate, even the nets in the tail of the wire-load distribution are sufficiently driven by the average gate. In such clusters, statistical wire-load modeling can be applied to local nets without resulting in significant discrepancies in critical-path timing between synthesis and layout. To estimate the size of a cluster that has reasonable worst-case wire-loads, we find the longest wire an average gate can drive and then determine the number of gates that fit in a rectangle with a half-perimeter equal to that wire length.

To determine the maximum output capacitance an average library gate can drive before it gets too slow, we use the process-independent model of gate delay described by Sutherland et al. [70], [82]. In this model, the delay of a single-stage gate consists of *parasitic delay* and *effort delay*. The *parasitic delay* is due to the internal parasitic capacitance of a gate, and is independent of the its drive-strength as well as output loading. For gates with significant output loading, the *effort delay* dominates its

total delay. The *effort delay* (or gate effort) of an average library gate is given by Equation 5.4. C_{out} is the output capacitance (wire-load + pin-load), C_{in} is the input capacitance, and g is the the *logical effort* of the gate. The *logical effort* of a gate represents how much its driving capability is weakened by its transistor topology, as compared to the ideal driving capability of an inverter. The characteristics of an average library gate are derived by profiling the usage of different types of gates in various designs, and using the profile to calculate the weighted mean of the C_{in} and g of all gates.

$$EffortDelay = \frac{g \times C_{out}}{C_{in}} \tag{5.4}$$

Thus, the maximum output capacitance that can be driven by an average gate is shown in Equation 5.5. Here, *MaxEffort* represents the budget on the maximum effort delay of a gate which is not too overloaded for its size. From [70], independent of the technology, the delay through a gate is minimum when its effort delay is 4, but remains near-minimum for effort delays from 2 to 8.

$$Budget.C_{out} \leq \frac{MaxEffort \times C_{in}}{g} \tag{5.5}$$

For an average fanout of *FanOut* per net, the maximum C_{out} seen by a gate in a sub-block with a half-perimeter P is given in Equation 5.6. The first term is the maximum wire capacitance given by the product of maximum wire-length and wire-capacitance per micron of length in the given technology. Here, the maximum length of a wire in a cluster is approximated as the half-perimeter P of the cluster.[1] The second term is the total input capacitance of the loading gates, which is the average fanout per net (*FanOut*) multiplied by the input capacitance of an average library gate.

$$Max.C_{out} \leq (P \times C_{wire/um}) + (FanOut \times C_{in}) \tag{5.6}$$

[1] This approximation is accurate for nets with up to 3 fanouts, and can be up to 100% inaccurate for 4 and 5 fanouts [51]. However, this estimate can be used here without incurring significant errors; since local wires, created by the partitioning scheme of Section 5.2, have a low average fanout of 2.8.

Thus, to ensure that an average library gate is not too slow when driving the worst-case local wire-load in a cluster, we need to ensure that the $Max.C_{out}$ of Equation 5.6 is lower than the $Budget.C_{out}$ of Equation 5.5 in that cluster. This gives a budget for the maximum half-perimeter of a cluster, as shown in Equation 5.7.

$$P \leq \frac{(MaxEffort - (FanOut \times g)) \times C_{in}}{g \times C_{wire/um}} \tag{5.7}$$

For placement quality, the aspect ratio of a cluster is limited to an upper limit S, which translates the half-perimeter of the cluster to its area A as shown in Equation 5.8. For a desired cluster utilization U, and the area of an average library gate A_{gate}, the maximum number of gates in a cluster is derived from its area using Equation 5.9.

$$A = \frac{P^2 \times S}{(S+1)^2} \tag{5.8}$$

$$Max\_Num\_Gates \leq \frac{A \times U}{A_{gate}} \tag{5.9}$$

Using Equation 5.8 and 5.9 in Equation 5.7, the maximum number of gates in a cluster is given by Equation 5.10.

$$Max\_Num\_Gates \leq \frac{((MaxEffort - (FanOut \times g)) \times C_{in})^2 \times S \times U}{(g \times C_{wire/um} \times (S+1))^2 \times A_{gate}} \tag{5.10}$$

For the $0.5\mu m$ standard-cell library used in this work, $C_{wire/um}$ was $0.2fF/\mu m$. For an average library gate, the C_{in} was $0.035pF$, g was 1.34, and the gate area was $281.6\mu m^2$. The average fanout of local nets within clusters was 1.3. Using these values in Equation 5.10 with conservative budgets on *MaxEffort*, S and U of 8, 8 and 0.65 respectively, the maximum cluster size is 152 gates. With lenient budgets of 14, 4 and 0.75 the maximum size is 1019 gates. To incorporate this budget into the *Nebula* design flow proposed in Chapter 4, we use a realistic aspect ratio of 4 that represents typical geometries of layout blocks. Also, we use conservative budgets of 8 and 0.65 for the *MaxEffort* and the area-utilization. These budgets give a cluster-size

budget of 246 gates.

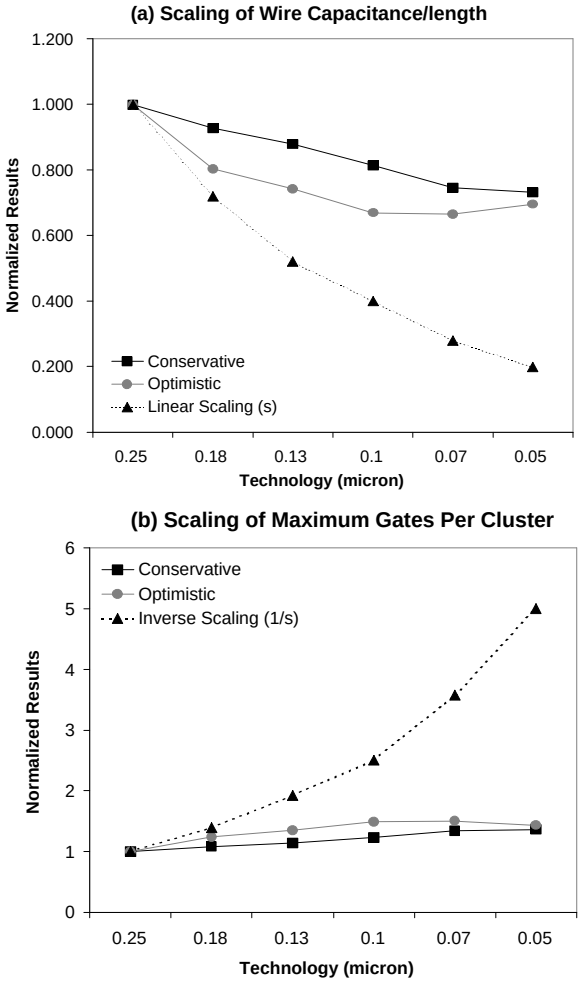


Figure 5.6: Scaling Predictions for $C_{wire/\mu m}$, and Max_Num_Gates

With technology scaling, C_{in} scales down linearly, A_{gate} scales down with the square of technology and $C_{wire/um}$ scales down very slowly [83]. Since the scaling of C_{in} and A_{gate} cancel each other in Equation 5.10, the budget for the maximum number of gates per cluster increases very slowly, inversely proportional to the square of $C_{wire/\mu m}$ scaling. Figure 5.6(a) shows both aggressive and conservative scaling predictions for $C_{wire/\mu m}$ scaling which uses corresponding scaling assumptions from the SIA road map [84], [85]. Compared to the linear scaling curve, shown in the same

graph; both the conservative and aggressive scaling assumptions give very slow scaling of $C_{wire/\mu m}$. Figure 5.6(b) shows the corresponding scaling for the maximum number of gates per cluster budget, and also shows the inverse scaling curve for comparison. It is evident here that the maximum number of gates per cluster increases much more slowly than the technology, hence the cluster-size budget can be assumed to remain constant with technology scaling. Hence, once the budget for the maximum number of gates per cluster is derived for a given library using some representative designs for data profiling, the same budget can be used for scaled down versions of that library.

5.3 Experimental Results

Table 5.1: Benchmark Designs

Design		# Gates
IO	Conv.	2164
	Proto.	2258
IOPI	Conv.	7537
	Proto.	7205
Magic	Conv.	12775
	Proto.	12043

Experiments were done on three control-logic blocks of the MAGIC chip designed by the FLASH multiprocessor team [13]. Table 5.1 shows the number of library cell-instances of the three designs (also referred to as number of gates here), when implemented in the conventional methodology (Conv.) and and the prototype system (Proto.). We collected experimental results to first derive the best partition size in the prototype system. Using this partition size, we did more experiments to compare design implementations generated by the prototype system with those generated by the conventional design flow.

5.3.1 Maximum Cluster Size for the Prototype System

The maximum number of gates per cluster given by Equation 5.10 was derived to make statistical wire-load modeling reasonably accurate on local wires, making local wire-loads more predictable during synthesis. This budget would work well in a design flow that treats partitions as soft boundaries in both synthesis and layout algorithms. However, the prototype design system uses conventional tools that treat partition boundaries as hard logical and physical boundaries during synthesis and layout, leading to reduced optimization space in both tools. The budget derived in Section 5.2.3 does not account for this overhead.

To come up with a meaningful budget for the prototype system, we did experiments on the three benchmark designs. We targeted these experiments to study the trade-offs between wire-load predictability given by small cluster sizes and optimization space gained by large cluster sizes. Using the prototype system, each design was implemented with different cluster sizes between the conservative budget of 152 and lenient budget of 1019 gates derived in Section 5.2.3. The results of these experiments for the IO design, containing around 2000 gates, are shown in Figures 5.7 through 5.9. The x-axis in these graphs represents the maximum cluster size in each experiment sorted in decreasing order. The largest size of 2K represents the conventional methodology with 1 cluster, and the remaining sizes correspond to further partitioning of the flat design with 2, 3, 5, 10 and 19 clusters respectively.

Since an ideal wire-load model would estimate synthesis wire-loads to be equal to the post-placement measurements, we did a linear regression analysis of the estimated and the measured wire-loads. Thus, accuracy of a wire-load model is represented by the correlation coefficient, which is the R^2 coefficient given by a least-square regression analysis [86]. Figure 5.7 shows this correlation coefficient for the different cluster-size budgets. Small cluster sizes create more partitions, making a higher fraction of nets as more accurately modeled global nets. Smaller clusters also give higher accuracy in local nets and local segments of global nets due to small layout geometries. Hence smaller cluster sizes give a better correlation between synthesis and post-placement wire-loads.

