ALGORITHMS FOR THE SCALING TOWARD NANOMETER VLSI PHYSICAL SYNTHESIS

A Dissertation

by

CHIN NGAI SZE

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

DOCTOR OF PHILOSOPHY

December 2005

Major Subject: Computer Engineering

ALGORITHMS FOR THE SCALING TOWARD NANOMETER VLSI PHYSICAL SYNTHESIS

A Dissertation

by

CHIN NGAI SZE

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

DOCTOR OF PHILOSOPHY

December 2005

Major Subject: Computer Engineering

ABSTRACT

Algorithms for the Scaling Toward
Nanometer VLSI Physical Synthesis. (December 2005)

Chin Ngai Sze, B.Eng., The Chinese University of Hong Kong;
M.Phil., The Chinese University of Hong Kong
Chair of Advisory Committee: Dr. Jiang Hu

Along the history of Very Large Scale Integration (VLSI), we have successfully scaled down the size of transistors, scaled up the speed of integrated circuits (IC) and the number of transistors in a chip - these are just a few examples of our achievement in **VLSI scaling**. It is projected to enter the nanometer ($10^{-9}m$) scale era in the nearest future. At the same time, the scaling has imposed new challenges to physical synthesis. Among all the challenges, this thesis focuses on the following problems:

- Increasingly domination of interconnect delay leads to a need in interconnect-centric design flows;
- Different design stages (e.g. floorplanning, placement and global routing) have unmatched timing estimation, which brings difficulty in timing closure;
- More and more VLSI circuits are designed in architectural styles, which require a new set of algorithms.

The paper consists of two parts, each of which focuses on several specific problems in VLSI physical synthesis when facing the new challenges.

- **Part-1 Place and route aware buffer Steiner tree construction**
  Efficient techniques are presented for the problem of buffered interconnect tree construction under blockage and routing congestion constraint. This part also contains

timing estimation and buffer planning for global routing and other early stages such as floorplanning. A novel path based buffer insertion scheme is also included, which can overcome the weakness of the net based approaches.

- **Part-2 Circuit clustering techniques with the application in Field-Programmable Gate Array (FPGA) technology mapping**

  The problem of timing driven n-way circuit partitioning with application to FPGA technology mapping is studied and a hierarchical clustering approach is presented for the latest multi-level FPGA architectures. Moreover, a more general delay model is included in order to accurately characterize the delay behavior of the clusters and circuit elements.

To my parent, YIU Wai Wah and SZE Ping Kwong.

ACKNOWLEDGMENTS

First, I would like to express my deepest gratitude to my advisor, Professor Jiang Hu for his guidance and kindness. He aroused my interest in the research of physical synthesis, piloted me when I was confused and encouraged me when I felt depressed. Besides, I would like to thank Professor Melvin R. Mercer, Professor Weiping Shi, Professor Sing-Hoi Sze and Dr. Charles J. Alpert for their advices.

Special thanks are given to Professor Ting-Chi Wang for his guidance when I first arrived at Texas A&M University.

Last, but not least, the greatest gratefulness is owed to my family. I am just nothing without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

A. Motivation and Aims

Very Large Scale Integration (VLSI) is the process of placing more than 10^4 of electronic components on a single chip, namely integrated circuits. It can generally be found in modern memories, computers, signal processors. Due to the very high complexity of VLSI processes, designs of VLSI chips using computer systems are essential in order to reduce the time-to-market (TTM) and cost per transistors, as well as improve total yield. This refers as to the Computer-Aid Design (CAD) or Electronic Design Automation (EDA).

Traditionally, the CAD for VLSI can simply be separated into three steps: high level synthesis (involving Behavioral Synthesis and Sequential Synthesis), logic synthesis (including technology mapping) and physical design synthesis[1, 2]. The detailed design process is shown in Figure 41 in Appendix A. High level synthesis consists of the construction of behavioral and functional specification and the conversion from specification into hardware descriptions such as Finite State Machine (FSM). Logic synthesis refers to the translation of high-level language descriptions into logic designs (a set of technology specific gates and interconnects, or netlist) and the optimization of the chip area, speed (delay) and testability. Physical design synthesis transforms the circuit representation into a geometric representation, the physical layout. It involves the process of circuit partitioning, floorplanning, placement, and routing. The objective of physical design is to minimize the chip area and to maintain chip performance. Since the design processes are very complicated, these steps often operate separately and, there is no interaction between them. However, the de-

This thesis follows the style of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*.

Table I. Projections of VLSI chips feature size

Year of 1^{st} Product shipment	1999	2002	2005	2008	2011	2014
Minimum feature size (nm) (DRAM half-pitch)	180	130	100	70	50	35

sign processes are changing due to the development of sub-micron and deep sub-micron VLSI technology in the last decade. And we are expecting the process continues toward nanometer VLSI technology which will bring us with more new problems.

B. Scaling Toward Nanometer VLSI Circuits

In 1965, Gordon E. Moore, the co-founder of Intel Corporation, predicted that “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year”. After the 70’s, this prediction is generally formulated as “the number of transistors on integrated circuits will be doubled every 18 months.” The projection is usually referred to as “The Moore’s Law”.

The Moore’s Law is roughly valid throughout the past several decades due to the rapid development of fabrication technology. We can observe that the VLSI feature size, which is usually attributed by DRAM half pitch of the chip, shrinks along with the process generations. The shrinking in chip element size and increase in chip density is generally referred to as the “scaling” of VLSI circuits. According to several reports [3, 4] , the feature size will be 70nm and 50nm in 2008 and 2011 respectively. The information is shown in Table I.

We can use the scaling factor S to describe the scaling effect for consecutive generations (Typically, $S = 1.5$). The gate delay can be characterized by Equation (1.1) as shown in Figure 1.

$$t_{gate} = \frac{C_{gate}V_{DD}}{I_{DS}} \tag{1.1}$$

When the scaling proceed, we have $\frac{C_{gate}}{S}$, $\frac{V_{DD}}{S}$ and $\frac{I_{DS}}{S}$. Therefore, gate delay is reduced by a factor S for each generation.

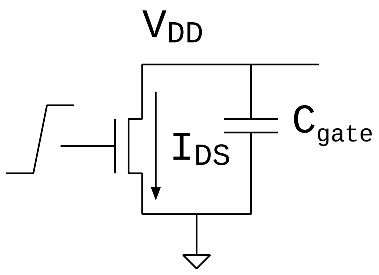


Fig. 1. Scaling of the gate delay

However, scaling has imposed a difference effect on the interconnect delay as shown in Figure 2. The interconnect delay can be formulated by Equation (1.2), where r is the unit length resistance, c is the unit length capacitance, k_1 and k_2 is the resistive and capacitive coefficient, w_s is the spacing between adjacent interconnects.

$$t_{interconnect} = rcl^2 = \frac{k_1}{wh}\frac{k_2 h}{w_s}l^2 \tag{1.2}$$

If the interconnect dimension and spacing is scaled uniformly, the local interconnect delay will become $rcS^2(\frac{l}{S})^2$ so that the delay remains unchanged during scaling. However, the length of global interconnect does not scale downward according to the shrink of feature size. On the contrary, it scales upward slightly with the chip size S_{chip}. As a result, the delay along global interconnect becomes $rcS^2(lS_{chip})^2$ which is a quadratic relationship to S and becoming much worse when comparing to the gate delay. Hence, as the VLSI technology scales downward, the interconnect delay has already dominated the total path delay, which includes interconnect and gate delays. (See Figure 3 [3]).

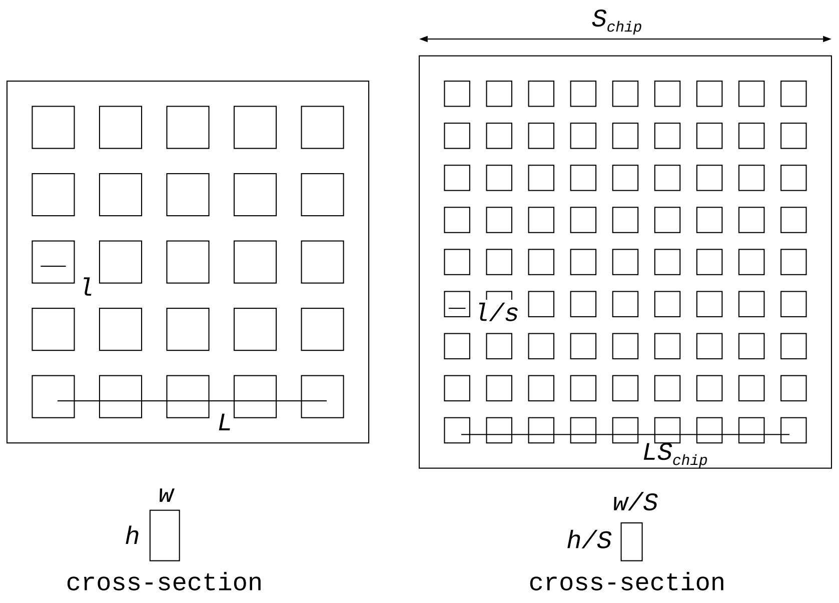


Fig. 2. Scaling of the local and global interconnect delay

The decrease in chip feature size and increase in complexity of chip design also bring the following effects:

- more hierarchical design schemes are needed.
- noise effect is exacerbated.
- power dissipation becomes unmanageable.

As a result, the link between each design stage is loosing. For example, a verified design in logic synthesis steps may violate the rules in physical design steps. Also, a placement with minimum “wirelength” does not produce a final layout with minimum clock cycle.

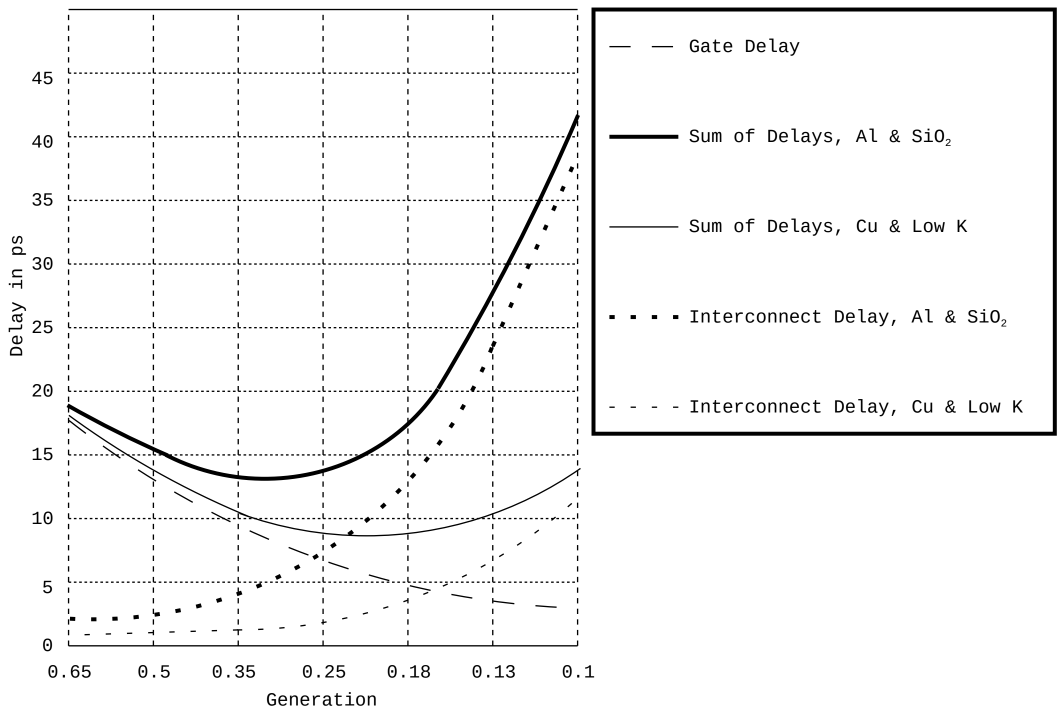


Fig. 3. Interconnection projections

1. Scaling in FPGA

Field-programmable gate array (FPGA) is one of the fastest growing semiconductor sectors. In fact, the growth of FPGA industry is faster than the average of the semiconductor industry. This can be justified by the following examples [5]. In 1988, the Altera MAX 5000 family had only 600 to 3,750 gates, while in 1999, each APEX 20K devices provide 60,000 to 1.5 million usable gates. Similarly, for the Xilinx XC2000 device family introduced in 1985, each device has 1200 to 1800 logic gates, while in 1998, Xilinx Virtex devices provides 58K to 4M gates.

Although the clock speed of FPGAs is comparatively slower than high-end custom-designed devices such as microprocessor, there are many problems related to the increasing complexity of FPGA. For example, there are becoming more hierarchical FPGA architectures to remedy the FPGA design difficulty. In fact, one example of hierarchical FPGA architectures can be found in the APEX20K device family which uses a two-level hierarchy (see Figure 23). Besides, the delay along interconnect is also increasing as the increase in chip size and decrease in size of the logic array blocks.

C. Contribution

This research project focuses on solving the physical design problems due to the rapid scaling of VLSI systems. The work can be divided into two parts: interconnect tree synthesis with buffer insertion and circuit clustering for FPGA technology mapping.

1. Interconnect Tree Synthesis with Buffer Insertion

As mentioned in previous section, interconnect delay is dominating the total delay in the circuit. Buffer insertion is one of the most effective methods to reduce wire delay. Moreover, since buffer insertion improves interconnect timing, the timing estimation in different stage using different buffering assumption can differ a lot. This actually is one of the reasons why the link between different design stages is getting loose. More importantly, sometimes an inaccurate timing estimation in early stages is vital to the timing closure of the design process.

Due to the high importance of buffer insertion to physical synthesis and timing, I have performed three projects related to buffering and interconnect tree synthesis.

- **Place and route aware buffered Steiner tree construction**

  A tree adjustment technique is proposed which modifies a given Steiner tree and simultaneously handles the objectives of timing, placement and routing congestion. To the best of my knowledge, this is the first study which simultaneously considers these three objectives for the buffered Steiner tree problem. Experimental results confirm the effectiveness of the algorithm while it achieves up to $20\times$ speed-up when comparing with the state-of-the-art algorithm. The project is described in Chapter III.

- **Accurate estimation of global buffer delay within a floorplan**

  The closed formed delay expressions of a buffered interconnect are extended to show how one can model the blocks into a simple delay estimation technique that applies both to two-pin and to multi-pin nets. Even though the formula uses one buffer type, it shows remarkable accuracy in predicting delay when compared to an optimal realizable buffer insertion solution. Potential applications include wire planning, timing analysis during floorplanning or global routing. Our experiments show that our approach accurately predicts delay when compared to constructing an realizable buffer insertion with multiple buffer types. The project is detailed in Chapter IV.

- **Path based buffer insertion**

  A novel path based buffer insertion scheme is introduced which can overcome the weakness of the conventional net based approaches. Experimental results show that our method can efficiently reduce buffer/gate cost significantly (by 71% on average) when compared to traditional net based approaches. In literature, This is the first work on path based buffer insertion and simultaneous gate sizing. The project is explained in Chapter V.

2. Circuit Clustering for FPGA Technology Mapping

Owing to the rapid growth of FPGA density, we have more logic blocks in a FPGA chip. The delay in interconnect is becoming more important. In the second part of this thesis, two projects are presented to solve physical design problems related to latest FPGA designs.

- **Optimal circuit clustering for delay minimization under a more general delay model**

  For the problem of area-constrained clustering for delay minimization, I proposed a more general delay model, which practically takes variable interconnect delay into account. The delay model is particularly applicable when allowing the back-annotation of actual delay information to drive the clustering process. An algorithm is presented to the clustering problem and can be proved to solve the problem optimally in polynomial time. This work is presented in Chapter VII.

- **Multi-level circuit clustering for delay minimization**

  An effective scheme is proposed for multi-level circuit clustering for delay minimization, which is applicable to hierarchical FPGAs. In fact, our algorithm is the first one for the general multi-level circuit clustering problem with more than two levels. The project is included in Chapter VIII.

CHAPTER II

PERFORMANCE-DRIVEN INTERCONNECT TREE SYNTHESIS AND BUFFER INSERTION

Due to the development of fabrication technology, the VLSI synthesis processes are entering the deep sub-micron range (feature size $<$ 500 nm). According to the projection in International Technology Roadmap for Semiconductors 2000, the VLSI chips feature size will drop to 100nm and 70nm in 2007 and 2010 respectively. Owing to the tremendous drop in chip feature size, the interconnect delay becomes a dominant part of the total path delay, which includes both the interconnect and gate delays. Since the importance of interconnect delay on VLSI timing optimization increases rapidly, some timing-driven interconnect routing methodologies adopted in current design cycles no longer produce feasible timing solution. Buffering in the interconnect tree is one of the effective techniques to optimize overall system performance and link between different design stages.

However, current buffer insertion techniques are too specific and their applications are very limited within the overall design cycle. As a result, my research is to develop a comprehensive interconnect tree synthesis algorithm which simultaneously handles multiple objectives such as buffer and wiring costs, obstacles, routing congestion, and signal integrity while the algorithm can be applied at different stages in the VLSI Computer-Aided Design cycles.

We focus on the following three objectives of the buffered tree synthesis problem in the next three chapters.

- **Place and route aware buffered Steiner tree construction**

  In order to achieve timing closure on increasingly complex IC designs, buffer insertion needs to be performed on thousands of nets within an integrated physical synthesis system. In most of previous works, buffers may be inserted at any open space. Even when there may appear to be space for buffers in the alleys between large blocks, these regions are often densely packed or may be useful later to fix critical paths. In addition, a buffer solution may inadvertently force wires to go through routing congested regions. Therefore, within physical synthesis, a buffer insertion scheme needs to be aware of both placement congestion and routing congestion of the existing layout and so it has to be able to decide when to insert buffers in dense regions to achieve critical performance improvement and when to utilize the sparser regions of the chip. With the proposed Steiner tree adjustment technique, this work aims at finding congestion-aware buffered Steiner trees. Our tree adjustment technique takes a Steiner tree as input, modifies the tree and simultaneously handles the objectives of timing, placement and routing congestion. To our knowledge, this is the first study which simultaneously considers these three objectives for the buffered Steiner tree problem. Experimental results confirm the effectiveness of our algorithm while it achieves up to $20\times$ speed-up when comparing with the state-of-the-art algorithm [6].

- **Accurate estimation of global buffer delay within a floorplan**
  Closed formed expressions for buffered interconnect delay approximation have been around for some time. However, previous approaches assume that buffers are free to be placed anywhere. In practice, designs frequently have large blocks that make the ideal buffer insertion solution unrealizable. The theory of [7] is extended to show how one can model the blocks into a simple delay estimation technique that applies both to two-pin and to multi-pin nets. Even though the formula uses one buffer type, it shows remarkable accuracy in predicting delay when compared to an optimal realizable buffer insertion solution. Potential applications include wire planning, timing analysis during floorplanning or global routing. Our experiments

show that our approach accurately predicts delay when compared to constructing an realizable buffer insertion with multiple buffer types.

- **Path based buffer insertion**

  Along with the progress of VLSI technology, buffer insertion plays an increasingly critical role on affecting circuit design and performance. Traditional buffer insertion algorithms are mostly net based and therefore often result in sub-optimal delay or unnecessary buffer expense due to the lack of global view. In this paper, we propose a novel path based buffer insertion scheme which can overcome the weakness of the net based approaches. We also discuss some potential difficulties of the path based buffer insertion approach and propose solutions to them. A fast estimation on buffered delay is employed to improve the solution quality. Gate sizing is also considered at the same time. Experimental results show that our method can efficiently reduce buffer/gate cost significantly (by 71% on average) when compared to traditional net based approaches. To the best of our knowledge, this is the first work on path based buffer insertion and simultaneous gate sizing.

CHAPTER III

A PLACE AND ROUTE AWARE BUFFERED STEINER TREE CONSTRUCTION

A. Introduction

It has been widely recognized that interconnect becomes a dominating factor for modern VLSI circuit designs. A key technology to improve interconnect performance is buffer insertion. A recent study by Intel [8] speculates that for $35nm$ technology, 70% of the cells on a chip will be buffers.

Early works on buffer insertion are mostly focused on improving interconnect timing performance. The most influential pioneer work is van Ginneken's dynamic programming algorithm [9] that achieves polynomial time optimal solution on a given Steiner tree under Elmore delay model [10]. In [11], Lillis *et al.* extended van Ginneken's algorithm by using a buffer library with inverting and non-inverting buffers, while also considering power consumptions.

The major weakness of the van Ginneken approach is that it requires a fixed Steiner tree topology which makes the final buffer solution quality dependent on the input Steiner tree. Even though it is optimal for a given topology, the van Ginneken algorithm will yield poor solutions when it is fed with a poor topology. To overcome this problem, several works have proposed to simultaneously construct a Steiner tree while performing buffer insertion [12, 13, 14]. Although the simultaneous algorithms generally yield high quality solution, their time complexities are very high. A different approach to solve the weakness of van Ginneken's algorithm is proposed by Alpert *et al.* [15]. They construct a "buffer-aware" Steiner tree, called C-Tree for van Ginneken's algorithm. Despite being a two-stage sequential method, it yields solutions comparable in quality to simultaneous methods, while consuming significantly less CPU time.

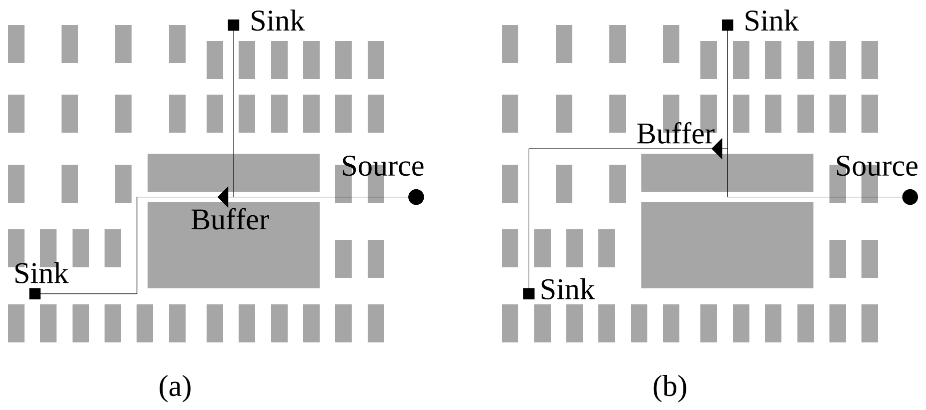


Fig. 4. A minimum Steiner tree as shown in (a) may force buffers being inserted at dense regions. A placement congestion aware buffered Steiner tree, which is shown in (b), will enable a buffer solution at sparse regions.

Recent trends towards hierarchical (or semi-hierarchical) chip design and system-on-chip design force certain regions of a chip to be occupied by large building blocks or IP cores so that buffer insertion is not permitted. These constraints on buffer locations can severely hamper solution quality, and these effects have to be considered in the buffered path [16, 17, 18] class of algorithms. Though optimal, they are only applicable to two pin nets. Works that handle restrictions on buffer locations while performing simultaneous Steiner tree construction and buffer insertion are proposed in [19, 20], which can provide high quality solutions though the runtimes are too exorbitant to be used in a physical synthesis system. In [21], a Steiner tree is rerouted to avoid buffer blockages before conducting buffer insertion. This sequential approach is fast, but sometimes unnecessary wiring detours may result in poor solutions. An adaptive tree adjustment technique is proposed in [22] to obtain good solution results efficiently.

The huge number of nets that require buffering means that resources have to be allocated intelligently. For example, large blocks closely placed together create narrow alleys that are magnets for buffers since they are the only locations that buffers can be inserted for those routes that cross over these blocks. But competition for resources for these routes is very fierce. Inserting buffers for less critical nets can eliminate space that is needed for more critical nets which require gate sizing or other logic transforms. Further, though no blockages lie in the alleys, these region could already be packed with logic and feasible space may not exist for the buffers. If the buffer insertion algorithm cannot recognize this scenario, after a buffer is inserted into this congested space, placement legalization may shift it out too far from its original location due to cell overlap. Hence, whenever possible one should avoid denser regions unless it is absolutely critical. For example, Figure 4(a) shows a multi-pin net which is routed through a dense region or “hot spot”, and (b) shows that the Steiner point is moved outside of the dense region in order to obtain an improved buffer insertion result.

Similarly, a buffer solution may inadvertently force wires to go through routing congested regions such as in Figure 5(a). Such solution causes wire detours in routing stage. Generally speaking, an L-shaped wire has flexibilities on avoiding congestions without increasing wirelength. These flexibilities may be ripped off when inserted buffers make the L-shaped wire into a set of straight connections. If we let the buffering algorithm be aware of the routing congestions, these flexibilities can be kept for congestion avoidance in later routing stage.

To the best of our knowledge, the only published work about placement congestion aware buffer insertion is the porosity aware buffered Steiner tree problem addressed in [6]. This work integrates the length-based buffer insertion [23] with a plate-based tree adjustment to obtain a Steiner tree at regions with greater porosity. However, the routing congestion is not considered and the runtime overhead is too large.

This work adopts the sequential method in constructing a Steiner tree and the tree is then fed into a van Ginneken style buffer insertion algorithm. However, before buffer insertion, a **timing-driven plate-based tree adjustment algorithm** is applied so that **both the placement and routing congestion** are considered. Buffered paths between nodes are

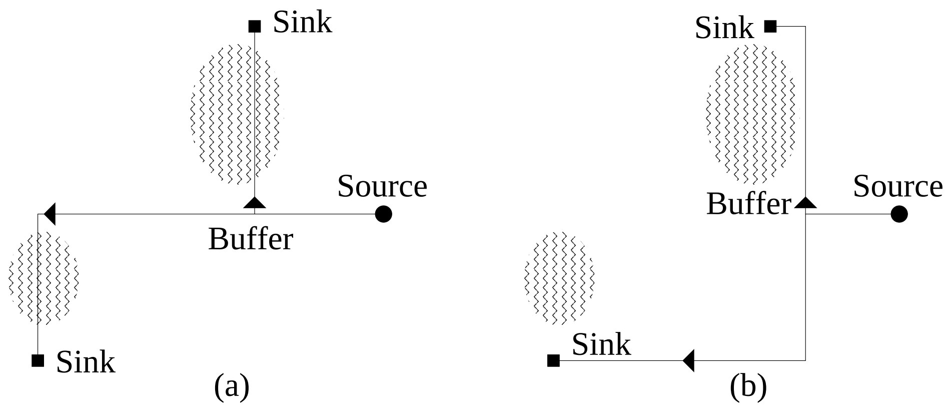


Fig. 5. A minimum buffered Steiner tree as shown in (a) may force wires going through routing congested regions. A routing congestion aware buffered Steiner tree, which is shown in (b), will enable a buffer solution at less congested regions.

found through utilizing the analytical form buffered path solution and a congestion cost driven maze routing. In [6], each solution during path search is characterized by its cost and downstream capacitance, thus, the solution set is a two-dimensional array while at the same time, the tree adjustment is not driven by timing optimization. In the maze routing of this work, only the congestion cost is considered for each candidate solution. Therefore, the candidate solution set is a one-dimensional array and this smaller-sized solution set enables a faster computational speed. In fact, our experiment shows that, when comparing with [6], our algorithm achieves 7% better timing, 9% lower total placement and routing congestion but runs up to 20 times faster.

B. Problem Formulation

In this paper, we use a tile graph to capture the placement and routing congestion information and at the same time reduce the complexity of our problem. A tile graph is represented as $G = (V_G, E_G)$ such that $V_G = \{g_1, g_2, ...\}$ is a set of tile and E_G is a set of boundaries each (g_i, g_j) of which is between two adjacent tile g_i and g_j.

If a tile $g_i \in V_G$ has an area of $A(g_i)$ and its area occupied by placed cells are $a(g_i)$, the placement density is defined as the area usage density $d(g_i) = \frac{a(g_i)}{A(g_i)}$. Let $W(g_i, g_j)$ be the maximum number of wires that can be routed across the tile boundary (g_i, g_j) and $w(g_i, g_j)$ be the number of wires crossing (g_i, g_j). Similarly, the boundary density is $d(g_i, g_j) = \frac{w(g_i,g_j)}{W(g_i,g_j)}$.

A net is represented as a set of sinks $V_{sink} = \{v_1, v_2, ..., v_n\}$ and a source node v_0. Each sink $v_i \in V_{sink}$ at location (x_i, y_i) is associated with a load capacitance $c(v_i)$ and a required arrival time $q(v_i)$. The source node is at (x_0, y_0) and is associated with a driver with driver resistance R_d. For simplification, a buffer type with input capacitance C_b, intrinsic delay t_b and output resistance R_b is used. The unit wire resistance is r and the unit wire capacitance is c. We use Elmore model [10] for interconnect delay and RC switch model for driver and buffer delay.

Problem Definition:(Buffered Steiner Tree for Placement and Routing Congestion Mitigation) *Given a net $N = \{v_0, v_1, \ldots, v_n\}$ with source v_0 and sinks $\{v_1, \ldots, v_n\}$, load capacitance $c(v_i)$ and required arrival time $q(v_i)$ for $1 \leq i \leq n$, tile graph $G(V_G, E_G)$, and a buffer type b, construct a Steiner tree $T(V, E)$, in which $V = N \cup V_{Steiner}$ and edges in E span every node in V, such that a buffer insertion solution that satisfies $q(v_i)$ is obtained with a minimum congestion cost S.*

The congestion cost can be formulated based on the application but the algorithm should have the flexibility to take any kind of congestion cost formulation. In this paper, we adopt the following definition. The placement cost $p(g_i)$ of placing a buffer in a tile g_i is the square of the density $d(g_i)$; while the routing cost $p(g_i, g_j)$ crossing a tile boundary (g_i, g_j) is the square of the boundary density $d(g_i, g_j)$. With this cost definition, we do not use an infinite cost for overflow. In reality, if the placement or routing on a dense region really helps improving slack or other design objectives, moving the previously placed and routed elements in the next design cycle would be more beneficial.

C. The Algorithm

1. Methodology Overview

Since simultaneous Steiner tree construction and buffer insertion is computationally expensive for practical circuit designs, we propose to solve the congestion aware buffered Steiner tree problem through the following three stages: 1)Initial timing-driven Steiner tree construction[1]; 2)Tree adjustment for congestion improvement; 3)Van Ginneken style buffer insertion.

Stage 1 can be accomplished by any heuristics while we apply “buffer aware” C-Tree algorithm[15]. Stage 2 contains the key ideas behind the algorithm. The tree adjustment phase modifies the existing timing-driven Steiner tree in an effort to reduce congestion cost while maintaining the tree’s high performance. It allows Steiner points to migrate outside of congested tiles into lower-congestion tiles while maintaining (if not improving) performance. Finally, in Stage 3 the resulting tree topology is fixed for van Ginneken style buffer insertion. Since we use known algorithms for Stages 1 and 3, the rest of the discussion focuses on stage 2, which is the main contribution of this work.

2. Algorithm Motivation

The basic idea for the tree adjustment is to perform a simplified simultaneous buffer insertion[2] and local tree topology modification so that the Steiner nodes and wiring paths can be moved to less congested regions without significant disturbance on the timing performance obtained in Stage 1. Also for the sake of simplification, we assume a single “typical” buffer

[1]We choose a timing-driven Steiner tree algorithm here since it is fast and easy to implement with our proposed tree adjustment technique. However, any Steiner tree (e.g., congestion-aware Steiner tree) can also be fed into stage 2 of our overall algorithm.

[2]Note that the “buffer insertion” in tree adjustment is for timing estimation. Actual buffer insertion is performed in stage 3 of our algorithm.

type, the Elmore delay model for interconnect and a switch level RC gate delay model for this tree adjustment. The tree adjustment traverses the given Steiner topology in a bottom-up fashion similar to van Ginneken's algorithm. During this process, candidate buffering and routing solutions are propagated from leaf nodes towards the source. At a Steiner node, candidate solutions from its two child branches are merged. Therefore, we can consider the propagation and merging process separately.

The buffered Steiner tree problem is inherently very difficult to solve and including congestions into account makes the complexity even more formidable. In this problem, three major factors (1) timing (2) load capacitance and (3) congestion cost have to be considered simultaneously on a two-dimensional Manhattan plane. Note that load capacitance needs to be evaluated and maintained for delay calculation even though it is not a part of objectives. In order to make the computation time practical, solution quality has to be sacrificed to a certain degree. Of course, the sacrifice on solution quality need to be as small as possible while the computation cost reduction has to be substantial. In [6], a length-based buffer insertion [23] scheme is employed to reduce complexity. Instead of maintaining all the timing and load capacitance information, only the maximum driving load for each buffer/driver is enforced as a rule of thumb. Therefore, the number of factors is reduced from three to two and the computation speed is acceptable. However, the experimental results in [6] show that runtime is almost doubled just because of considering congestions. The runtime bottleneck is due to the fact that buffering solution has to be searched along with node-to-node[3] paths in a two-dimensional plane since low congestion paths have to be found at where the buffers are needed.