Figure 5.8 shows the layout overheads of partitioning. The y-axis represents values

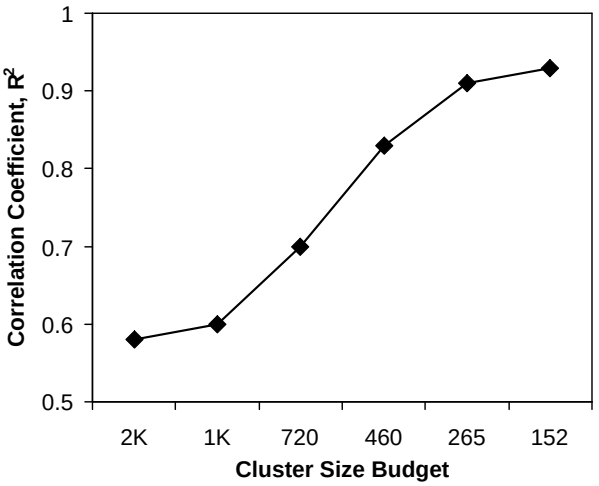


Figure 5.7: Wire-load Correlation Vs. Maximum Cluster-Size

of various results normalized with respect to those of the flat methodology, represented by the first data point of the corresponding curve. The first curve from the bottom gives average utilization of cluster area, which decreases with smaller cluster sizes because the placement tool has smaller optimization space. The total layout area shown by the next curve increases with reducing cluster sizes due to two factors: the reduced cluster-area utilization shown by the first curve, and the fragmentation of layout area. Fragmentation of layout is mostly an artifact of the layout tool we used. The LSI tool suit required all odd cluster columns to align with odd columns of the top-level layout, all columns to be of equal width, and the partitions to be rectangular regions for automatic floorplanning. The requirement for column alignments wasted one column of area in clusters with odd number of columns. After re-synthesis the sizes of clusters changed but the floorplan could not re-align cluster boundaries, due to the requirement of having strictly rectangular regions. This resulted in fragmentation of the layout area, giving poorer overall utilization. The third curve in Figure 5.8 gives total post-placement wire-load capacitance in the design. Smaller clusters give higher total wire-load capacitance due to the area overhead of partitioning, and due to a higher number of global nets created by more partitions.

Figure 5.9 shows the worst-case timing violation for a 9ns clock-cycle target. The two curves show the timing violations estimated by resynthesis, and those measured

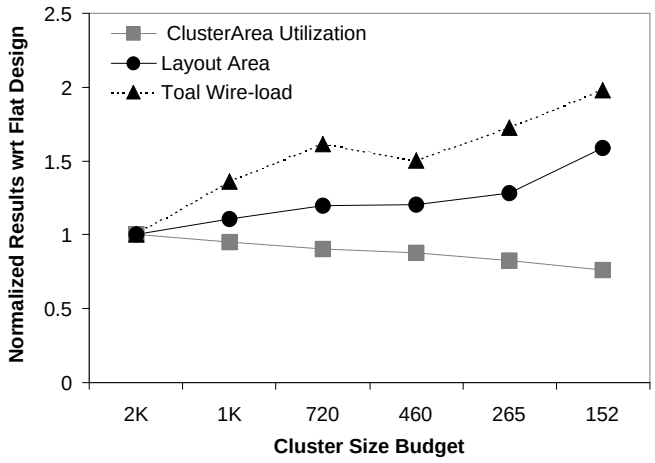


Figure 5.8: Layout Overheads Vs. Maximum Cluster-Size

after detailed placement. The post-layout timing violation determines the speed of the design. Thus, the speed of the design depends on the initial solution given by post-synthesis timing estimate, and the gap between synthesis and layout timing.

The gap between synthesis and layout timing is determined by the accuracy of wire-load approximations, layout area overheads, and characteristics of individual floorplans. Both the largest and the smallest cluster sizes have a large timing gap. With 2K gates per cluster, this gap is due to inaccurate wire-load modeling during synthesis. With 152 gates per cluster, the layout overheads discussed above result in the large gap between post-synthesis and post-layout timing. Cluster sizes of 460 and 265 combine a large optimization space for synthesis and placement with a small cluster size for high correlation between synthesis and layout; resulting in the best post-layout timing. Since the cluster size of 460 also has lower area and total wire-capacitance from Figure 5.8, it is the best size to use for the technology and tools used in this work.

The next section gives experimental results that compare the conventional methodology of flat synthesis and layout with the prototype system using the best cluster size of 500 gates derived here.

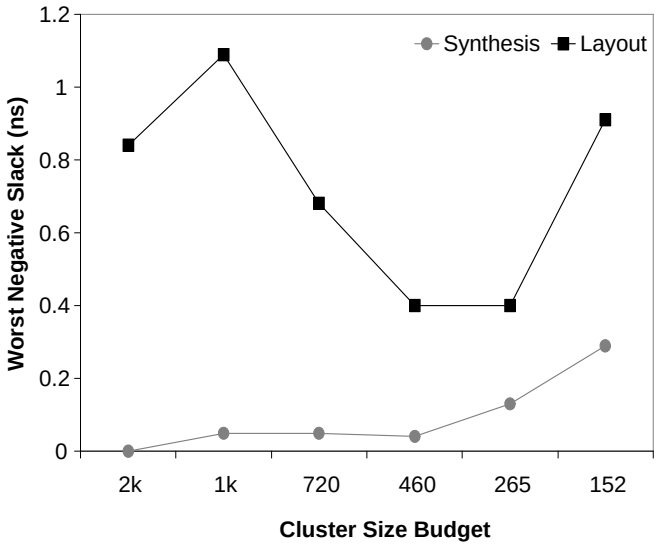


Figure 5.9: Timing Violations Vs. Maximum Cluster-Size

5.3.2 Comparison of the Gap Between Synthesis and Layout

We designed each of the three bench mark design twice: once using the prototype system and once using the conventional design flow, and compared the two implementations. The *Verilog* description of each design was synthesized in a $0.5\mu m$ standard-cell library from LSI logic (*lcb500k*). Custom statistical wire-load models were used for logic synthesis in the conventional methodology, as well as for the first flat synthesis step of the prototype system. Designs were partitioned in the prototype design flow, using the maximum cluster-size of 500 derived in Section 5.3.1. Table 5.2 shows the number of partitions created in each design.

Table 5.2: Number of Partitions Created in the Benchmark Designs

Design		# Gates	# Partitions
IO	Conv.	2164	1
	Proto.	2258	5
IOPI	Conv.	7537	1
	Proto.	7205	16
Magic	Conv.	12775	1
	Proto.	12043	27

For the resynthesis step of the prototype system, we enhanced the *Synopsys Design Compiler* to use the hybrid wire-load model described in Section 5.1. Floorplanning and placement were done using LSI Design's *CMDE* tools. The *SILO* tool suit of LSI Design was used for its ability to flatten cluster macros at the top hierarchical level while respecting the floorplan created earlier. The ability to flatten individual clusters at the top-level enables the prototype system to extract wire-loads of global nets in a design after initial or detailed cell-placement occurs within each cluster. We used the Vex netlist database system [78] to implement the remaining steps of the prototype system; such as partitioning, creation of segmented back-annotation for the hybrid wire-load model, and pin-assignment in the floorplan.

The following discussion compares various parameters between the two design flows, to illustrate the overheads and advantages of the prototype system. Results for the prototype system were taken from netlists generated at the end of resynthesis and detailed placement steps in Figure 4.3. Results for the conventional flow were taken after the first flat synthesis and detailed placement steps in Figure 2.5.

Experimental Results: Overheads in the Prototype Design Flow

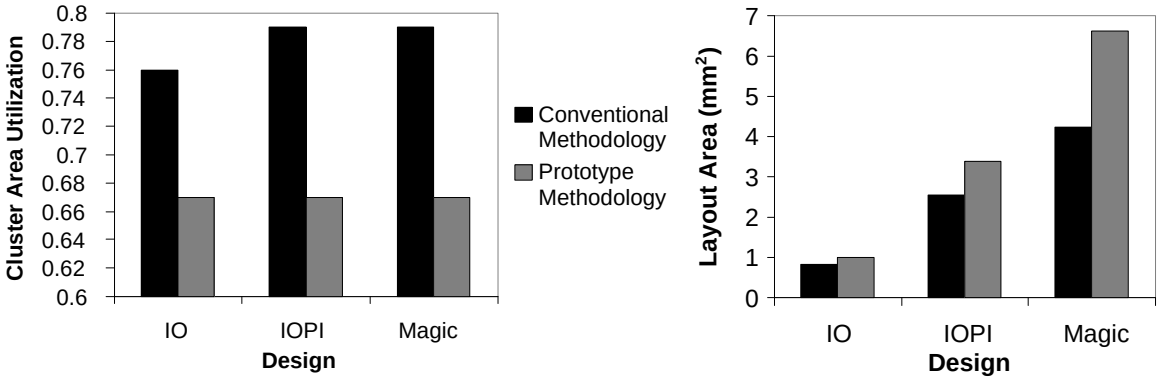


Figure 5.10: Layout Overhead in the Prototype System

Figure 5.10 compares the cluster-area utilization in the prototype system with the total area utilization of the conventional methodology, and also compares the total layout areas of the two methodologies. The three benchmark designs are arranged by

increasing size on the x-axis. The area utilization of clusters is lower in the prototype system due to reduced optimization space for placement algorithms, and remains the same for all three designs because the same maximum cluster size is used for all three. The area utilization in the conventional methodology improves for larger designs due to larger optimization space for placement algorithms, and it tops off at 0.79.

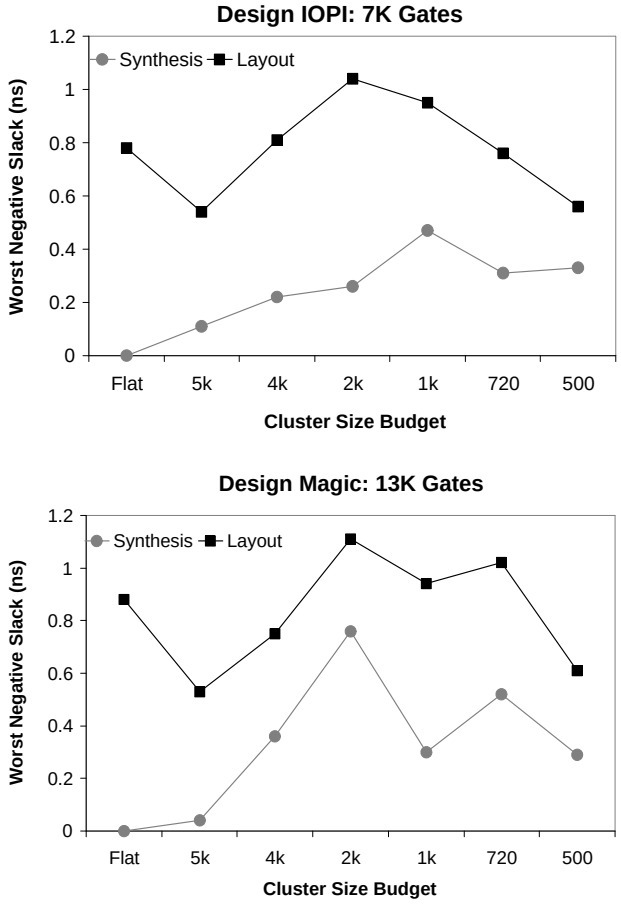


Figure 5.11: Synthesis Overhead in the Prototype System

Reduced area utilization within clusters, combined with the fragmentation of layout area, results in much larger total layout area in the prototype system. The worse-case utilization overhead of the prototype system is 15%, whereas the maximum area overhead is 36%. The maximum overhead is in the largest design, since it has the maximum number of clusters that result in the maximum fragmentation of the floorplan. This area overhead also leads to longer wires, which affects the timing

of the design.