If we can predict where buffers are needed in advance, then we can merely focus on searching low congestion paths and the number of factors to be considered can be further re-

[3]The node may be the source node, a sink node or a Steiner node of degree greater than two. Thus, degree-2 Steiner nodes are not included here.

duced to one. If we diagnose the mechanism on how buffer insertion improves interconnect timing performance, it can be broken down into two parts: (1) regenerating signal level to increase driving capability for long wires and (2) shielding capacitive load at non-critical branches from the timing critical path. In a Steiner tree, buffers that play the first role are along a node-to-node path while buffers for the second purpose are normally close to a branching Steiner node. The majority of buffer insertion algorithms such as van Ginneken's method are dynamic programming based and have been proved to be very effective for both purposes. However, optimal buffer solutions along a node-to-node path can be found analytically if the driving resistance for this path is known [24, 25]. This fact suggests that we may have a hybrid approach in which buffers along paths are placed according to the closed form solutions while the buffers at branching nodes are still solved by dynamic programming, i.e., analytical buffered path solutions replace both the wire segmenting [24] and candidate solution generations at segmenting points in the bottom-up dynamic programming framework. The only problem of this approach is that the driving resistance of a path to be processed is not known in this bottom-up procedure. This can be solved by sampling a set of anticipated upstream resistance values and generating candidate buffering solutions for each anticipated value. Different sampling rate may result in different solution quality and runtime tradeoffs. Computing candidate buffered paths analytically is faster than dynamic programming, because the complexity of a dynamic programming approach has a quadratic dependence on the segmenting size while analytical approach has only linear dependence on the upstream resistance sampling size.

3. Steiner Node Adjustment

For a Steiner node, we find a few nearby tiles with the least congestion. In each of these tiles, we consider an alternative Steiner node there. Therefore, the candidate solutions from child branches are propagated to not only the original Steiner node but also these

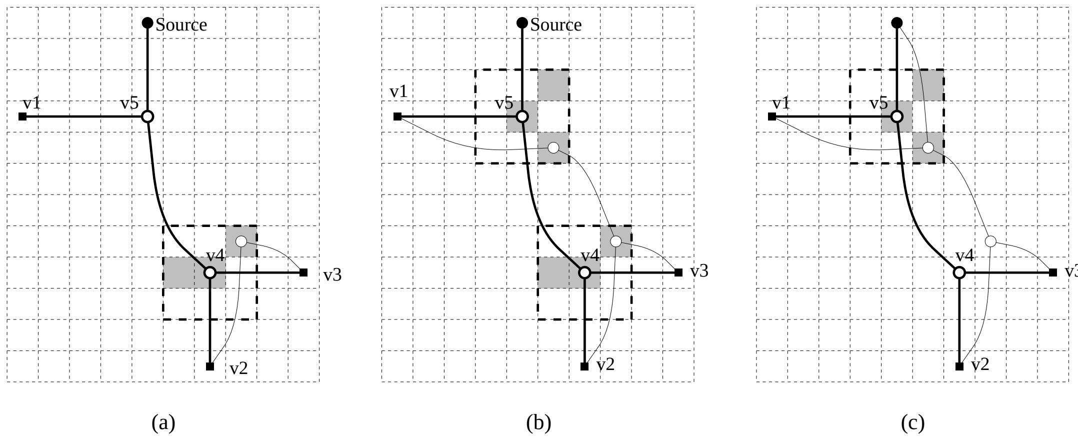


Fig. 6. (a) Candidate solutions are generated from v_2 and v_3 and propagated to every shaded tile for v_4. (b) Solutions from v_1 and every shaded tile for v_4 are propagated to v_5 and its alternative nodes. (c) Solutions from v_5 and its alternative nodes are propagated to the source and the thin solid lines indicate an alternative tree that may result from this process.

alternative Steiner nodes. For a node v, we define *expanded node set* as its alternative nodes as well as the node itself and denote this set as $P(v)$. The selection of alternative nodes in $P(v)$ can be controlled by the placement cost and wiring cost of the tiles[4] or for different objectives of Steiner node adjustment, other selection schemes may be applied. After candidate solutions from child branches are merged at the original Steiner node and each of the alternative Steiner nodes, the merged solutions are propagated further towards the source. This process is illustrated in Figure 6 where the tiles for the expanded node set are shaded. The alternative Steiner nodes enable alternative tree topologies and only the topology that is part of the best solution at the root will be finally selected. Therefore, this tree adjustment is a dynamic selection.

Usually, we restrain the alternative Steiner nodes to be close to the original Steiner node so that the perturbation to the original timing driven Steiner tree is limited. We define

[4]See Section D for the $P(v)$ selection details adopted in our experiment.

the adjustment flexibility which represent what the maximum distance of the alternative Steiner nodes can be away from the original Steiner node. For example, in Figure 6, all alternative nodes are enclosed in 3×3 tiles and so we refer it as the 3×3 tiles adjustment flexibility. By this definition, we can have a larger flexibility for the selection of expanded node set if, for instance, we adopt the 5×5 tiles adjustment flexibility in our program. A larger adjustment flexibility would lead to a higher possibility for the original Steiner node to migrate outside of the congested area.

The main difference between our proposed technique and the work of [6] is that a regular array of tiles are considered for alternative Steiner node in [6] while our selection on alternative Steiner nodes is according to the congestion for nearby tiles. This is based on our observation that only the nearby tiles with relatively low buffer placement or routing congestion cost worth considering to be the alternative Steiner node. Moreover, due to the irregularity and flexibility of expanded node set in our algorithm, we have more choices and can pick alternative Steiner nodes other than the immediate neighbor nodes for better congestion reduction and higher efficiency. On the contrary, since [6] features a regular array of tiles, a larger flexibility for alternative Steiner nodes is desired, the runtime would become unbearable. This can be shown by our experiments in Section D.

4. Minimum Cost Buffered Path

Our work distinguishes from [6] significantly on the candidate solution propagation between two nodes. In [6], a length based buffer insertion is integrated with the minimum congestion cost path search, so the process is not timing-driven. However, our algorithm has the required arrival time information in an intermediate solution during the bottom-up propagation and pruning process, and hence our work is capable of handling not only the congestion reduction but also the timing optimization. In order to achieve these two objectives, we separate the buffer insertion for timing from the minimum congestion cost path

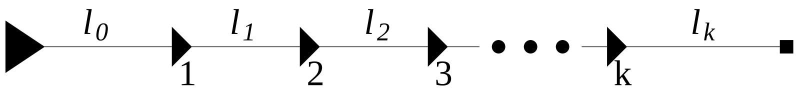


Fig. 7. Buffer positions along a path.

search. For a path of length l with driver resistance R_d at one end and a load capacitance C_L on the other end, the number of buffers k that minimizes the path delay is obtained in [24] as:

$$k = \left\lfloor -\frac{1}{2} + \frac{1}{2}\sqrt{1 + \frac{2(rcl - r(C_b - C_L) - c(R_b - R_d))^2}{rc(R_b C_b + t_b)}} \right\rfloor$$

The k buffers separate the path into $k+1$ segments of length l_0, l_1, ...l_k as illustrated in Figure 7. According to [24], The length of each segment can be obtained through:

$$\begin{aligned} l_0 &= \frac{1}{k+1}\left(l + \frac{k(R_b - R_d)}{r} + \frac{C_L - C_b}{c}\right) \\ l_1 &= \ldots = l_{k-1} = \frac{1}{k+1}\left(l - \frac{R_b - R_d}{r} + \frac{C_L - C_b}{c}\right) \\ l_k &= \frac{1}{k+1}\left(l - \frac{R_b - R_d}{r} - \frac{k(C_L - C_b)}{c}\right) \end{aligned} \tag{3.1}$$

Then, we explain our buffered path routing technique by an example. For the thickened path in Figure 8(a), if we know the driving resistance at v_1 and load capacitance at v_4, we may obtain the optimal buffer positions at v_2 and v_3. However, if we connect v_1 and v_4 in a two-dimensional plane, there are many alternative paths between them and the optimal buffer locations form rows along diagonal directions. The tiles for the optimal buffer locations are shaded in Figure 8(a). Therefore, if we connect v_1 and v_4 with any monotone path and insert a buffer whenever this path passes through a shaded tile, the resulting buffered path should have the same minimum delay. The thin solid curve in Figure 8(a) is an example of an alternative minimum delay buffered path. Certainly, different buffer paths may have different congestion cost. Then the minimum congestion cost buffered path can be found by running the Dijkstra's algorithm on the tile graph which is demonstrated

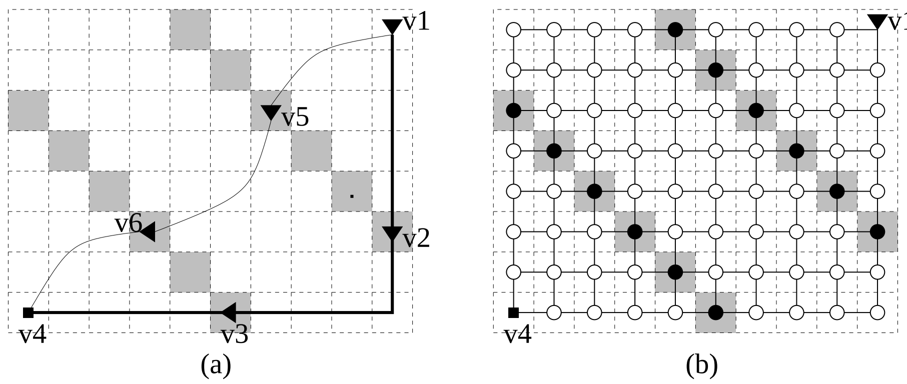


Fig. 8. Find low congestion path with known buffer positions indicated by the shaded tiles.

in Figure 8(b). In Figure 8(b), each solid edge corresponds to a tile boundary and its edge cost is the corresponding wiring congestion cost (=square of its boundary density). There are two types of nodes, the empty circle nodes that have zero cost and filled circle nodes that have cost equal to the placement congestion cost in corresponding tile. In conclusion, the shortest path obtained in this way produces a buffered path with both good timing and low congestion cost.

An issue need to be handled by this approach is that the upstream resistance R_d is unknown in the bottom-up solution propagation process. However, we are aware that the lower bound on the upstream resistance is $\underline{R} = \min(R_d, R_b)$ and the upper bound $\overline{R}$ is $\max(R_d, R_b)$ plus the upstream wire resistance[5]. Then, we can sample a few values between $\underline{R}$ and $\overline{R}$, and find the minimum cost buffered path for each value. Since the timing result is not sensitive to the upstream resistance, normally the sampling size is very limited.

5. Overall Algorithm

In our algorithm, each intermediate buffer solution is characterized by a 4-tuple $s(v, c, q, w)$ in which v is the root of the subtree, c is the downstream load capacitance seen from v, q

[5]The maximum upstream wire resistance can be derived from the length of maximum buffer-to-buffer interval. This is also mentioned in [23].

```
Procedure: $FindCandidates(v)$
Input:    Current node $v$ to be processed
Output: Candidate solution set $S(P(v))$
Global:  Steiner tree $T(V, E)$
           Tile graph $G(V_G, E_G)$
1. If $v$ is a sink
      $S(v) \leftarrow \{(v, c(v), q(v), 0)\}$
      $S(P(v)) \leftarrow \{S(v)\}$
      Return $S(P(v))$
2. $v_l \leftarrow$ left child of $v$
   $S(P(v_l)) \leftarrow FindCandidates(v_l)$
3. $S_l(P(v)) \leftarrow Propagate(S(P(v_l)), P(v))$
4. If $v$ has only one child
      Return $S_l(P(v))$
5. $v_r \leftarrow$ right child of $v$
   $S(P(v_r)) \leftarrow FindCandidates(v_r)$
6. $S_r(P(v)) \leftarrow Propagate(S(P(v_r)), P(v))$
7. $S(P(v)) \leftarrow Merge(S_l(P(v)), S_r(P(v)))$
8. $Prune(S(P(v)))$
9. Return $S(P(v))$
```

Fig. 9. Core algorithm.

is the required arrival time at v and w is the accumulated congestion cost. A solution $s_i(v, c_i, q_i, w_i)$ is said to be dominated by another solution $s_j(v, c_j, q_j, w_j)$, if $c_i \geq c_j, q_i \leq q_j$ and $w_i \geq w_j$. A set of buffer solutions $S(v)$ at node v is a non-dominating set when there is no solution in $S(v)$ dominated by another solution in $S(v)$.

The complete algorithm descriptions are given in Figures 9 and 10 where the basic operations are defined as follows.

- $Merge(S_l(v), S_r(v))$: merge solution set from left child of v to the solution set from the right child of v to obtain a merged solution set $S(v)$. For a solution $s_{i,l}(v, c_{i,l}, q_{i,l}, w_{i,l})$ from the left child and a solution $s_{j,r}(v, c_{j,r}, q_{j,r}, w_{j,r})$, they are merged to $s_k(v, c_k = c_{i,l} + c_{j,r}, q_k = \min(q_{i,l}, q_{j,r}), w_k = w_{i,l} + w_{j,r})$.

```
Procedure: $Propagate(S(P(v_i)), P(v_j))$
Input:  Candidate solutions at $P(v_i)$
        Expanded node set $P(v_j)$
Output: Candidate solution set $S(P(v_j))$
Global: Tile graph $G(V_G, E_G)$
1. $S(P(v_j)) \leftarrow \emptyset$
2. For each node $v_l \in P(v_j)$
3.   $S(v_l) \leftarrow \emptyset$
4.   For each node $v_k \in P(v_i)$
5.     For each anticipated upstream resistance $R_u$ at $v_l$
6.       $(\mathcal{L}, c, q) \leftarrow FindBufferPositions(R_u, v_l, v_k)$
         $w \leftarrow FindMinCostPath(v_l, v_k, G, \mathcal{L})$
         $S(v_l) \leftarrow S(v_l) \cup \{s(v_l, c, q, w)\}$
7.   $Prune(S(v_l))$
     $S(P(v_j)) \leftarrow S(P(v_j)) \cup S(v_l)$
8. Return $S(P(v_j))$
```

Fig. 10. Subroutine of propagating candidate solutions from one node set to another.

- $Prune(S(v))$: remove any solution $s_i \in S(v)$ that is dominated by another solution $s_j \in S(v)$.

- $FindBufferPositions(R_u, v_u, v_d)$: apply equations of (3.1) to find the set of buffer positions $\mathcal{L}$ for the minimum delay from node v_u to v_d assuming driving resistance R_u at v_u. The required arrival time q and downstream capacitance c at v_u are also returned.

- $FindMinCostPath(v_u, v_d, G, \mathcal{L})$: apply Dijkstra's algorithm to find the minimum cost path connecting v_u and v_d on tile graph $G(V_G, E_G)$. On tile graph G, the cost of each node $g \in V_G$ is $d^2(g)$ if $g \in \mathcal{L}$; otherwise zero. The cost of an edge in G corresponding to boundary between tiles g_i and g_j is $d^2(g_i, g_j)$. Return the path cost finally. (An example is shown in Section III.D.)

$FindCandidates(v)$ (Figure 9) is a recursive procedure which is similar to van Ginneken's algorithm such that at each node, we propagate the solutions from its children, merge and prune the solutions. However, the procedure $Propagate(S(P(v_i)), P(v_j))($ (Figure 10) for solution propagation adopts our analytical equations and shortest path algorithm described in Section III.D, which accelerates our algorithm when compare with the van Ginneken's dynamic programming approaches. When our algorithm terminates, we obtain a set of solutions with different timing and congestion cost tradeoffs at the root.

D. Experimental Results

All experiments are performed on a Sun Ultra Sparc 450 machine running in 400 MHz. Our experiments adopt the following parameters: $r = 0.184\Omega/\mu m$, $c = 0.124fF/\mu m$, $R_b = 246.3\Omega$, $C_b = 7.2fF$ and $t_b = 27.46ps$. The sampling size of upstream resistance is set to 3.

We implement our algorithm "Place and Route Aware Buffered Steiner Tree Construction" (PRAB) and the algorithm "Porosity Aware Buffered Steiner Tree Construction" in [6][6] (namely POROSITY in this paper) with C++ and compare them based on a set of industrial nets which is also used in [15] and [22]. In the benchmark nets, the driver resistance ranges from 271.8Ω to 2557.5Ω while the loading capacitance ranges from $2.9fF$ to $55.6fF$.

For each net, after we construct a tile graph, we randomly produce a set of buffer blockages and then calculate the placement density for each tile which range from 0 to 1. The boundary density for each tile boundary is also generated randomly ranging from 0 to 0.25 so that the importance of buffer placement cost and wiring congestion cost balance each other in the buffered Steiner trees.

[6]We compare our work to the algorithm in [6] since it is the latest published work with almost the same problem formulation and similar objectives.

In order to ensure a fair comparison, the buffered Steiner tree generation for both implementations follows the three-stage scheme mentioned in Section C-1 and therefore we can focus on the comparison between the congestion-aware tree adjustment step for both algorithms. After the tree adjustment in stage 2, our PRAB algorithm generates a set of solutions with different timing and congestion cost tradeoff but POROSITY produces a single solution which only considers the minimization of estimated congestion cost. Therefore, before running the van Ginneken style buffer insertion, we only pick the one with least estimated cost in our solution set for the comparison with POROSITY.

1. Results for Real Multi-Sink Nets

Table II. Information of all testcases

net	sink	graph size (row×col)	Stage 1 req/ps
mcu0s5	18	32×34	5949.31
mcu1s9	19	38×29	5936.46
n1071	17	23×37	1723.04
n18905	29	67×53	-1062.66
n313	19	36×45	646.43
n7866	32	112×45	-635.27
n8692	21	79×27	284.87
n8702	43	106×68	-2031.09
n8730	20	55×33	-538.76
pointer3	20	64×53	-613.75

In the first experiment, we pick the expanded node set from the 3×3 neighbor tiles of each Steiner node – 3×3 tiles adjustment flexibility. Each expanded node set consists of three tiles: the tile g_1 containing the original Steiner node, the tile g_2 with the smallest buffer placement cost $p(g_2)$, and the tile g_3 with the smallest wiring cost average, which is defined as $\frac{\sum_{g' \in N(g_3)} p(g_3, g')}{|N(g_3)|}$ where $N(g) = \{g' | (g, g') \in E_G\}$. To achieve the same Steiner node adjustment flexibility, we implement POROSITY with the “plate size” to be 3×3. Table II shows the information of all testcases. For each net, the second column shows the number of sinks. “graph size” displays the number of columns and rows in the tile graph which encloses the routing region of the net. “Stage 1 req” represents the required arrival time at source node which is propagated from all sinks along the initial timing driven Steiner tree without buffer insertion (just after Stage 1). Table III shows the comparison between the POROSITY algorithm and our PRAB algorithm. Columns 2-7 represents the results of POROSITY [6] while columns 8-13 shows our results. “req” represents the required arrival time at source node for the buffered Steiner tree generated by all three stages for both algorithms. Then, “imp” means the timing improvement by each three-stage implementation when comparing to the first-stage Steiner tree. (Note that both algorithms generate a negative improvement for the net $mcu0s5$ since in the process of congestion mitigation, the total wire-length increases and in turn decreases the required arrival time at source node.) “cost” shows the total congestion cost that is a sum of wiring cost “w.cost” and buffer placement cost “b.cost”, which are induced by the final buffered Steiner tree on the tile graph. Although we perform all three stages for both implementations, only the CPU time for second stage is shown for proper comparisons.

From Table III, we observe the following:

- Our algorithm for tree adjustment outperforms POROSITY [6] in the sense of both timing improvement (7%) and congestion cost evaluation(7%). Particularly, we are better than POROSITY by 15% in buffer placement cost. This justifies our claim that our tree adjustment algorithm not only operates in a timing-driven manner but also simultaneously handles the wiring and placement congestion while [6] does not.

- PRAB runs with about 15 times speed-up. The main reason for the efficiency is that we selectively pick the expanded node set according to the congestion nearby the

Table III. Comparison between POROSITY and PRAB under 3×3 tiles adjustment flexibility.

net	POROSITY[6]						Our algorithm PRAB					
	req/ps	imp/ps	cost	w.cost	b.cost	CPU/s	req/ps	imp/ps	cost	w.cost	b.cost	CPU/s
mcu0s5	5919.85	-29.46	3.71	1.99	1.72	6.23	5922.10	-27.21	4.07	1.99	2.08	1.88
mcu1s9	6047.52	111.06	4.13	1.50	2.63	9.89	6082.39	145.93	2.67	1.10	1.57	0.65
n1071	1787.23	64.19	3.58	1.08	2.50	7.64	1792.37	69.33	3.08	1.45	1.63	1.93
n18905	-869.89	192.77	5.06	2.88	2.19	23.83	-791.84	270.82	4.69	3.07	1.63	4.69
n313	870.89	224.46	2.78	1.90	0.88	6.02	869.68	223.25	2.66	1.77	0.90	0.65
n7866	-20.83	614.44	3.19	2.24	0.95	141.67	32.30	667.56	2.69	2.02	0.68	8.06
n8692	639.72	354.85	3.39	1.81	1.59	18.27	660.48	375.61	4.36	1.74	2.62	1.00
n8702	347.54	2378.63	7.10	2.45	4.65	261.26	386.53	2417.62	5.64	2.63	3.01	11.89
n8730	99.45	638.21	8.29	1.61	6.68	11.42	169.78	708.54	8.02	1.53	6.49	1.12
pointer3	-328.44	285.30	3.56	1.99	1.57	41.36	-289.58	324.16	2.77	1.77	1.00	3.46
sum		4834.45	44.80	19.46	25.34	527.59		5175.61	40.65	19.06	21.59	35.33
ratio		1	1	1	1	1		1.07	0.91	0.98	0.85	0.067

node containing the original Steiner node; And, buffer location is determined by an analytical formula so that node-to-node routing becomes very fast.

2. Results when More Choices for Expanded Node Set

The second experiment is intended to show that our algorithm is capable in handling the situation when a greater flexibility is needed for the Steiner node adjustment. As stated in Section C-3, the set of alternative Steiner nodes is defined by the expanded node set. In Table IV, we have a 5×5 tiles adjustment flexibility while keeping the size of expanded node set to be 3, which is the same as the first experiment. Similarly, POROSITY is implemented with each plate consisting of 5×5 tiles.

Table IV consolidates our claims in the last experiment in the way that:

- PRAB generates better timing and congestion cost results than POROSITY.
- PRAB runs 16 times faster.

Table IV. Comparison between POROSITY and PRAB under 5×5 tiles adjustment flexibility.

net	POROSITY[6]						Our algorithm PRAB					
	req/ps	imp/ps	cost	w.cst	b.cst	CPU/s	req/ps	imp/ps	cost	w.cst	b.cst	CPU/s
mcu0s5	5913.20	-36.11	4.55	1.68	2.87	14.77	5918.15	-31.16	4.63	1.95	2.68	2.07
mcu1s9	6047.59	111.13	4.19	1.70	2.50	19.48	6040.83	104.37	2.02	1.10	0.92	0.67
n1071	1783.42	60.38	2.88	1.07	1.81	16.91	1791.09	68.05	4.07	1.57	2.50	1.41
n18905	-973.32	89.34	5.39	2.63	2.76	48.17	-1026.91	35.75	3.95	2.80	1.15	10.25
n313	829.23	182.80	4.11	1.86	2.25	13.46	861.31	214.88	2.63	1.74	0.90	0.81
n7866	-18.14	617.13	3.19	2.24	0.95	163.36	18.42	653.69	2.56	2.06	0.50	8.21
n8692	625.81	340.94	5.22	1.59	3.63	25.52	651.07	366.20	4.11	1.74	2.37	1.04
n8702	338.17	2369.26	6.90	2.26	4.65	295.52	386.53	2417.62	5.55	2.54	3.01	11.77
n8730	101.29	640.06	5.81	1.57	4.24	23.25	163.90	702.66	4.70	1.51	3.19	1.36
pointer3	-336.62	277.12	2.62	2.12	0.50	56.93	-301.86	311.89	2.64	1.78	0.86	3.77
sum		4652.05	44.87	18.73	26.14	677.37		4843.95	36.88	18.80	18.08	41.36
ratio		1	1	1	1	1		1.04	0.82	1.00	0.69	0.061

Table IV demonstrates that our algorithm achieves a much faster Steiner tree adjustment with better solution quality and it also reveals the fact that for exploring a larger flexibility in Steiner node adjustment, our scheme of picking 3 tiles (irregularity) to be the expanded node set not only makes our algorithm very efficient but also accomplishes promising Steiner tree adjustment with congestion awareness.

Table V. Summary of total timing improvement and congestion cost for all 10 nets under different adjustment flexibilities.

adjustment flexibility	POROSITY[6]			Our algorithm PRAB					
	imp/ps	cost	CPU/s	imp/ps	ratio	cost	ratio	CPU/s	ratio
3×3 tiles	4834.45	44.80	527.59	5175.61	1.07	40.65	0.91	35.33	0.067
5×5 tiles	4652.05	44.87	677.37	4843.95	1.04	36.88	0.82	41.36	0.061
7×7 tiles	4275.41	37.76	859.89	4749.34	1.11	37.15	0.98	42.45	0.049

3. Results for Different Adjustment Flexibilities

We summarize Table III and IV in Table V. It also includes the results for the 7×7 tiles adjustment flexibility setting with the size of expanded node set to be 3. The columns in Table V shows the corresponding total value of all 10 nets in previous experiments and the ratio of our PRAB algorithm comparing to POROSITY. Table V shows that for larger adjustment flexibility, our PRAB algorithm together with the expanded node set selection scheme provides a larger speed-up in run-time while its solution quality is still better than [6] and the results agree with our claim that selecting a small number of tiles for the expanded node set according to congestion helps improving efficiency but it does not hurt the solution quality.

4. Summary

Experimental results confirm the effectiveness of our algorithm in producing solutions with better timing and less placement and routing congestion cost when comparing to the POROSITY algorithm. In our experiment, we use the Steiner tree construction algorithm proposed in [15] in both two algorithms. However, any Steiner tree (e.g., a congestion-aware Steiner tree) can be taken as an input to our plate-based tree adjustment algorithm, and the similar solution quality of our tree adjustment technique can be expected.

E. Conclusion

In this paper, we proposed a new tree adjustment technique for congestion driven buffered Steiner tree construction. For a Steiner tree, a tree adjustment is performed such that the tree is perturbed to less congested region with very limited impact on its timing performance. In order to anticipate the request from buffer insertion, we applied analytical form buffering solution ahead of low congestion route search. To our knowledge, this is the first work which simultaneously considers the objectives of timing, placement and routing congestion for the buffered Steiner tree problem. Compared with previous work, this method is faster, more practical, and generates better solution quality.

CHAPTER IV

ACCURATE ESTIMATION OF GLOBAL BUFFER DELAY WITHIN A FLOORPLAN

A. Introduction

Buffer insertion is becoming an ever critical component of physical synthesis for timing closure and design planning (see Cong et al. [26] for a survey). Saxena et al. [8] estimate that the distance between buffers continues to shrink rapidly. One must be able to efficiently and accurately assess the impact of buffer insertion on a design, whether in terms of floorplanning, resource allocation, timing estimation or within an actual buffer insertion heuristic.

To this end, several works (e.g., [24, 25, 27, 28, 7]) have explored closed form expressions for buffer insertion on a line. None of these works model blockages in the layout. Given the advent of SoC chip design and the trends toward large memory arrays, IP cores, and hierarchical design, an ever increasing percentage of the layout is covered by blocks in which buffers cannot be inserted (though routes may cross over). A large blockage can cripple a route's ability to meet timing since delay is quadratic in length when no buffers are inserted, but linear in length for optimal buffer insertion. Blockages are now a first order delay effect and must be taken into account for any buffer estimation technique to be sufficiently accurate. Several works, e.g., [19, 14, 22, 20], have explored the problem of buffer insertion when there are constraints on the buffer positions, but none of them address the problem of delay estimation.

Our work begins from Otten's theoretical result [7] that in an optimal buffering, delay is linear in terms of length. With large blockages and multi-fanout nets, a linear delay solution is not necessarily realizable. It is not at all immediately obvious how to overcome these limitations, and whether such an extension (even if it were possible) would even be valid in a real design methodology. The primary contribution of this work is to extend Otten's theory to predict interconnect delays for multi-fanout nets in the presence of blockages, and to validate it on real industrial test-cases. The end result of this work is a fast and simple formula that is proved on real design scenarios, and can be of practical use in early design planning.

The following key assumptions actually impose little error when compared to an actual buffer insertion solution:

- Smaller blocks ignored: Let L_{opt} be the spacing between consecutive buffers that obtains optimal signal propagation speed. Blocks with width less than can safely be ignored. While this may cause buffers to be inserted at distance less than the optimal spacing, having multiple buffers in the library allows the optimal linear delay to still be achievable. An example is shown in Figure 11. As long as the blocks are smaller than L_{opt}, the optimal realizable buffering in (b) will have delay very close to the ideal buffering of (a). Hence we can assume the delay model of (a) to approximate (b).

- Single buffer type: A single buffer type that yields the fastest point to point delay is sufficient for modeling. It turns out that the more buffer types that are actually in the library, the better the single buffer type approximation. As shown in Figure 11(b) different size buffers may be required to buffer distances that are less than L_{opt}. Having additional buffers in the library allows the situation in (b) better recover from its perturbation from the ideal buffering in (a).

- Block locations ignored: whether blocks are closer to the source or sink has little effect on the actual buffered delay, so we ignore this effect.

- Infinitesimal decoupling buffers: Buffers for decoupling capacitance off the critical path can be modeled to have zero input capacitance. This may lead to slight underestimation of delay, but the effect is almost negligible.

- Larger block front-to-back buffering: A block with width larger than L_{opt} will cause a linear delay model to break down. To optimize the delay across such a block, and optimal buffering will almost assuredly place a buffer right before and right after the block. Hence the delay across the large block can be modeled separately from the rest of the Steiner route.

Despite the inaccuracy it would seem these assumptions impose, our simple linear time estimation technique shows high accuracy when compared to realizing a buffer insertion solution with van Ginneken's algorithm [9].

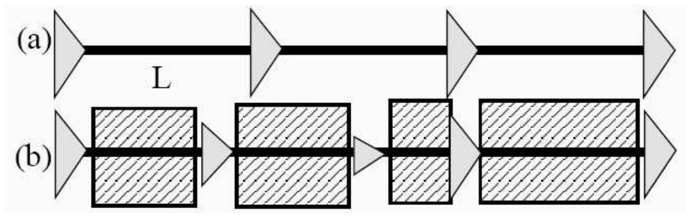


Fig. 11. Example of (a) an optimally buffered line with equal spaced buffers and (b) an optimal realizable buffering when blockages are present. Note that unequal buffer sizes may be used here.

This approach has several potential applications. For example, it can assess the timing cost of different block configurations during floorplanning or assess different Steiner routes during wire planning. A global router can use this to decide which of several possible Steiner tree constructions is likely to yield the best timing result. One may want to do timing analysis of a floorplan and/or placement without having to actually perform the buffer insertion for a net. One could embed the formula into a placement algorithm. Finally, the recent work of applies this approach in a Steiner tree construction that navigates the environment as a precursor to buffer insertion.

B. Closed Form Formula for Two-Pin Nets

Consider buffering a line of length L. Assume a given "ideal" buffer [7] (or inverter) b that is optimal for signal propagation speed on a wire. Let the intrinsic resistance and input capacitance of b be given by R_b and C_b, respectively. Assume the intrinsic buffer delay is zero since this is a first order approximation. One could also add an intrinsic resistance term and derive alternative formulas.

The unit wire resistance and capacitance are given by R and C, respectively. We make the following assumptions:

- The driver of the net has the same driver resistance as b. If this assumption is incorrect, that may indicate a design flaw. Too large a resistance means the driver should probably be powered up until the resistance is close to R_b. Even if the gate cannot be properly sized, it will likely need a few buffers (in which the last buffer is of size b) as close as possible to the driver to power up the signal in order to drive the line. Similarly, if the resistance is much lower than b, this indicates that the gate is likely overpowered and can be powered down.

- The sink of the net has input capacitance C_b. A different value should not significantly change the value, especially for a long line. Significantly different input capacitances also may potentially indicate an ill-sized sink. Overall, we find that sink capacitance gets overshadowed by wire capacitance on nets that require buffering.

- The intrinsic buffer delay is zero. This terms tends to be dominated by the R_bC_b term. However, an intrinsic delay term can easily be incorporated if desired.