Figure 5.11 shows the effect of synthesis optimization space on timing in the prototype system. The two curves show the same data as Figure 5.9, for designs *IOPI* and *MAGIC*. The x-axis shows cluster sizes ranging from flat, all the way to the optimal cluster size of around 500 gates. 500 gates per cluster gives the best post-layout timing due the narrowest gap between synthesis and layout timing. However, a much larger cluster size of 5K also gives the best post-layout timing even though the gap between synthesis and layout is large. This is due to the fact that the post-synthesis timing for 5K gates per cluster is much better than that for 500 gates. It is evident from Figure 5.11 that as cluster sizes decrease, resynthesis does a worse job of optimizing the design. This is due to the fact that the Synopsys *Design Compiler* does limited optimizations across partition boundaries in a hierarchical netlist, consisting of inversion propagation and removal of redundant nets. Hence, the optimization space available to synthesis decreases with smaller clusters, resulting in worse timing.

Experimental Results: Advantages of the Prototype Flow

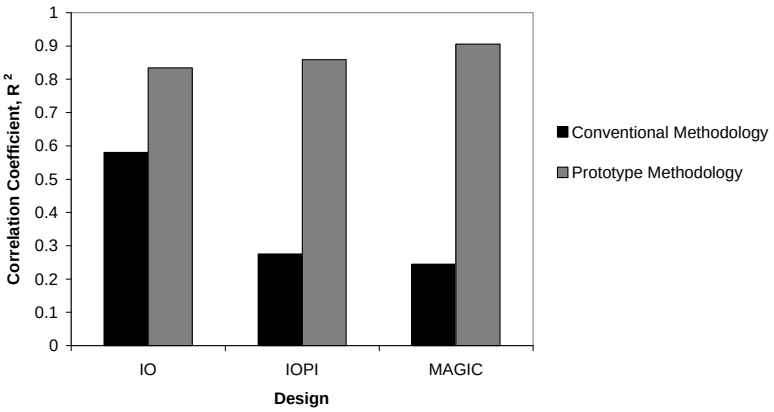


Figure 5.12: Comparison of Wire-load Correlation

Figure 5.12 compares the correlation coefficient (R^2 coefficient) of estimating the actual wire-loads measured after layout with synthesis wire-load models. The hybrid wire-load model used for resynthesis in the prototype system is significantly more accurate than the custom statistical model used in the conventional methodology,

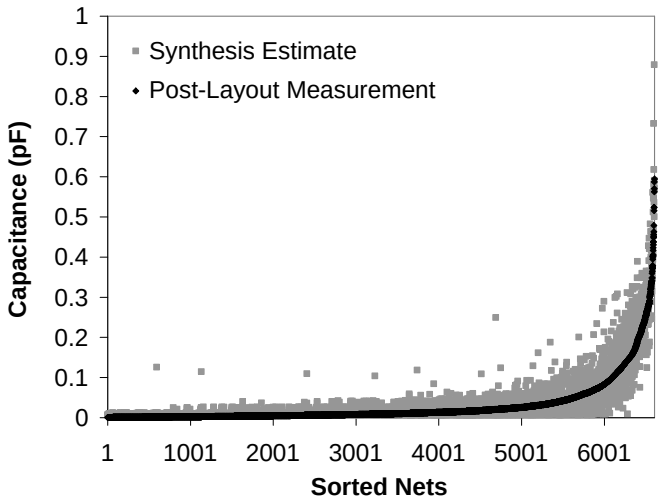


Figure 5.13: Accuracy of Wire-load Models in the Prototype System

resulting in higher correlation coefficients in all three designs. In our experiments we also found that global wire-loads had much higher correlation coefficients than local wire-loads in the prototype system. This is expected from the way global and local wires are estimated by the hybrid wire-load model. As we go to larger designs in the prototype system, the higher correlation of global wires dominates the overall wire-load correlation. Hence, the correlation coefficient marginally improves for larger designs in the prototype system. We expect this trend to saturate pretty quickly for even larger designs. The correlation in the conventional methodology gets worse for larger designs since wire-loads vary in bigger geometries, making the tail of the wire-load distribution of Figure 2.4 wider and taller for each fanout.

This difference in wire-load modeling is illustrated by Figure 5.13, which compares synthesis estimates and post-layout measurements of wire-loads using the prototype system on the same design as the one used in Figure 2.4. The x-axis gives all nets sorted by increasing fanout and post-layout wire-load, and the y-axis gives wire-load capacitance values in pF. Comparing Figures 2.4 and 5.13, it is evident that the limitation of statistical wire-load modeling is removed in the prototype system.

Figure 5.14 compares synthesis estimates and post-placement measurements of the longest delays in the two methodologies. There is a large disparity between post-

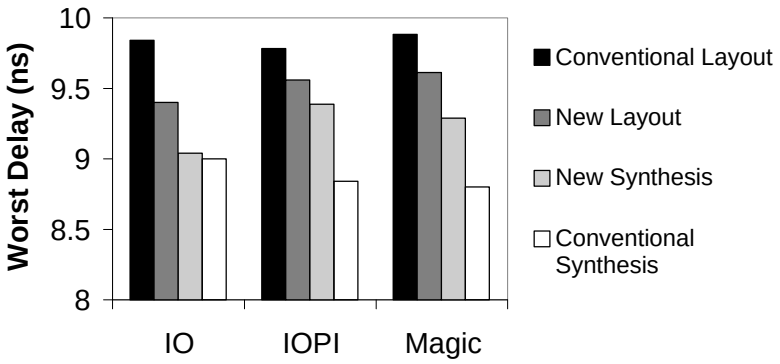


Figure 5.14: Comparison of Worst-case Delays

synthesis and post-placement timings seen in the conventional methodology, which leads to iterations between synthesis and layout discussed in Section 2.2. Figure 5.12 showed that statistical wire-load models of the conventional design flow are significantly less accurate than the prototype flow, especially for larger designs. However, the timing gap between synthesis and layout is only marginally worse in the conventional methodology. This is due to the fact that the advantage of predictable wire-loads in the prototype flow are diminished by the overheads of partitioning. Also, for the $0.5\mu m$ technology used here, even the largest benchmark design does not have wire delays that are comparable to gate delays. Hence, errors in wire-load modeling are only seen as increased gate delays after layout. With larger designs or smaller technologies, this gap between synthesis and layout timing would be higher in the conventional methodology.

Accurate wire-load modeling in the prototype system results in realistic timing prediction by synthesis, which is very close to the timing measured after placement. Thus, in spite of the overheads of using partitioning with conventional tools, the prototype system successfully reduces the gap between synthesis and layout timing by accurately predicting wires during synthesis. This results in fewer surprise critical paths after placement, reducing the need for iterations between synthesis and layout before timing constraints are met.

5.4 Summary

This chapter presented a hybrid wire-load model for accurately modeling wires using part back-annotation and part statistical models that are derived from partial layout of a partitioned netlist. Since this model accurately estimates inter-cluster wires, the corresponding partitioning scheme partitions a netlist along those nets that are unpredictable due to their netlist structure. The maximum size of each cluster is calculated to ensure that an average library gate can drive the longest local wire in the cluster. This enables statistical modeling of local wires without having a significant impact on the timing of the netlist due to errors in wire-load estimation. Thus, this partition scheme gives the optimal trade-off between accuracy of wire-load models and ability to incrementally optimize the netlist in low-level design iterations proposed by the *Nebula* design flow.

Results show that the prototype system gives high predictability in wire-lengths and timing during synthesis. Using partitioning with conventional synthesis and layout tools results in overheads in the quality of synthesis optimizations and in the layout area. This limits the advantage of the predictability. Overcoming the overheads due to reduced optimization space in cell-placement and logic synthesis calls for algorithms that optimize netlists across partition boundaries. Also, fragmentation of floorplanned area calls for a floorplanner that creates soft cluster boundaries with arbitrary shapes, since such shapes can be re-aligned without loosing any area. Some of the requirements for synthesis and layout tools that treat partitions as soft boundaries were discussed as part of the *Nebula* flow in Chapter 3. Chapter 7 discusses ideas for future work in creating such tools.

As a first step towards creating soft partition boundaries in the low-level iterations of the *Nebula* flow, a heuristic needs to be developed for moving logic across partitions based on the latest timing information are needed. The next chapter presents such a heuristic for timing-driven repartitioning, implemented within the framework of the prototype system.

Chapter 6

Timing-driven Repartitioning

Netlist partitioning is used at several stages of the chip-design process. Netlists are functionally partitioned early in the design cycle to distribute design responsibility across teams of designers and to do early planning of chip area and characterization of global wire-lengths. Functional blocks are further partitioned during floorplanning to generate sub-modules with sizes managed by synthesis and layout tools. This partitioning is usually done by a traditional partitioning algorithm such as mincut [76]. Netlist-structure based partitioning is used within synthesizable blocks; to reduce the complexity of placement [80], [7] and to manage the complexity of graph-matching during technology mapping [51].

A partitioning scheme applied at an early stage in any design flow relies on incomplete or inaccurate timing information. This pushes the prediction problem from logic synthesis to partitioning, and creates discrepancies in timing viewed by the initial partitioning and the final layout. The partitioning criteria may cluster gates in such a way that some gates end up in non-optimal locations; these gates can not be optimized by placement or synthesis due to the physical boundaries created by partitioning. If a critical path goes through such a gate, there is potential to improve its timing by moving the gate to another partition which contains other gates on the path.

This chapter presents a heuristic for timing-driven repartitioning, to improve critical timing paths that are created due to early partitioning choices. Based on timing

analysis done after detailed placement, this heuristic identifies logic that adds the most global wire-length to the most critical paths. Next, the heuristic looks for a partition where this logic should be relocated in order to improve the timing of all critical paths going through it. If the heuristic finds such a partition, the logic is relocated to that partition. Instead of considering only the worst critical path, all paths worse than a desired maximum timing violation (also referred to as timing slack) are considered in this technique. This global view of timing paths gives incremental timing improvements, not only reducing the longest delay in the design, but also reducing the number and total delay of all critical paths considered.

Timing is improved by reducing global wire-loads along critical paths. Since critical gates are relocated together, such repartitioning also gives the opportunity for later synthesis transforms to synthesize larger clusters of critical logic together within one partition. Hence, such repartitioning is an important part of the low-level synthesis and layout iterations that are part of the *Nebula* flow proposed in Chapter 4.

Section 6.1 describes the heuristic for choosing gate relocations and motivates the extent of timing-analysis required to make the heuristic effective in improving post-layout timing. Section 6.2 presents the implementation of the heuristic as an automated tool for timing-driven repartitioning. This heuristic was implemented in the prototype CAD system of Figure 4.3 to improve timing after the detailed cell-placement step. Experimental results in Section 6.3 show the effectiveness of the heuristic as a layout-driven synthesis technique for timing improvements at a late stage in the design cycle.

6.1 Heuristic for Gate Relocation

Beginning with the placement and timing information of a partitioned netlist, the heuristic identifies gates that contribute most to the overall timing quality of the design due to the locations of their parent partitions in the netlist floorplan. Such gates are relocated to another partition which reduces the global wire-length along critical timing paths. Those gates that give the most timing improvement along the

most critical paths, while not hurting other critical paths, are chosen for relocation. This is quantified by a heuristic called the *Potential for Timing Improvement* (*PTI*) of relocating a gate.

The gate with the maximum *PTI* is chosen for relocation along with the cluster of gates, belonging to the same partition and sharing critical paths with the chosen gate. This cluster is relocated to another partition that contributes the most timing improvement to the *PTI* of the chosen gate. Small clusters are preferred over large ones since they would cause smaller perturbation to the original layout. The chosen cluster is moved to another partition containing gates that appear just before or just after the chosen cluster along many critical paths. Therefore, the hop to the cluster's current partition is removed from those paths. To minimize adverse effects on other paths, higher priority is given to relocating clusters that are loosely connected to their parent partitions, but tightly connected to the new partitions. Also, clusters with fewer desired new locations are preferred over those with many conflicting relocation requirements.

Since this heuristic falls into the class of layout-driven synthesis techniques discussed in Section 3.2, two key issues were addressed while developing the heuristic: the accuracy of timing analysis, and convergence to the final design implementation. Accurate timing analysis is achieved by using the LSI layout tool's *net-length extractor* followed by the *delay calculator* to accurately estimate the timing from a placement, rather than using lumped RC models for wires or simplistic models for gate delays. To avoid creating new timing paths while fixing the most critical path, all paths with timing slacks, higher than a desired maximum, are treated as critical paths. Thus, all timing-paths within a window of timing slack are enumerated in order to choose gates for relocation. To assure convergence, each relocation is followed by a layout-merge and timing analysis. If the relocation doesn't improve timing after being merged into the existing layout, it is rejected.

The following subsection motivates the need for path enumeration with the use of an example netlist and critical-path information from benchmark designs of the MAGIC chip. Next, Section 6.1.2 derives the formula for *PTI*.

6.1.1 Motivation for Path Enumeration

Fixing only the worst or the top few critical paths at a time has the inherent limitation that new critical paths may arise due to a fix. This is especially true if the timing of several paths is very close and the paths share many gates [45]. Table 6.1 and Figure 6.1 illustrate this for the longest timing paths, seen after detailed placement in the three benchmark designs used for the experiments in Chapter 5.

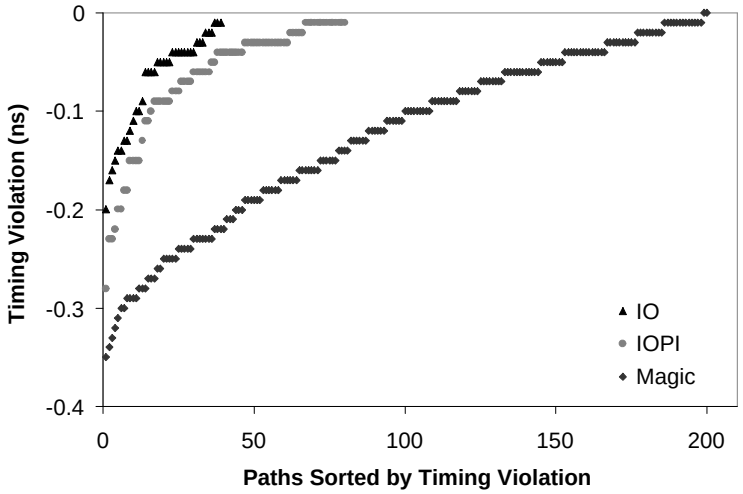


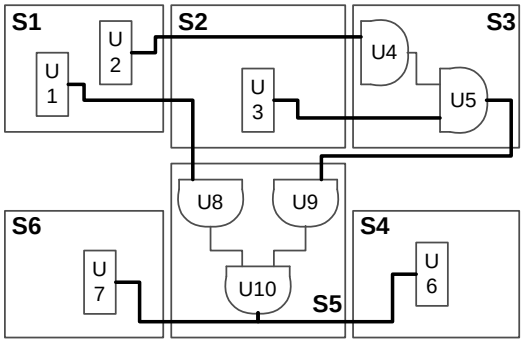
Figure 6.1: Paths Leading to the Worst-Case Endpoint

Table 6.1: Paths Leading to the Worst-Case Endpoint

Design	Paths	Sources	Instances	Paths/Gate
IO	39	5	68	26
IOPI	81	8	113	36
Magic	200	8	104	101

Figure 6.1 shows the timing violation of paths leading to the worst-case endpoint in each design on the y-axis, and sorted by decreasing negative slack on the x-axis. There are many paths with very similar timing slacks leading to the same endpoint. Table 6.1 shows detailed statistics of these paths. The four columns represent the number of critical paths that are in violation of a 9ns clock-cycle target, the number

of different timing sources and total number of distinct library cell-instances (gates) on these paths, and the average number of paths going through each of those gates. We can see that critical paths have a lot of shared logic, since many paths start from a common timing source and each logic gate belongs to several different critical paths. Hence, relocating gates to fix the worst path, without considering other paths closely related in timing and logic, is likely to result in worse timing on the other paths.



Path	Slack
P1: U2,4,5,9,10,6	-1ns
P2: U2,4,5,9,10,7	-1ns
P3: U3,5,9,10,6	-0.95ns
P4: U3,5,9,10,7	-0.95ns
P5: U1,8,10,6	-0.9ns
P6: U1,8,10,7	-0.9ns

Figure 6.2: Example Netlist for Timing-driven Repartitioning

Figure 6.2 shows an example floorplan of a netlist containing partitions *S1-S6*, six critical paths *P1-P6*, and their timing slacks. Only those netlist connections that belong to one of these critical paths are shown in the figure. Gate *U10* belongs to all six critical paths and contributes the most to the global lengths of these paths due to its location in the floorplan. If the cluster *U9,U10* is relocated to partitions *S3* or *S4*, the global wire-length along paths *P1*, *P3* and *P5* would be significantly reduced, improving their timing. However, the global wire-length along *P2*, *P4* and *P6* would increase, making their timing worse. Similarly, moving the cluster *U8,U10* to the left side of the floorplan would improve the timing of *P2*, *P4* and *P6*; however, this would make the other three paths worse. Since the timing violations of all six paths are very close, the worse-case violation in the netlist could end up being worse than 1ns after either of these relocations. Hence, there is no right place to relocate gate *U10*.

On the other hand, gate *U5* belongs to four of the six critical paths and contributes as much global wire-length as *U10* to those paths. Relocating the cluster *U4,U5* to *S2* or *S5* would reduce the global wire-length along *P1,P2,P3* and *P4*, while not affecting paths *P5* or *P6*.

The heuristic presented here considers this interplay of critical paths; all paths with negative timing slacks worse than a maximum desirable slack are treated as critical paths, and all critical paths are enumerated in order to choose gates for relocation.

6.1.2 Potential For Timing Improvement

PTI is the metric for estimating the Potential for Timing Improvement of candidate logic clusters and their new locations. We derive a formula for *PTI* here that considers various timing, layout and netlist parameters, and accurately estimates the relative potential for timing improvement of various relocation choices.

A timing path can be viewed as a series of gate clusters. Each cluster belongs to a single partition called its *parent partition*. Clusters along a timing path are separated by global wires. Along a path, each cluster has a *source partition* driving the global net input of the first gate of the cluster, and a *destination partition* containing logic that is driven by the global net output of the last gate of the cluster. For example, path *P1* in Figure 6.2 has four logic clusters *U2, U4,U5, U9,U10, U6* separated by three global nets. Along this timing path, *S3* is the parent partition of logic cluster *U4,U5*, whereas *S1* and *S5* are its source and destination partitions.

Let $PTI(P, C)$ denote the potential for timing improvement along timing path P, by moving cluster C from its parent partition S_C to its source or destination partition. This *PTI* can be estimated by the timing slack of P multiplied by the ratio of the total global length along P over the global length, if the hop to S_C was removed from P. Here the global length of a path is the total Manhattan distance between centers of consecutive partitions and primary IO pin locations along the path. This is shown in Equation 6.1.

$$PTI(P, C) = Slack(P) \times \frac{GlobalLen(P)}{GlobalLen(P - S_C)} \tag{6.1}$$

However, from Table 6.1, every gate in a cluster may belong to several other clusters as part of other critical paths. Hence, the *PTI* of relocating a gate g is the sum of *PTI*s of each cluster it belongs to along each critical path through g. This is

shown in Equation 6.2, where C_i is the cluster of local logic of g along path P_i, for N paths going through g.

$$PTI(g) = \sum_{i=1}^{N} PTI(P_i, C_i) \tag{6.2}$$

If there are many gates in cluster C, this move is likely to perturb many other critical paths. To encourage moving small clusters over large ones, the $PTI(P, C)$ is divided by the size of the cluster (in number of gates) in Equation 6.3.

$$PTI(P, C) = Slack(P) \times \frac{GlobalLen(P)}{GlobalLen(P - S_C)} \times \frac{1}{Size(C)} \tag{6.3}$$

If a gate has a lot of direct fanin or fanout (fanIO) terminals in its parent partition as opposed to the source or destination partitions, it should be discouraged from moving since it is likely to affect many other paths in the parent partition. Similarly, if the gate has a lot of fanIO in the source or destination partition it should be encouraged to relocate. However, if different critical paths through a gate have different source and destination partitions, there would be many conflicting locations where the gate could be relocated. As illustrated by the example of Figure 6.2, moving such a gate could result in worse timing if the timing slacks of paths with different source and destination partitions are similar. These criteria are taken into account in the $PTI(g)$ in Equation 6.4. Here $IO(g, S)$ is the number of fanIO terminals of g in partition S, S_P is the parent partition of g, and S_{S_i} and S_{D_i} are the source and destination partitions of g along path P_i. $NS_{S,D}(g)$ is the number of distinct combinations of source and destination partitions where g can be relocated along all critical paths going through it.

$$PTI(g) = \frac{1}{NS_{S,D}(g)} \times \sum_{i=1}^{N} \left(PTI(P_i, C_i) \times \frac{IO(g, S_{S_i}) + IO(g, S_{D_i})}{IO(g, S_P)} \right) \tag{6.4}$$

Equation 6.4 is used to sort all gates on all critical paths of a design by their PTI. The gate g_R with the maximum PTI is selected for relocation. For each partition $S_K \in \{S_{D_i}, S_{S_i} \forall i = 1, 2 \ldots N\}$ to which a gate g_R can possibly be relocated, the $PTI(g_R, S_K)$ is computed from Equation 6.5. This equation uses the same summation

formula as Equation 6.4, however the summation is over only those K critical paths along which S_K is either the source or destination partition of g_R.

$$PTI(g_R, S_K) = \sum_{i=1}^{K} \Big(PTI(P_i, C_i) \times \frac{IO(g_R, S_{S_i}) + IO(g_R, S_{D_i})}{IO(g_R, S_P)} \Big) \tag{6.5}$$

The partition S_R with the maximum $PTI(g_R, S_R)$ is chosen as the new location for gate g_R. All gates that fall into the parent cluster S_P of g_R, and belong to any of the K critical paths considered in computing $PTI(g_R, S_R)$, are combined in a relocation cluster C_R along with g_R. C_R is relocated to S_R to ensure that the hop to S_R is removed from all critical paths shared by g_R and S_R, having S_R as the source or destination partition of g_R.

Experiments with the different parameters in this heuristic show that the $PTI(g)$ from Equation 6.4 gives the best overall timing improvements, which consists of: the worst-case timing slack, the total number of critical paths above a desired maximum timing slack, and the total timing slack of those paths. The basic formula of Equation 6.2 gives the same improvement in the total number of critical paths as the Equation 6.4 but inferior worst-case timing and hence inferior total timing slack of all critical paths. Both worst-case and total timing slacks improve when the cluster size is added to $PTI(P, C)$ in Equation 6.3, and improve even further when the conflicting relocation requirements and the relocating gate's fanout are taken into account by Equation 6.4. Thus, considering only the timing slack of paths and global length added due to a partition prunes out as many critical paths as the final equation. However, adding the remaining netlist-structure based parameters identifies better relocations that give more timing improvements on those paths.