1. Delay Formula with No Blockages

The following result was also derived by Otten [7]. We present it here for completeness.

Theorem 1: The delay $D(L)$ function of an optimally buffered line of length L with no blockages asymptotically approaches the linear function of the design and buffer parasitics given by:

$$D(L) \;=\; L(R_bC + RC_b + \sqrt{2R_bC_bRC}) \tag{4.1}$$

Proof: Let k be number of stages (so there are $k-1$ buffers) that results in the optimal delay along L, as in Figure 11(a). Several works have proved that the optimal buffer configuration spaces the buffers at equal distances. Since the source and sink have the same parasitics as b, all stages have the same delay. The length of wire between consecutive stages is $\frac{L}{k}$. The delay on the line is given by k times the sum of the buffer delay and the wire delay.

$$D(L) \;=\; k(R_b(\frac{CL}{k} + C_b) + \frac{RL}{k}(\frac{CL}{2k} + C_b)) \tag{4.2}$$

We wish to find the optimal number of buffers k. Taking the derivative with respect to k, setting the expression to zero, and solving for k yields the optimal number of buffers

$$k \;=\; L\sqrt{\frac{RC}{2R_bC_b}} \tag{4.3}$$

Obviously, if k is not an integer one may need to try rounding k up or down and see which yields the best delay. Substituting Equation (3) into Equation (2) yields the theorem.

Observe that $D(L)$ is a lower bound on the realizable delay. A nice property of Theorem 1 is that it is independent of the number of buffers. Of course, this will introduce some

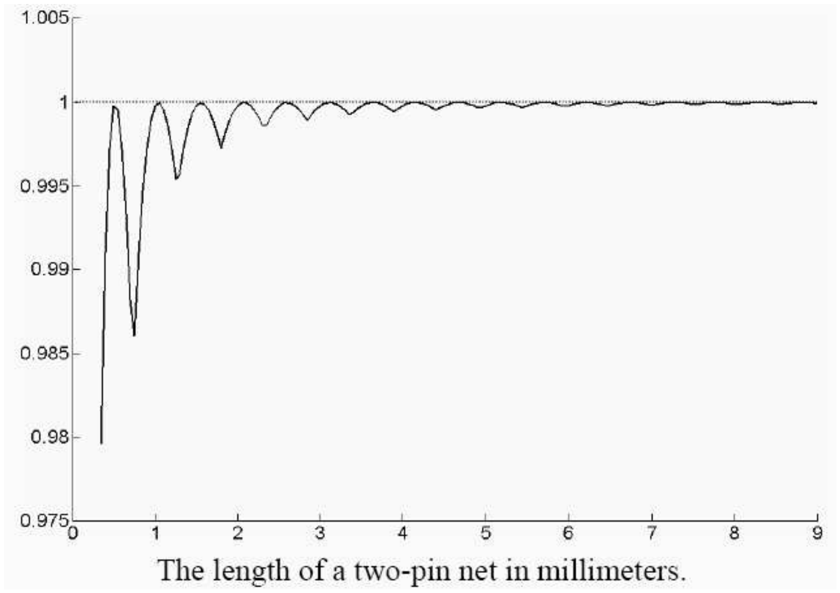


Fig. 12. Ratio of Equation (1) to (2) as a function of wirelength.

error because the delay in (1) is not realizable when the optimal number of stages is not an integer. However, the error is actually quite small, as illustrated by Figure 12.

The figure presents the ratio of the delay according to the estimation formula in Equation (1) to the delay of Equation (2). As L goes to infinity, the ratio goes to one, meaning the error goes to zero. Note that the maximum error occurs when the optimum number of buffers k is not an integer. However, the realizable delay will actually be even less than in Equation (2) if one permits additional buffer sizing. For example, if the number of buffers $k = 3.5$, then one may size up the buffer type until the ideal number is three or size down until it is four, thereby achieving a slightly better delay. Even without sizing though, Equation (1) is within 0.5% of Equation (2) when more than one buffer is required.

This result is perhaps not surprising, given the observations of Cong et al. [29]. They show that a fairly large “feasible region” exists for each buffer to be manipulated without suffering significant degradation in timing. Our example bears this out as buffers are shifted slightly when k is not an integer in order to get equal spacing between buffers. This shifting results in close to the ideal delay of Equation (1).

Corollary 1: The optimum spacing L_{opt} between buffers is

$$L_{opt} = L\sqrt{\frac{2R_bC_b}{RC}} \tag{4.4}$$

2. Delay Formula with Blockages

Next we consider inserting a block of width w somewhere on the line L. This notion can be generalized to include jogs and bends as in Figure 13. Let $l(u,v)$ be the length on the route from u to v. In the figure, we consider L to be $l(s_0,s_1)$ and w to be $l(x_0,x_1)$. We wish to derive a delay formula that is a function solely of L and w. Our strategy is as follows:

- For $w < L_{opt}$, we assume that buffers can be placed (and potentially sized) in such a way as to avoid the blockage while only suffering a nominal delay penalty. Hence, we ignore w and just use Theorem 1.

- For $w \geq L_{opt}$, we try placing a buffer immediately before and after a blockage. This minimizes the quadratic effect on delay that the width of the blockage has. For the rest of the line, we simply invoke Theorem 1, again accepting the potential error from the inability to have the optimal number of buffers be a non-integer.

For the second scenario ($w \geq L_{opt}$), the buffered delay is given by the buffered delays of the unblocked wires plus the delay needed to cross the blockage. The latter term is given by the Elmore delay:

$$ED(w) = R_b(Cw + C_b) + Rw(\frac{Cw}{2} + C_b) \tag{4.5}$$

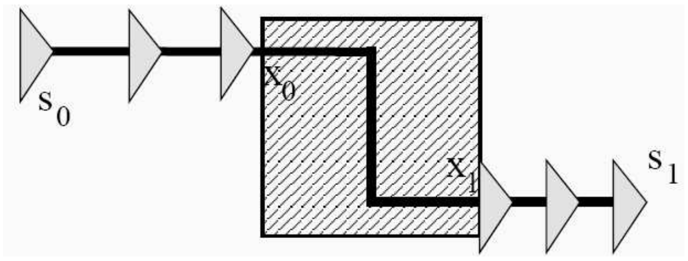


Fig. 13. Buffering scheme on a route of length $L = l(s_0, s_1)$ with a single blockage spanning length $w = l(x_0, x_1)$.

Given a set of blockages W with crossing width $w \geq L_{opt}$, then one can assume the existence of a single buffer before and after each blockage and summing the pieces together. We overload the D function so that $D(L)$ is the formula in Equation (1) for no blockages and $D(L, W)$ is the following delay for a set of blockages W.

Blockage Buffered Delay Formula:

$$D(L, W) \quad = \quad D(L - L_w) + \sum_{w \in W} ED(w) \text{ ,where } L_w = \sum_{w \in W} w \qquad (4.6)$$

Unlike Theorem 1, this formula is not a lower bound on the actual achievable delay. It could conceivably over-estimate delay since inserting buffers right after one blockage and right before another may result in overly tight buffer spacing. However, the delay from optimal buffered solution is rarely smaller than that from Equation (6) when $w \geq L_{opt}$, as we demonstrate in the next section.

C. Two-Pin Experiments

We use the following parameters from 100 nanometer technology [27]: $R = 0.184\Omega/\mu m$, $C = 0.0715fF/\mu m$, $R_b = 246.3\Omega$, and $C_b = 72.fF$. For these values, Equation (4) yields $L_opt = 519\mu m$. Assume that the sink and source are both buffers b. We first illustrate the accuracy of Equation (6) is accurate, even though it is independent of the blockage location. For each possible blockage location, we compute $D(L, W)$ and the optimal delay according to van Ginneken's algorithm (using only buffer type b).

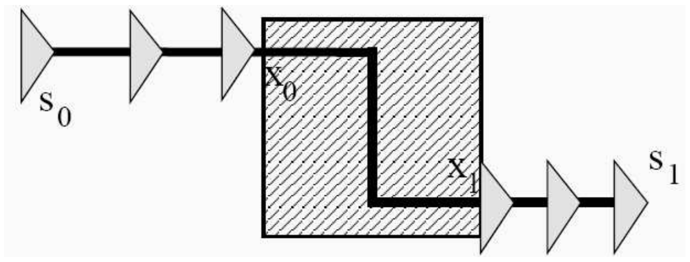


Fig. 14. Comparison of the blockage buffered delay formula with van Ginneken's algorithm for the case of a single blockage on a 2-pin net with length $10mm$.

Figure 14 shows this for a ten mm wire for blockages with widths 0.5, 2.0, 3.0, and $4.0mm$. The horizontal axis give the location of the blockage in terms of the distance from the source. We observe the following. First, for all examples, the closed form of Equation (6) is a tight lower bound. Next, the error is less than 1% for all blockage widths and blockage placement. Finally, the maximum error occurs at the tail ends because this is where one may have to either insert two buffers that are too close together or drive a distance that is actually longer than the blockage. With a library of multiple buffer types for van Ginneken's algorithm, this small error is reduced even further. Thus, for a single blockage the error is insignificant.

Next, consider the scenario when multiple blockages cover a significant part of the wire. We generated ten different instances with either 3 or 4 blockages, covering a large percentage of a $12mm$ wire. The ten cases are shown in Table VI, where the second column gives the blockage widths and the third column gives the corresponding distances from the source. For each case, we report the Blockage Buffered Delay formula and the optimal

Table VI. Comparison of the blockage buffered delay formula with van Ginneken's algorithm for multiple blockages on a 2-pin net with length $12mm$.

test case	Block widths (mm)	Positions (mm)	Eq.(6) (ps)	van Gin delay (ps)	Error (%)
1	1.8/4.0/2.9	0.1/2.2/6.7	437.0	438.5	0.35
2	2.5/4.0/2.9	0.3/3.2/8.7	451.9	452.5	0.11
3	0.5/4.7/2.1	1.3/2.2/9.7	440.6	441.5	0.21
4	3.5/4.7/2.0	0.0/4.2/9.7	497.0	497.8	0.14
5	4.5/0.7/3.0	0.5/6.2/8.7	454.1	454.7	0.12
6	2.5/2.1/2.9/1.1	0.3/3.2/6.7/10.0	390.9	391.6	0.16
7	2.5/1.1/5.9/0.5	0.0/3.2/4.7/11.0	527.7	528.1	0.08
8	2.6/4.4/0.9/1.8	0.3/3.2/8.7/10.2	448.5	449.2	0.15
9	1.5/3.3/0.9/4.2	0.3/2.2/5.7/7.3	456.5	457.8	0.28
10	1.5/3.3/3.9/2.2	0.0/2.2/5.7/9.8	456.5	461.7	0.11

delay according to van Ginneken's algorithm in columns four and five. From the last error column we see that our formula is well within one percent of optimal for all ten cases.

We have effectively shown that for single and multiple blockages, the error from our formula is insignificant.

D. Linear Time Estimation for Trees

We now show how to extend the two-pin formulae to trees in the presence of blockages. Two convenient properties of the following estimation technique are that:

- It can be decomposed into a summation of piecewise components, just like the Elmore delay, thereby enabling efficient optimization algorithms.
- The delay can be broken into the sum of the delays on a given path, allowing one to compute the worst slack of the tree in a single bottom-up traversal.

Let $T(V, E)$ be a Steiner tree with n nodes and source node s_0 and sinks $s_1, ..., s_k$. Let $RAT(s_i)$ be the required arrival time for sink s_i and let $p(v)$ be the parent of node

$v \in V - \{s_0\}$. The quality of a given buffer solution is typically measured by the slack at the source node, which is given by

$$q(s_0) \quad = \quad \min_{1 \le i \le k} \{RAT(s_i) - Delay(s_0, s_i)\} \tag{4.7}$$

where $Delay(s_0, s_i)$ is the buffered delay from the s_0 to s_i.

The key idea is to assume that if a path from s_0 to s_i is the most critical, that all subtrees off the critical path will be decoupled. To achieve the lowest delay to the critical sink, the decoupling buffer should have the minimum possible input capacitance; we assume input capacitance zero. We show in Section 5, that this is a second order effects, as compared to the first order blockage effect.

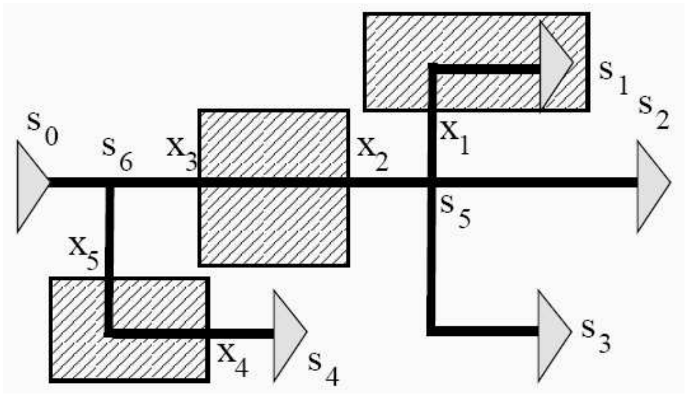


Fig. 15. Multi-sink tree with only unblocked Steiner points.

1. Case 1: Unblocked Steiner Points

Consider the case where all Steiner points are unblocked, as in the four-sink example of Figure 15. Here, all decoupling of branches can be accomplished by placing the buffer right near the Steiner point. Hence, the delay to a given sink can be broken piecewise into the sum of its sub-paths. For example, the delays in Figure 15 are given by:

$$\begin{aligned} Delay(s_0, s_1) &= D(l(s_0, s_1), \{l(x_3, x_2), l(x_1, s_1)\}) \\ Delay(s_0, s_2) &= D(l(s_0, s_2), \{l(x_3, x_2)\}) \\ Delay(s_0, s_3) &= D(l(s_0, s_3), \{l(x_3, x_2)\}) \\ Delay(s_0, s_4) &= D(l(s_0, s_4), \{l(x_5, x_4)\}) \end{aligned} \tag{4.8}$$

where $D(L, W)$ is given by Equation (4.6).

2. Case 2: Blocked Steiner Points

Now consider when Steiner points may lie inside blockages, as in Figure 16. In this case, decoupling may only occur outside of the blockage after incurring potentially significant wirelength. This is modeled by keeping track of the off-path capacitance and multiplying it by the upstream resistance inside the blockage. Also, we need to add the delay from the extra capacitance loading on the (imaginary) driving buffer. Define the function $OD(l_r, l_c)$ to be the delay from the off-path capacitance inside a blockage as:

$$OD(l_r, l_c) = (Rl_r + R_b)(Cl_c) \tag{4.9}$$

For example, some of the delays in Figure 16 are given by:

$$\begin{aligned}
Delay(s_0, s_1) &= D(l(s_0, s_1), \{l(x_6, x_5), l(x_1, s_1)\}) + OD(l(x_6, s_5), l(s_5, x_2) + l(s_5, x_3)) \\
&\quad + OD(l(x_6, s_6), l(s_6, x_4)) \\
Delay(s_0, s_4) &= D(l(s_0, s_4), \{l(x_6, x_4)\}) \\
&\quad + OD(l(x_6, s_6), l(s_6, x_5) + l(s_5, x_2) + l(s_5, x_3)) \qquad (4.10)
\end{aligned}$$

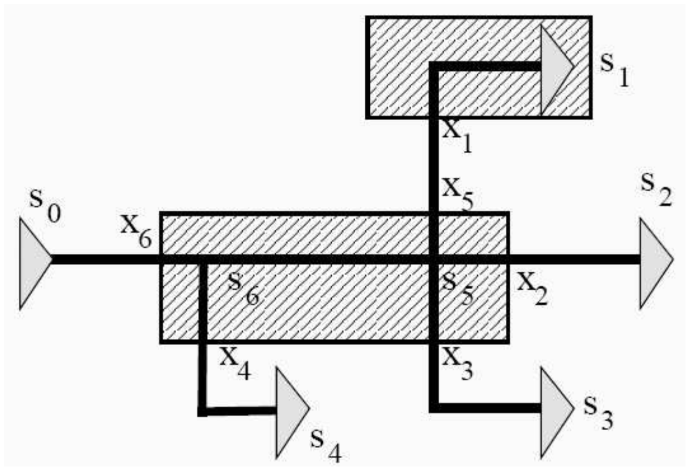


Fig. 16. Multi-sink tree example with blocked Steiner points.

3. Linear Time Estimation Algorithm

The examples of Figure 15 and Figure 16 show that the estimation can be expressed as a formula to find the delay to any sink. It is not as clear how to compute the slack at the source without having to compute the delay to each individual sink. We now present a linear time algorithm to compute the slack at the source in a single bottom-up tree traversal. The key component is to recognize that at any given Steiner point, the most critical downstream path can be determined because any upstream delay will be the same for all sinks downstream from the Steiner point.

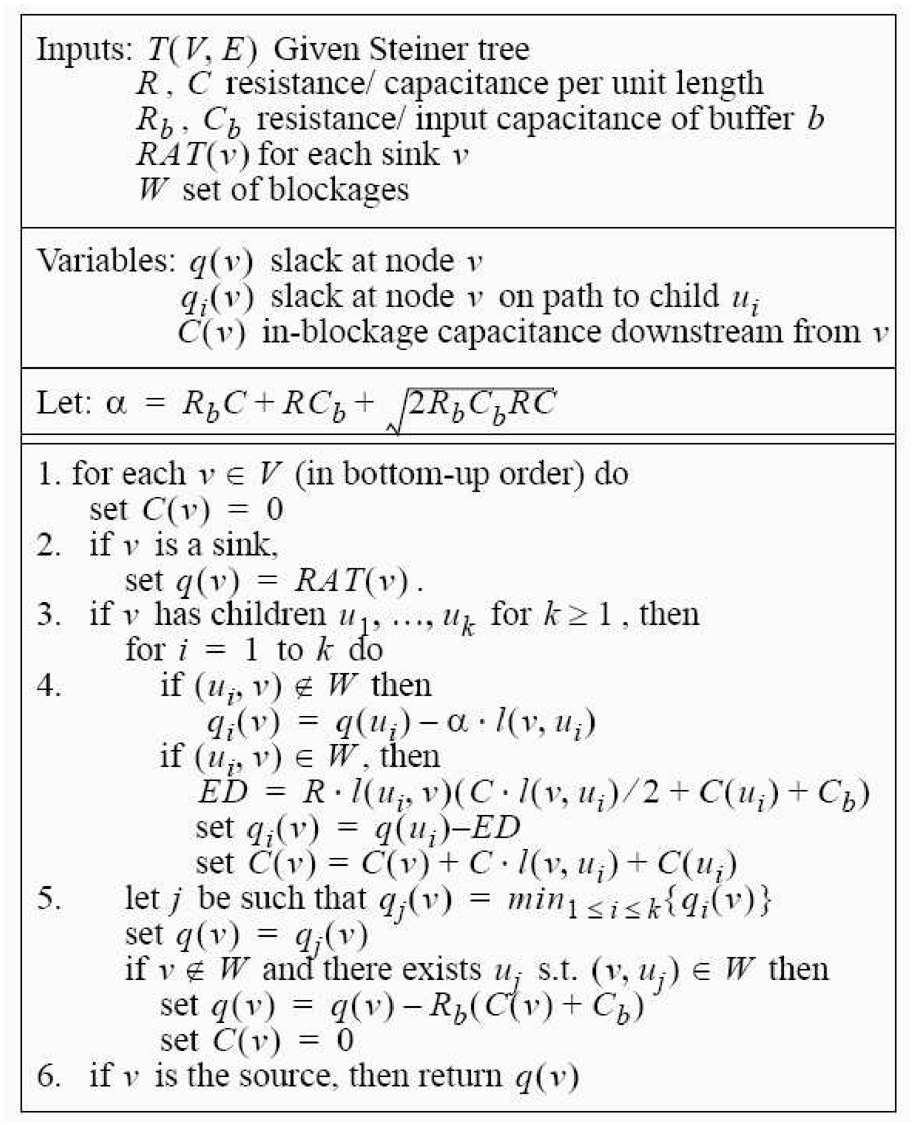
For each node v, let $q(v)$ denote the slack at node v, and let $C(v)$ denote the sub-tree capacitance downstream from v that is in the same blockage as v. In Figure 16,

$$\begin{aligned} C(s_5) &= C(l(s_5,x_3)+l(s_5,x_5)+l(s_5,x_2)) \text{ and} \\ C(s_6) &= C(s_5)+C(l(s_6,x_4)+l(s_6,s_5)) \end{aligned} \tag{4.11}$$

Note that the input buffer capacitance C_b is not stored in $C(v)$, but is only invoked when making a delay calculation. We only want to consider this additional capacitance on the critical path, but not for the whole sub-tree, since the non-critical paths can be decoupled with much smaller buffers.

We assume the edges in the tree are segmented such that whenever a blockage in intersects a tree edge, the edge is broken into two edges incident to an intermediate boundary node (as the x_i's are in Figure 15 and Figure 16). So each edge (u, v) lies either completely inside or outside a blockage in W. Boundary nodes lie outside W. A node lies inside W only if it is completely inside a block in W, e.g., in Figure 16 only s_5 and s_6 lie inside W.

The algorithm is shown in Figure 17. Instead of using the formulae from Section 2, the delay is computed piecewise since this affords the simplest direct implementation. Step 1 visits each node v in a bottom-up tree traversal, initializing the downstream capacitance $C(v)$ to zero. Step 2 handles the case where v is a sink, initializing the slack to the required arrival time. Step 3 handles multiple children and iterates through the children $u_1, ..., u_k$ of v. Step 4 updates upstream information when going from node u_i to v via the intermediate variable $q_i(v)$. If the edge (u_i, v) is not in a blockage, the downstream in-blockage length is zero and the slack is updated by the linear delay from v to u_i. The slack is updated to include the Elmore delay from v to the critical node downstream from u_i that is just



```
Inputs: $T(V, E)$ Given Steiner tree
        $R$, $C$ resistance/ capacitance per unit length
        $R_b$, $C_b$ resistance/ input capacitance of buffer $b$
        $RAT(v)$ for each sink $v$
        $W$ set of blockages

Variables: $q(v)$ slack at node $v$
           $q_i(v)$ slack at node $v$ on path to child $u_i$
           $C(v)$ in-blockage capacitance downstream from $v$

Let: $\alpha = R_bC + RC_b + \sqrt{2R_bC_bRC}$

1. for each $v \in V$ (in bottom-up order) do
     set $C(v) = 0$
2.   if $v$ is a sink,
       set $q(v) = RAT(v)$.
3.   if $v$ has children $u_1, \ldots, u_k$ for $k \geq 1$, then
       for $i = 1$ to $k$ do
4.       if $(u_i, v) \notin W$ then
           $q_i(v) = q(u_i) - \alpha \cdot l(v, u_i)$
         if $(u_i, v) \in W$, then
           $ED = R \cdot l(u_i, v)(C \cdot l(v, u_i)/2 + C(u_i) + C_b)$
           set $q_i(v) = q(u_i) - ED$
           set $C(v) = C(v) + C \cdot l(v, u_i) + C(u_i)$
5.     let $j$ be such that $q_j(v) = min_{1 \leq i \leq k}\{q_i(v)\}$
       set $q(v) = q_j(v)$
       if $v \notin W$ and there exists $u_j$ s.t. $(v, u_j) \in W$ then
         set $q(v) = q(v) - R_b(C(v) + C_b)$
         set $C(v) = 0$
6.   if $v$ is the source, then return $q(v)$
```

Fig. 17. Linear time estimation algorithm for trees.

outside the blockage containing v. Finally, for edges (u_i, v) that lie within blockages, all downstream capacitance is summed in $C(v)$.

Step 5 then identifies the child u_j of v that is the ancestor of the most critical sink, and the slack at v is then set. Finally, if v is not in a blockage, but the edge (u_j, v) is, then one must incorporate into the slack the additional delay required for a buffer just outside the blockage to drive. Since v is not outside the blockage, its downstream capacitance is then set to zero. Finally, Step 6 returns the slack at the source. The time complexity is linear in the number of nodes.

E. Experiments for Multi-Sink Nets

We call our estimation algorithm in Figure 17 BELT for Blockage Estimation in Linear Time. We consider three other buffered slack calculations.

- One can compute the estimation formula while ignoring blockages. This in effect reduces to the estimations of [29, 7], whereby one just looks at the length of each path and performs an optimal buffering as if it were a 2-pin net. Since this is essentially the BELT estimation without the blockages, we call this formula ELT.

- As in Section 3, we run van Ginneken's algorithm using the single buffer type b, and call this VG1 since there is one buffer type.

- In practice, we have the ability to run actual buffer insertion with additional buffers types. We generated three additional smaller buffers (since b is already a larger buffer) to use with van Ginneken framework. We call this algorithm VG4.

All codes were written in C++, and compiled using g++ version 2.95 on a Sun Ultra-4 running SunOS 5.7.

For the following experiments, the required arrival times were chosen to be the same for each sink since this actually increases the likelihood of error in the delay estimation formula. If one sink is substantially more critical, then this sink will have all off-path branches decoupled, making it an easier problem. Consequently, instead of reporting slack, we report the maximum path delay (which also makes interpreting results more intuitive).

1. Results on Random Nets

Our first experiment examines randomly generated nets. First we created a simple artificial floorplan. The plan has 16 high level square blocks, each five millimeters on a side. The blocks are arranged in a regular pattern on a square layout that is 21 millimeters on a side. Thus, there is sufficient space in the alleys (wide) between blocks to allow buffer insertion. This type of layout loosely corresponds to the kind of behavior one might expect from a large chunky hierarchical design.

Next, we generated nine nets each of size three through ten pins. We ran the four different algorithms, ELT, BELT, VG1 and VG4, on each net and summarize the results in Table VII. In each case we set the driver strength and the sink size to be equal to buffer *b*. The Steiner topology was generated using the C-Tree algorithm which ignores blockages [15]. Delay calculations are for 0.10 micron technology [27].

The table presents a single row summarizing the average of nine different nets each having the specified number of sinks. For each net, we report the average wirelength and percent of the net that was blocked. For the four algorithms we present the average maximum delay for the net. The solutions of VG4 are also evaluated by SPICE model and the results are shown in the rightmost column. The SPICE results are based on 50% Vdd signal delay. Column 5 gives the ratio of ELT to BELT delay as a percentage. Note that by definition ELT will always be less than BELT. The percentage ratio of BELT to VG4 delay is listed in column 7 for each case. It is well known that the Elmore delay is an upper

Table VII. Experiments on randomly generated nets. Each row represents the average of nine different nets.

net sinks	WL (μm)	% Blk	ELT	%ELT/ BELT	BELT	%BELT/ VG4	BELT ln2	%BELT_ln2/ VG4_SPICE	VG1	VG4	VG4 SPICE
3	29104	88.5	552.8	45.0	1225.9	99.4	849.7	93.7	1235.0	1232.8	906.9
4	41983	93.3	617.0	43.5	1416.6	99.6	981.9	92.6	1428.7	1422.1	1060.6
5	39904	90.2	512.8	42.1	1216.6	99.2	843.3	93.2	1230.2	1225.9	904.5
6	46559	90.2	569.8	43.9	1295.6	99.4	898.1	93.0	1308.0	1303.7	965.5
7	50373	88.9	548.6	42.2	1299.3	99.2	900.6	93.5	1314.2	1309.4	963.0
8	59190	91.1	663.0	43.0	1541.7	98.9	1068.6	95.7	1558.0	1551.2	1117.0
9	54659	90.5	539.0	39.8	1353.0	99.4	937.8	94.0	1375.3	1368.2	997.7
10	65350	94.0	595.1	41.7	1426.4	98.9	988.7	95.1	1449.1	1441.8	1040.2

bound of real delay and people often multiply $ln2$ with the Elmore delay to reflect 50% Vdd signal delay. When comparing the timing performance of two Steiner trees for a same net, the scaling of $ln2$ does not affect the conclusion of the comparison. In order to have more fair comparison with the SPICE results, we report the ratio of the BELT results scaled by $ln2$ to the SPICE based VG4 results in column 9. We observe the following:

- By comparing the ratio of ELT to BELT in column 5, observe if one ignores blockage, the errors are typically off by over a factor of two. Of course the degree of the error will depend on the size of the blocks. Clearly, ignoring blocks causes gross underestimation of the achievable delay.
- Comparing BELT to VG1, we see that the delay estimation is quite accurate, and tends to underestimate the achievable delay by 1.1% on average.
- Comparing BELT to VG4, we see that the error is reduced even further, to 0.8% on average.
- When compared to SPICE based VG4, the error of BELT is always less than 8%.

Clearly, the accuracy of BELT is sufficient while the accuracy of an estimation technique that is not blockage aware begins to suffer fairly significant underestimation. This

Table VIII. Experiments for 13 nets from an industry design.

net name	WL (μm)	Sinks	% Blk	ELT	BELT	%BELT/ VG4	BELT ln2	%BELT_ln2/ VG4_SPICE	VG1	VG4	VG4 SPICE
mcu0	50540	18	89.9	380	822	95.1	569.9	88.6	872	864	643.4
mcu1	41780	19	96.2	492	1052	97.4	729.3	92.6	1084	1080	787.3
n107	14870	17	97.6	257	361	91.4	250.2	85.8	396	395	291.7
n189	64700	29	83.8	573	1486	96.9	1029.8	91.9	1556	1532	1120.9
n313	69430	19	96.6	587	1821	99.0	1262.3	95.6	1850	1840	1319.7
n786	53110	32	96.4	1126	3574	92.3	2477.2	85.1	3880	3873	2911.3
n869	42180	21	96.6	1042	2605	92.6	1805.3	85.1	2816	2813	2122.5
n870	45230	21	97.3	972	2326	93.1	1612.4	85.4	2498	2498	1887.3
n873	49290	43	78.0	527	1363	99.1	944.6	92.1	1381	1375	1026.1
poi3	63600	20	96.8	1256	3746	97.4	2596.6	89.6	3854	3847	2898.1
big1	195300	88	85.8	1143	5920	97.6	4103.6	99.8	6115	6063	4111.0
big2	122500	79	93.1	545	1577	96.0	1093.1	95.9	1657	1643	1139.9
big3	95320	63	94.1	403	1415	96.8	980.5	96.7	1478	1460	1014.0

effect becomes magnified when the blockage map has large blocks that may correspond to IP cores or memory.

2. Results on Large Real Nets

Our next experiments use the Steiner trees for a set of the industrial nets reported in [15] and [22]. We perform the same set of experiments as in Section 1 and report the results in Table VIII. This time the nets are listed on an individual basis.

We observe the following:

- On average, the ELT/BELT percentage is 36.2%, which means that blockage has on average about a 64% impact for these nets.
- For some cases, the impact of blockage is not that significant, e.g., for net n107 ELT is a reasonable estimate. For others, it is quite large, e.g., netbig1 has an ELT/BELT percentage of 19.3%. In this case, we see that ELT underestimates the (realizable) VG4 delay by 81%, while BELT underestimates the VG4 delay by 3%.
- On average the error of BELT compared to VG4 is 5.2%, while on average the VG4

delay is almost a factor of three higher than that predicted by ELT.

- Compared to SPICE based VG4, the error of BELT is 9% on average.

These experiments illustrates that our estimation technique is sufficiently accurate for design planning, while ignoring blockages is prohibitively costly.

Finally, note how efficient the estimation technique is. The total runtime in seconds for running the above 13 test cases was 0.24, 23.0 and 29.0 for BELT, VG1 and VG4, respectively. In other words, BELT is about 100 times faster than running an actual buffer insertion algorithm. For medium sized nets, such as n786 and n869, the runtime of VG4 plus SPICE simulation is over 10000 times slower than the runtime of BELT.

F. Conclusion

We presented closed form formulae for estimating the achievable buffered delay when buffering restrictions exist in the layout. We demonstrate that adding blockages to the layout can cause significant error in estimation techniques that ignore the blockage terrain. We also showed that our technique is a lower bound, has an error of less than one percent for two-pin nets, and has only a few percent error for multi-sink nets.

CHAPTER V

PATH BASED BUFFER INSERTION

A. Introduction

Buffer insertion is widely recognized as an essential technique for interconnect optimization [30] while interconnect is a fundamental limit [31] for VLSI technology progress. The importance of buffer insertion has resulted in numerous algorithmic and methodologic works. Perhaps the most influential work is the classic van Ginneken's algorithm [9]. Given a Steiner tree spanning a signal net and candidate buffer locations on the tree, van Ginneken's dynamic programming (VGDP) algorithm can find the maximum timing slack solution optimally in quadratic time. This algorithm is extended to handle buffer cost and buffer library in [11]. The noise avoidance issue is addressed in buffer insertion in [32]. Higher order delay models are adopted in buffer insertion in [33]. For 2-pin nets, quadratic programming based approach [34] and closed form buffering solutions are proposed in [28, 35]. Recently, an $O(n \log n)$ buffer insertion algorithm is developed [36].

Recently, an industry study [37] predicts that 35% of the cells on a chip will be buffers at $65nm$. The huge number of buffers may affect various aspects of circuit design and performance including timing [30], power dissipation [11], signal integrity [32], placement and routing congestion [37]. Therefore, buffer insertion needs to be conducted in a more elaborated manner to push the envelope of performance.

In fact, most of the previous works on buffer insertion are net based, i.e., buffer insertion is performed on one net after another individually. Even though the buffer insertion problem with net based formulations is relatively easy to solve, it may lead to sub-optimal path delay or unnecessary buffer usage due to the lack of global view. The weakness of net based buffer insertion can be illustrated through a very simple example in Figure 18. Consider two nets A and B along a critical path in the circuit. If we perform buffer insertion on net B first, 4 buffers are needed on B and then no buffer is needed on A for satisfying the critical path timing constraint. This is denoted by solution $S1$. However, the constraint can also be satisfied by putting 1 buffer on both A and B, denoted as $S2$. Optimizing the entire path may get a better solution and certainly can lead to a better deployment of buffering resources. Usually, net based dynamic programming algorithms such as [11] do not have a global view, nets which are processed first tend to over-consume buffer resources.

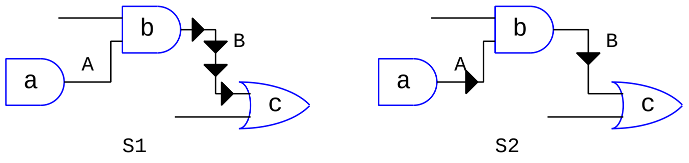


Fig. 18. Net based buffer insertion solutions depend on net ordering.

Despite their current popularity, the net based buffer insertion methods, without global view of the whole combinational circuit, will become inadequate for future technologies. In [38, 39], network based buffer insertion algorithms are proposed. In these approaches, buffer insertion is performed on all nets between PI/registers and registers/PO simultaneously through Lagrangian relaxation. However, both works include a restrictive assumption that buffers are inserted at every branch node to simplify the calculation of delay. In practice, whether or not buffers are necessary at certain branch node depends on timing constraints of related paths. Due to path re-convergence, it is very difficult to perform network based buffer insertion without the assumption. Although their works usually produces good results, they do not scale very well. Indeed, in [38], the CPU time consumption explodes for the larger testcases. When buffer insertion is considered beyond the limit of nets or gates, it is natural to consider gate sizing at the same time. In [40], a greedy heuristic on integrated gate sizing and buffer insertion is proposed. However, it neglects wire delays.

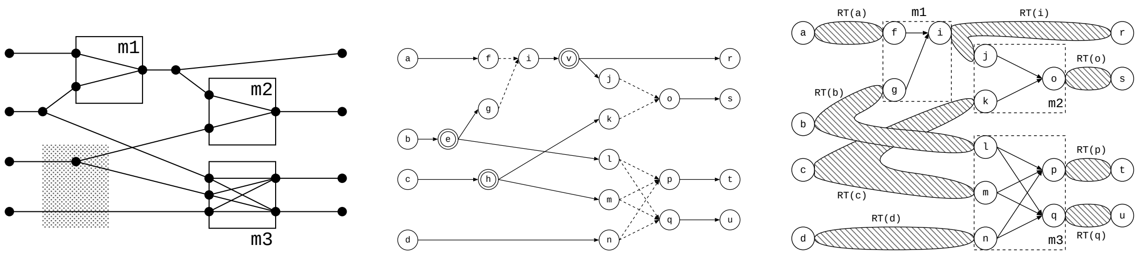
The network based methods make severe oversimplifications in order to achieve a solution, such as ignoring wire delays entirely. Our method takes advantage of some global optimization without sacrificing any of the modeling accuracy that the net based approaches provide.

In this paper, we propose a path based buffer insertion heuristic in order to minimize buffer/gate cost subject to path timing constraints. This approach is in the middle between net based and network based methods. However, it can achieve both better solution quality and faster computation runtime. Since our path based approach can easily handle false paths, the solution quality can be much better than network based methods. Besides, instead of relying solely on static timing analysis, a fast estimation on buffered delay is applied on the entire network so that a better global view is obtained. Our path based buffer insertion algorithm is based on the VGDP algorithm since it is robust and sophisticated enough to handle different instances. However, directly using VGDP may induce problems and we successfully solve those problems by a set of techniques such as off-path required arrival time estimation and gate sizing at sinks. Experimental results show that the usage in buffer/gate cost is reduced by 71% on average through our approach compared with traditional net based algorithms. The runtime is also reasonably fast.

The conference version of our paper can be found in [41]. However, this paper contains more details in algorithm description and new experimental results. Moreover, experimental results on ISCAS circuits are also included.

B. Problem Formulation

A placed and routed circuit can be formulated as a directed acyclic graph (DAG) $G = (V, E)$. An example is shown in Figure 19. A vertex $v \in V$ can be either (1) a primary input (PI) / primary output (PO) (e.g., nodes a and r in Figure 19(b)), or (2) a pin of a



(a) An example of a combinational circuit

(b) The corresponding DAG

(c) The circuit model with routing tree abstraction

Fig. 19. Example of combinational circuit models

module (e.g., f and i), or (3) a Steiner node on the route (e.g., e, shown as double circle in Figure 19(b)), or (4) a candidate buffer insertion location (not shown in Figure 19(b)). An edge is either an interconnect wire (solid line) or an input-to-output path (dotted line) within a module.

In this paper, interconnect wires are modeled as a distributed RC network and we adopt the Elmore delay model. Thus, an interconnect w is annotated by the corresponding resistance $R(w)$ and capacitance $C(w)$. A buffer b_i in the buffer library is defined by its load capacitance $C(b_i)$, intrinsic delay $T(b_i)$ and output resistance $R(b_i)$, while $W(b_i)$ represents the buffer cost which can be either buffer area or power.

A module is identified by the delay $T_{j,k}$ for each input-to-output path, the capacitance C_j of the input pin and the resistance R_k of the output pin. If x is the size of the module, $C_j = \hat{C}_j x + f_j$ and $R_k = \hat{R}_k / x$, where $\hat{C}_j$, $\hat{R}_k$, f_j are the unit size output resistance, unit size gate area capacitance and gate perimeter capacitance of the module. In this paper, the size of each gate is selected from the set $S = \{x_1, x_2, ..., x_n\}$. Each primary input is annotated with a user-defined arrival time while each primary output a required arrival time.

There may exist buffer blockages in the floorplan, for example, the shaded box in Figure 19(a). When a buffer insertion candidate location overlaps with the region of the buffer blockages, it is restricted such that no buffer can be placed. In other words, there exists no buffer candidate location within the buffer blockage region.

Signal slew rate is also considered in our path based buffer insertion algorithm. For a signal propagating along a wire, we employ a simple metric of $propagation\_slew = \ln 9 \cdot Elmore\_delay$ [30]. The slew rate at the receiving end of a wire depends on both the propagation slew rate s_p and the launching slew rate s_l at the driving end of the wire and is given in [42] as $\sqrt{s_p^2 + s_l^2}$. In our buffer insertion algorithm, any buffer solution with receiving slew rate greater than a certain threshold will be discarded.

The problem of **simultaneous gate sizing and buffer insertion** is defined as follows. Given a DAG which represents a placed and routed combinational circuit, a buffer library, a set of buffer candidate locations, a set of buffer blockage regions, find a buffering solution such that the overall cost of buffers and gate sizes is minimized. Buffering solution is in terms of the locations and types of buffers inserted, and sizes of the gates. At the same time, the solution is subject to the constraint of both the arrival time at each primary input and the required arrival time at each primary output as well as the slew rate requirement.

As mentioned in Section A, van Ginneken's dynamic programming (VGDP) approach is very flexible and efficient so that it can be easily applied to buffer blockage avoidance (by selectively setting candidate buffer locations) while considering buffer cost (i.e., area/power), buffer polarity and slew rate [43]. In order to utilize the flexibility and efficiency of VGDP, we intend to use net based buffer insertion algorithm as a building block for path based buffer insertion. In this way, we abstract the routing tree of the circuit and ignore all the details (i.e., Steiner node and interconnect tree structure, etc.) within the routing tree. An example of our circuit model is shown in Figure 19(c). In our model, we abstract all interconnect routing so that vertices only represent PI/PO of the circuit and input/output pins of modules (we use this definition for vertex hereafter in this paper) while edges only for input-to-output paths within a module. The routing tree is identified by its root vertex. For example, the routing tree $RT(c)$ is rooted at vertex c and with sinks k and m. In a combinational circuit, the root of a routing tree is either a PI vertex or an output pin of a module, while a sink is either a PO vertex or an input pin of a module.

C. Net Based Buffer Insertion

For a routing tree, if the required arrival time (RAT) at each sink vertex is given, VGDP algorithm traverses every candidate buffer location v_i of the tree in a bottom-up manner, propagating a set of solutions in the form of (c, q, w) which stands for downstream load capacitance, RAT, and total buffer cost respectively. Each solution reflects the intermediate results of a buffering solution on the subtree rooted at v_i. When the propagation reaches the driver (i.e., root vertex), a set of solution with different cost-RAT tradeoff is obtained. If the arrival time (AT) at the root vertex is given, we pick the solution with minimum buffer cost while timing requirement is satisfied.

Conventionally, net based buffer insertion for the whole circuit is accomplished by iteratively performing the following steps:

1. Static timing analysis (STA) and obtain the RAT and AT at every vertex
2. Perform buffer insertion for a routing tree with the AT and RAT obtained from STA (according to a specific net order. Discussion of net order is in Section F.)
3. Update STA results (of the fanin/fanout cone of the buffered routing tree) and perform buffer insertion for the next routing tree

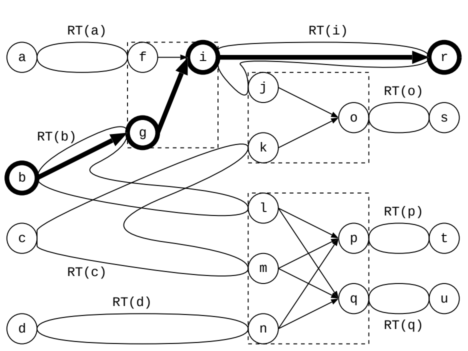
However, this is in fact a greedy algorithm and in our experiment, we found that the buffering solution of this approach is far from optimal even we let the iteration run unboundedly in order to refine the buffering solution for buffer cost reduction. Our observation to the problems of net based buffering include:

- STA is usually not buffer aware so the timing estimation at the first iterations are inaccurate and the circuit is very timing critical. Hence, the first processed nets tend to over-consume buffer resources (the case of net B in $S1$ of Figure 18).
- Since the algorithm is in nature greedy and a poor buffering decision to a routing tree $RT(a)$ due to incorrect timing estimation may lead to a poor buffering decision at other routing trees $RT(b)$ where b is a transitive fanin/fanout of a. Thus, earlier buffering decision may have degraded the quality of the whole buffering process, which cannot be improved in latter iterations. This is especially true for those nets along a critical path from PI to PO.
- Buffering solution of a routing tree is highly depending on the criticality of the sinks, which in turn depends on buffering of their fanout routing tree. Therefore, the criticality can be substantially different from the results of STA due to buffer blockages, which brings significant error in final buffering solution.

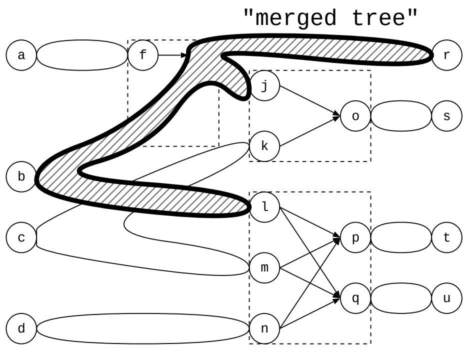
D. Path Based Buffer Insertion

In this section, we propose our path based buffer insertion algorithm (PBBI) as described in this section. The key elements of our buffer insertion algorithm are (i) buffer aware static timing analysis, (ii) path based VGDP buffer insertion, and (iii) off-path required arrival time estimation.

Different from net based buffering insertion schemes, our PBBI algorithm starts with buffer aware static timing analysis. The steps are as follows:



(a) A critical path on a combinational circuit



(b) The "merged tree"

Fig. 20. Example of algorithm overview

(1) The buffer aware static timing analysis takes care of buffer blockages and therefore the resultant AT and RAT is comparatively much more accurate than ordinary STA and it would not over-consume buffer resources even at the very beginning.

(2) A list of k most critical paths is obtained based on the buffer aware STA.

(3) Merge all routing trees of the vertices along the path into one big routing tree and then VGDP is applied to the "merged trees".

(4) repeat step (3) for every path found in step (2).

For example, in Figure 20(a), if the critical path is $\{b \rightsquigarrow g \rightarrow i \rightsquigarrow r\}$, the "merged tree" are formed from $RT(b)$ and $RT(i)$, shown in Figure 20(b). In our approach, the root (e.g., b) and the sink (e.g., r) along the path have a fixed AT and RAT respectively, which in turn produces a relatively good buffering solution. For all other sinks, namely off-path sinks (e.g., j, l), we propose an approach to adjust their RAT values to become more accurate and then the adjusted RAT are fed into the VGDP algorithm. In this section, we assume an efficient VGDP algorithm is given such that it considers buffer blockage, buffer polarity, buffer area/power and slew rate.

1. Buffer Aware Static Timing Analysis

Critical path method is widely used as a tool for static timing analysis (STA) [44]. It propagates the static delay information throughout the circuit. However, the delay along interconnect changes during the process buffer insertion which is a main source of error of net based buffer insertion algorithms, as mentioned in Section A. A work which predicts the post-buffering delay is in [45]. The work derives delay equations along a buffered wire segment considering buffer blockages and applies the equations for delay estimation upon multi-pin nets. Experimental results show that the delay estimation merely produces insignificant errors. Toward our problem, we verify in our experiments that integrating this buffer aware delay estimation with STA provides a good guide for buffer insertion using VGDP. With buffer aware STA, early buffering step will not over-consume buffer resource which trap the overall solution into a local optimum.

Buffer aware STA not only provides a good basis for VGDP algorithm on a path, but it is also a crucial element of the whole path based buffer insertion algorithm. For example, if we are performing buffer insertion on a critical path p_c from PI to PO, it propagates a set of solutions from the PO vertex with accurate RAT information (since the RAT at PO is fixed by user specification). During the propagation, there may exist some branch paths such that the RAT of those paths is needed to compute the solution sets. In such a way, if the RAT of branch paths is not accurate since STA is not aware of buffering along those branches, VGDP may think that the branches are more critical than p_c, which destroys the solution quality of the path based buffer insertion algorithm.

2. Path Based Buffer Insertion

In order to accomplish path based buffer insertion, a list of distinct paths must be first obtained. With the help of buffer aware STA, k most critical paths can be found using a polynomial-time algorithm in [46] (k can be changed during the progress of the algorithm). The parameter k is a tradeoff between quality and speed while k can be determined on the fly - stop finding the next critical path if the slack of the path is greater than a specific value. Note that the accuracy of our algorithm can also be improved when false paths are detected and only sensitizable critical path are selected. In the whole circuit, each routing tree have to be processed once. Therefore, if the list of paths are overlapping with each other, we delete the common vertices from the less-critical path and cut it into different distinct paths. For example, if the 3 most critical paths in Figure 19(c) is $\{b, g, i, r\}; \{a, f, i, j, o, s\}; \{c, k, o, s\}$, after removing common vertices, the list of distinct paths becomes $\{b, g, i, r\}; \{a, f\}; \{j, o, s\}; \{c, k\}$.

After getting a list of distinct paths, for each path, the algorithm treats all routing trees along the path as one big routing tree. The merging process is simple since the routing trees is cascaded together such that the sink (an input pin of a module) along the path merges into the root (an output pin of the same module) of the fanout routing tree. The merged vertex is treated as a candidate buffer location such that a special buffer must be inserted. The parameters of the special buffer corresponds to the capacitance/delay/resistance of the pin-to-pin path of the module. In Figure 19(c), the merged routing tree on the path $\{b, g, i, r\}$ consists of three sinks l, j, r rooted at b. g and i are merged into one buffer location with a special buffer b_s according to input-to-output path of $M1$ (The delay model of a buffer is similar to that of an input-to-output path of a module). After all, VGDP algorithm is applied to the merged routing tree. After all paths have been processed, there may exist some routing trees in which no buffer insertion has been performed. They are all comparatively non-critical and net based buffer insertion can be carried out for each of those nets. In summary, PBBI only processes each of merged trees (which are generated by distinct paths) once. In other words, our algorithm performs buffer insertion on each net only once in one single pass.

3. Off-Path Required Arrival Time Estimation

Since the PBBI algorithm performs buffer insertion in a path-by-path basis, the buffering solution may violate the timing constraints. An example is shown in Figure 21. The straight line represents input/output path within a module and dotted curly line stands for a path. Assume that a is a PI vertex and z is a PO vertex and the algorithm processes the path $p_1 = \{a \rightsquigarrow g \rightarrow h \rightsquigarrow p \rightarrow r \rightsquigarrow z\}$ prior to another path $p_2 = \{i \rightsquigarrow q\}$. We define the off-path sinks of a path p_1 as the sinks of the merged routing tree derived from p_1 such that the sinks are not along p_1. For example, for the path $p_1 = \{a \rightsquigarrow g \rightarrow h \rightsquigarrow p \rightarrow r \rightsquigarrow z\}$, i is an off-path sink of p_1. When applying VGDP on the path p_1, actual buffering solution may reduce the delay along $g \rightarrow h \rightsquigarrow p$ to a value which is less than our delay estimation using buffer aware STA. Since the delay between a and z is bounded above by the an user specified RAT and AT, the delay along the paths $a \rightsquigarrow g$ and $r \rightsquigarrow z$ could be larger than our estimation. In such case, it could happen that even with the minimum-delay buffering along the path $i \rightsquigarrow q$, the delay along $p_3 = \{a \rightsquigarrow g \rightarrow i \rightsquigarrow q \rightarrow r \rightsquigarrow z\}$ still violates the timing constraint. After each time we perform path based buffering insertion along a path, the AT and RAT of fanin and fanout cone for each vertex of the path are updated, so the only situation which causes the problem is that there exists re-convergence along the path we are performing buffer insertion. As in the previous example, we are inserting buffer along the path p_1, assuming that there exists a path $i \rightsquigarrow q$ in the list of distinct paths. The required arrival time of the off-path sink i (denoted as RAT_i) does not reflect the buffering solution along $r \rightsquigarrow z$ since the final buffering solution is unknown until the whole path p_1

is done. In this consideration, an equation for spreading out the slack of a path to all the routing tree is needed, which is presented in Theorem 1. In the following, we first derive the equation and revisit the problem in the example at the end of this section.

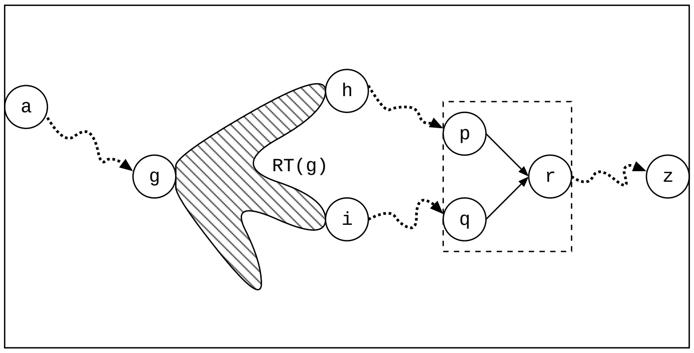


Fig. 21. A problem of path based buffer insertion algorithm

Considering a circuit which only contains a cascade of two-pin routing trees $\{RT(s_1), RT(s_2), ..., RT(s_k)\}$ as shown in Figure 22. For each routing tree $\{RT(s_i)\}$ with root s_i, the only sink is t_i. With optimal buffering, we can find the minimum delay $\underline{d_i}$ for each $RT(s_i)$. And based on the minimum delay, for each s_i, AT_{s_i} and RAT_{s_i} is propagated from PI and PO respectively according to the user specified timing constraint. Note that since the circuit is a sequence of 2-pin net, the slack $RAT_{s_i} - AT_{s_i}$ must be the same for all s_i. If $AT_{s_i} = RAT_{s_i}$, the circuit has zero slack and only the minimum-delay buffering solution can fulfill the timing constraint. However, if slack> 0, different buffering with smaller buffer cost is possible. We can denote $RAT_{s_i} - AT_{s_i}$ as “useful slack” resource such that buffering algorithm uses it to reduce the total buffer cost. Intuitively, since the ratio of timing improvement to buffer cost is usually greatest around the middle region of the cost-delay tradeoff curve, a buffering solution for the whole circuit with minimum cost tends to spread the “useful slack” to each $RT(s_i)$. Based on a buffering solution

B with minimum cost, we can calculate the delay d_i^B , $AT_{s_i}^B$ and $RAT_{s_i}^B$ for each $RT(s_i)$ accordingly. Note that $AT_{s_1}^B = AT_{s_1}$ and $RAT_{t_k}^B = RAT_{t_k}$. Ideally, $AT_{s_i}^B = RAT_{s_i}^B$ since the solution B would consume "useful slack" completely.

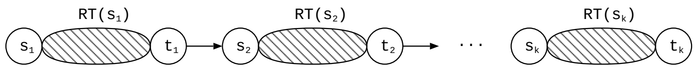


Fig. 22. Example of a circuit for Theorem 1

The following theorem is to quantify the spreading of "useful slack" and it matches with our experimental results of buffering for minimum cost. The minimum delay to PO is $RAT_{t_k}^B - RAT_{s_i}^B$ with buffering solution B and that is $RAT_{t_k} - RAT_{s_i}$ with optimal buffering solution for minimum delay.

Theorem 1 If the "useful slack" evenly distributes to every $RT(s_i)$ for $i = 1, ..., k$ in a manner that the ratio of minimum delay to PO with buffering solution B to that with optimal buffering solution is a constant among all s_i, then

$$RAT_{s_i}^B = RAT_{s_i} - \frac{(RAT_{t_k} - RAT_{s_i})(RAT_{t_k} - AT_{t_k})}{AT_{t_k} - AT_{s_1}}. \tag{5.1}$$

Proof From the assumption that the delay ratio is a constant, we have

$$\frac{RAT_{t_k}^B - RAT_{s_i}^B}{RAT_{t_k} - RAT_{s_i}} = \frac{RAT_{t_k}^B - RAT_{s_1}^B}{RAT_{t_k} - RAT_{s_1}} \tag{5.2}$$

$$\frac{RAT_{s_i} - RAT_{s_i}^B}{RAT_{t_k} - RAT_{s_i}} = \frac{RAT_{s_1} - RAT_{s_1}^B}{RAT_{t_k} - RAT_{s_1}} \tag{5.3}$$

$$RAT_{s_i} - RAT_{s_i}^B = \frac{(RAT_{t_k} - RAT_{s_i})(RAT_{s_1} - AT_{s_1})}{RAT_{t_k} - RAT_{s_1}} \tag{5.4}$$

$$RAT_{s_i} - RAT_{s_i}^B = \frac{(RAT_{t_k} - RAT_{s_i})(RAT_{t_k} - AT_{t_k})}{RAT_{t_k} - RAT_{s_1}} \tag{5.5}$$

$$RAT_{s_i} - RAT_{s_i}^B = \frac{(RAT_{t_k} - RAT_{s_i})(RAT_{t_k} - AT_{t_k})}{AT_{t_k} - AT_{s_1}} \tag{5.6}$$

Equations (5.2)-(5.6) shows the derivation of Theorem 1. From (5.2) to (5.3), we substitute $RAT^B_{t_k}$ with RAT_{t_k} and subtract 1 from both side. (5.4) is based on the fact that $RAT^B_{s_1} = AT_{s_1}$ with zero slack and (5.5) is due to equal slack along the 2-pin routing trees. Finally, we have (5.6) because $RAT_{t_k} - RAT_{s_1} = AT_{t_k} - AT_{s_1}$ which is delay from s_1 to t_k. ■

Theorem 1 provides a method for adjusting the required arrival time of i in Figure 21 and the equation is shown in (5.7), where RAT_z is the required arrival time at z and AT_z is the arrival time at z based on buffer aware STA. Although the slack $RAT_z - AT_z$ is due to the path $\{a \leadsto g \rightarrow h \leadsto p \rightarrow r \leadsto z\}$ which is less than the slack at i, we can use $RAT_z - AT_z$ as a lower bound estimation. In such case, spreading the value of $RAT_z - AT_z$ over path $a \leadsto g \rightarrow i \leadsto q \rightarrow r \leadsto z$ gives a good adjustment to RAT_i and make the sink i a little bit more critical in the routing tree $RT(g)$.

$$\text{adjusted } RAT_i = RAT_i - \frac{(RAT_z - RAT_i)(RAT_z - AT_z)}{AT_z - AT_a} \tag{5.7}$$

E. Path Based Buffer Insertion and Simultaneous Gate Sizing

The framework of our PBBI algorithm provides us the flexibility in integrating gate sizing into PBBI. It is due to the fact that when we cascade several routing trees into one merged tree, input pin and output pin of a module along the processing path is treated as one single vertex v , which is a special candidate buffer location and a buffer must be inserted at v while the resistance/delay/capacitance characteristic of the buffer is derived from that of the module's input-to-output path. In this point of view, if we also consider gate sizing with n choices of size, then we can derive n special buffers according to the sizes. Thus, by restricting the VGDP algorithm to insert one and only one buffer at v choosing from the n special buffers, path based simultaneous buffer insertion and gate sizing is accomplished.

Indeed, treating the gate sizing as selecting a buffer from a buffer library may cause error because a gate can have more than one input and more than one output. When the algorithm changes the size of a gate along a path, for those input pins and output pins which are not along the path, their capacitance or resistance values change simultaneously without considering the impact resulted from the change. However, such sacrifice helps maintaining the solution quality of the processing path (which must be more critical than the unprocessed paths) and we have verified this claim by our experimental results such that the algorithm can substantially reduce overall area of buffers and gates.

1. Gate Sizing at the Sinks

Along a path, using path based simultaneous buffer insertion and gate sizing on the merged routing tree cannot solve the problem when the root of the merged routing tree also need gate sizing. It is especially true when the root of the merged routing tree is an output pin of a module. For example, in Figure 19(c), after we already finished buffer insertion for the merged routing tree of $RT(b)$ and $RT(i)$, if we are processing the path $\{o, s\}$, the merged routing tree is just $RT(o)$ itself. In addition to applying VGDP to $RT(o)$, we have to perform gate sizing for the module $m2$ with pins $\{j, k, o\}$.

Sizing up a gate reduces the output resistance and so the delay is reduced. At the same time, the input capacitance increases which in turn raises the delay of upstream interconnect. Such delay increase can be handled by adding a delay penalty when doing gate sizing as proposed in [47]. However, the main problem about this issue is that the increase in load capacitance may alter the other path delays of the previously buffered routing tree. In the above example, if we size up the module $m2$, the increase in input capacitance at j may increase the downstream load capacitance of $RT(i)$ and it may in turn increase the delay along the path $\{i, r\}$.

In order to fix this problem, we perform gate sizing at all sinks of the merged routing tree while we apply VGDP. If the sink vertex is not a PO, and if the module at the sink is not sized previously, we perform gate sizing for the module at the sink. At a sink s without gate sizing, VGDP starts to propagate one solution with the corresponding load capacitance, the required arrival time RAT_s, and zero cost (which means no buffer has been inserted up to s). With gate sizing at s, we propagate n solutions (n is the total number of different gate sizes), each of which have a scaled load capacitance, the gate size as its cost, and an modified required arrival time $RAT_s'(x_i)$. Assume that the original output resistance of the gate is R_0 and that of a sized gate is R_i, while R_b and C_b is the output resistance and input capacitance of the buffer used in buffer aware STA. For simplicity, we assume that there is a linear relationship between the delay change $(RAT_s'(x_i) - RAT_s)$ and the change in resistance $(R_i - R_0)$. Empirically, we found that (5.8) gives a good and effective calculation for $RAT_s'(x_i)$, where L_{opt} is the optimal buffer interval which is also used in [45]. Intuitively, $(R_i - R_0)(L_{opt}C + C_b)$ gives the change in delay if there exists a buffer in the downstream of s while the length between s and the buffer is L_{opt}.

$$\begin{aligned} RAT_s' &= RAT_s + (R_i - R_0)(L_{opt}C + C_b) \\ RAT_s' &= RAT_s + (R_i - R_0)\left(\sqrt{\frac{2R_bC_bC}{R}} + C_b\right) \end{aligned} \tag{5.8}$$

F. Experimental Results

We have implemented a very efficient VGDP buffer insertion algorithm according to [48] which uses approximation techniques to improve the efficiency. We have performed experiments on 12 combinational circuits which are randomly generated based on real nets from IBM. For simplicity, buffer cost refers to the number of buffers inserted, which approximately stands for buffer area. All experiments are running on a Debian Linux machine with 2.4 GHz processor and 1GB RAM. The size of each testcase is summarized in Table IX.

Table IX. Summary of testcases

circuit	Circuit Size			
	# mod	# edge	total	# B candidate
a1	53	68	121	1449
a2	154	160	314	1880
a3	259	272	531	3190
a4	328	352	680	3897
a5	465	480	945	4316
a6	564	592	1156	6625
a7	742	768	1510	6810
a8	766	816	1582	8658
a9	893	928	1821	8083
a10	999	1072	2071	11623
a11	1958	2136	4094	20432
a12	2983	3120	6103	25738

The second column refers to the number of modulus in the circuit. The third column refers to the number of edges, which equals the total number of sinks for all routing trees. The column “total” is “mod”+“edge” which is a measure of the circuit size. “# B candidate” is the total number of buffer candidate locations for each circuit.

1. Path Based Buffer Insertion

We first compare our PBBI algorithm with the net based buffer insertion algorithm using the same implementation of VGDP.

In order to know the minimum achievable delay of the circuit with optimal buffer insertion, for each net in the circuit, a minimum-delay buffer insertion algorithm is applied, namely “Net based (min delay)”. The minimum delay in ns and the number of buffers inserted are listed in the second and third columns of Table X. We then set the delay constraint of a circuit to be its minimum achievable delay. For all the following comparisons between VGDP and PBBI, the worst slacks of the circuit for both methods are similar and are not shown in the tables. For net based buffering, we tried several different ordering (see Section G) and made the conclusion that, on average, all tested ordering performs similarly in terms of total buffer cost. As a result, in our experimental results, we used the ordering based on ascending order of worse slack for comparison. The results are shown in Table X. The column “B cost” stands for the total number of buffers inserted while “% redu” is the percentage of buffer cost reduction when PBBI is compared to net based buffer insertion (min cost) which minimizes buffer cost while subjects to delay constraint, as described in Section C.

We also performed similar experiments on net based buffer insertion with buffer aware static timing analysis. The result is shown in the last column of Table X. From the results, net based buffer insertion (min cost) with/without buffer aware STA perform similarly in term of buffer resource allocation. This implies the buffer resource allocation of net based buffer insertion cannot be easily improved by only applying simple slack distribution methods.

From Table X, we have the following conclusions.

- The average reduction in buffer area is 15% for PBBI algorithm when comparing to the net based buffer insertion (min cost).
- The average reduction is 72% for PBBI when comparing to the net based buffer insertion (min delay).
- Although there is more than $5\times$ increase in CPU-time, the total CPU time for the biggest circuit with $3k$ modules is still less than 1 minute. In fact, the CPU time is empirically linear to the size of the circuit and the buffer candidate locations.