The next section describes our implementation of this heuristic as an automatic tool for applying post-layout timing-driven repartitioning to a design.

6.2 Implementation

Figure 6.3 shows the flowchart of the timing-driven repartitioning tool, as it was implemented in the prototype system. The tool begins by reading the floorplan

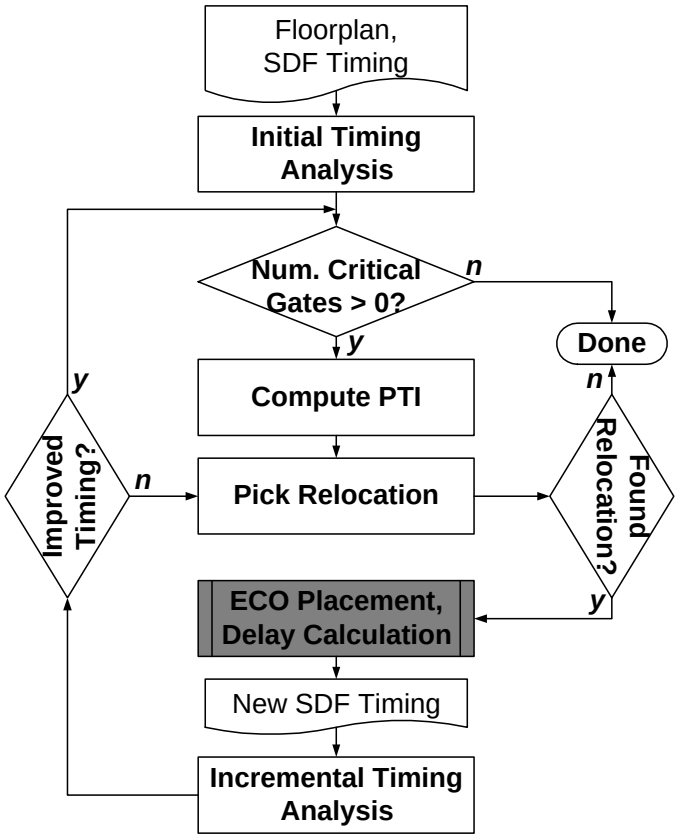


Figure 6.3: Flowchart of Timing-Driven Repartitioning Tool

locations of partitions, timing constraints, and detailed timing of the design. Timing information is generated from the detailed placement engine using LSI's *Net-length estimator* and *Delay Calculator* tools. This generates timing in the Standard Delay Format (*SDF*), which gives the delay of every timing edge in the design including gate delay, wire delay, flip-flop setup/hold, and clock-Q edges. The following subsections discuss each step of the flowchart.

6.2.1 Initial Timing Analysis

A *Vex* routine converts the *SDF* file into a directed acyclic timing graph of the netlist. Gate pins and primary I/O pins of the netlist make up the vertices, and net connections and connections through gates make up the edges of the timing graph. Any cycles in the graph are broken at flip-flops by not adding edges corresponding to

setup times. Instead, setup times are stored as arrival-time constraints on the vertex representing the data input of the flip-flop. Thus, primary inputs of the design and clock pins of flip-flops are timing sources in this graph; whereas primary outputs of the design and data inputs of flip-flops are timing sinks. The rise and fall delay of each edge and its monotonicity are annotated to avoid false paths in static timing analysis due to illegal rise/fall transitions.

The timing graph is topologically ordered using the depth-first search algorithm [47]. Furthermore, the graph is traversed twice in forward and reverse topological order to annotate arrival times, required times, and worst-case slacks at vertices. The complexity of topological sort is linear, $O(N+E)$ and of each traversal is also linear, $O(N)$ where N is the number of vertices and E is the number of edges in the graph. Static path sensitization analysis techniques can be used to enhance this static timing analysis, to ignore the timing of false paths and delay edges that are never exercised during the actual operation of the netlist [87], [88]. All timing sources and vertices with negative timing slacks worse than a desired maximum are treated as critical.

Critical paths are enumerated by doing a depth-first search for critical vertices in the fanout of every critical timing source. The algorithm for enumeration stops the depth-first search at timing sinks and non-critical vertices, and prunes out non-critical partial and full paths. During path enumeration, the parameters necessary to compute the PTI of Equation 6.4 are annotated on the timing paths, logic clusters, and individual gates visited. For every critical path, its worst negative slack (WNS) for rising and falling transitions are annotated as the slack of the endpoint of the path. Also, the total global wire-length along the path is annotated. For every local cluster of gates along a path, the number of members in the cluster and the global wire-length along the path up to that cluster are annotated. For every gate, the number of fanin/fanout connections in its source, destination, and parent partitions are annotated.

6.2.2 Computing *PTI*

This step computes the PTI of all critical gates using the parameters annotated by *Initial Timing Analysis* on critical paths, critical gates and local clusters of critical paths. For each critical gate, the annotations on each critical path going through the gate and the annotations on the corresponding local cluster of the gate are used to compute $PTI(P, C)$ of Equation 6.3. Here, the global wire-length due to a local cluster is calculated as the global wire-length up to the cluster in its source partition subtracted from the global wire-length up to the cluster in its destination partition. $NS_{S,D}(g)$, the number of different source and destination partitions along all paths going through the gate is calculated while looking up the annotations along each path for the above calculation. Also, the $PTI(g, S)$ of the source and destination partitions of the gate along each path are calculated. PTIs calculated along individual paths are added to calculate the total $PTI(g)$ of a gate and the corresponding $PTI(g, S)$ of each source and destination partition, as shown in Equations 6.4 and 6.5 respectively.

6.2.3 Picking a Relocation

This step sorts all critical gates by their $PTI(g)$ and also sorts all potential new partitions each gate can be relocated to by their $PTI(g, S)$, as discussed in Section 6.1.2. The gate g_R with the maximum $PTI(g_R)$ is chosen for relocation to the partition S_R with the maximum $PTI(g_R, S_R)$. Next, all local clusters of critical paths going through g_R that have S_R as as the source or destination partition of g_R are accumulated in one relocation cluster C_R. If C_R is larger than a desired maximum size or contains gates that have already been relocated, the destination S_R is rejected and the partition with the next highest $PTI(g_R, S)$ is chosen for relocation. If none of the source or destination partitions of g_R give an acceptable cluster for relocation, the gate with the next highest $PTI(g)$ is chosen for relocation. If none of the gates give an acceptable relocation, the algorithm exits.

6.2.4 ECO Placement

The ECO placer of LSI Logic's *CMDE* tools suit was used to update the existing placement with the relocation. The ECO placer is constrained to place gates of the relocation cluster C_R in a bounding box created by locations of C_R's fanin and fanout terminals in the new partition S_R. Existing gates within the bounding box are nudged within a maximum limit to create space for the relocated gates. If this bounding box is too small for the total gate area in C_R, it is expanded around the center of the original bounding box until the gates can be placed by the ECO placer.

If the ECO placer can not place a relocated gate within the bounding box, it places it in the best location it can find close to the gate's fanout. This decision is not timing-driven and does not always end up in the desired new partition. After ECO placement, the *CMDE net-length extractor*, followed by the *delay calculator*, are run to generate new timing in *SDF* format.

6.2.5 Incremental Timing Analysis

Using the new *SDF* file, incremental timing analysis accurately measures the impact of the relocation. Arrival times are updated in the fanout of all edges that have changed delays. Required times and path slacks are updated in the fanin of all edges with changed delays. Timing sources that end up with positive slack after incremental timing analysis are deleted from the list of critical timing sources. All paths leaving new critical timing sources or critical timing sources with updated slacks are re-enumerated. The *Initial Timing Analysis* step is expected to be slow due to exponential complexity of doing path enumeration on the entire timing graph. However, *Incremental Timing Analysis* only works on the incremental timing graph of vertices affected by the last relocation. As will be seen in the experimental results, the number of gates relocated at a time is usually very small; hence *Incremental Timing Analysis* is expected to be much faster than *Initial Timing Analysis*.

6.2.6 Measuring Timing Improvement

Timing improvement is measured in terms of the worst negative slack (*WNS*) of the most critical path and the total negative slack (*TNS*) of all critical paths. The *WNS* reflects the speed of the design. Whereas, the *TNS* reflects the number and amount of timing violation of all timing paths, which need to be fixed before the design meets its timing constraints. A relocation may end up with worse timing if the ECO placer is unable to find a suitable layout merge without significantly perturbing the existing layout, resulting in some gates being placed far from their original or desired new locations.

If a relocation results in worse *WNS* or the same *WNS* but worse *TNS*, it is rejected. The next best relocation is attempted by going down the list of sorted $PTI(g)$s. If a relocation improves the timing, it is accepted and the tool loops back to recompute new *PTI*s to account for changes made to the layout. The tool stops when it has attempted to relocate all critical gates with non-zero *PTI*.

6.3 Experimental Results

We experimented with the same three benchmarks designs, described in Section 5.3, as the earlier experiments with the partitioning scheme. Timing-driven repartitioning through gate relocation was applied to the final detailed placement generated by the prototype system, as shown in Figure 4.3. Timing analysis and integration of the technique as one automated tool were done using *Vex*.

6.3.1 Incremental Timing Improvements

Figures 6.4, 6.5, and 6.6 show the improvement in timing achieved by each accepted cluster relocation in the three designs. The x-axis represents the total number of gates relocated. The y-axis represents the number of critical paths, *TNS*, and *WNS*, which are normalized with respect to their initial values measured after the detailed placement. Even though the maximum cluster size was set to 25 gates, each relocation moved a very small cluster in the range of 1 to 11 gates. The first few relocations gave

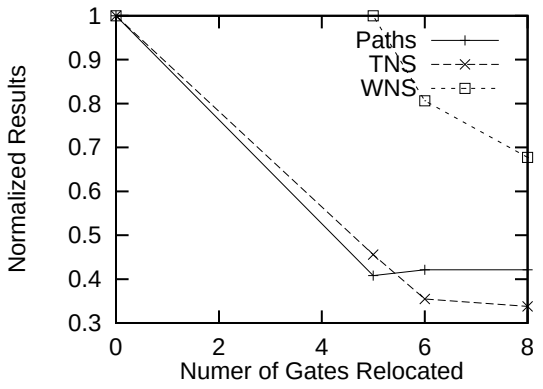


Figure 6.4: Timing Improvements Through Successive Relocations : *IO*

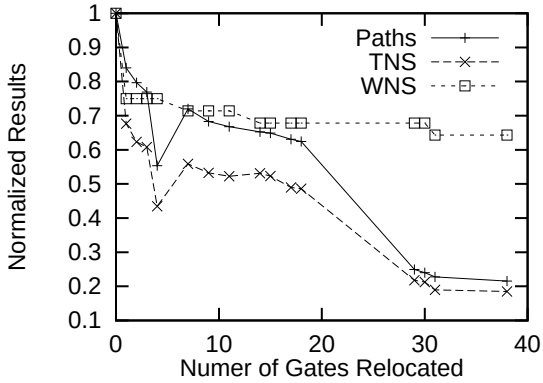


Figure 6.5: Timing Improvements Through Successive Relocations : *IOPI*

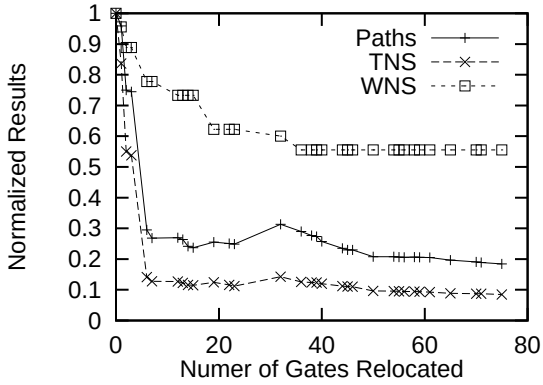


Figure 6.6: Timing Improvements Through Successive Relocations : *Magic*

significant improvements in all three timing parameters. Later moves improved the overall timing quality of the designs by reducing the number of critical paths and the *TNS*; however, the *WNS* did not improve significantly. The last few moves reached a point of diminishing returns.

6.3.2 Design Speed

Table 6.2: Timing Results After Relocation

Design	# Gates		%Delay in Global Nets		Worst Violation (ns)		%Clock-Cycle	
	Total	Reloc	Init	Reloc	Init	Reloc	Init	Reloc
IO	2164	8	24.2	15.3	-0.31	-0.21	3.4	2.3
IOPI	7537	38	30.2	14.8	-0.28	-0.18	3.1	2
Magic	12775	75	30.6	15.5	-0.45	-0.25	5	2.8

Table 6.2 compares the timing of the initial placement (Init), generated after the detailed placement stage of the prototype system, with the timing of the final placement (Reloc), generated after attempting to relocate all gates with non-zero *PTI*. The first column gives the total number of gates in each design and the total number relocated, which is less than 1% of total gates in all three designs. The second column shows the contribution of global nets to the total delay of all critical paths, which goes down by around 50% after relocation. This delay contribution of global nets is measured as the cumulative delays of global interconnect edges and of gates driving global nets. However, the design speed, determined by the worst-case timing violation shown in the last column in both nanoseconds and % of clock-cycle time, improves only marginally. This is due to the fact that relocation can only fix one of the several sources of timing slacks along critical paths i.e. the contribution of global wire-length. When timing of critical paths is mostly due to the number of logic stages, it can not be improved further by gate relocation. Logic restructuring would be more suitable for improving the timing of such paths. Moreover, we found two other reasons why the relocations failed to give further improvements in the worst-

case timing violation: several conflicting relocation requirements on critical gates, and the limited scope of ECO placers to merge relocation clusters into the existing layout. Ideas for addressing these limitations will be discussed as part of future work in Section 6.4.

6.3.3 Overall Timing Quality

Paths that violate timing constraints at the end of post-layout timing fixes need to be fixed through manual intervention. Thus, the quality of a placement can be judged by the amount of manual intervention which is required due to the number and the timing violation of its critical paths.

Table 6.3 shows the overall timing quality of the designs before and after relocation. The timing quality is measured in terms of the number of critical paths in violation of a 9ns clock-cycle time in the first column and the total negative slack (*TNS*) of all critical paths in the second column. Even though relocation can not reduce the worst critical slack beyond a small amount, it prunes out many of the near-critical paths. The number of critical paths is reduced by an average of 73%. Due to the combined effect of fewer critical paths and lower global interconnect delay in critical paths, the *TNS* drops by 80% on average. The timing improvement is more significant for the larger designs since those have more partitions, offering more opportunities for relocation.

Table 6.3: Timing Quality After Relocation

	# Critical Paths			Total Violation (ns)		
	Init	Reloc	%	Init	Reloc	%
IO	76	32	58	-7.8	-2.6	65
IOPI	325	70	78	-26.2	-4.8	81
Magic	1069	197	82	-161.4	-13.6	92

This is further illustrated in Figure 6.7 which compares the quality of placement of design *IOPI* before and after relocation. The x-axis represents the timing violation in ns, and the y-axis represents the number of paths with that violation. While some

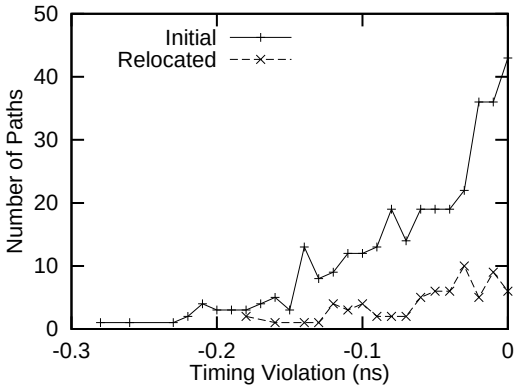


Figure 6.7: Timing Quality: *IOPI*

long paths remain, the overall number and violation of critical paths is significantly reduced.

The following experiments explore widening the scope of post-layout timing improvements gained here, by applying relocation along with driver-sizing.

6.3.4 Applying Relocation with Driver-Sizing

Since the scope for timing improvement of gate-relocation is limited to global wires, its timing improvement hits a wall when the critical path is removed from global wires and is determined by the logic of local clusters. However, if used with other post-layout timing optimization techniques that can optimize the drive strength and structure of logic in the local clusters, relocation can be very effective in making designs more manageable for manual timing fixes. The strength of gate relocation is that it does not added extra area, gates, or power due to more or larger gates in the design as opposed the other post-layout optimization techniques. By pruning out many critical paths from a design, relocation makes the job of other post-layout techniques or designers doing manual path fixing easier. To study the validity of this, we ran experiments to compare the relative effectiveness of one of the post-layout optimization techniques - driver sizing with and without gate relocation.

Figure 6.8 shows the worst negative slack in the three benchmark designs after three different scenarios:

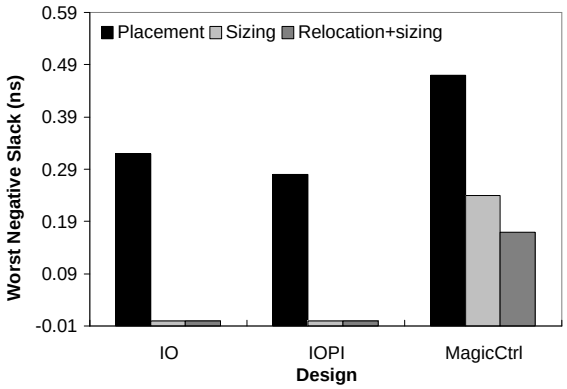


Figure 6.8: Comparison of Worst Negative Slack

1. After detailed placement step of the prototype system.

2. Detailed placement followed by driver sizing run in Synopsys Design Compiler followed by a layout-merge.

3. Detailed placement followed by gate relocation followed by driver sizing and layout-merge.

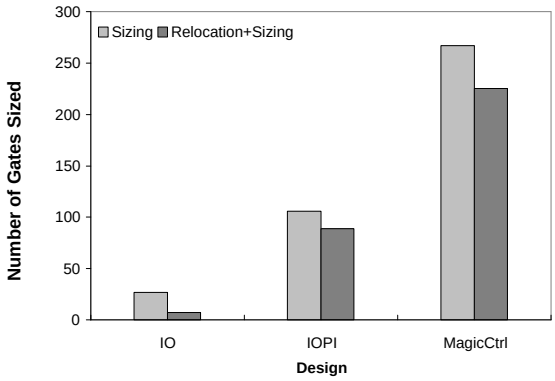


Figure 6.9: Comparison of Number of Gates Sized

Figure 6.9 shows the number of gates that needed to be sized in scenarios 2 and 3 in order to get the corresponding timing improvements of Figure 6.8. For the two smaller designs *IO* and *IOPI*, both scenarios 2 and 3 reduce the *WNS* to 0. However, the number of gates sized to achieve the timing is lower with relocation. When relocation

was applied before sizing in the largest design, *MagicCtrl*, lower *WNS* was achieved even though fewer gates needed sizing. Thus, not only does applying relocation first make the job of driver sizing easier by requiring fewer sizing changes, it gives timing improves that could not be achieved by sizing alone. Chapter 7 discusses taking this concept further in the future, with a heuristic that gets the most timing improvements by combining several different post-layout timing optimizations techniques in one tool.

6.4 Future Work

Experimentation with gate relocations opened up two interesting avenues that would extend its scope in the future: and getting more out of relocations through logic duplications, and making more relocations successful after layout merges. These are covered in the following two subsections.

6.4.1 Extending the Scope of Relocations with Logic Duplication

Logic duplication on high-fanout gates is used by Lu *et al.* in [89], as part of post-placement timing improvement transforms. Their technique duplicates high-fanout gates that have more than twice the delay on their fanout nets as compared to their fanin nets. This identifies sufficiently driven gates that are over-loaded. The fanout is split between the two duplicated instances, if majority of the fanout is critical. Otherwise, only the critical fanout is driven by one of the instances and the rest of the fanout is driven by the other instance.

In the context of the timing-driven repartitioning heuristic, the global wire-length along some paths can not be reduced further if one or more gates have several conflicting locations for relocation. Such gates can not be relocated to any one place to improve the worst-case timing violation, as was the case for gate *U10* in the example of Figure 6.2. Such gates are also candidates for duplication, since each duplicated instance can be relocated to one of the conflicting new locations. Candidates for duplications can be identified from Equation 6.4, by separating the summation term

from number of conflicting source and destination pairs $NS_{SD}(g)$. Gates with a high potential for timing improvement, given by the summation term, and a high conflict in relocation, given by the $NS_{SD}(g)$ term, should be duplicated.

However, logic duplication increases the gate loading on the fanin due to the additional gate, unless the duplicated gates are sized down to keep the total capacitive loading on the fanin gate the same. When applied along with relocation, it also increases the wire-load of inputs of the duplicated gate; those nets have to be routed to two partitions instead of one. Formulation of the heuristic for logic duplication should take this into account. Gates with stronger drive in the immediate fanin should be given higher priority for duplication, since they are less likely to adversely affect the timing. If inputs of a candidate gate have fanout in the new partitions, the corresponding gate should be given higher priority since no new global routes would be introduced by relocating its duplicated instances.

The two new instances should be relocated to the top two members of the set of relocation partitions, which are located on opposite sides of the gate's parent partition in the floorplan. Each relocated cluster should contain a duplicated instance, and gates from its parents partition that share critical paths with the instance and its new partition.

6.4.2 Extending the Scope of Layout Merges

We found that on average, 2.5% of all gates had non-zero *PTI* in the three benchmark designs. However, only 0.5% were successfully relocated due to the inability of ECO placement to merge relocations into the existing placement. Figure 6.10 shows all gates with non-zero *PTI* in design *IOPI*. The y-axis represents the *PTI* values of gates, which are sorted by decreasing *PTI* on the x-axis. It is evident here that many gates with high *PTI* are not successfully relocated. This problem of merging relocations into the existing layout gets worse when larger clusters are relocated, since the ECO placer has to find room for more gates in a given bounding box.