In our next experiment, we consider buffer blockages and the results are shown in Table XI. From the results, we obtained an even greater buffer cost reduction of 29% when

Table X. Comparison of net based and path based buffer insertion

	Net based (min delay)		Net based (min cost)		Path based (PBBI)			Net based w/ BaSTA
circuit	B cost	min D (ns)	B cost	CPU(s)	B cost	% redu	CPU(s)	B cost
a1	204	10.5	127	0.7	96	24.41	0.7	105
a2	274	20	144	0.5	138	4.17	1.1	142
a3	484	21	216	1.0	177	18.06	2.0	206
a4	592	26.5	317	1.3	288	9.15	2.8	325
a5	645	43	357	1.0	338	5.32	3.6	354
a6	1009	28.3	285	2.1	232	18.60	4.3	262
a7	1033	61.5	384	1.5	307	20.05	8.4	384
a8	1314	33	434	2.6	359	17.28	5.5	430
a9	1238	84	461	1.8	363	21.26	16.1	462
a10	1700	54	492	3.7	414	15.85	13.6	489
a11	2991	105	841	5.9	766	8.92	39.0	803
a12	3925	122	1048	5.1	852	18.70	38.1	1046
Total	15409	—	5106	27.1	4330	15.20 (avg)	135.2	5008

Table XI. Comparison of net based and path based buffer insertion considering buffer blockages

	Net based (min cost)		Path based (PBBI)		
circuit	B cost	CPU(s)	B cost	% redu	CPU(s)
a1	103	0.5	83	24.10	0.7
a2	128	0.5	114	12.28	1.1
a3	195	0.9	151	29.14	1.9
a4	294	1.1	241	21.99	2.8
a5	309	0.9	304	1.64	3.8
a6	272	1.8	186	46.24	4.3
a7	343	1.3	248	38.31	8.5
a8	401	2.2	314	27.71	5.7
a9	423	1.7	294	43.88	16.1
a10	463	3.3	336	37.80	13.4
a11	787	5.6	657	19.79	31.2
a12	961	5.2	712	34.97	33.9
Total	4679	25.2	3640	(avg)28.54	123.4

Table XII. Comparison of net based and path based buffer insertion considering gate sizing

	Net based (min cost)				Path based (PBBI+GS)					(PBBI+GS) w/o sink sizing		
circuit	B cost	Δ G	Δ A	CPU(s)	B cost	Δ G	Δ A	% redu	CPU(s)	B cost	Δ G	Δ A
a1	115	36	151	0.6	81	8	89	41.06	1.5	109	9	118
a2	189	20	209	0.5	102	24	126	39.71	2.5	100	28	127
a3	278	44	322	0.9	151	10	161	50.00	7.0	161	12	173
a4	391	72	463	1.2	222	94	316	31.75	10.3	224	108	332
a5	488	112	600	0.9	218	61	279	53.50	15.7	328	70	398
a6	641	159	800	1.9	197	16	213	73.38	18.1	224	18	243
a7	779	198	977	1.3	235	34	269	72.47	44.9	250	39	289
a8	839	166	1005	2.4	310	52	362	63.98	19.2	416	60	476
a9	973	310	1283	1.7	279	42	321	74.98	61.9	299	48	348
a10	1089	216	1305	3.5	322	39	361	72.34	38.7	394	45	439
a11	2097	331	2428	6.2	587	68	655	73.02	93.2	664	78	742
a12	3149	837	3986	6.0	709	110	819	79.45	119.4	723	127	850
Total	11028	2501	13529	27.2	3413	558	3971	70.65 (avg)	432.5	3892	642	4533

comparing to the net based buffering. From our observation, it is due to the fact that buffer insertion with blockages becomes more complicated, and an algorithm with a global view would perform better.

2. Simultaneous Gate Sizing and Buffer Insertion

We have performed the similar experiments for simultaneously gate sizing and buffer insertion. For net based approach, we have implemented the delay penalty scheme [47] which includes driver sizing into net based buffer insertion process. However, the delay penalty formula used in the paper is mainly for solution comparison in VGDP. When applying the delay penalty as a mean to estimate the delay increase in upstream interconnect, we have to scale the delay penalty empirically. The results is shown in Table XII. In the table, "Δ G" is the total size change in all gates, "Δ area" is the total increase in cost (area) from the buffer insertion/gate sizing, and "% redu" is the percentage of total cost reduction when PBBI+GS is compared to the net based approach.

From Table XII, we found that the overall cost reduction by PBBI algorithm is 71%

when comparing to net based gate sizing/buffer insertion. As the same time, our PBBI+GS algorithm is reasonably efficient as the runtime is within 2 minutes for the biggest testcase.

Our novel technique of performing gate sizing at sink results in significant improvement in solution quality. For [47], the delay penalty is calculated at the driver of a routing tree. It is simple and effective but it only reflects the delay increase in the upstream interconnect toward the gate and the routing tree itself. We have performed experiments for our PBBI algorithm using different gate sizing schemes, results show that gate sizing using delay penalty takes 14% more area than our gate sizing at sink technique (shown in the last three columns of Table XII). In conclusion, the lack of a global view results in the trapping into a local optimal solution and the error exacerbates when the complexity of the problem increases.

3. PBBI on ISCAS Circuits

From [49], we obtained the layout and parasitic information for some placed and routed ISCAS85 circuits in $180nm$ technology. In order to emulate the latest technology, we scale the interconnect by a constant factor. The size of each testcase is summarized in Table XIII.

Table XIII. Summary of ISCAS placed and routed circuits

	Circuit Size		
circuit	# mod	# edge	# B candidate
c1355	619	1096	2377
c1908	938	1523	4647
c2670	1642	2292	7982
c3540	1741	2961	8971
c432	203	343	1015
c499	275	440	1413
c5315	2608	4509	13427
c7552	3828	6253	19291
c880	469	755	1944

We performed experiments on the ISCAS circuit with settings similar to Table X. The results are shown in Table XIV. We found that the results are similar to Table X while PBBI outperforms net based buffer insertion for buffer cost minimization by 16% on average. One special observation is that runtime of PBBI is actually very similar to the net based counterpart.

We found that the efficiency is due to that fact that the PI to PO paths in the ISCAS circuits are comparatively shorter than our random testcases in terms of number of modules along the paths. In such a case, the resultant merged routing trees for VGDP are smaller.

Table XIV. Comparison of net based and path based buffer insertion

	Net based (min delay)		Net based (min cost)		Path based (PBBI)		
circuit	B cost	min D (ns)	B cost	CPU(s)	B cost	% redu	CPU(s)
c1355	585	23.9	74	0.19	67	9.46	0.18
c1908	1027	50	106	0.3	82	22.64	0.38
c2670	2030	65	85	0.47	82	3.53	0.54
c3540	2082	82.7	100	0.54	77	23.00	0.64
c432	212	19.9	24	0.06	17	29.17	0.12
c499	300	19.6	39	0.09	31	20.51	0.14
c5315	3322	74.7	150	0.85	140	6.67	1.02
c7552	4411	75.6	240	1.15	187	22.08	1.50
c880	395	16.9	21	0.14	21	0.00	0.16
Total	14364	—	839	3.79	704	(avg)16.09	4.68

G. Different Orderings for Net Based Buffer Insertion

In order to find out the the best ordering for net based buffer insertion in terms of buffer cost minimization subject to timing constraints, we performed a set of experiments on 20 randomly generated circuits. In the experiment, we have tested the following ordering:

a the topological order from PI to PO;

b the topological order from PO to PI;

c first process the net with the smallest worse slack;

d random;

e pick a path according to the descending order of its criticality, then pick a net from PI to PO;

f pick a path according to the descending order of its criticality, then pick a net from PO to PI;

g according to the descending order of the total load capacitance of the nets;

h pick a path according to the descending order of its criticality, then according to the descending order of the total load capacitance of the nets;

i first process the net such that most critical paths are passing through it;

j assign a cost value to each path representing its criticality, then first process the net such that it has the biggest total cost from all the paths passing through it;

The results are shown in Table XV. Each column represents one circuit and all numbers shown in the table are numbers of buffer inserted. From the table, we observed that no single ordering outperforms other orderings for most testcases. In fact, all tested ordering performs on average similarly in terms of total buffer cost.

H. Conclusion

The VLSI technology scaling requests increasingly more buffers in circuit designs and therefore buffer insertion needs to be carried in more elaborated manner. As a departure

Table XV. Number of buffers inserted for 20 testcases

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
a	148	117	54	128	126	86	84	180	167	133
b	135	117	82	135	113	85	83	180	163	145
c	116	103	69	120	119	89	82	172	135	127
d	145	125	64	131	130	104	114	178	180	153
e	136	110	73	128	126	104	81	169	135	124
f	116	103	70	128	119	89	65	168	140	115
g	146	115	44	128	128	107	78	168	132	110
h	136	110	73	128	126	104	81	169	132	116
i	136	105	52	118	118	104	78	164	143	130
j	132	101	52	118	118	104	78	164	143	124

	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20
a	118	99	104	80	65	47	227	559	467	382
b	133	116	100	61	52	37	229	455	375	329
c	114	82	106	101	74	44	236	446	378	292
d	120	100	104	98	66	42	228	483	487	417
e	114	92	106	95	74	44	227	443	380	287
f	113	85	106	102	72	45	236	456	392	294
g	84	83	105	92	67	57	227	526	498	429
h	114	92	107	96	75	52	227	445	382	289
i	103	96	109	86	49	42	160	424	381	289
j	103	96	109	81	45	42	159	424	381	289

from traditional net based buffer insertion methods, we propose a path based buffer insertion approach which can obtain a better buffer usage efficiency due to its global view. To the best of our knowledge, this is the first work on path based buffer insertion. Compared to network based methods, our approach is more practical in terms of computation complexity. Several techniques are proposed along with the path based buffering including buffer aware static timing analysis, slack spreading along off-paths and simultaneous sink sizing. Experimental results show that our approach can reduce buffer/gate cost by 71% on average compared to net based methods.

CHAPTER VI

CIRCUIT CLUSTERING FOR FPGA TECHNOLOGY MAPPING

Circuit clustering is defined as assigning circuit elements into clusters under different design constraints, such as area and pin constraints [50, 51, 52, 53]. In this way, the circuit clusters are smaller compared to the original circuit, and hence manipulation and synthesis of the clusters are easier. Most circuit clustering algorithms aim at either minimizing the circuit delay or the inter-cluster connections. One major application of circuit clustering is FPGA technology mapping. In this problem, each cluster is then mapped to a lookup table (LUT) which is the basic logic element in a FPGA chip.

We focus on the following two objectives of the circuit clustering problem in the next two chapters.

- **Optimal circuit clustering for delay minimization under a more general delay model**

  As the interconnect delay dominates the path delay in VLSI technology advancement, we can no longer use simplified delay model [50, 51] for circuit clustering. Our work is to propose a delay model in order to handle the importance of interconnect delay which is vital for new FPGA architectures. The research also aims at considering the area-constrained clustering of combinational circuits for delay minimization under the proposed delay model, which practically takes variable interconnect delay into account. The delay model is particularly applicable when allowing the back-annotation of actual delay information to drive the clustering process.

- **Multi-level circuit clustering for delay minimization**

  In [5], it is stated that the growth of FPGA industry is much faster than the average of the semiconductor industry. For example, Altera launched the MAX5000 devices with only 600 to 3750 usable gates while it shipped the APEX 20K devices family with up to 51480 logic elements (each of which is a 4-input LUT) and 0.3 to 1.5 million usable gates [54]. Dealing with the high and ever growing complexity of the FPGA devices, Altera employs a 2-level (namely MegaLAB) architecture as shown in Figure 23. In the structure, each MegaLAB consists of 10-24 of logic array blocks (LABs) which contains 10 LUTs connected by local interconnect array. We can treat the LAB as the first-level cluster and the MegaLAB as the second-level cluster. This design improves timing performance by manipulating the fast local interconnect within a LAB and semi-global interconnect within a MegaLAB. In order to cope with the new architectures, we propose an algorithm towards the problem of multi-level circuit clustering for delay minimization and applicable to hierarchical FPGAs.

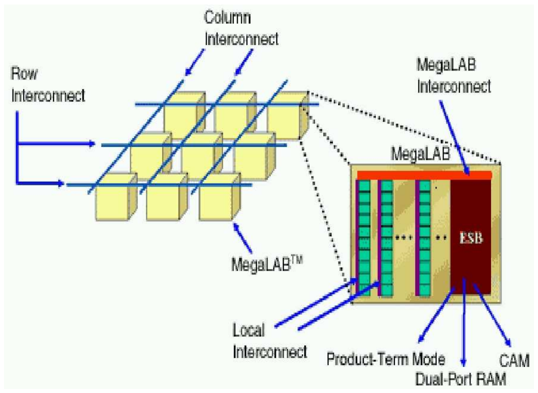


Fig. 23. The MegaLAB structure in Altera APEX 20K devices

CHAPTER VII

OPTIMAL CIRCUIT CLUSTERING FOR DELAY MINIMIZATION UNDER A MORE GENERAL DELAY MODEL*

A. Introduction

Circuit clustering is to assign circuit elements into a number of clusters under different design constraints, such as area and/or pin constraints [51, 52, 53, 50]. In this way, the circuit clusters are smaller compared to the original circuit and manipulation of these clusters is easier. Circuit clustering algorithms usually aim at minimizing either the circuit delay or the inter-cluster connections.

In this paper, we focus on the problem of combinational circuit clustering for delay minimization subject to area constraints. This problem is first studied in [50]. The authors formulate the problem in the unit delay model, in which no delay is associated with any gate or with any connection within a cluster and a unit delay is assigned to each inter-cluster connection. They propose a polynomial time algorithm to solve the problem optimally. Recently, most researchers adopt the general delay model [53], in which each gate is associated with a delay value, no delay is for each connection within the same cluster, and a constant delay is for each inter-cluster connection. An algorithm for the circuit clustering problem under the general delay model is proposed in [51]. It is proved that the algorithm can optimally solve the problem in polynomial time.

However, when wire delay starts to dominate the total delay in a circuit, the general delay model is no longer capable of handling more practical problems in current technol-

*Reprinted with permission from "Optimal Circuit Clustering for Delay Minimization under a More General Delay Model" by C. N. Sze and T.-C. Wang, 2003. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 22, Issue: 5, pp. 646-652. COPYRIGHT 2003 by IEEE.

ogy. Hence, it is necessary for the delay model to be more sophisticated such that a delay value is also associated with each connection within a cluster. As a result, we propose a new delay model in this paper, which is practical for the delay back-annotation techniques, in which the actual delay information of the circuit after place and route is fed-back to drive the clustering process. In fact, our delay model is more general because the general delay model adopted in [51] is a special case of our model (with all connections within the same cluster set to zero).

We demonstrate several trivial extensions of the algorithm in [51] and show that they cannot optimally solve the circuit clustering problem under our proposed delay model. Details are discussed in Section C.

Besides, we present a vertex grouping technique, and integrate it with the algorithm in [51] such that our algorithm can be proved to optimally solve the area-constrained combinational circuit clustering problem for delay minimization under our delay model in polynomial time.

The paper is organized as follows. The next section gives the problem definition. In Section C, we present a brief review of the algorithm in [51] and several trivial extensions. Section D describes our vertex grouping technique while the overall algorithm is discussed in Section E. Analysis of the algorithm and conclusion are included in the last two sections.

B. Problem Definition

A combinational circuit can be represented as a directed acyclic graph (DAG) $G = (V, E)$. V is the set of vertices which represent the functional blocks (e.g., gates) in the graph and E is the set of edges which stand for the connections among the blocks. In the graph, PI is the set of vertices with out-going edges only, and on the contrary, PO vertices have in-coming edges only. A vertex u is a predecessor (successor) of a vertex v if there exists a path from

u to v (from v to u). A vertex u is an immediate predecessor (immediate successor) of a vertex v if there exists an edge from u to v (from v to u).

For each vertex $v \in V$, let $w(v)$ represent its area. A cluster $C \subseteq V$ is a set of vertices $\{v_1, v_2, ..., v_k\}$ which satisfies the area bound M, where M is a given constant. For each cluster C, its area $w(C)$ is defined as the sum of area of all vertices in it and must be no more than M. That is,

$$w(C) = \sum_{v \in C} w(v) \leq M$$

In the input (unclustered) graph, delay values are associated with all vertices and edges. For each vertex $v \in V$, let $\delta(v)$ represent its intrinsic delay. For each edge $(u, v) \in E$, it is associated with a delay $\delta(u, v)$ (Note that, $\delta(u, v) = 0$ in [51]). For the graph in Figure 24(a), the numbers beside the vertices and edges indicate the delay values associated with them. For example, the vertex delay of e is 1 ($\delta(e) = 1$), and the edge delay of (c, e) is 4 ($\delta(c, e) = 4$).

A clustering S on the graph G is defined as a set of clusters, $S = \{C_1, C_2, ..., C_m\}$, such that all the clusters in S satisfy the following condition.

$$\forall i \in \{1, ..., m\}, C_i \subseteq V \text{ , s.t. } \begin{cases} w(C_i) \leq M, \\ \bigcup_{i=1}^{m} C_i = V \end{cases}$$

Note that node duplication (i.e., a node appearing in more than one cluster) may happen in S. Let G' be the clustered graph induced by a clustering S on the graph G. The delay associated with G' is evaluated as follows. For each edge (u, v) within the same cluster, it still has the original delay $\delta(u, v)$. However, for each edge connecting two vertices in different clusters, its edge delay is replaced by a fixed value D. For the clustered graph G' shown in Figure 24(b), the set of boxes indicates a clustering (which contains three clusters C_1, C_2, C_3) on the graph in Figure 24(a). This clustering contains node duplication - node a appears in both C_1 and C_2. Since the vertices a, c, e are in the same cluster C_1, the edge delays are $\delta(a, c) = 6$ and $\delta(c, e) = 4$. (But for the delay model adopted in [51], there is no delay associated with the edge (a, c) or (c, e) in this example.) However, the edge (e, g) is across two clusters C_1 and C_3, so $\delta(e, g)$ becomes a predefined value D. Practically, we assume $\delta(u, v) \leq D$ for each edge (u, v) in the input graph.

To calculate the delay of a path from vertex u to vertex v, we always include all vertex delays and edge delays along the path. The *path delay* at a vertex v is defined as the maximum delay of all paths from PIs to v.

The delay of a clustered circuit G' is defined as the maximum delay of all paths from PIs to POs, which is equal to the maximum path delay at all PO vertices. For example, in Figure 24(b), the delay of G' is $15 + D$ along the path $a \rightarrow c \rightarrow e \rightarrow g$. Based on the definitions, the circuit clustering problem considered in this paper is presented in the following.

Circuit clustering with variable interconnect delay

Given a graph G, find a clustering S, a set of clusters, such that the delay of the clustered circuit is minimized.

C. Previous Work and Pitfalls in Trivial Extensions

1. Previous Work

In this section, we discuss the algorithm in [51], which solves the circuit clustering problem optimally under the general delay model. In the algorithm, each cluster has one and only one root vertex r and we denote the cluster rooted at r (which is found by the algorithm) as $cluster(r)$[1]. Let G_r be the set of all predecessors of r together with the vertex r. Note that

[1]All vertices in the cluster are predecessors of the root vertex.

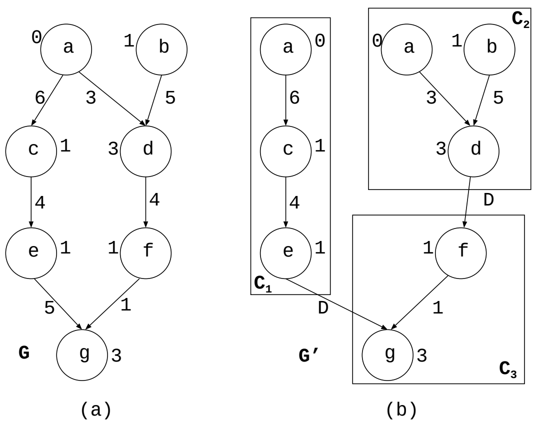


Fig. 24. A circuit with variable interconnect delay and a clustered circuit

in the algorithm, $cluster(r)$ is a subset of G_r. When the context is not ambiguous, we also denote the subgraph induced by G_r by the vertex set G_r.

The algorithm consists of two phases: labeling phase and clustering phase. For each vertex $u \in V$, the label of u, $l(u)$, is defined as the minimum path delay at u among all possible clusterings on the graph G_u. In the labeling phase, for each vertex r in a topological order, the algorithm finds $cluster(r)$ from G_r such that it would make the path delay at r become the minimum among all possible clusterings on G_r, and at the same time, the algorithm obtains $l(r)$.

In order to get $l(r)$, the algorithm first calculates $l'(u)$ of each predecessor u of r. $l'(u)$ is defined as the sum of $l(u)$ and the maximum delay of the paths from the output of u to the output of r in the input graph[2]. Then, a vertex u with the highest l' value is repeatedly found and included into the cluster rooted at r until the cluster area violates the area constraint.

[2]When calculating the delay of a path from the output of u to the output of r, the vertex delay of u is not included.

After finding the $cluster(r)$, the algorithm continues to calculate l_1, the maximum l' value of the PI vertices which are inside the cluster, and find l_2, the maximum $l' + D$ value of the vertices outside the cluster. After that, the label of r, $l(r)$, can be found by getting the greater value of l_1 and l_2.

The clustering phase constructs clusters from POs to PIs according to the cluster information generated in the labeling phase. First, for each PO vertex v, the corresponding $cluster(v)$ is included into the clustering S. Then, for each vertex u outside S, which is an immediate predecessor of any vertex inside S, $cluster(u)$ is also included into the clustering S. The procedure is repeated until all vertices in G are included in S.

2. Pitfalls in Trivial Extensions

This section shows that the algorithm in [51] cannot be "trivially" extended to deal with the circuit clustering problem with variable interconnect delay.

The labeling phase in [51] is based on the fact that in the general delay model, for each vertex $u \in G_r - r$, $l'(u) + D$ effectively represents the path delay at r due to u when u is not included in $cluster(r)$. Therefore, to make the resultant path delay at r as small as possible, adding vertices into $cluster(r)$ is done in the non-increasing order of the l' values. However, in our delay model, it requires replacing the original connection delay with a constant inter-cluster delay D; hence, instead of $l'(u) + D$, the expression $l'(u) + D - \delta(u, g(u))$ represents the path delay at r due to u when u is not included in $cluster(r)$. Note that for each vertex u, $g(u)$ is defined as the immediate successor of u such that the delay on the path from the output of u to the output of r passing through $g(u)$ is maximum among all immediate successors of u[3]. Therefore, if $l'(u) > l'(v)$, it is not always true that $l'(u) + D - \delta(u, g(u)) > l'(v) + D - \delta(v, g(v))$. As a result, we cannot

[3]If more than one immediate successor of u satisfies the definition of $g(u)$, we just pick any one of them to be $g(u)$.

use l' value to select vertices.

To simplify our presentation, we define $l^*(u)$ to be $l'(u) + D - \delta(u, g(u))$ for each vertex $u \in G_r - r$. In other words, for a root vertex r, $l^*(u)$ indicates the maximum delay of the paths through the vertex u when u is not in the cluster rooted at r. Obviously, using $l^*(u)$ rather than $l'(u)$ for vertex selection is another trivial extension for applying the algorithm in [51] under our delay model. However, problems still exist in this extension. An example is shown in Figure 25(a). In the figure, $M = 3, D = 7$, each vertex has a unit area and the number beside each vertex or edge is the delay value associated with it. It is easy to calculate that $l^*(e) = 22$, $l^*(f) = 24$, $l^*(c) = 21$, $l^*(d) = 21$, $l^*(a) = 23$, $l^*(b) = 20$ with respect to the root vertex g. Under this calculation, we know that vertex selection based on the value of l^* should be in the order of f, a, e, c, d, b. Based on this ordering, we can eventually form the clustering $\{\{g, f, a\}$, $\{e, c, a\}$, $\{d, b\}\}$, and the path delay at g is 22 which is shown in Figure 25(b). However, as in Figure 25(c), if we form the clustering $\{\{g, f, e\}$, $\{c, a\}$, $\{d, b\}\}$ instead, the path delay at g becomes 21. In fact, $\{g, f, e\}$ is an optimal choice for $cluster(g)$ while $l(g) = 21$. The problem of using the value of l^* to select vertices is that the vertex set of the resultant cluster may not be connected and this may increase the circuit delay. This example shows that such a trivial extension of algorithm in [51] is also unable to produce optimal solutions.

D. Vertex Grouping

In this section, we first define the function $l''(x, y)$ for each edge (x, y) and then present our vertex grouping algorithm.

Definition 1

Given a root vertex r and its associated G_r, the l'' value of an edge (x, y), which represents the path delay at r due to x if x is not included into the cluster of r

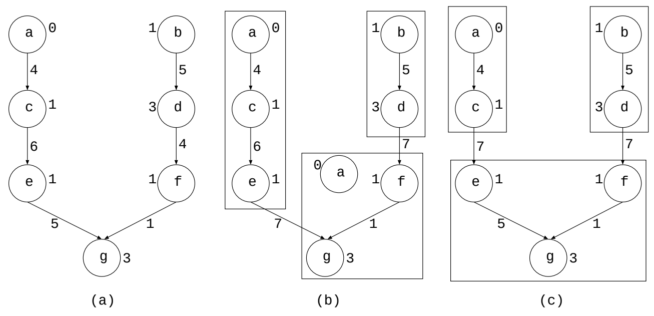


Fig. 25. Examples of circuit clustering

and y is included, is defined as

$$l''(x, y) = l(x) + D + \Delta(y, r),$$

where $\Delta(y, r)$ is defined as the maximum delay (including gate and edge delays) along any path from y to r, in the input graph (assuming all gates are in the same cluster).

By calculating $l''(x, y)$, we obtain the delay due to x in the situation where x is excluded from the cluster and y is included into the cluster rooted at r. The reason why l'' is calculated for each connection, but not for each vertex, is that the delay on each fanout edge of a vertex can differ. If $M = 3, D = 7$, the l'' values for the graph in Figure 26(a) (for root vertex g) are shown in Figure 26(b). In the figure, it is clear that when c is included into the cluster while a is excluded, the path delay at g is greater than the situation when we add a into $cluster(g)$ since $l''(a, c) = 23 > l''(c, e) = 21$. In other words, if we cannot include a and c at the same time, we should not choose to include c. By this observation, vertices should be added into a cluster in a "group" basis. Hence, we propose the following grouping strategy.

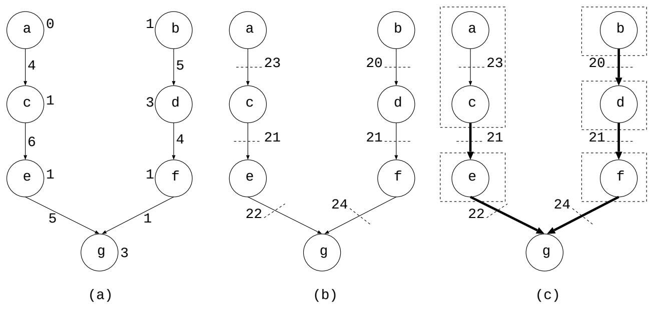


Fig. 26. Illustration of vertex grouping

Definition 2 (Vertex Grouping)

Given a root vertex r, its associated G_r, and an edge (a_0, x) in G_r, $group(a_0, x)$ is a subset of G_r such that a predecessor a_i of a_0 must be assigned into $group(a_0, x)$ if there exists a path from a_i to a_0, denoted $a_i \to a_{i-1} \to ... \to a_1 \to a_0$, such that

$$l''(a_k, a_{k-1}) > l''(a_0, x), \forall k = 1, 2, ..., i.$$

Also, a_0 is always assigned into $group(a_0, x)$.

Note that in Definition 2, all vertices on the path from a_i to x are the predecessors of r, i.e., $x, a_0, ..., a_i \in G_r$. Based on this grouping definition, a is assigned into $group(c, e)$, shown in Figure 26(c), because there exists a path $a \to c$ such that $l''(a, c) > l''(c, e)$. In fact, it indicates that c and a should be assigned to the cluster rooted at g at the same time.

Definition 2 describes the condition whether a vertex a_i in G_{a_0} is in $group(a_0, x)$. In order to get $group(a_0, x)$, we can check all vertices in G_{a_0}. However, given the subgraph G_r rooted at r, it is not necessary to get $group(a, b)$ for every edge (a, b) in G_r. Therefore, we propose the following vertex grouping algorithm which assigns vertices into $group(a, b)$ for some edges (a, b) only. The algorithm is shown below, followed by a detailed discussion.

```
ALGORITHM Grouping($r$,$G_r$)
Input  : vertex $r$, $G_r$
Output : list $P_r$ of edges,
         $group(a,b)$ for each edge $(a,b) \in P_r$
begin
    leader_set={};
    FOR each immediate predecessor $u$ of $r$
        /* i.e., $(u,r) \in E$ */
        put edge $(u,r)$ into leader_set;
    END FOR
    WHILE (leader_set is not empty)
        remove an edge $(a,b)$ from leader_set and
        put it into $P_r$;
        $group(a,b) = \{a\}$;
        FOR each immediate predecessor $x$ of $a$
            Group_vertex($x$,$a$,$a$,$b$);
        END FOR
    END WHILE
    return $P_r$, $group(a,b)$ for each edge $(a,b) \in P_r$;
end

Group_vertex($x$,$y$,$a$,$b$)
begin
    IF ($x$ in $group(a,b)$)
        return;
    ELSE IF ($l''(x,y) \leq l''(a,b)$)
        add $(x,y)$ into leader_set;
        return;
    ELSE IF ($l''(x,y) > l''(a,b)$)
        add $x$ into $group(a,b)$;
        FOR each immediate predecessor $w$ of $x$
            Group_vertex($w$,$x$,$a$,$b$);
        END FOR
    END IF
end
```

In $Grouping()$, each edge (u,r) is first stored in the "leader_set". When all vertices in $group(u,r)$ are found, the edge (u,r) has been removed from "leader_set" and put into a list P_r. We denote all edges in both "leader_set" and P_r as "leader edges". $Group\_vertex(x,y,a,b)$ shows the process of checking whether the vertex x should be added into $group(a,b)$. If $x \in G_a$ does not comply with the definition of $group(a,b)$, the edge (x, y) is added into the "leader_set". The algorithm terminates when "leader_set" is empty.

In this algorithm, not all edges in G_r may be added to P_r, and we only obtain $group(a, b)$ for each "leader edge" $(a, b) \in P_r$. After $Grouping()$, all predecessors of r are divided into groups and the "leader edge" of each group is stored in the list P_r.

For the circuit in Figure 26(a), the results after $Grouping()$ are shown in Figure 26(c). In the figure, the thick lines are edges in P_r (leader edges) while totally five groups are formed. With the grouping algorithm, we are able to obtain an important property in which adding a group (no longer a vertex) into a cluster without increasing the path delay at the root vertex r is possible.

E. The Algorithm

In this section, we present our algorithm (shown below) based on the grouping strategy, which is a "non-trivial" extension of the algorithm in [51], for the circuit clustering problem under our new delay model.

```
ALGORITHM Circuit_clustering($G$)
Input  : graph $G = (V, E)$
Output : a set of clusters $S = \{C_1, C_2, ..., C_n\}$
1.  begin
2.  compute the maximum delay matrix $\Delta$,
    where $\Delta(i, j)$ is the maximum delay along
    any path from $i$ to $j$, (including gate
    and edge delays) assuming all gates are
    in the same cluster;
3.  FOR each PI $i$, $l(i) = \delta(i)$;
4.  sort the non-PI vertices of $G$ in a
    topological order to obtain list $T$;
5.  WHILE $T$ is not empty
6.      remove the first vertex $r$ from $T$;
7.      compute $G_r$;
8.      FOR each edge $(u, w) \in E$
         s.t.  $u \in G_r - \{r\}$ and $w \in G_r$
9.           $l''(u, w) = l(u) + \Delta(w, r) + D$;
```

```
        END FOR
10.     $P_r =$ Grouping($r, G_r$);
11.     $P'_r =$ sort the edges in $P_r$ in
              non-increasing order of $l''$;
12.     Labeling($r, P'_r$);
    END WHILE
13. $L =$ all PO vertices;
14. $S = \phi$ ;
15. WHILE $L$ is not empty
16.     remove a vertex $r$ from $L$;
17.     $S = S \cup \{$cluster$(r)\}$;
18.     FOR each vertex $x \in (V -$cluster$(r))$,
         s.t. $x$ is an input of cluster($r$)
              and cluster$(x) \notin S$
19.         $L = L \cup \{x\}$;
        END FOR
    END WHILE
20. end

Labeling($r, P$)
Input  : vertex $r$, list $P$
Output : $l(r)$, cluster$(r)$
1.  begin
2.  cluster($r$)=$\{r\}$;
3.  WHILE $P$ is not empty
4.     remove the first edge $(u, w)$ in $P$;
5.     IF($w($cluster$(r) \cup group(u, w)) \leq M$ )
6.         cluster$(r)$ =cluster$(r) \cup group(u, w)$;
7.     ELSE
8.         insert $(u, w)$ back to the head of $P$;
9.         break;
       END IF
    END WHILE
10. $l_1(r) =$ max$\{l''(x, g(x)) - D + \delta(x, g(x))|$
                                  $x \in ($cluster$(r) \cap (PI))\}$;
11. $l_2(r) = (l''(u, w)|(u, w)$ is the first edge in $P)$;
12. $l(r) =$ max$\{l_1(r), l_2(r)\}$;
13. end
```

Similar to [51], our algorithm consists of two parts: the labeling phase (lines 3-12) and the clustering phase (lines 13-19). The clustering phase works in the same way as [51].