As the relocation heuristic progresses, critical logic is accumulated in fewer partitions, making the average size of relocation clusters larger. Our experiments found

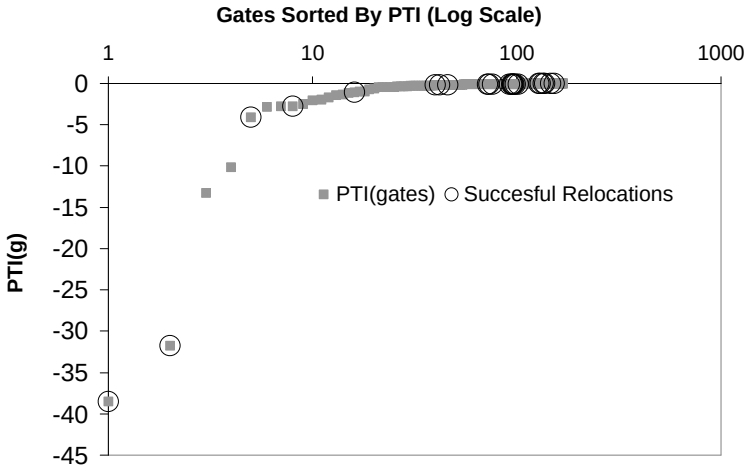


Figure 6.10: *PTI* of gates and Successful Relocations

that relocation reduced the average number of partitions per critical path from around 5 to less than 3, by accumulating logic in local clusters that increased in size from 10 gates on average to 16. To further reduce the number of partitions per path, larger clusters had to be relocated. Since each partition had around 500 gates, relocating clusters larger than 10-12 gates usually resulted in worse timing. Typical ECO placers, including the LSI ECO placer, can not merge large relocations into small bounding boxes without significantly perturbing the existing layout.

If more relocations can be made successful, relocation can be more effective in improving the timing of designs. This can be done by exchanging critical gates with non-critical ones between the parent partition and the new partition of the relocation cluster, essentially creating more layout space that can absorb the relocation. The non-critical gates should be picked by a parameter opposite to the *PTI*, i.e. their *Ability to be Relocated without any Timing Disruption*, *ARTD*.

Gates that belong inside or close to the bounding box of the target ECO placement should have high *ARTD*, since they would create space in the right place for the critical gates. If a gate is connected to the incoming relocation cluster, it should not be exchanged. Gates with high positive slacks should have higher *ARTD*, since they are least likely to introduce new timing paths. This is not computationally expensive,

since it does not require path-enumeration. The *ARTD* should also depend on how strongly the gate is connected in the new partition (i.e. parent partition of the relocation cluster) as opposed to its parent (i.e. the relocation partition).

Beginning with the gate with the maximum *ARTD*, new gates should be added to the set of non-critical gates that would be exchanged; until the total area freed up by non-critical gates is equal to the area of the incoming relocation cluster. The bounding box for ECO placement of the non-critical gates should be the current locations of the gates in the relocation cluster.

6.5 Summary

This chapter presented a timing-driven repartitioning heuristic for correcting early partitioning decisions based on the latest post-layout timing information. The heuristic relocates gates across partition boundaries in order to reduce the contribution of global nets to the timing of critical paths. This is done using a metric called *PTI*, the *Potential for Timing Improvement*, that evaluates different choices of gates and their new partitions for relocation. By applying this heuristic after detailed placement in the prototype system, results show an improvement in the quality of the placement, by reducing the total number and timing violation of critical paths. Apart from improving post-layout timing, the heuristic re-groups critical gates within a cluster. When used in the low-level iterations of the *Nebula* design flow, this gives later synthesis steps a better chance of optimizing critical logic together. However, the scope of timing-driven repartitioning is limited since it can only be used if the relocated gates can be merged into the layout of another partition.

Chapter 7

Conclusions

The conventional methodology for computer-aided design of random-logic blocks consists of iterations between netlist-level optimizations in logic synthesis and layout. This methodology suffers from a lack of convergence to the final design implementation, due to two main factors: the inaccuracy of fanout-based statistical wire-load models used by early synthesis optimizations, and the inability of layout tools to accept incremental changes during later iterations. This lack of convergence costs a lot of manual design effort and time before a design is completed, and is expected to get worse as technology scaling continues to motivate more complex chip designs.

One approach to solving this problem is to create tools that simultaneously perform synthesis and layout optimizations on each low-level element of the design netlist, thus reducing the reliance on wire-load estimates and the uncertainty of incorporating synthesis changes into the layout. This thesis proposes *Nebula*, our vision of a CAD flow targeting timing convergence through simultaneous synthesis and layout. Netlist partitioning is a key step in *Nebula*, which creates an optimal trade-off between the conflicting needs for accurate wire-load models and incremental layout optimizations. We implemented a prototype design system that emulates this design flow using commercially available synthesis and layout tools; this prototype provided an experimentation platform to explore implementation challenges in creating a CAD system targeting timing convergence.

Conventional synthesis and layout tools give sub-optimal results when they are

used with partitioned designs, since their optimization space is limited to individual clusters of the netlist. Furthermore, using rectangular boundaries in a floorplanning tool fragments the layout area when sizes of individual partitions change after synthesis optimizations. However, partitioning enables the creation of a hybrid wire-load model that accurately models global nets and approximates local nets to leave room for future local optimizations. This model leads to a better wire-load correlation between synthesis models and post-layout measurements in the prototype system. Compared to the conventional design flow, better wire-load correlation in the prototype gives better correlation between post-synthesis and post-layout timing. In spite of the overheads of partitioning, the narrower gap between synthesis and layout in the prototype flow leads to a better post-layout timing than the conventional flow. Thus, accurate wire-load modeling in the prototype makes each iteration more effective, leading to faster convergence.

As a first step towards softening the view of hard partition boundaries, the heuristic for timing-driven repartitioning reduces the contribution of global wire-loads along critical paths, by relocating logic across partitions. Data from benchmark designs reinforces the importance of optimizing several timing paths together, in order to ensure convergence in a post-layout timing optimization technique. Since several critical paths are made up of interconnections of a few critical gates, relocating a very small fraction of all gates in a design prunes out most of its critical paths. However, relocation does not give significant improvement in the longest delay of the design since its scope is limited to reducing global wire-loads in critical paths. The scope of timing improvements gained by relocation is further limited due to two main reasons: critical paths containing gates with several conflicting relocation requirements can not be optimized further; and ECO placers have limited ability to merge changes in the existing layout.

7.1 Future Work

The findings in this dissertation lead to two major open areas for future research: expanding the scope of post-layout optimization techniques, and creating the notion

of soft boundaries in synthesis and layout optimizations.

7.1.1 Synergy of Post-Layout Optimization Techniques

Results in Section 6.3 show that applying relocation and driver-sizing on a design widens the scope of both of these techniques, giving better timing with fewer changes in the layout. Huang *et al.* reported similar results for the application of simultaneous rewiring and buffer insertions, followed by driver sizing [5]. Thus, post-layout timing optimization can be made more effective if several common transforms are applied simultaneously. The algorithm should look at all critical paths or gates in a design and identify the culprit gates and nets, contributing the maximum timing along the most critical paths. Depending on the cause of the target timing problem, an appropriate post-layout optimization transform should be applied. Transforms that are easier to merge in the existing layout, such as relocation and driver-sizing, should be preferred over more layout-intensive transforms such as buffer insertion and logic duplication. Logic restructuring should be used as the last resort since it requires the most changes to an existing layout.

7.1.2 Soft Boundaries in Synthesis and Layout Optimizations

Partitioning overheads limit the advantage of accurate wire-load models in the prototype design system. Future research is required in creating synthesis and layout algorithms that overcome this overhead, before a CAD system can be built around the *Nebula* design flow.

Algorithms for creating irregularly-shaped partition boundaries for floorplanning have been implemented [9, 10]. These are soft-macro placement algorithms that determine not only the size and placement of the macros but also determine irregular shapes for their boundaries. Such boundaries maximize layout area utilization by removing all fragmentation of area due to misaligned partition boundaries. Using a soft-macro placer for floorplanning in *Nebula* would reduce the total area overhead seen in the prototype system.

Another layout-area overhead comes from lower area-utilization of cell-placement

algorithms when they are applied to individual partitions. This overhead requires a cell-placement algorithm that allows creation of region constraints on clusters of logic. The placement engine keeps all cells in a cluster within a certain distance of its region constraint but allows neighboring regions to overlap at the edges. This allows the placement engine to optimize the entire design, improving its area-utilization. Creating region-constraints on gates exists as a user-controlled option in commercial CAD tools such as Cadence [39] and was used for design planning in the past [90]. However, detailed implementation of such algorithms and their area-utilization, compared to flat and partitioned placements, have not been published.

There has not been an effort to create synthesis algorithms that regains optimization space in a partitioned design by treating partition boundaries as "soft" boundaries. In future, a synthesis algorithm that would optimize logic across partition boundaries needs to be created. In our view, the timing-aware technology mapping step in synthesis is the right place to create the notion of soft boundaries. When synthesis creates trees of logic for technology mapping, it should attempt to keep most trees local within clusters. However, logic that is structurally close to a cluster's boundary should be combined with other logic close to another cluster's boundary if the two sets of logic gates share several critical paths. When optimizing the technology map of an inter-cluster logic tree, new gates, that do not belong to any cluster in the current layout, will be created. This will collapse some existing global nets, and new global nets with unknown wire-loads will form. Basic layout knowhow should be added to synthesis to determine the best parent cluster for the new gates, in order to estimate the wire-loads of new global nets. The following layout step should take hints from synthesis for placing the new gates.

By solving these open challenges, there is a potential to develop a design flow that creates nebulous clusters of logic as a way to create properties essential for convergence while not incurring any overheads.

Bibliography

[1] "Timing convergence, iDesign methodology serivces of cadence Design Systems," *http://www.cadence.com/methodology_services/meth_tc_l2_index.html.*

[2] K. Sato et al., "Post-layout optimization for deep submicron design," in *Proceedings of 33rd Design Automation Conference*, June 1996, pp. 740–5.

[3] L. N. Kannan, P. R. Sauris, and Hong-Gee Fang, "A methodology and algorithms for post-placement delay optimization," in *Proceedings of 31st ACM/IEEE Design Automation Conference*, June 1994, pp. 327–32.

[4] W. Chuang and I. N. Hajj, "Delay and area optimization for compact placement by gate resizing and relocation," in *Proceedings of IEEE International Conference on Computer Aided Design*, Nov. 1994, pp. 145–8.

[5] R. Huang, Y. Want, and K-T. Cheng, "Libra - a library-independent framework for post-layout performance optimization," in *International Symposium on Physical Design*, 1998, pp. 135–140.

[6] M. Lee et al., "Incremental timing optimization for physical design by interacting logic restructuring and layout," in *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis*, May 1998, pp. 508–13.

[7] G. Stenz et al., "Timing driven placement in interaction with netlist transformations," in *Proceedings of the International Symposium on Physical Design*, April 1997, pp. 36–41.

[8] A. Salek, J. Lou, and M. Pedram, "A DSM design flow: putting floorplanning, technology mapping and gate placement together," in *Proceedings of 35th Design Automation Conference*, June 1998, pp. 287–90.

[9] H.-P. Su, A.C-H. Wu, and Y.-L. Lin, "Performance-driven soft-macro clustering and placement by preserving HDL design hierarchy," in *Proceedings of 36th Design Automation Conference*, 1999.