In the labeling phase, we get a cluster $cluster(r)$ for each vertex r in a topological order. For each non-PI vertex r, we first compute the value $l''(u,w)$ for each edge $(u,w) \in E$ which connects vertices in G_r. Based on the l'' values, we apply our vertex grouping algorithm to divide the vertices in G_r into groups. After that, each group of vertices is considered for adding to $cluster(r)$ based on the l'' value of its leader edge. Thus, the leader edges are sorted in advance according to their l'' values.

For example, applying our algorithm to the circuit in Figure 26(a), the resultant clustering contains three clusters: $cluster(g) = \{g, f, e\}$, $cluster(c) = \{a, c\}$, $cluster(d) = \{b, d\}$, and $l(g) = 21$ along the path $a \rightarrow c \rightarrow e \rightarrow g$ or $b \rightarrow d \rightarrow f \rightarrow g$. In fact, the clustering is the same as the optimal one shown in Figure 25(c).

The main difference between our algorithm and the algorithm in [51] is that we employ the grouping strategy and add vertices to a cluster in a group basis, which guarantees a minimum circuit delay under our delay model. Besides, in order to apply the grouping strategy, we calculate l'' for every edge while the algorithm in [51] calculates l' for every vertex for selecting vertices which has been shown not applicable to our delay model.

For our algorithm, we have the following lemma describing some important properties of the sorted list P'_r that is generated in line 11 of $Circuit\_clustering()$.

Lemma 1 For any non-PI vertex r, the following properties $P1$, $P2$ $P3$ and $P4$ are correct. Let (w,u) and (x,y) be any two edges in P'_r (note that w, u, x, y are in G_r).

P1. If (w,u) is positioned before (x,y) in P'_r, then

$$l(w) + \Delta(u,r) + D \geq l(x) + \Delta(y,r) + D$$

P2. If x is the predecessors of both w and u (y and w may be the same vertex) such that $y \in group(w,u)$ and $x \notin group(w,u)$, then

$$l(w) + \Delta(u,r) + D \geq l(x) + \Delta(y,r) + D$$

P3. If a vertex z is the predecessor of both w and u such that $z \in group(w, u)$, there must exist a path from z to w such that all edges (p, q) along the path satisfy $p \in group(w, u)$ and $q \in group(w, u)$ and they must also satisfy the following inequality.

$$l(p) + \Delta(q, r) + D > l(w) + \Delta(u, r) + D$$

P4. If x is the predecessor of both w and u such that $y \in group(w, u)$ and $x \notin group(w, u)$, there must exist a path from y to w such that all edges (p, q) along the path satisfy $p \in group(w, u)$ and $q \in group(w, u)$ and they must also satisfy the following inequality.

$$\begin{aligned} l(p) + \Delta(q, r) + D > l(w) + \Delta(u, r) + D \\ \geq l(x) + \Delta(y, r) + D \end{aligned}$$

Proof $P1$ is trivial and it is simply because the list P'_r is sorted. For $P2$, it is true because $Group\_vertex()$ stops including the predecessor x only when $l''(w, u) \geq l''(x, y)$. The same reasoning is applicable for $P3$; before $Group\_vertex()$ stops including the predecessors into $group(w, u)$, each examined edge (p, q) must have $l''(p, q) > l''(w, u)$. $P4$ is obtained by joining $P2$ and $P3$. □

F. Optimality and Complexity of the Algorithm

The authors in [51] state that in any clustering S on a graph G, the path delay at any vertex v should be greater than or equal to the path delay at v in an optimal clustering S_{op} on G_v (**Lemma 1** in [51]). It can be easily proved that this lemma is also applicable to our new delay model. To prove the optimality of our algorithm, we first prove that (1) the label of each vertex, $l(v)$ (which is calculated in the labeling phase), is the lower bound of the path delay at v in any optimal clustering, and (2) the clustering phase (lines 13-19) is able to construct a clustering such that the path delay at v equals $l(v)$. From (1) and (2), together with **Lemma 1** in [51], it can be then proved that the clustering S generated by our algorithm is an optimal clustering.

Before proving (1) and (2), we first explore an important lemma for vertices within a cluster.

Lemma 2 For any cluster $cluster(v) \supset \{v\}$ generated in $Labeling()$, for each edge (x, y) such that $x, y \in G_v$, $x \in (G_v - cluster(v))$ and $y \in cluster(v)$, if we arbitrarily divide $cluster(v)$ into two sets A and B such that $A \neq \phi, B \neq \phi$, $A \cup B = cluster(v)$ and $A \cap B = \phi$, there must exist an edge (z, w) connecting vertices in A and B (without loss of generality, assume $z \in A$ and $w \in B$) such that $l''(z, w) \geq l''(x, y)$.

Proof In the labeling phase, groups of vertices are assigned into $cluster(v)$ in the non-increasing order of the leader edges' l'' values. According to $P4$ in Lemma 1, if a vertex z is assigned into $cluster(v)$ (that means $cluster(v) \supset \{v\}$), there must exist an immediate successor w of z and an edge (j, k) such that $z \in group(j, k)$ and $l''(z, w) \geq l''(j, k)$, while the edge (j, k) must have the value $l''(j, k)$ greater than or equal to the l'' values of all leader edges of those groups that are not yet assigned to $cluster(v)$. Therefore, $l''(z, w) \geq l''(j, k) \geq l''(x, y)$. $\square$

From Lemma 2, we know that $l''(z, w) \geq l''(x, y)$. Besides, we have $l_2(v) = max\{l''(x, y)|x, y \in G_v, x \in (G_v - cluster(v)), y \in cluster(v)\}$ (which is generated in line 11 of $Labeling()$). The following Corollary can be easily derived.

Corollary 1 For any cluster $cluster(v) \supset \{v\}$ generated in $Labeling()$, if we arbitrarily divide $cluster(v)$ into two sets A and B such that $A \neq \phi, B \neq \phi$, $A \cup B = cluster(v)$ and $A \cap B = \phi$, there must exist an edge (z, w) connecting vertices in A and B (without loss of generality, assume $z \in A$ and $w \in B$) such that $l''(z, w) \geq l_2(v)$.

Lemma 3 For any vertex v, the path delay at v in any optimal clustering of the subgraph

G_v, denoted by $delay(v)$, is greater than or equal to $l(v)$.

Proof It is proved by induction.

(1. Induction basis) For any PI vertex v, $l(v) = \delta(v)$. It is obvious that $l(v) = delay(v)$.

(2. Induction step) Assume that the statement, $l(x) \leq delay(x)$, is true for all vertices $x \in (G_v - \{v\})$, and we are going to prove that the statement is also true for vertex v, i.e., $l(v) \leq delay(v)$.

The value of $l(v)$ is the maximum value of $l_1(v)$ and $l_2(v)$ (in line 12 of $Labeling()$). We consider the two cases separately.

- (Case 1) $l(v) = l_1(v) = max\{l''(x, g(x)) - D + \delta(x, g(x)) | x \in (cluster(v) \cap (PI))\} = max\{l(x) + \Delta(g(x), v) + \delta(x, g(x)) | x \in (cluster(v) \cap PI)\}$. Since x is a PI, $l(v) = max\{\delta(x) + \Delta(g(x), v) + \delta(x, g(x)) | x \in (cluster(v) \cap PI)\} = max\{\Delta(x, v) | x \in (cluster(v) \cap PI)\}$, which is the maximum delay among the paths from all PIs in $cluster(v)$ to v, and by assumption, $l_1(v)$ is greater than or equal to the delay value of all other paths involved in calculating $l_2(v)$. Hence, the path delay at v in any optimal clustering of the subgraph G_v cannot be smaller than $l_1(v)$.

- (Case 2) $l(v) = l_2(v) = max\{l(x) + \Delta(w, v) + D | (x, w) \in E, w \in cluster(v)$ and $x \in (G_v - cluster(v))\}$. This case implies $cluster(v) \subset G_v$. Here, we prove by contradiction.

  Assume $delay(v)$ is smaller than $l(v)$, i.e., $l(v) > delay(v)$. And we denote the corresponding cluster rooted at v in an optimal clustering as C_v. Without loss of generality, we assume $C_v \subseteq G_v$. There are three cases.

  (a) If $C_v = cluster(v)$ ($\subset G_v$), there must exist two edges (a, b), (a', b') (they may be the same) such that $a, a' \in (G_v - C_v)$, $b, b' \in C_v$, $l(v) = l(a) + \Delta(b, v) + D$ and $delay(v) = delay(a') + \Delta(b', v) + D$. This is depicted in Figure 27(a). First, $delay(a') + \Delta(b', v) + D = delay(v) \geq delay(a) + \Delta(b, v) + D$. Second, based on the induction hypothesis that $l(x) \leq delay(x)$ for all vertices $x \in (G_v - \{v\})$, we have $delay(a) \geq l(a)$. Thus, $l(v) \leq delay(v)$ but it contradicts our assumption. (See Figure 27(a).)

(b) If $C_v \supset cluster(v)$ (which implies $C_v - cluster(v) \neq \phi$), there must exist an edge (a, b) with $a \in G_v - cluster(v)$ and $b \in cluster(v)$ such that $l''(a, b) = l(v)$. Let E_C denote the set of all such edges. For each edge $(a, b) \in E_C$, both a and b are in C_v because $delay(v) < l(v)$. So, $b \in cluster(v)$ and $a \in C_v - cluster(v)$. We consider the first edge (u, w) remaining in P in line 11 of $Labeling()$ with $l(v) = l''(u, w)$. Since $l(v) > delay(v)$, the vertex u must be in $C_v - cluster(v)$ and $w \in cluster(v)$, i.e., $(u, w) \in E_C$. In this case, $Labeling()$ should add $group(u, w)$ into the $cluster(v)$ if the area of $cluster(v) \cup group(u, w)$ does not exceed the area constraint M. However, $u \in G_v - cluster(v)$ implies that the area of $cluster(v) \cup group(u, w)$ exceeds M. When $u \in C_v$ and the area of $cluster(v) \cup group(u, w)$ is greater than M, there must exist an edge (f, g) such that $f \in group(u, w)$, $g \in group(u, w)$, $f \in (G_v - C_v)$ and $g \in C_v$. The situation is depicted in Figure 27(b). Based on the induction hypothesis, $delay(f) + \Delta(g, v) + D \geq l(f) + \Delta(g, v) + D$. Then, by $P3$ of Lemma 1, $delay(v) \geq delay(f) + \Delta(g, v) + D \geq l(f) + \Delta(g, v) + D > l(u) + \Delta(w, v) + D = l(v) > delay(v)$ which is impossible. (See Figure 27(b).)

(c) If $(cluster(v) - C_v) \neq \phi$, we can divide $cluster(v)$ into two disjoint subsets $cluster(v) - C_v$ and $cluster(v) \cap C_v$. By Corollary 1, there exists an edge (s, t) such that $s \in (cluster(v) - C_v)$ and $t \in cluster(v) \cap C_v$ such that $l(s) + \Delta(s, t) + D = l''(s, t) \geq l_2(v) = l(v)$. This is depicted in Figure 27(c). Since we know $delay(v) \geq delay(s) + \Delta(s, t) + D$, due to the induction hypothesis that $delay(s) + \Delta(s, t) + D \geq l(s) + \Delta(s, t) + D$, we have

$$delay(v) \geq delay(s) + \Delta(s, t) + D \geq l(s) + \Delta(s, t) + D \geq l(v)$$

which contradicts the assumption $delay(v) < l(v)$. (See Figure 27(c).)

As a result, the statement is also true for vertex v. □

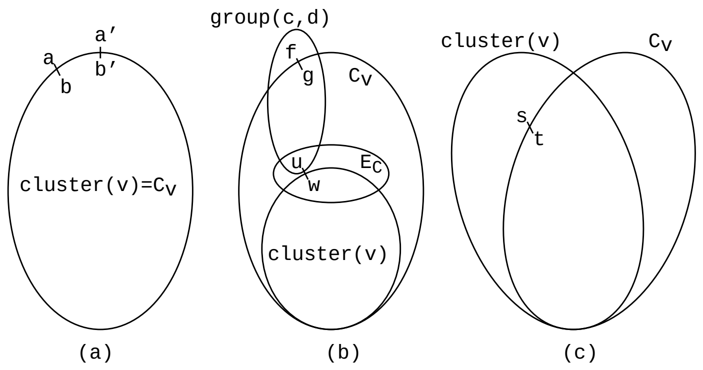


Fig. 27. Illustration of Proof of Lemma 3

Lemma 4 In our algorithm, for any vertex v in the clustering S generated by the clustering phase (lines 13-19), the path delay at v is less than or equal to $l(v)$.

Proof Our delay model is different from that in [51], but the clustering phase in our algorithm is the same as that of [51], so the proof is the same. Details can be found in [51]. □

Based on Lemma 3 and Lemma 4, we can easily derive the following theorem.

Theorem 1 The clustering S generated in our algorithm is an optimal clustering for any instance of the problem described in Section B.

Proof In Lemma 3, it is shown that for each vertex v, the label $l(v)$ in our algorithm is less than or equal to the path delay at vertex v in any optimal clustering; Lemma 4 states that our algorithm is able to generate a clustering with the path delay at v less than or equal to $l(v)$ which is the lower bound of the path delay at vertex v in any optimal clustering. Together with Lemma 1 in [51], the clustering S generated by our algorithm is an optimal clustering. $\square$

We analyze the complexity of our algorithm. In $Grouping()$, $Group\_vertex()$ would run at most $|V|$ times, so the time complexity of the WHILE loop is $O(|V||E|)$. In $Circuit\_clustering()$, finding the maximum delay matrix Δ takes $O(|V|(|V| + |E|))$, finding a topological order in line 4 takes $O(|V| + |E|)$ time, the sorting in line 11 takes time $O(|E|lg(|E|))$, and $Labeling()$ takes only $O(|E|)$ time. So, the first WHILE loop of $Circuit\_clustering()$ takes $O(|V|(|E|lg(|E|) + |V||E|))$ time. Clustering phase (lines 13-19) takes time $O(|V| + |E|)$. So the overall time complexity is $O(|V|(|E|lg(|E|) + |V||E|)) = O(|V|^2|E|)$.

Remarks: In fact, our algorithm can also handle the case where the inter-cluster delay D is a variable value (say $D(x, y), \forall(x, y) \in E$). It is because the calculation of $l''(x, y) = l(x) + D + \Delta(y, r)$ includes the value of D such that if D becomes a variable $D(x, y)$, the calculation becomes $l''(x, y) = l(x) + D(x, y) + \Delta(y, r)$ and it still correctly represents the situation when (x, y) becomes an inter-cluster edge. Besides, the optimality of the algorithm still holds because all the theoretical results remain true and can be proved similarly.

G. Conclusion

In this paper, we have introduced a new delay model which is more general and practical than the general delay model [53]. Under our new delay model, a circuit clustering algorithm based on a novel vertex grouping technique is proposed and is proved to optimally solve the area-constrained combinational circuit clustering problem for delay minimization in polynomial time.

CHAPTER VIII

MULTI-LEVEL CIRCUIT CLUSTERING FOR DELAY MINIMIZATION[*]

A. Introduction

Circuit clustering is defined as assigning circuit elements into clusters under different design constraints, such as area and pin constraints [51, 52, 53, 50]. In this way, the circuit clusters are smaller compared to the original circuit, and hence manipulation and synthesis of the clusters are easier. Most circuit clustering algorithms aim at either minimizing the circuit delay or the inter-cluster connections.

In this paper, we focus on the problem of combinational circuit clustering for delay minimization subject to area constraints. This problem is first studied in [50]. The authors formulate the problem in the unit delay model in which no delay value is associated with any connection within a cluster or any gate while unit delay is assigned to each inter-cluster connection. A polynomial time algorithm is also proposed to solve the problem optimally. Recently, most researchers adopted the general delay model [53], in which each gate is associated with a delay value, no delay is for each connection within the same cluster, and a constant delay is for each inter-cluster connection. An algorithm which solves the circuit clustering problem based on the general delay model is proposed in [51]. It is proved that the algorithm can optimally solve the problem in polynomial time. The problems considered in [51, 53, 50] are referred to as *single-level* circuit clustering.

The necessity of a solution to the *multi-level* (or *hierarchical*) circuit clustering prob-

[*]Reprinted with permission from “Multi-Level Circuit Clustering for Delay Minimization ” by C. N. Sze, T.-C. Wang and L.-C. Wang, 2004. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume: 23, Issue: 7, pp. 1073-1085. COPYRIGHT 2004 by IEEE.

lem is increasing when more and more designs are built on hierarchical FPGA architectures. The two-level clustering problem with area constraints is studied in [55]. The problem formulation requires the division of a circuit into clusters (second-level clusters) and each cluster is then further divided into smaller clusters (first-level clusters). It is proved that two-level circuit clustering for delay minimization is an NP-hard problem. Hence, they propose a heuristic which is extended from [51]. Their algorithm constructs a candidate second-level cluster rooted at each node and then covers the whole circuit based on the clusters. During the construction of each candidate second-level cluster, the first-level clusters within it are formed at the same time. Both first-level and second-level clusters are constructed according to the same criterion – nodes are chosen by comparing the maximum delay of the paths from primary inputs to the cluster root passing through them.

However, the heuristic in [55] is not effective enough according to our experiments. The main reason may be related to the restriction in which it does not allow node duplication within a second-level cluster. Node duplication within a second-level cluster has two contrasting effects on delay minimization. On one hand, it may reduce the circuit delay since a node can be included into different clusters so that the number of inter-cluster connections may be reduced. On the other hand, each cluster is constrained by an area bound and node duplication consumes area, so less different nodes can be included into a second-level cluster and then the circuit delay may increase. However, we have shown by experiment that properly allowing node duplication within second-level clusters is beneficial to delay minimization. Moreover, since the algorithm in [55] performs first-level and second-level clusterings at the same time, for each node included into a first-level cluster, all the data for first-level and second-level clusters (e.g., lists of candidate nodes and immediate successor with maximum delay) should be updated accordingly. It makes their algorithm hardly extensible to solve the circuit clustering problem with more than two levels.

In order to cope with the difficulties mentioned above, we propose an algorithm for the general combinational circuit clustering problem with any arbitrary number of levels. Our algorithm constructs clusters for each level separately, from the first level to the desired level. The clustering of each level is performed on a contracted graph which only captures the most important delay information from the clustering of the previous level. Besides, since we only perform circuit clustering on the contracted graph formed from the previous level, a simple but effective single-level graph clustering algorithm can be employed. As a result, our algorithm effectively handles the multi-level problem by repeating the single-level graph clustering algorithm and the graph contraction technique.

Although we employ a single-level clustering algorithm which is extended from [51], our overall algorithm is *not* merely a trivial extension of [51], because without our graph contraction technique, the single-level clustering algorithm cannot be repeatedly applied to the circuit to obtain a multi-level clustering. As a result, the graph contraction algorithm plays a critical role in our work and it successfully links every two successive levels of circuit clustering.

Taking the two-level clustering problem as an example, our algorithm first divides the circuit into a set of first-level clusters with node duplication, so that the node duplication within second-level clusters can be later guided by those first-level clusters. In this way, node duplication within a second-level cluster happens only when the duplication helps minimizing the delay of the resultant first-level clustering. In fact, our implementation and experimental results show that, under this mechanism, allowing node duplication within second-level clusters indeed further reduces the delay values. We are able to achieve 12% more delay reduction when comparing with the algorithm in [55], in which node duplication within second-level clusters is not allowed.

In the rest of the paper, Section 2 defines the notation and the problem formulation. In Section 3, our algorithm as well as the most essential graph contraction technique is explained in detail. Analysis of the algorithm is presented in Section 4. Section 5 discusses some postprocessing techniques to reduce the area of the clustered circuit and Section 6 describes the implementation of our algorithm and the experimental results. Finally, we conclude the paper in the last section.

B. Definitions and Problem Formulation

A combinational circuit can be represented as a directed acyclic graph (DAG) $G = (V, E)$. V is the set of nodes which represent the functional blocks (e.g., gates) in the circuit and E is the set of edges which stand for the connections among the blocks. In the graph, PIs are nodes with out-going edges only, and on the contrary, POs have in-coming edges only.

An area function $w(v)$ is defined for each node $v \in V$. The value of $w(v)$ represents the area of the corresponding functional block.

A first-level cluster $C^1 \subset V$ is a set of nodes $\{v_1, v_2, ..., v_k\}$ which satisfies the first-level area bound M_1, and a second-level cluster C^2 is a set of first-level clusters $\{C^1_1, C^1_2, ..., C^1_l\}$ which satisfies the second-level area bound M_2. More generally, an i-th-level cluster C^i is a set of $(i-1)$-th-level clusters $\{C^{i-1}_1, C^{i-1}_2, ..., C^{i-1}_r\}$ and its area bound is denoted as M_i. For each first-level (second-level) cluster, its area function is defined as the sum of area of all nodes (the sum of first-level area bounds) in the cluster. That is,

$$w(C^1) = \sum_{v \in C^1} w(v) \text{ and } w(C^2) = \sum_{C^1 \in C^2} M_1$$

In general, for each i-th-level cluster C^i, we have

$$w(C^i) = \sum_{C^{i-1} \in C^i} M_{i-1},\ i \in \{2, ..., n\}$$

where n is the desired level of circuit clustering.

In the definition of the cluster area for an i-th-level cluster C^i, all $(i-1)$-th-level clusters inside C^i are considered to have the same area M_{i-1}. So totally no more than $\frac{M_i}{M_{i-1}}$ $(i-1)$-th-level clusters can be included into one i-th-level cluster[1].

Besides area constraints, there are delay values associated with all nodes and edges. For each node $v \in V$, a delay function $\delta(v)$ is defined as a non-negative value which represents the delay of the functional block. The notation $\delta(a, b)$ represents the edge delay from node a to node b. For each edge within the same first-level cluster, it is associated with a fixed delay D_1. For each edge connecting two nodes in different first-level clusters but in the same second-level cluster, the edge delay is assigned to a fixed delay D_2. Generally, for each edge connecting two nodes in different $(i-1)$-th level clusters but in the same i-th-level cluster, the edge delay is associated with a fixed delay D_i. And, each edge, which connects two nodes between two different n-th-level clusters, has a fixed delay D_{n+1}. Practically, we have $D_1 < D_2 < D_3 < ... < D_{n+1}$ for an n-level circuit clustering.

For the delay of a path from node a to node b, we always include all node delays and edge delays along the path. The path delay at a node v is defined as the maximum delay of all paths from PIs to v. The delay of a clustered circuit is defined as the maximum path delay at all PO nodes; in other words, it is the maximum delay of all paths from PIs to POs within the clustered circuit. According to the above definitions, the multi-level circuit clustering problem with area constraints is presented in the following.

Problem (Multi-Level Circuit Clustering)

Divide the graph G into a set $S_1 = \{C_1^1, C_2^1, ..., C_{m_1}^1\}$ of first-level clusters, divide the set of all first-level clusters into a set $S_2 = \{C_1^2, C_2^2, ..., C_{m_2}^2\}$ of second-level clusters, and recursively divide all the $(i-1)$-th-level clusters into a set $S_i = \{C_1^i, C_2^i, ..., C_{m_i}^i\}$ of i-th-level clusters until a set of n-th-level

[1]It seems that our area definitions of clusters are different from [55], but in fact, the TLC implementations we obtained from the authors of [55] follow our definitions here. Details are discussed in Section F.

clusters is obtained, such that the delay of the clustered circuit is minimized. The clusters of each level may have common elements, but the clustered circuit must be logically equivalent to the original circuit. The clusters of each level should satisfy the following conditions.

$$\forall j \in \{1, ..., m_1\}, C_j^1 \subseteq V \text{ , s.t. } \begin{cases} w(C_j^1) \leq M_1, \\ \bigcup_{j=1}^{m_1} C_j^1 = V \end{cases}$$

$$\forall j \in \{1, ..., m_2\}, C_j^2 \subseteq S_1 \text{ , s.t. } \begin{cases} w(C_j^2) \leq M_2, \\ \bigcup_{j=1}^{m_2} C_j^2 = S_1 \end{cases}$$

and,

$$\forall i \in \{3, ..., n\}, \forall j \in \{1, ..., m_i\}, C_j^i \subseteq S_{i-1} \text{ ,}$$
$$\text{s.t. } \begin{cases} w(C_j^i) \leq M_i, \\ \bigcup_{j=1}^{m_i} C_j^i = S_{i-1} \end{cases}$$

An example is shown in Figure 28. In the figure, there are 17 nodes in the graph and a three-level circuit clustering is shown. In the clustering, the delay of edge (j, k) is D_4 since j and k are in different third-level clusters. The delay values of (l, q) and (i, j) are D_3 and D_2, respectively. Since e and f are in the same first-level cluster, the delay associated with (e, f) is D_1. If each node is associated with unit area and $M_1 = 2, M_2 = 6, M_3 = 12$, the area w of each first-level cluster in the clustering is 2 except for the clusters containing i,j or o, whose w equals 1. The second-level cluster containing $\{a, b, c, d, e, f\}$ has an area of 6. The area of the second-level cluster containing $\{g, h, i, j\}$ is also 6 since it consists of three first-level clusters and no more first-level cluster can be further filled into it. In fact, the area of each second-level cluster in Figure 28 is 6 except the one containing $\{k, l\}$ (whose area is 2). The area of each third-level cluster in the graph is the same, which is 12.

In the example, we assume $D_1 = 1, D_2 = 3, D_3 = 7, D_4 = 17$ and $\delta(v) =$

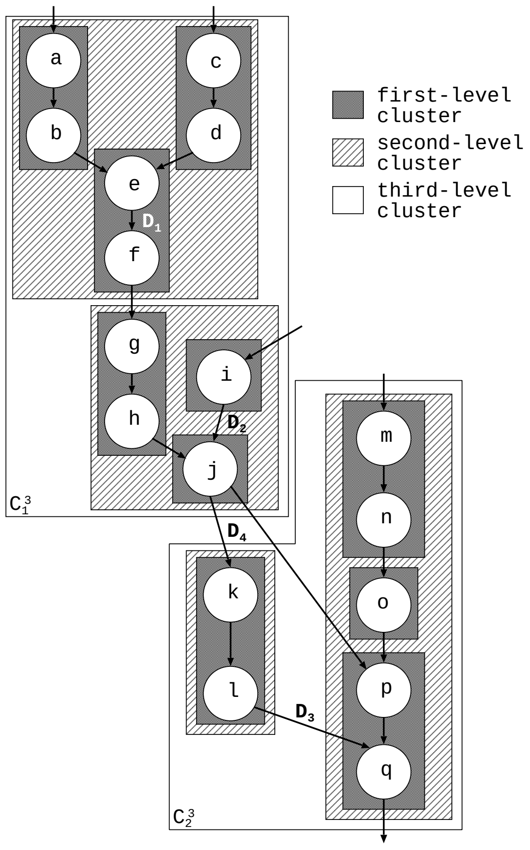


Fig. 28. An example of circuit clustering with three levels

1 for each node v. The delay of the clustered circuit is equal to 85, which is along the path $a \to b \to e \to f \to g \to h \to j \to k \to l \to q$, including the edge entering a and the one leaving q (The delays of both are D_4.).

C. The Algorithm

The flow of our algorithm, Multi-Level circuit Clustering (MLC), is depicted in Figure 29. We divide the multi-level clustering problem into two subproblems: single-level graph clustering and graph contraction. In next subsections, the subproblems and their solutions are discussed in detail. Before that, a brief overview of our algorithm and some new notations for the subproblems are introduced.

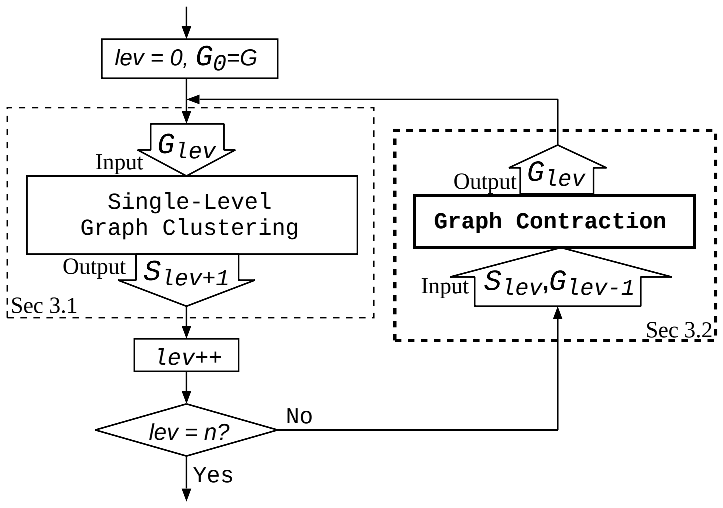


Fig. 29. The flow of our algorithm

At the beginning of our algorithm, the level index (denoted by lev) is set to 0, which means the circuit is not yet clustered. So, we have $G_0 = G$, while in $G_0 = (V_0, E_0)$, we denote the node delay $\delta_1(v)$ for each node $v \in V_0$ and edge delay $\delta_1(a, b)$ for each edge $(a, b) \in E_0$, where $\delta_1(v) = \delta(v)$ for each node $v \in V_0$ and $\delta_1(a, b) = D_1$ for each edge

$(a, b) \in E_0$.

Then, G_0 is the input to the single-level graph clustering algorithm and S_1, a first-level clustering, is returned. Then, lev is increase by 1 ($lev = 1$) to indicate that the algorithm has obtained a first-level clustering. If we want a single-level clustering (i.e., $n = 1$), it is needless to perform graph contraction step and the algorithm terminates. But if $n > 1$, G_0 and S_1 are then used to generate the contracted graph G_1. In graph contraction, we treat each first-level cluster C_i^1 as an independent supernode. In other words, G_1 is built in the way that each node represents a supernode, which in fact stands for a cluster in S_1.

The contracted graph G_1 is completely different from the original circuit G_0 because for each node in G_1, there must exist a corresponding cluster in the clustering S_1 on G_0. So, in the contracted graph $G_1 = (V_1, E_1)$, we define the node delay as $\delta_2(v)$ for each node $v \in V_1$ and the edge delay as $\delta_2(a, b)$ for each edge $(a, b) \in E_1$, which are different from the node delay and edge delay $\delta_1(.)$ defined in G_0.

In general, each time the single-level graph clustering algorithm takes the contracted graph G_{lev} as input and it outputs a set of clusters, S_{lev+1}[2]. After that, lev is increased by 1. If lev is equal to the required number of levels, n, the algorithm terminates. Otherwise, graph contraction is performed on G_{lev-1} based on S_{lev} and returns a new contracted graph $G_{lev} = (V_{lev}, E_{lev})$ (and a new set of corresponding node and edge delays with notation δ_{lev+1}). Then, single-level graph clustering is once again performed on the new graph G_{lev}, and the output is a set S_{lev+1} of clusters. The algorithm iterates until the desire level n is obtained.

The details of the single-level graph clustering and graph contraction algorithms are presented in Sections 3.1 and 3.2. The **graph contraction algorithm** is the main contribution in the paper since without graph contraction, single-level graph clustering cannot be

[2]Actually, in the subproblem of *Single-Level Graph Clustering*, we use the notation $\hat{S}_{lev+1}$ instead of S_{lev+1}. Their difference is explained in Section 1.

repeatedly applied to the circuit G.

1. Single-Level Graph Clustering

In this section, the definition of the single-level graph clustering subproblem is first presented. Then, we present our algorithm to this subproblem.

a. Definition of the Subproblem

In the subproblem, given a graph $G_{lev} = \{V_{lev}, E_{lev}\}$, it is required to construct the graph clustering $\hat{S}_{lev+1}$ which is to divide the graph G_{lev} into a set $\hat{S}_{lev+1} = \{\hat{C}_1^{lev+1}, \hat{C}_2^{lev+1}, ..., \hat{C}_{m_{lev+1}}^{lev+1}\}$ of clusters, such that the delay of the clustered graph is minimized. The delay of each node a is defined by $\delta_{lev+1}(a)$ while the edge delay is $\delta_{lev+1}(a, b)$ for each edge $(a, b) \in E_{lev}$. When the end vertices a and b of an edge are assigned into different clusters, the edge delay will become $(\delta_{lev+1}(a, b) + (D_{lev+2} - D_{lev+1}))$ instead [3]. The clusters may have common nodes, but the clustered graph must retain the predecessor-successor[4] relationship of the original graph. The clusters must also satisfy the following conditions.

$$\forall i \in \{1, ..., m_{lev+1}\}, \hat{C}_i^{lev+1} \subseteq V_{lev} \ ,$$
$$\text{s.t.} \begin{cases} w(\hat{C}_i^{lev+1}) \le \frac{M_{lev+1}}{M_{lev}} \ , \\ \bigcup_{i=1}^{m_{lev+1}} \hat{C}_i^{lev+1} = V_{lev} \ , \end{cases}$$

where $w(\hat{C}_i^{lev+1})$ is defined as the number of nodes in $\hat{C}_i^{lev+1}$ for $lev \ge 1$, $w(\hat{C}_i^1) = \sum_{v \in \hat{C}_i^1} w(v)$ and $M_0 = 1$.