[10] "Monterey Design," *http://www.mondes.com.*

[11] W. Gosti et al., "Wireplanning in logic synthesis," in *IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1998, pp. 26–33.

[12] "Magma Design Automation," *http://www.magma-da.com.*

[13] J. Kuskin et al., "The Stanford FLASH multiprocessor," in *Proceedings of 21st International Symposium on Computer Architecture*, April 1994, pp. 302–13.

[14] R. Simoni and M. Horowitz, "Dynamic pointer allocation for scalable cache coherence directories," in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991, pp. 72–81.

[15] H. Heinlein et al., "Integration of message passing and shared memory in the stanford flash multiprocessor," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[16] V. Soundararajan et al., "Flexible use of memory for replication/migration in cache-coherent dsm multiprocessors," in *Proceedings of 25th International Symposium on Computer Architecture*, July 1998.

[17] "Pci local bus specification," *http://www.pcisig.com/tech/availspecs.html.*

[18] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kulver Academic Publishers, 1991.

[19] D. Perry, *VHDL*, McGraw-Hill, 1991.

[20] B. Kerninghan and D. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J. : Prentice Hall, 1998.

[21] B. Stroustrup, *The C++ Programming Language, Third Edition*, Addison Wesley Longman Inc., 1997.

[22] L. Semeria and G. De Micheli, "SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C," in *Proceedings of IEEE International Conference on Computer Aided Design*, Nov. 1999, pp. 321–6.

[23] "Cynapps," *http://www.cynapps.com.*

[24] "Coware," *http://www.coware.com.*

[25] "C level design inc.," *http://www.cleveldesign.com.*

[26] "Synopsys: Systemc," *http://www.systemc.com.*

[27] "Tera Systems," *http://www.terasystems.com.*

[28] V. Nagasamy, N. Berry, and C. Dangelo, "Specification, planning, and synthesis in a VHDL design environment," *IEEE Design & Test of Computers*, vol. 9, no. 2, pp. 58–68, June 1992.

[29] D. W. Knapp and A. C. Parker, "The ADAM design planning engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 7, pp. 829–46, July 1991.

[30] S. V. Venkatesh, "Hierarchical timing-driven floorplanning and place and route using a timing budgeter," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1995, pp. 469–72.

[31] B. Preas and M. Lorenzetti, Eds., *Physical Design Automation of VLSI Systems*, Benjamin Cummings Publishing Company, 1988.

[32] T. Koide and S. Wakabayashi, "A timing-driven global routing algorithm with pin assignment, block reshaping, and positioning for building block layout," in

Proceedings of 1998 Asia and South Pacific Design Automation Conference Yokohama, Feb. 1998, pp. 577–83.

[33] E. L. Le-Chin and C. Sechen, "Multi-layer chip-level global routing using an efficient graph-based steiner tree heuristic," in *Proceedings. European Design and Test Conference ED & TC*, March 1997, pp. 311–18.

[34] D. Gajski, *Silicon Compilation*, Addison-Wesley, 1988.

[35] G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Eds., *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1986.

[36] R. Senthinathan et al., "A 600MHz IA-32 microprocessor with enhanced data streaming for graphics and video," in *IEEE International Solid-State Circuits Conference*, Feb. 1999, pp. 98–101.

[37] A. Scherer et al., "An out-of-order three-way superscalar multimedia floating-point unit," in *IEEE International Solid-State Circuits Conference*, Feb. 1999, pp. 94–5.

[38] "Synopsys Inc.," *http://www.synopsys.com.*

[39] "Cadence Design Systems," *http://www.cadence.com.*

[40] "Avanti Corp.," *http://www.avanti.com.*

[41] "Lsi Design," *http://www.lsil.com.*

[42] "Physical synthesis," *Synopsys Inc., http://www.synopsys.com/products/phy_syn/phy_syn.html.*

[43] "Single-pass VDSM," *Avanti Corp., http://www.avanti.com/Avant!/SolutionsProducts/Produts/Item/1,1172,49,00.html.*

[44] H. Kapadia and M. Horowitz, "Using partitioning to help convergence in the standard-cell design automation methodology," in *Proceedings of 36th Design Automation Conference*, 1999.

[45] N. P. Jouppi, "Timing analysis and performance improvement of MOS VLSI designs," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 4, pp. 650–65, July 1987.

[46] R. B. Hitchcock Sr., G. L. Smith, and D. D. Cheng, "Timing analysis of computer hardware," *IBM Journal of Research and Development*, vol. 26, no. 1, pp. 100–16, Jan 1982.

[47] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1993.

[48] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, 1989.

[49] H. H.-F. Jyu and S. Malik, "Prediction of interconnect delay in logic synthesis," in *Proceedings of the European Design and Test Conference ED&TC*, March 1995, pp. 411–5.

[50] H. Kapadia and H. Vaishnav, "Wire load modeling with structural netlist parameters," in *Cadence Technical Conference*, 1997.

[51] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1991.

[52] R. Rudell, "Tutorial: Design of a logic synthesis system," in *Proceedings of 33rd Design Automation Conference*, June 1996, pp. 191–6.

[53] M. Sarrafzadeh and C. K. Wong, Eds., *An Introduction to VLSI Physical Design*, McGraw Hill, 1996.

[54] B. Preas and P. Karger, "Automatic placement: A review of current techniques," in *Proceedings of 23rd Design Automation Conference*, June 1986, pp. 622–8.

[55] N. R. Quinn Jr. and M. A. Breuer, "A forced directed component placement procedure for printed circuit boards," *IEEE Transaction on Circuits and Systems*, vol. 26, no. 6, pp. 377–87, June 1979.

[56] K. M. Hall, "An r-dimensional quadratic placement algorithm," *Management Science*, vol. 17, pp. 219–29, Nov. 1970.

[57] D. G. Schweikert, "A 2-dimensional placement algorithm for the layout of electrical circuits," in *Proceedings of 13th Design Automation Conference*, June 1976, pp. 408–16.

[58] F. K. Hwang, "On steiner minimal trees with rectilinear distance," *SIAM Journal on Applied Mathematics*, vol. 30, no. 1, pp. 104–14, Jan. 1976.

[59] J. Barra et al., "Application of data analysis methods and of simulated annealing for the automatic layout of circuits," *Computer Systems Science and Engineering*, vol. 2, no. 1, pp. 3–15, January 1987.

[60] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *Proceedings of International Conference on Computer Aided Design*, Nov. 1998, pp. 203–11.

[61] R. Ho et al., "Interconnect scaling implications for CAD," in *International Conference on Computer-Aided Design*, Nov. 1999.

[62] M. Burstein and M. N. Youssef, "Timing influenced layout design," in *Proceedings of 22nd Design Automation Conference*, June 1985, pp. 124–30.

[63] W. K. Luk, "A fast physical constraint generator for timing driven layout," in *Proceedings of 28th ACM/IEEE Design Automation Conference*, June 1991, pp. 626–31.

[64] C.-S. Choy, T.-S. Cheung, and K.-K. Wong, "Incremental layout placement modification algorithms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 5, pp. 437–45, April 1996.

[65] M. H. Cynn and S. M. Kang, "Incremental node extraction algorithms for incremental layout system," in *Proceedings of International Symposium on Circuits and Systems*, April 1995, pp. 1691–4.

[66] K. Keutzer, A. R. Newton, and N. Shenoy, "The future of logic synthesis and physical design in deep-submicron process geometries," in *Proceedings of the International Symposium on Physical Design*, April 1997, pp. 218–24.

[67] R.H.J.M. Otten and R.K. Brayton, "Planning for performance," in *Proceedings 35th Design and Automation Conference*, June 1998, pp. 122–7.

[68] W. Gosti et al., "Wireplanning in logic synthesis," in *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis*, May 1998, pp. 520–9.

[69] D. Brand, R. F. Damiano, L.P.P.P. van Ginneken, and A. D. Drumm, "In the driver's seat of booledozer," in *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1994, pp. 518–21.

[70] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufman, 1999.

[71] W. C. Elmore, "The transient response of damped linear networks with particular regard to wide-band amplifiers," *Journal of Applied Physics*, vol. 19, pp. 55–63, 1948.

[72] J. Rubinstein, P. Penfield Jr., and M. Horowitz, "Signal delay in RC tree networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, no. 3, pp. 202–11, July 1983.

[73] Y. C. Ju and R. A. Saleh, "Incremental techniques for identification of statically sensitizable critical paths," in *Proceedings of 28th ACM/IEEE Design Automation Conference*, June 1991, pp. 541–6.

[74] Y-M. Jiang, A. Krstic, K-T. Cheng, and M. Marek-Sadowska, "Post-layout logic restructuring for performance optimization," in *Proceedings of 34th Design Automation Conference*, 1997, pp. 663–5.

[75] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," in *Bell System Technical Journal*, February 1970, vol. 49, pp. 291–308.

[76] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of 19th ACM/IEEE Design Automation Conference*, June 1982, pp. 174–81.

[77] H.-P. Su, A.C-H. Wu, and Y.-L. Lin, "Performance-driven soft-macro clustering and placement by preserving HDL design hierarchy," in *Proceedings of the International Symposium on Physical Design*, April 1998.

[78] "Vex CAD toolbox," *http://www-flash.stanford.edu/~bergmann/tools.*

[79] J. Bergmann and M. Horowitz, "Vex - a CAD toolbox," in *Proceedings of 36th Design Automation Conference*, 1999.

[80] J. Cong and X. Dongmin, "Exploiting signal flow and logic dependency in standard cell placement," in *Proc. Asia and South Pacific Design Automation Conf.*, Aug. 1995, pp. 399–404.

[81] J. Cong et al., "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering," in *Proc. IEEE Int'l Conf. on Computer-Aided Design*, Nov. 1997, pp. 441–46.

[82] I. Sutherland and B. Sproull, "The theory of logical effort: designing for speed on the back on an envelope," in *Advanced Research in VLSI Conference*, 1991.

[83] H. B. Bokaglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, 1990.

[84] "The national technology roadmap for semiconductors: Interconnect," *Sematech Corporation: http://www.sematech.org, http://www.itrs.net/ntrs/publntrs.nsf*, 1997.

[85] M. Horowitz, R. Ho, and K. Mai, "Interconnect technology beyond the roadmap: The future of wires," in *The SRC/MARCO/SEMATECH Interconnects for Systems on a Chip - Projected Performance Technology Requirements Workshop : http://www.src.org/areas/nis/5_22_ws.dgw*, May 1999.

[86] W. Mendenhall and T. Sincich, *Statistics for engineering and the sciences*, Englewood Cliffs, N.J. : Prentice-Hall, 1995.

[87] D. H. Du, S. H. Yen, and S. Ghanta, "On the general false path problem in timing analysis," in *Proceedings of 36th ACM/IEEE Design Automation Conference*, June 1989, pp. 555–60.

[88] J. Benkoski et al., "Timing verification using statically sensitizable paths," *IEEE Transactions on Computer Aided Design*, vol. 9, no. 10, pp. 1073–83, Oct. 1989.

[89] A. Lu et al., "Combining technology mapping with post-placement resynthesis for performance optimization," in *Proceedings International Conference on Computer Design*, Oct. 1998, pp. 616–21.

[90] J.Y. et al. Sayah, "Design planning for high performance asics," *IBM Journal of Research and Development*, vol. 40, no. 4, pp. 431–52, July 1996.