Since for each time we only deal with single-level graph clustering, the definition of "$(lev + 1)$-th-level" clustering $\hat{S}_{lev+1} = \{\hat{C}_1^{lev+1}, \hat{C}_2^{lev+1}, ..., \hat{C}_{m_{lev+1}}^{lev+1}\}$ in this subproblem

[3]The reasoning of the edge delay calculation is explained in Section D.

[4]In a graph, a vertex u is a predecessor (successor) of a vertex v if there exists a path from u to v (from v to u). Similarly, a vertex u is an immediate predecessor (immediate successor) of a vertex v if there exists an edge from u to v (from v to u).

is slightly different from $S_{lev+1} = \{C_1^{lev+1}, C_2^{lev+1}, ..., C_{m_{lev+1}}^{lev+1}\}$ in Section B. For each cluster $\hat{C}_i^{lev+1} \in \hat{S}_{lev+1}$, it only contains a number of nodes (or namely supernodes) in the contracted graph G_{lev} while the corresponding $C_i^{lev+1} \in S_{lev+1}$ contains a set of lev-th-level clusters. However, S_{lev+1} can be easily converted from the corresponding $(lev+1)$-th-level clustering solution $\hat{S}_{lev+1}$ of the single-level graph clustering subproblem. The conversion is further explained in Section 3. Figure 30 shows an example. In the figure, there are two third-level clusters $\hat{C}_1^3$ and $\hat{C}_2^3$ on the “contracted graph” which is constructed from the second-level clustering of the circuit shown in Figure 28.

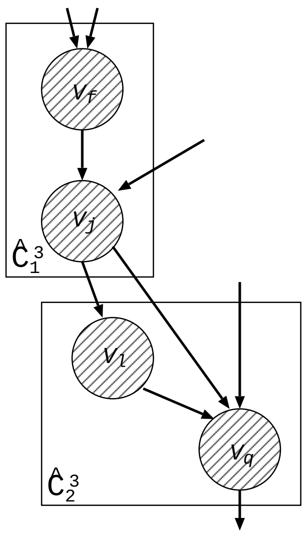


Fig. 30. The corresponding $\hat{S}_3$ of S_3 in Figure 28

b. Solution to the Subproblem

The subproblem can be solved by applying the single-level circuit clustering algorithm in [51] with modifications[5]. The delay model adopted in [51] is slightly different from the delay model that we consider in this paper. There is no delay value for each edge within the same cluster in the delay model in [51]. But, in our model, the delay of each edge within the same cluster can be any number. (Note that, $\delta_1(a, b) = D_1$ for each edge $(a, b) \in E_0$ at the beginning of the algorithm when $lev = 0$. But, after the first iteration, $lev \geq 1$, the delay $\delta_{lev+1}(a, b)$ of each edge $(a, b) \in E_{lev}$ is defined by the last graph contraction and the delay value may be different for every edge.) However, taking the first iteration with $lev = 0$ as an example, if we assume each edge has the delay D_1 during the calculation of the delay matrix Δ_1, which stores the maximum delay (including node and edge delays) of the paths between any two nodes, single-level clustering under our model can be solved optimally[6] in a similar way. The pseudo-code of the modified single-level graph clustering algorithm is shown in Figure 31.

At the first time the algorithm is invoked, the circuit is unclustered. We have $lev = 0$, $\delta_1(v) = \delta(v)$ for each node $v \in V_0$, and $\delta_1(x, y) = \delta(x, y) = D_1$ for each edge $(x, y) \in E_0$ in the graph G_0. After the first iteration ($lev \geq 1$), the values of each $\delta_{lev+1}(v)$ and each $\delta_{lev+1}(x, y)$ are determined in the last graph contraction step (discussed in the next section). Moreover, the calculation of delay matrix Δ_{lev+1}, which stores the maximum delay (including node and edge delays) of paths between any two nodes in V_{lev}, is calculated based on the values of $\delta_{lev+1}(v)$ and $\delta_{lev+1}(x, y)$ defined in the input graph G_{lev}.

Each cluster generated by the Single-Level_Graph_Clustering (SLGC) algorithm has

[5]Although the single-level circuit clustering algorithm in [56] can also be applied to our subproblem, we did not use it because of its higher computational complexity.

[6]The optimality is discussed in Section D.

```
ALGORITHM Single-Level_Graph_Clustering($G_{lev}$)
Input  : graph $G_{lev} = (V_{lev}, E_{lev})$
Output : graph clustering $\hat{S}_{lev+1} = \{\hat{C}_1^{lev+1}, \hat{C}_2^{lev+1},\ \dots\ , \hat{C}_{m_{lev+1}}^{lev+1}\}$
1.  begin
2.  compute the delay matrix $\Delta_{lev+1}$, where $\Delta_{lev+1}(i,j)$
    is the maximum delay (including node and
    edge delays) of the paths from $i$ to $j$ ;
3.  FOR each PI $i$, DO $l_{lev+1}(i) = \delta_{lev+1}(i)$;
4.  Sort the non-PI nodes of $V_{lev}$ in a topological order
    to obtain list $T$;
5.  WHILE $T$ is not empty
6.      Remove the first node $v$ from $T$;
7.      Compute $N_v$;
8.      FOR each node  $u \in N_v - \{v\}$  do
9.          $l'(u) = l_{lev+1}(u) + \Delta_{lev+1}(u,v) - \delta_{lev+1}(u);$
        END FOR
10.     $P =$ Sort the nodes in $N_v - \{v\}$ in decreasing order of $l'$
11.     Labeling($v, P$);
    END WHILE
12. $L =$ all PO nodes;
13. $S\ =\ \phi$ ;
14. WHILE $L$ is not empty
15.     Remove a node $v$ from $L$;
16.     $S\ =\ S \cup \{$cluster$(v)\}$;
17.     FOR all nodes $x \in V_{lev}-$cluster$(v)$, such that $x$ is
        an input of cluster($v$) and cluster($x$) $\notin\ S$
18.         $L = L \cup \{x\}$;
        END FOR
    END WHILE
19. return $\hat{S}_{lev+1} = S$;
20. end

Labeling($v, P$)
Input  : node $v$, list $P$
Output : $l_{lev+1}(v),$ cluster$(v)$
1.  begin
2.  cluster($v$)=$\{v\}$;
3.  WHILE ($P$ is not empty)
4.      Remove the first node $u$ in $P$;
5.      IF($w($cluster$(v) \cup \{u\})\ \ \leq\ \ \frac{M_{lev+1}}{M_{lev}}$)
6.          cluster$(v)$ =cluster$(v) \cup \{u\}$;
7.      ELSE
8.          break;
        END IF
    END WHILE
9.  $l_{lev+1}^1(v) =$ max$\{l'(x) | x \in$cluster$(v) \cap (PI)\}$;
10. $l_{lev+1}^2(v) =$ max$\{l'(u) + (D_{lev+2} - D_{lev+1}) | u \in P\}$;
11. $l_{lev+1}(v) =$ max$\{l_{lev+1}^1(v), l_{lev+1}^2(v)\}$;
12. end
```

Fig. 31. The pseudo-code of our single-level graph clustering algorithm

one and only one root vertex v and we denote the cluster rooted at v as $cluster(v)$[7]. Let N_v be the set of all predecessors of v together with the vertex v. Note that, $cluster(v)$ is a subset of N_v. When the context is not ambiguous, we also denote the subgraph induced by N_v by the vertex set N_v.

The algorithm consists of two phases: labeling phase and clustering phase. For each vertex $v \in V_{lev}$, the label of v, $l_{lev+1}(v)$, is defined as the minimum path delay at v among all possible clusterings on the subgraph N_v. In the labeling phase (lines 2-11 of SLGC), for each vertex v in a topological order, the algorithm finds $cluster(v)$ from N_v such that it would make the path delay at v become the minimum among all possible clusterings on N_v, and at the same time, the algorithm obtains $l_{lev+1}(v)$.

In order to get $l_{lev+1}(v)$, the algorithm first calculates $l'(u)$ locally for each predecessor u of v (lines 8-9 of SLGC). $l'(u)$ is defined as the sum of $l_{lev+1}(u)$ and the maximum delay of the paths from the output of u to v in the input graph[8]. Then, a vertex with the highest l' value is repeatedly picked and included into the cluster rooted at v until the cluster area violates the area constraint (lines 3-8 of Labeling). Note that the cluster area $w(.)$ in line 5 is defined the same as that in Section a.

After finding the $cluster(v)$, the algorithm continues to calculate l^1_{lev+1}, the maximum l' value of the PI vertices (denoted by the set PI) which are inside the cluster, and to find l^2_{lev+1}, the maximum $l' + (D_{lev+2} - D_{lev+1})$ value of the vertices outside the cluster. After that, the label of v, $l_{lev+1}(v)$, can be found by getting the greater value of l^1_{lev+1} and l^2_{lev+1} (lines 9-11 of Labeling).

The clustering phase (lines 12-19 of SLGC) constructs clusters from POs to PIs according to the cluster information generated in the labeling phase. First, for each PO vertex

[7]All vertices in the cluster are predecessors of the root vertex.

[8]When calculating the delay of a path from the output of u to v, the vertex delay of u is not included.

v, the corresponding $cluster(v)$ is included into the clustering S. Then, for each vertex x outside S, which is an immediate predecessor of any vertex inside S, $cluster(x)$ is also included into the clustering S. The procedure is repeated until all vertices in G_{lev} are included in S. Finally, S is returned and stored as $\hat{S}_{lev+1}$.

As an example, a sample circuit is shown in Figure 32, while all PI nodes and PO nodes are not included in the figure. In this example, we assume $M_1 = 3, M_2 = 6, D_1 = 1, D_2 = 5, D_3 = 10, \delta_1(v) = 1, w(v) = 1$ for each node v. For this sample circuit, we perform the SLGC algorithm and the resultant first-level clustering result is depicted in Figure 33. As shown in the figure, node duplication is allowed between first-level clusters so as to reduce the delay of the resultant circuit. For example, node f is duplicated and located in two clusters $\hat{C}_f^1$ and $\hat{C}_i^1$. In the first-level clustering $\hat{S}_1 = \{\hat{C}_b^1, \hat{C}_c^1, \hat{C}_d^1, \hat{C}_e^1, \hat{C}_f^1, \hat{C}_g^1, \hat{C}_i^1, \hat{C}_j^1, \hat{C}_k^1, \hat{C}_n^1\}$, the first-level clusters are named according to the root node of each cluster for legibility.

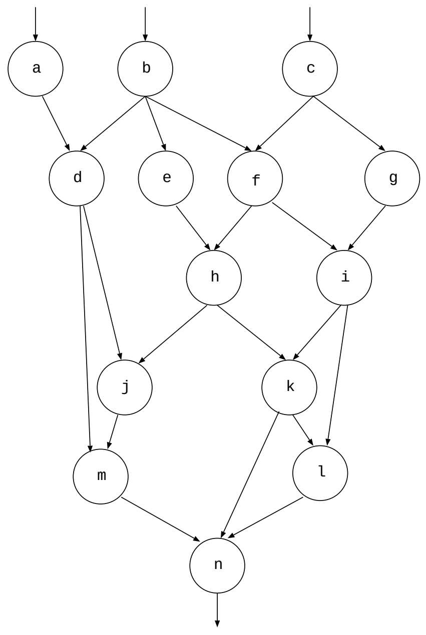


Fig. 32. A sample circuit G_0

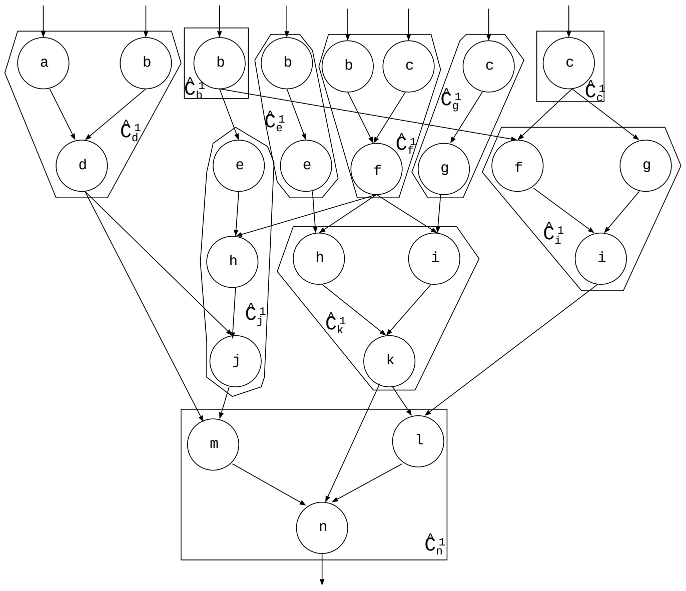


Fig. 33. First-level clustering $\hat{S}_1$ for Figure 32

2. Graph Contraction

The problem definition of the graph contraction subproblem and our solution are presented in this section.

a. Definition of the Subproblem

After constructing the set $\hat{S}_{lev}$ of lev-th-level clusters, the second subproblem is to build a contracted graph $G_{lev} = \{V_{lev} = \{v_1, v_2, ..., v_{m_{lev}}\}, E_{lev}\}$ from the graph G_{lev-1} and the clustering $\hat{S}_{lev} = \{\hat{C}_1^{lev}, \hat{C}_2^{lev}, ..., \hat{C}_{m_{lev}}^{lev}\}$, and at the same time, assign node delay $\delta_{lev+1}(v_s)$ to each node $v_s \in V_{lev}$ and edge delay $\delta_{lev+1}(v_j, v_k)$ to each edge $(v_j, v_k) \in E_{lev}$, such that each node v_i in G_{lev} corresponds to a lev-th-level cluster $\hat{C}_i^{lev}$ in $\hat{S}_{lev}$ on the graph G_{lev-1} and its path delay in G_{lev} is the same as the path delay at the root node r_i of $\hat{C}_i^{lev}$ in the clustered graph of G_{lev-1}, in other words, the label $l_{lev}(r_i)$ of r_i.

b. Solution to the Subproblem

In the subproblem, since clusters may have different numbers of nodes and subgraph structures, and there may be several edges connecting nodes inside two different clusters, delay assignments to each supernode and each edge connecting the supernodes are not straightforward. The general algorithm for constructing G_{lev} from graph G_{lev-1} and clustering $\hat{S}_{lev}$ is shown in Figure 34.

```
ALGORITHM Graph_Contraction($G_{lev-1}$, $\hat{S}_{lev}$)
Input  : Graph $G_{lev-1}$, Clustering $\hat{S}_{lev}$=$\{\hat{C}_1^{lev}, \hat{C}_2^{lev}, ..., \hat{C}_{m_{lev}}^{lev}\}$
Output : $G_{lev}$
1.  begin
2.  FOR each cluster $\hat{C}_i^{lev}$ in $\hat{S}_{lev}$
3.     construct a node $v_i$ in $G_{lev}$
4.     $\delta_{lev+1}(v_i) = D(\hat{C}_i^{lev})$;
       /* $\delta_{lev+1}(v_i)$ is assigned in $G_{lev}$ */
    END FOR
5.  FOR each edge $e = (a, b)$ in $G_{lev-1}$
6.     IF ($a$ and $b$ are not in the same cluster of $\hat{S}_{lev}$)
7.        assume $b$ in $\hat{C}_i^{lev}$ which is rooted at $r$;
8.        find the cluster $\hat{C}_j^{lev}$ rooted at $a$;
9.        IF (there exists no edge from $v_j$ to $v_i$ in $G_{lev}$)
10.          add an edge from $v_j$ to $v_i$ in $G_{lev}$;
11.          $\delta_{lev+1}(v_j, v_i) = \delta_{lev}(a, b) + (D_{lev+1} - D_{lev})$
                       $-\delta_{lev+1}(v_i) + \Delta_{lev}(b, r)$;
             /* Note that $\delta_{lev+1}(v_j, v_i)$ is defined in $G_{lev}$ */
             /* and $\delta_{lev}(a, b)$ is defined in $G_{lev-1}$ */
12.       ELSE IF (there exists an edge from $v_j$ to $v_i$ in
             $G_{lev}$)
13.          IF ($\delta_{lev+1}(v_j, v_i) < \delta_{lev}(a, b) + (D_{lev+1} - D_{lev})$
                       $-\delta_{lev+1}(v_i) + \Delta_{lev}(b, r)$)
14.             $\delta_{lev+1}(v_j, v_i) = \delta_{lev}(a, b) + (D_{lev+1} - D_{lev})$
                       $-\delta_{lev+1}(v_i) + \Delta_{lev}(b, r)$;
             END IF
          END IF
       END IF
    END FOR
15. return $G_{lev}$;
16. end
```

Fig. 34. The pseudo-code of our graph contraction algorithm

In the first FOR loop (lines 2-4), each new node (or namely supernodes) v_i in G_{lev}

is first created in such a way that v_i is corresponding to a cluster $\hat{C}_i^{lev}$ in $\hat{S}_{lev}$. The delay $\delta_{lev+1}(v_i)$ of each new node v_i is set to $D(\hat{C}_i^{lev})$, which is defined as the maximum delay of paths from any node within $\hat{C}_i^{lev}$ to the root of the cluster, in order to keep the maximum delay values of the clusters in the new graph.

The second FOR loop (lines 5-14) is to create edges between the supernodes and to calculate the delay value $\delta_{lev+1}(v_j, v_i)$ for each edge (v_j, v_i) in G_{lev}. For each inter-cluster edge induced by the original graph G_{lev-1} and $\hat{S}_{lev}$, we build an edge between the two corresponding supernodes and assign the delay value such that the maximum delay of the paths passing through that inter-cluster edge is maintained. This is the most complicated part for the graph contraction algorithm since different edges may connect different nodes inside two clusters. Our work successfully accomplishes the delay assignment to each edge (lines 9-14) which is depicted in Figure 35. (Note that the notation $\delta_{lev}(a, b)$ represents the edge delay between nodes a and b in graph G_{lev-1}. At the first time graph contraction is performed, when a and b are nodes in the original circuit $G_0(= G)$ and $\hat{S}_1(= S_1)$ is a first-level clustering, $\delta_1(a, b)$ is set to D_1 for each edge (a, b) in G_0. For the case where a and b are nodes generated by the previous graph contraction step, all corresponding $\delta_{lev}(a, b)$ values are also assigned at the same time.) In Figure 35, the “height” of a node, an edge, or a cluster represents its delay. It is obvious that $x + \delta_{lev+1}(v_i) = \delta_{lev}(a, b) + \Delta_{lev}(b, r)$, so we have $x = \delta_{lev}(a, b) - \delta_{lev+1}(v_i) + \Delta_{lev}(b, r)$. Since the edge between a and b becomes an inter-cluster edge connecting two lev-th-level clusters (originally it connects two $(lev-1)$-th-level clusters), its delay value has to be adjusted such that $\delta_{lev+1}(v_j, v_i) = x + D_{lev+1} - D_{lev}$ which leads to the equation in line 11 and line 14 of *Graph_Contraction*.

Finally, for every two supernodes, if there exists more than one edge linking them, we only keep the edge with maximum delay value and remove all the others in the new graph G_{lev}. Note that in the clustering generated by our single-level graph clustering algorithm mentioned in Section b, inter-cluster edges only connect from the roots of the predecessor clusters.

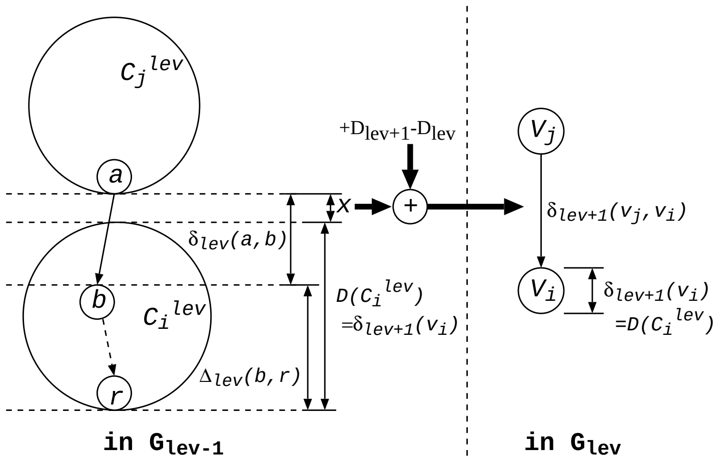


Fig. 35. Illustration of edge delay calculation in $Graph\_Contraction$

We describe in Section D that in this graph contraction algorithm, some crucial delay information of each new node $v_i \in V_{lev}$ is extracted from the root of the corresponding cluster $\hat{C}_i^{lev}$. Since this delay information is retained in the contracted graph, finding the next higher-level clustering from the contracted graph for delay minimization can be achieved by the SLGC algorithm in polynomial time.

For the first-level clustering $\hat{S}_1$ in Figure 33 and G_0 in Figure 32, the contracted graph G_1 is shown in Figure 36 as an example. Again, we assume $M_1 = 3, M_2 = 6, D_1 = 1, D_2 = 5, D_3 = 10, \delta_1(v_i) = 1, w(v_i) = 1$ for each node v_i. Each cluster in $\hat{S}_1$ becomes a node in G_1. For example, v_k in G_1 represents $\hat{C}_k^1$ in $\hat{S}_1$. The delay value $\delta_2(v)$ assigned to each node v in G_1 is calculated from $D(C)$, the maximum delay among all paths within the corresponding cluster C. For example, $\delta_2(v_j) = D(\hat{C}_j^1) = 5$ along the path $e \rightarrow h \rightarrow j$ and $\delta_2(v_n) = D(\hat{C}_n^1) = 3$ along the path $m \rightarrow n$. The edges in G_1 are also constructed accordingly. For example, between clusters $\hat{C}_k^1$ and $\hat{C}_n^1$, there are two edges $\{(k,n),(k,l)\}$. Then, we construct an corresponding edge (v_k, v_n) in G_1. The edge delay $\delta_2(v_k, v_n)$ is

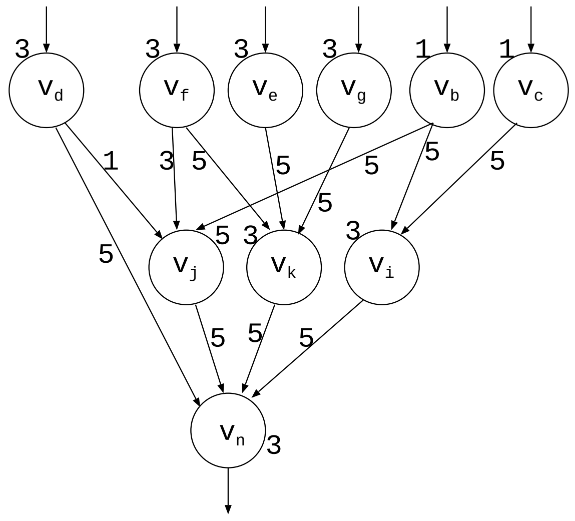


Fig. 36. The contracted graph G_1 constructed from $\hat{S}_1$ and G_0

assigned by comparing the two edges $\{(k,n),(k,l)\}$. Figure 37 shows the calculation of $\delta_1(k,l)-\delta_2(v_n)+\Delta_1(l,n)+(D_2-D_1)=1-3+3+5-1=5$. Similarly, $\delta_1(k,n)-\delta_2(v_n)+\Delta_1(n,n)+(D_2-D_1)=1-3+1+5-1=3$. According to the above comparison, we assign the edge delay $\delta_2(v_k,v_n)=\delta_1(k,l)-\delta_2(v_n)+\Delta_1(l,n)+(D_2-D_1)=5$. Note that, in this example, we use $\delta_1(k,l)$ from the circuit (the original graph G_0) in Figure 32 to calculate $\delta_2(v_k,v_n)$ for the contracted graph G_1.

Moreover, the path delay at node n, which equals $l_1(n)$, in the clustered graph of G_0 in Figure 33 is 19 which is the same as the path delay at v_n in G_1. This example shows that our graph contraction algorithm can retain the crucial delay information in the contracted graph by keeping the path delay at a node v_i in G_{lev} the same as that at the root node of the corresponding cluster $\hat{C}_i^{lev}$ in the clustered graph of G_{lev-1}.

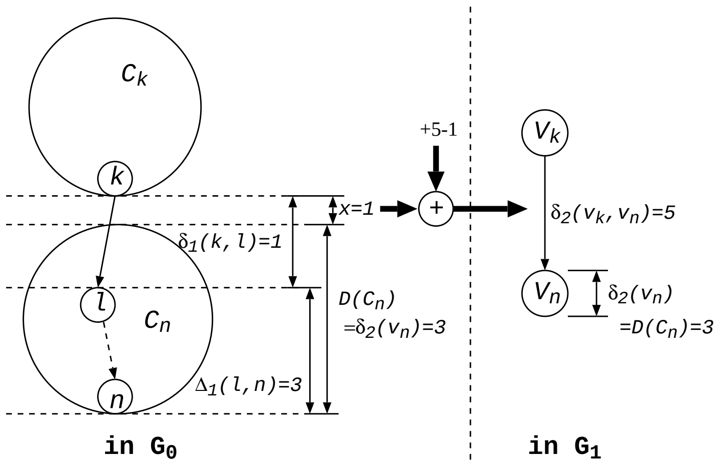


Fig. 37. Example of delay calculation of the new edge (v_k, v_n) in G_1

3. Remarks

A multi-level clustering is achieved when we iteratively perform the single-level graph clustering and graph contraction to get the information of all clusterings $\hat{S}_1, \hat{S}_2, ..., \hat{S}_n$. However, due to limited space, we only demonstrates the generation of 2-level clustering in this section as an example.

After the contracted graph is constructed based on the first-level clustering $\hat{S}_1$ (it equals S_1 because the first-level circuit clustering on G is the same as the single-level graph clustering on G_0), a second-level clustering $\hat{S}_2$ can be found from the contracted graph G_1 by the single-level graph clustering algorithm in Section b with $lev = 1$. This time, the algorithm takes $D_3 - D_2$ for the calculation of label l_2^2 value (in line 10 of "Labeling"), $\frac{M_2}{M_1}$ for area bound (in line 5 of "Labeling"), and edge and node delays, δ_2, in G_1 step for the calculation of Δ_2. The resultant second-level clustering of G_0 in Figure 32, obtained by performing the SLGC algorithm on G_1 in Figure 36, is shown in Figure 38. In the example, we add $(D_3 - D_2) = 5$ to the delay of each edge connecting two second-level clusters when calculating the delay of the clustered circuit.

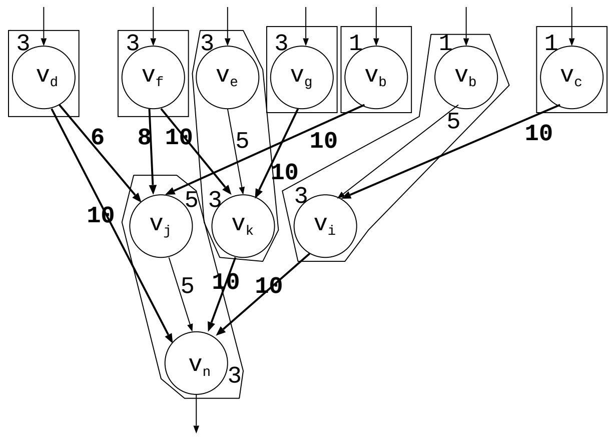


Fig. 38. Second-level clustering $\hat{S}_2$ on G_1

As mentioned previously, the definition of lev-th-level clusters in the subproblem is slightly different from that in Section B. A lev-th-level cluster C_i^{lev} in the multi-level clustering problem contains several $(lev - 1)$-th-level *clusters* while a lev-th-level cluster $\hat{C}_i^{lev}$ in the single-level graph clustering subproblem contains several *nodes* in the contracted graph G_{lev-1}. However, the conversion is straightforward since each vertex in the contracted graph G_{lev-1} represents a $(lev-1)$-th-level cluster in the $(lev-1)$-th-level clustering. For example, in Figure 38, the second-level cluster containing two supernodes v_e and v_k actually consists of two first-level clusters $\hat{C}_e^1$ and $\hat{C}_k^1$ shown in Figure 33. At the same time, this second-level cluster contains 5 nodes $\{b, e, h, i, k\}$ in V. The conversion of this second-level cluster is depicted in Figure 39.

After the second-level clustering $\hat{S}_2$ (or in general, the i-th-level clustering $\hat{S}_i$) is obtained, a contracted graph G_2 (G_i) is constructed. The third-level clustering $\hat{S}_3$ ($(i+1)$-th-level clustering $\hat{S}_{i+1}$) can then be generated similarly. In conclusion, our overall algorithm can be easily employed recursively to get an n-level circuit clustering.

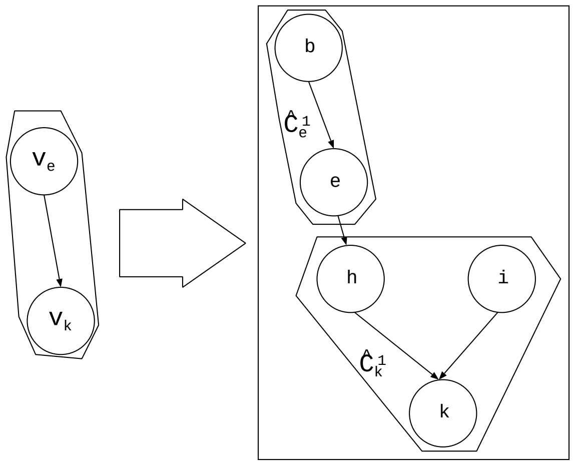


Fig. 39. A simple example about conversion from $\hat{C}_i^{lev}$ to C_i^{lev}

D. Analysis of the Algorithm

In our algorithm, the contracted graph G_{lev} is constructed such that the path delay at v_i in G_{lev} equals the label value $l_{lev}(r_i)$ of the root node r_i of the corresponding cluster $\hat{C}_i^{lev}$ in the clustered graph of G_{lev-1}. We discuss the correctness of our graph contraction algorithm in Theorem V-1. Before that, a lemma has to be first stated.

Lemma V-1 The contracted graph $G_{lev} = \{V_{lev}, E_{lev}\}$ generated by the algorithm *Graph_Contraction* is acyclic if the input graph G_{lev-1} is acyclic.

Proof It is proved by the contrapositive.

Assume the contracted graph G_{lev} is cyclic. There exists a cycle in G_{lev} and without loss of generosity, we let the cycle pass along the path $v_1, v_2, ..., v_n, v_1$ such that each vertex appears once along the path except v_1. We denote the corresponding clusters as $\hat{C}_1^{lev}, \hat{C}_2^{lev}, ..., \hat{C}_n^{lev}, \hat{C}_1^{lev}$, and the root node of those clusters as $r_1, r_2, ..., r_n, r_1$. Since all inter-cluster edges are incident from a root node, there must exist a cycle in the graph G_{lev-1} passing along the path $r_1, ..., r_2, ..., r_n, ..., r_1$. This proves that if G_{lev} is cyclic, then path delay at v_i in G_{lev}

$$\left.\begin{array}{rl} = & \max_{v_j \in V_{lev}, (v_j, v_i) \in E_{lev}} \left\{ \text{path delay at } v_j \text{ in } G_{lev} + \delta_{lev+1}(v_j, v_i) + D(\hat{C}_i^{lev}) \right\} \\ = & \max_{r_j \in V_{lev-1}, (r_j, b) \in E_{lev-1}, b \in \hat{C}_i^{lev}} \{ l_{lev}(r_j) + \delta_{lev}(r_j, b) \\ & \qquad + \Delta_{lev}(b, r_i) + (D_{lev+1} - D_{lev}) \} \\ = & l_{lev}(r_i) \end{array}\right] \quad (8.1)$$

G_{lev-1} is also cyclic, which is the contrapositive of Lemma V-1. □

Theorem V-1 For any node v_i in the contracted graph $G_{lev} = \{V_{lev}, E_{lev}\}$ generated by the algorithm *Graph_Contraction*, the path delay at v_i in G_{lev} is equal to the label, $l_{lev}(r_i)$, of the root node r_i of the corresponding cluster $\hat{C}_i^{lev} \in \hat{S}_{lev}$.

Proof From Lemma V-1, we know that the contracted graph in each level is a directed acyclic graph, so the theorem can be proved by induction.

(1. Induction basis) For any PI node v_i in G_{lev}, in the corresponding cluster $\hat{C}_i^{lev} \in \hat{S}_{lev}$, all nodes of $N_{r_i} \cap PI$ must be also within the cluster $\hat{C}_i^{lev} \in \hat{S}_{lev}$ since $\hat{C}_i^{lev} = N_{r_i}$. So $D(\hat{C}_i^{lev})$ is equal to the maximum delay from any node in $N_{r_i} \cap PI$ to r_i. In other words, the path delay at v_i in G_{lev} ($= \delta_{lev+1}(v_i) = D(\hat{C}_i^{lev})$) is equal to the label, $l_{lev}(r_i)$, of the root node r_i of the corresponding cluster $\hat{C}_i^{lev} \in \hat{S}_{lev}$.

(2. Induction step) Assume that the statement is true for all nodes $v_j \in V_{lev}$ (corresponding to clusters $\hat{C}_j^{lev}$ rooted at r_j) which are immediate predecessors of node $v_i \in V_{lev}$ (corresponding to cluster $\hat{C}_i^{lev}$ rooted at r_i), we are going to prove that the statement is also true for node v_i.

According to Equation (8.1), path delay at v_i in G_{lev} is equal to $l_{lev}(r_i)$.

As a result, the statement is also true for node v_i. □

An example of Theorem V-1 is that in Figure 33, the label, $l_1(k)$, of k is 16 ($= D_2 + \delta_1(b) + D_1 + \delta_1(f) + D_2 + \delta_1(h) + D_1 + \delta_1(k)$ on the path "a PI node"$\rightarrow b \rightarrow f \rightarrow h \rightarrow k$, assuming all PI nodes are not associated with any gate delay), which is exactly the same as the path delay at v_k ($=$ 16 on the path "a PI node"$\rightarrow v_f \rightarrow v_k$) in Figure 36.

Based on Theorem V-1, it is trivial that two observations can be made to describe the relationship between $\hat{S}_{lev}$ and S_{lev} in the algorithm.

Consider two sets: (1) the set $\hat{S}_{lev+1} = \{\hat{C}_1^{lev+1}, \hat{C}_2^{lev+1}, ..., \hat{C}_{m_{lev+1}}^{lev+1}\}$ which is a single-level graph clustering on the contracted graph $G_{lev} = \{V_{lev} = \{v_1^{lev}, v_2^{lev}, ..., v_{m_{lev}}^{lev}\}, E_{lev}\}$, and (2) the set $S_{lev+1} = \{C_1^{lev+1}, C_2^{lev+1}, ..., C_{m_{lev+1}}^{lev+1}\}$ which is a $(lev+1)$-th-level clustering on the lev-th-level clustered circuit of G (with all lev-th-level clusters denoted by the set $S_{lev} = \{C_1^{lev}, C_2^{lev}, ..., C_{m_{lev}}^{lev}\}$),

Observation V-1: The path delays at the root nodes of two corresponding clusters (e.g., $\hat{C}_i^{lev+1}$ and C_i^{lev+1}) are equal, where the root node of the cluster C_i^{lev+1} in the set S_{lev+1} is defined as a node n_x in the unclustered circuit G such that n_x is covered by C_i^{lev+1} and none of its immediate successors is covered by C_i^{lev+1}.

Observation V-2: The delays of the clustered graph induced by $\hat{S}_{lev+1}$ and the clustered circuit induced by S_{lev+1} are the same.

According to Theorem V-1, these observations reveal that performing single-level graph clustering on the lev-th-level contracted graph is the same as performing $(lev+1)$-th-level clustering based on the corresponding lev-th-level clustering and it allows the single-level graph clustering algorithm to be repeatedly applied to obtain the clustered circuit to any level n with correct delay information.

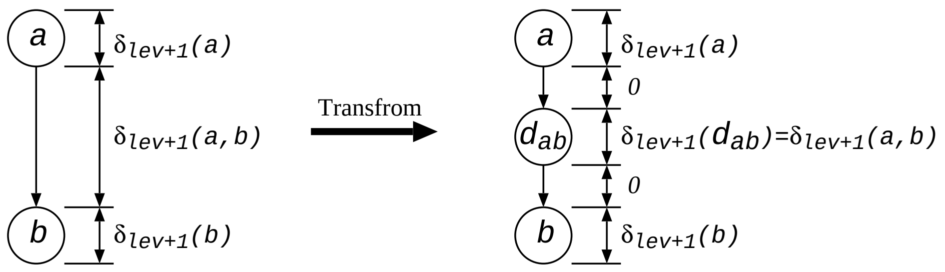
Then, we further derive the *local optimality* of our algorithm in Theorem V-2.

Theorem V-2 Given a contracted graph $G_{lev} = \{V_{lev}, E_{lev}\}$ generated by the graph contraction algorithm, our single-level graph clustering algorithm generates a graph clustering, $\hat{S}_{lev+1} = \{\hat{C}_1^{lev+1}, \hat{C}_2^{lev+1}, ..., \hat{C}_{m_{lev+1}}^{lev+1}\}$, which minimizes the delay of the resultant clustered graph.

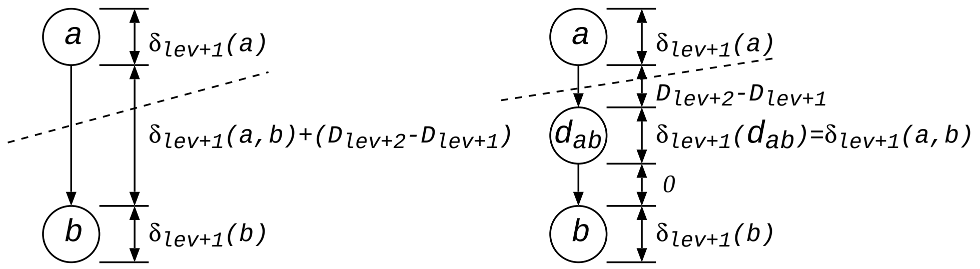
Proof From Lemma V-1, it is obvious that the contracted graph in each level is a directed acyclic graph. The optimality of our algorithm is based on the optimality of single-level circuit clustering algorithm in [51] because they are structurally similar. The main difference between the problem definition in [51] and our single-level graph clustering subproblem is the delay model. There is no delay associated with each interconnect linking two gates in the same cluster in general delay model [51] while in our problem, every edge $(a, b) \in E_{lev}$ linking two nodes in the same cluster would have a delay value $\delta_{lev+1}(a, b)$. When the end vertices a and b are assigned into different clusters, the edge delay will become $(\delta_{lev+1}(a, b) + (D_{lev+2} - D_{lev+1}))$ instead. As mentioned in [51], our delay model can be transformed to the general delay model by adding a dummy node (namely d_{ab}) with zero area and delay $= \delta_{lev+1}(d_{ab}) = \delta_{lev+1}(a, b)$ for each edge $(a, b) \in E_{lev}$. The transformation of an edge (a, b) is depicted in Figure 40a. For the case when a and b are assigned in different clusters, Figure 40b shows that the delay between a and b retains the same after transformation.

In our single-level graph clustering algorithm, we first calculate the delay matrix Δ_{lev+1}, which stores the maximum delay (including node and edge delays) of paths between any two nodes in V_{lev}. Since we use Δ_{lev+1} to store the delay of paths in G_{lev} for all the delay calculation throughout the algorithm, the calculation of this delay matrix has the same function as the transformation shown in Figure 40. As a result, the optimality of algorithm in [51] can also be applied to our single-level graph clustering algorithm. $\square$

Based on Theorem V-1, Theorem V-2, Observation V-1 and Observation V-2, it can be shown that our algorithm generates an optimal $(lev + 1)$-th-level circuit clustering S_{lev+1} on a circuit G if the lev-th-level clustering S_{lev} is given. The local optimality described in Theorem V-2 does not guarantee a final globally optimal solution but the local optimality of each recursive step tends to maintain the circuit with a small delay value. In fact, our experiments have shown that the delay reduction achieved by our algorithm is much better than the state-of-the-art algorithms.



a) when a and b are in the same cluster


a
b
$\delta_{lev+1}(a)$
$\delta_{lev+1}(a,b)+(D_{lev+2}-D_{lev+1})$
$\delta_{lev+1}(b)$
a
d_{ab}
b
$D_{lev+2}-D_{lev+1}$
$\delta_{lev+1}(d_{ab})=\delta_{lev+1}(a,b)$
0

b) when a and b are in the different clusters

Fig. 40. Transformation to general delay model

For the time complexity of the algorithm, the modified single-level circuit clustering algorithm takes $O(|V|^2 log(|V|) + |V||E|)$ time [51], and each graph contraction takes $O(|V| + |E|)$ time, where V and E are the node and edge sets of a given graph respectively. And for each graph contraction, the number of edges and number of vertices both are non-increasing. So for each level, the complexity of single-level graph clustering algorithm is still bounded above by $O(|V|^2 log(|V|) + |V||E|)$. Therefore, the overall time complexity of our algorithm is $O(n|V|(|V|log(|V|) + |E|))$ for the multi-level circuit clustering problem with n levels.

E. Postprocessing Techniques

In the experiment, our algorithm generates large clustered circuits since node duplications occur frequently in order to minimize the circuit delay. So, we present two simple post-processing techniques to reduce the area of a clustered circuit while the circuit delay is unchanged. The techniques are employed after the clustered circuit is generated, and they do not increase the delay of the clustered circuit. Assuming we have an n-level clustered circuit, the first technique locates those n-th-level clusters each of which is a proper subset of another n-th-level cluster and so they can be deleted without changing the delay and functionality of the whole circuit.

The second technique packs several n-th-level clusters into one single cluster if the area constraint is not violated. This can be done when the original clusters are small. The technique is based on the First Fit Decreasing method for the bin packing problem (which is also mentioned in [53]). All the n-th-level clusters are sorted in the non-increasing order of the area. We assume each bin has the capacity of M_n. Then, it starts to place clusters one by one into the bins. Each time we place a cluster in the leftmost bin that still has enough space for it, and start a new bin if necessary.

F. Implementation and Experimental Results

We test our algorithm upon a two-level hierarchy which is based on Altera's APEX FPGA architecture [55] and compare our algorithm (namely MLC) to the UCLA TLC implementations which are obtained from the authors of [55]. Based on the timing extraction in [55], we use the same parameters, i.e., $M_1 = 10$, $M_2 = 160$, $D_1 = 0.36ns$, $D_2 = 0.85ns$, $D_3 = 1.57ns$, $\delta(v) = 0.61ns$, $w(v) = 1$ for each node v (refer to [55] for the details of timing extraction). We evaluated our algorithm on this 2-level hierarchy mainly because this architecture is the latest FPGA model for the multi-level circuit clustering problem.

Experiments are performed on MCNC benchmark circuits which are also used by UCLA TLC [55]. The benchmarks are pre-processed and mapped into 4-input LUT networks by UC Berkeley SIS and UCLA RASP systems. Each benchmark circuit is clustered into a two-level clustering by the two TLC implementations (No node duplication and Full node duplication) and our algorithm. The first TLC implementation does not allow any node duplication among different second-level clusters while the latter one does. However, both TLC implementations do not allow node duplication within a second-level cluster. Both TLC implementations return the clustering information while the one with node duplication also returns a new circuit which is functionally equivalent to the original circuit.

In order to carry out a fair and objective comparison, our implementation strictly follows the same problem formulation as the TLC implementations in [55]. First, in our implementation, each PI node or PO node forms a second-level cluster by itself and it is excluded from any cluster rooted at any other node which is neither PI node nor PO node. As a result, the edge delay from each PI node to any of its immediate successors is always D_3, while similarly the same delay D_3 is always associated with the edge from any node to a PO node. Secondly, our problem formulation and the problem formulation in [55] seem to be different in the area calculation of second-level clusters. We limit the total number of first-level clusters within a second-level cluster while [55] limits the total number of nodes in a second-level cluster. So, if some first-level clusters contain fewer nodes, more first-level clusters can be included into a second-level cluster in the problem formulation of [55]; While in our problem formulation, the maximum number of first-level clusters within a second-level cluster is always fixed. However, our problem formulation is more appropriate to the APEX FPGA architecture where one MegaLAB (=second level cluster) can hold no more than 16 logic array blocks, LABs (=first level clusters), even when some of the LABs are not full. In fact, we find that the TLC implementations we obtain from the authors of [55] follow our definition on the second-level cluster area bound, so this ensures a fair comparison.

In our implementation, we have also imposed a constraint on the maximum number of inputs for each first-level cluster. In the specification of Altera APEX FPGA devices, a first-level cluster (LAB) cannot have more than 22 inputs. Hence, we control the number of inputs to a first-level cluster by adding a condition in the IF statement in line 5 of "Labeling" in Figure 31, such that we stop adding new nodes into each first-level cluster when the number of inputs to the cluster is more than 22. This constraint is also considered in the UCLA TLC implementations.

The experimental results are shown in Table XVI. Columns 2-4 show the results of TLC with no node duplication (No ND). Columns 5-8 show the best results of TLC in which node duplication is allowed among second-level clusters (Full ND). Columns 9-13 list the results of our algorithm. For the "delay" columns, they represent the delays of the clustered circuits in ns. "% de" columns list the percentage of delay reduced when comparing to the TLC (No ND) implementation. "CPU" columns show the CPU time (in second) consumed by each implementation on SUN Ultra4 workstations. "area" columns record the numbers of second-level clusters which reflect the total area in each clustered circuit. Moreover, "area-p" column shows the number of second-level clusters in each clustered circuit after applying our postprocessing techniques. Note that the clusters containing only a PI node or a PO node are not counted in the calculation of “area” and “area-p” in all implementations.

Table XVI. Comparison between two TLC implementations and our algorithm

	UCLA TLC (No ND)			UCLA TLC (Full ND)				Our Algorithm MLC					1-level	
Circuits	delay	CPU	area	delay	%de	CPU	area	delay	%de	CPU	area	area-p	delay	%di
alu	23.38	0.26	22	21.22	9.2	0.34	59	16.61	29.0	1.13	168	74	15.89	4.5
apex2	17.08	0.91	141	14.92	12.6	1.97	181	12.73	25.5	6.96	311	59	12.01	6.0
apex6	10.79	0.23	99	10.56	2.1	0.34	99	9.09	15.8	0.37	104	14	9.09	0.0
C1908	15.66	0.22	25	15.43	1.5	0.26	25	13.21	15.6	0.30	93	43	12.49	5.8
C5315	16.38	1.50	109	14.45	11.8	1.42	136	13.70	16.4	1.79	206	67	13.46	1.8
C880	18.32	0.14	26	16.38	10.6	0.13	26	15.40	15.9	0.24	63	30	15.40	0.0
dalu	12.72	0.25	16	11.28	11.3	0.35	16	9.81	22.9	0.71	83	24	9.58	2.4
des	12.72	2.06	245	10.56	17.0	5.32	245	10.30	19.0	14.97	610	142	9.58	7.5
i10	23.89	1.50	230	22.45	6.0	3.85	288	18.07	24.4	9.69	521	217	17.35	4.1
i9	11.03	0.19	63	10.31	6.5	0.19	63	8.12	26.4	0.35	63	63	8.12	0.0
k2	14.67	0.37	48	13.95	4.9	0.72	101	11.27	23.2	2.47	246	64	10.55	6.8
large	16.36	0.69	116	15.41	5.8	1.43	145	12.73	22.2	5.99	307	66	12.01	6.0
misex3	14.18	0.69	45	12.97	8.5	1.42	63	11.27	20.5	4.98	262	62	10.55	6.8
too_large	12.51	0.13	3	12.51	0.0	0.11	3	10.30	17.7	0.15	40	10	10.06	2.4
vda	11.77	0.18	39	10.56	10.3	0.26	39	9.81	16.7	0.71	127	38	9.58	2.4
x3	9.08	0.22	99	8.12	10.6	0.33	99	8.12	10.6	0.36	102	13	8.12	0.0
Average					8.1				20.1					3.5
Total		9.54	1326			18.44	1588			51.17	3306	986		

The results demonstrate that our algorithm achieves, on average, 12% more delay reduction than the TLC (Full ND) implementation. Moreover, our results are constantly better or the same for all benchmarks. Although our algorithm runs comparatively slower, the total run time for all 16 circuits is still less than one minute.

Due to more node duplication, our resultant area is greater when comparing to the TLC implementations before applying the postprocessing techniques. However, our post-processing techniques effectively reduce the number of second-level clusters, on average, by 70% (from 3306 to 986). The effectiveness of our techniques is due to that most second-level clusters are not fully occupied. In fact, 60% of second-level clusters are less than half full in our results before postprocessing.

Since the Altera Quartus package is not available to us, we are unable to integrate our MLC algorithm into it to get the post-routing delay information. However, according to the comparisons of the post-routing results in [55] and the pre-routing results in this paper, we observe that the TLC implementation with full node duplication has better delay results than the TLC implementation with no node duplication for most cases (13 out of 16 in [55] and 15 out of 16 in this paper), and this indicates that better pre-routing results are likely to introduce better post-routing results. Since our algorithm generates much better pre-routing results than both the TLC implementations for almost all cases, it is likely that our post-routing results are also better.

Our work aims at minimizing the circuit delay, and we do successfully push the delay close to the minimum. This can be seen in columns 14-15 of Table XVI. The columns show the delay achieved by our algorithm with $n = 1$ ("one" level clustering only) and only D_1, D_2 (without D_3) used for edge delays, together with the percentage difference ("%di") when comparing to the delays achieved by the our algorithm ($n = 2$ two-level clustering). For the $n = 1$ case, the single-level graph clustering algorithm is only performed once and no graph contraction is performed. In fact, we can take these 1-level clustering results as a "loose" *lower bound* for our two-level clusterings. From the last column, it is shown that our results produce only 3.5% more delay than the 1-level results. In fact, out of 16 benchmarks, we obtain 2-level clustering solutions of the same delay as the 1-level results for 4 circuits (whose "%di" values equal to 0.0).

G. Conclusion

We have presented an effective algorithm for the general multi-level circuit clustering problem for delay minimization. The experimental results upon a two-level hierarchy shows that our algorithm achieves better delay reduction over the recent two-level circuit clustering algorithm [55]. Our future work aims at studying the area minimization for multi-level circuit clustering subject to delay constraints.

CHAPTER IX

SUMMARY AND CONCLUSIONS

This thesis focuses on five important problems in physical synthesis when the VLSI technology scales. The problems fall into two aspects: (1) Place and route aware buffer Steiner tree construction, and (2) Circuit clustering techniques with the application in Field-Programmable Gate Array (FPGA) technology mapping.

When the VLSI technology approaches the nanometer era, we have encountered more and more problems in physical synthesis, which includes power dissipation, process variations, crosstalk noises, etc. After my graduation, I will work at IBM Austin Research Laboratory and continue to devote myself in the research and development of the electronic design automation.

REFERENCES

[1] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer Academic Publishers, 1996.

[2] N. Sherwani, *Algorithms for VLSI Physical Design Automation*. Norwell, MA: Kluwer Academic Publishers, 1999.

[3] Semiconductor Industry Association, “National technology roadmap for semiconductors,” San Jose, CA, 1997.

[4] Semiconductor Industry Association, “International technology roadmap for semiconductors,” Sematech Inc., Austin, TX, 2000.

[5] J. Cong and S. Xu, “Synthesis challenges for next-generation high-performance and high-density plds,” in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2000, pp. 157–162.

[6] C. Alpert, G. Gandham, M. Hrkic, J. Hu, and S. Quay, “Porosity aware buffered Steiner tree construction,” in *Proceedings of the ACM International Symposium on Physical Design*, 2003, pp. 158–164.

[7] R. H. J. M. Otten, “Global wires harmful?” in *Proceedings of the ACM International Symposium on Physical Design*, 1998, pp. 104–109.

[8] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, “The scaling challenge: can correct-by-construction design help?” in *Proceedings of the ACM International Symposium on Physical Design*, 2003, pp. 51–58.

[9] L. P. P. P. van Ginneken, “Buffer placement in distributed RC-tree networks for minimal Elmore delay,” in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1990, pp. 865–868.

[10] W. C. Elmore, “The transient response of damped linear networks with particular regard to wideband amplifiers,” *Journal of Applied Physics*, vol. 19, pp. 55–63, Jan. 1948.

[11] J. Lillis, C. K. Cheng, and T. Y. Lin, “Optimal wire sizing and buffer insertion for low power and a generalized delay model,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, Mar. 1996.

[12] J. Lillis, C. K. Cheng, and T. Y. Lin, “Simultaneous routing and buffer insertion for high performance interconnect,” in *Proceedings of the Great Lake Symposium on VLSI*, 1996, pp. 148–153.

[13] M. Hrkic and J. Lillis, “S-tree: A technique for buffered routing tree synthesis,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2002, pp. 578–583.

[14] M. Hrkic and J. Lillis, “Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages,” in *Proceedings of the ACM International Symposium on Physical Design*, 2002, pp. 98–103.

[15] C. Alpert, G. Gandham, M. Hrkic, J. Hu, A. Kahng, J. Lillis, B. Liu, S. Quay, S. Sapatnekar, and A. Sullivan, “Buffered Steiner trees for difficult instances,” *IEEE Transactions on Computer-Aided Design*, vol. 21, no. 1, pp. 3–14, Jan. 2002.

[16] A. Jagannathan, S.-W. Hur, and J. Lillis, “A fast algorithm for context-aware buffer insertion,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2000, pp. 368–373.

[17] M. Lai and D. Wong, “Maze routing with buffer insertion and wiresizing,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2000, pp. 374–378.

[18] H. Zhou, D. F. Wong, I.-M. Liu, and A. Aziz, “Simultaneous routing and buffer insertion with restrictions on buffer locations,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 1999, pp. 96–99.

[19] J. Cong and X. Yuan, “Routing tree construction under fixed buffer locations,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2000, pp. 379–384.

[20] X. Tang, R. Tian, H. Xiang, and D. Wong, “A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2001, pp. 49–56.

[21] C. J. Alpert, G. Gandham, J. Hu, J. L. Neves, S. T. Quay, and S. S. Sapatnekar, “A Steiner tree construction for buffers, blockages, and bays,” *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 4, pp. 556–562, Apr. 2001.

[22] J. Hu, C. Alpert, S. Quay, and G. Gandham, “Buffer insertion with adaptive blockage avoidance,” in *Proceedings of the ACM International Symposium on Physical Design*, 2002, pp. 92–97.

[23] C. J. Alpert, J. Hu, S. S. Sapatnekar, and P. G. Villarrubia, “A practical methodology for early buffer and wire resource allocation,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2001, pp. 189–194.

[24] C. J. Alpert and A. Devgan, “Wire segmenting for improved buffer insertion,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 1997, pp. 588–593.

[25] C. C. N. Chu and D. F. Wong, “Closed form solution to simultaneous buffer insertion/sizing and wire sizing,” in *Proceedings of the ACM International Symposium on Physical Design*, 1997, pp. 192–197.

[26] J. Cong, L. He, C.-K. Koh, and P. H. Madden, “Performance optimization of VLSI interconnect layout,” *Integration: the VLSI Journal*, vol. 21, pp. 1–94, 1996.

[27] J. Cong and Z. Pan, “Interconnect performance estimation models for design planning,” *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 6, pp. 739–752, June 2001.

[28] S. Dhar and M. A. Franklin, “Optimum buffer circuits for driving long uniform lines,” *IEEE Journal of Solid-State Circuits*, vol. 26, no. 1, pp. 32–38, Jan. 1991.

[29] J. Cong, T. Kong, and D. Z. Pan, “Buffer block planning for interconnect-driven floorplanning,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1999, pp. 358–363.

[30] H. B. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*. Reading, MA: Addison-Wesley, 1990.

[31] J. A. Davis, R. Venkatesan, A. Kaloyeros, M. Beylansky, S. J. Souri, K. Banerjee, K. C. Saraswat, A. Rahman, R. Reif, and J. D. Meindl, “Interconnect limits on gigascale integration (GSI) in the 21st century,” *Proceedings of IEEE*, vol. 89, no. 3, pp. 305–324, Mar. 2001.

[32] C. J. Alpert, A. Devgan, and S. T. Quay, “Buffer insertion for noise and delay optimization,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 1998, pp. 362–367.

[33] C. J. Alpert, A. Devgan, and S. T. Quay, “Buffer insertion with accurate gate and interconnect delay computation,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 1999, pp. 479–484.

[34] C. C. N. Chu and D. F. Wong, “A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing,” *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 6, pp. 787–798, June 1999.

[35] C. C. N. Chu and D. F. Wong, “Closed form solution to simultaneous buffer insertion/sizing and wire sizing,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 3, pp. 343–371, July 2001.

[36] W. Shi and Z. Li, “An $O(n \log n)$ time algorithm for optimal buffer insertion,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2003, pp. 580–585.

[37] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, “Repeater scaling and its impact on CAD,” *IEEE Transactions on Computer-Aided Design*, vol. 23, no. 4, pp. 451–463, Apr. 2004.

[38] I.-M. Liu, A. Aziz, D. F. Wong, and H. Zhou, “An efficient buffer insertion algorithm for large networks based on Lagrangian relaxation,” in *Proceedings of the IEEE International Conference on Computer Design*, 1999, pp. 614–621.

[39] I.-M. Liu, A. Aziz, and D. F. Wong, “Meeting delay constraints in DSM by minimal repeater insertion,” in *Proceedings of Design, Automation and Test in Europe Conference*, 2000, pp. 436–441.

[40] Y. Jiang, S. S. Sapatnekar, C. Bamji, and J. Kim, “Interleaving buffer insertion and transistor sizing into a single optimization,” *IEEE Transactions on VLSI Systems*, vol. 6, no. 4, pp. 625–633, Dec. 1998.

[41] C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, “Path based buffer insertion,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 2005, pp. 509–514.

[42] C. V. Kashyap, C. J. Alpert, F. Liu, and A. Devgan, “Closed-form expressions for extending step delay and slew metrics to ramp inputs for RC trees,” *IEEE Transactions on Computer-Aided Design*, vol. 23, no. 4, pp. 509–516, Apr. 2004.

[43] J. Lillis, “Algorithms for performance driven design of integrated circuits,” PhD Thesis, University of California at San Diego, 1996.

[44] S. S. Sapatnekar, *Timing*. Norwell, MA: Kluwer Academic Publishers, 2004.

[45] C. J. Alpert, J. Hu, S. S. Sapatnekar, and C. N. Sze, “Accurate estimation of global buffer delay within a floorplan,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2004, pp. 706–711.

[46] Y.-C. Ju and R. A. Saleh, “Incremental techniques for the identification of statically sensitizable critical paths,” in *Proceedings of the ACM/IEEE Design Automation Conference*, 1991, pp. 541–546.

[47] C. J. Alpert, C. Chu, G. Gandham, M. Hrkic, J. Hu, C. Kashyap, and S. T. Quay, “Simultaneous driver sizing and buffer insertion using delay penalty estimation technique,” *IEEE Transactions on Computer-Aided Design*, vol. 23, no. 1, pp. 136–141, Jan. 2004.

[48] Z. Li, C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, “Making fast buffer insertion even faster via approximation techniques,” in *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference*, 2005, pp. 13–18.

[49] X. Lu and W. Shi, “Layout and parasitic information for iscas circuits,” <http://dropzone.tamu.edu/˜xiang/iscas.html> Assessed April, 2005.

[50] E. Lawler, K. Levitt, and J. Turner, “Module clustering to minimize delay in digital networks,” *IEEE Transactions on Computers*, vol. C-18, no. 1, pp. 47–57, January 1966.

[51] R. Rajaraman and D. F. Wong, “Optimum clustering for delay minimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 12, pp. 1490–1495, 1995.

[52] H. Yang and D. F. Wong, “Circuit clustering for delay minimization under area and pin constraints,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 9, pp. 976–986, 1997.

[53] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, “On clustering for minimum delay/area,” *IEEE Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 6–9, 1991.

[54] *APEX 20K Programmable Logic Device Family Data Sheet*, http://www.altera.com/literature/ds/apex.pdf, March 2004.

[55] J. Cong and M. Romesis, “Performance-driven multi-level clustering with application to hierarchical FPGA mapping,” *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 389–394, 2001.

[56] C. N. Sze and T.-C. Wang, “Optimal circuit clustering for delay minimization under a more general delay model,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 5, pp. 646–652, 2003.

APPENDIX A

VLSI DESIGN CYCLE

The overview of the VLSI systems design cycle is shown in Figure 41 which is extracted from [2]. The cycle can be generally divided into three parts: high-level synthesis, logic synthesis and physical design synthesis.

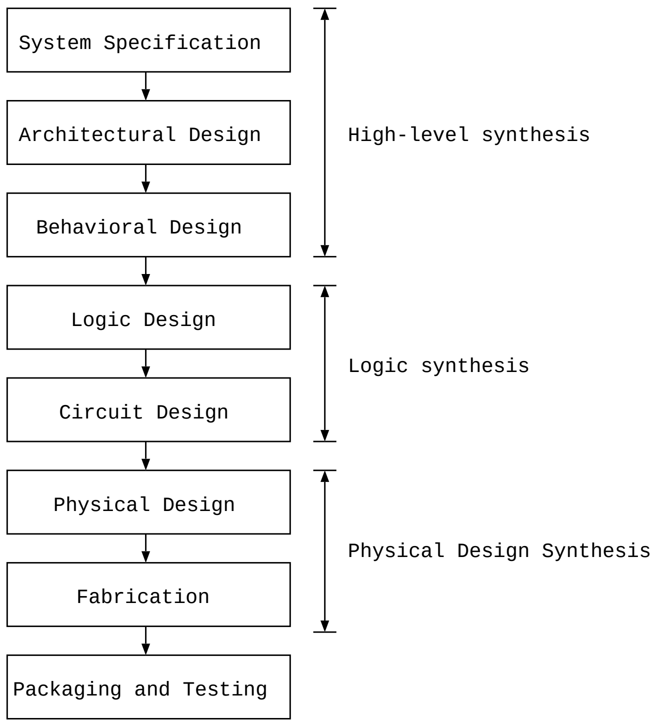


Fig. 41. A simplified VLSI design cycle

In high-level synthesis, the first steps are system specification and architectural design. The specification is a high-level representation of the system and it initiates the whole design process. As a result, it should consider all factors inside the design process such as, functionality, performance, technology, market value, etc. For the Architectural design, instruction sets and system elements such as ALUs, caches are specified. The Micro-Architectural Specification is the output of architectural design step.

The next step is the functional design and behavioral synthesis. In this step, all functional unit and the connection between the units are defined. Besides, for each unit, the system requirements are specified while the limitations are estimated.

For logic design, the output is the Register Transfer Level (RTL) description such as Verilog, Hardware Description Language (HDL) and VHDL. The description specifies logic expressions of each functional unit. In this step, logic and timing simulation and testing are performed.

The circuit design step is intended to convert logic specification into circuit representation. The representation is always called a netlist. It represents all circuit elements including gates and connections. The conversion is always guided by timing and power limitation.

For physical design processes, circuit level representation is converted into geometric representation. The process includes partitioning, floorplanning, placement and routing. The output of physical design is a layout. Throughout the processes, the conversion should strictly satisfy some design rules, such as metal width, size, layers and chip area specification. Verification is very important for layout quality assurance. If the limitation cannot be fulfilled, engineering changes must be performed.

Fabrication, packaging and testing are the last steps of the design cycle. In the process, wafers are fabricated and diced into chips. The chips should be packaged and tested before delivery. The final product should satisfy all system specification and performance requirement.

VITA

Chin Ngai SZE
Born - Hong Kong - 1977

Permanent Address

Room 5, 8/F, Block A, 395 King's Road, North Point, Hong Kong

Education

2001 - 2005
Ph.D. in Computer Engineering
Department of Electrical Engineering, Texas A&M University, College Station
1999 - 2001
M.Phil. in Computer Science and Engineering
The Chinese University of Hong Kong
Dissertation Title: Efficient Alternative Wiring Techniques and Applications
1996 - 1999
B.Eng. in Computer Engineering (Minor: Mathematics)
The Chinese University of Hong Kong
Degree Project: Routing for Low Earth Orbit Satellite Systems

Working and Teaching Experience

2005 Summer – Research Intern in IBM Austin Research Lab, Austin, TX
2004 Summer – Software Developer Intern in Synplicity Inc., Sunnyvale, CA
2001-2005 – Research/Teaching Assistant in Texas A&M University, College Station
1999-2001 – Teaching Assistant in the Chinese University of Hong Kong

Awards and Honors

- Design Automation Conference (DAC) Graduate Scholarship, 2004-2005
- Best Paper Nomination, Asia and South Pacific Design Automation Conference, 2004

Publications

- 6 journal and 14 conference/workshop papers published

The typist for this thesis was Chin Ngai Sze.