ALGORITHMS FOR VLSI CIRCUIT OPTIMIZATION AND GPU-BASED PARALLELIZATION

A Dissertation

by

YIFANG LIU

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

May 2010

Major Subject: Computer Engineering

ALGORITHMS FOR VLSI CIRCUIT OPTIMIZATION AND GPU-BASED PARALLELIZATION

A Dissertation

by

YIFANG LIU

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

Approved by:

Chair of Committee,	Jiang Hu
Committee Members,	Donald Friesen
	Weiping Shi
	Jun Zou
	Gi-Joon Nam
Head of Department,	Costas Georghiades

May 2010

Major Subject: Computer Engineering

ABSTRACT

Algorithms for VLSI Circuit Optimization and GPU-based Parallelization.

(May 2010)

Yifang Liu, B.S., Univ. of Elec. Science and Technology of China, China

M.S., Univ. of Maryland, Baltimore County, USA

Chair of Advisory Committee: Dr. Jiang Hu

This research addresses some critical challenges in various problems of VLSI design automation, including sophisticated solution search on DAG topology, simultaneous multi-stage design optimization, optimization on multi-scenario and multi-core designs, and GPU-based parallel computing for runtime acceleration.

Discrete optimization for VLSI design automation problems is often quite complex, due to the inconsistency and interference between solutions on reconvergent paths in directed acyclic graph (DAG). This research proposes a systematic solution search guided by a global view of the solution space. The key idea of the proposal is joint relaxation and restriction (JRR), which is similar in spirit to mathematical relaxation techniques, such as Lagrangian relaxation. Here, the relaxation and restriction together provides a global view, and iteratively improves the solution.

Traditionally, circuit optimization is carried out in a sequence of separate optimization stages. The problem with sequential optimization is that the best solution in one stage may be worse for another. To overcome this difficulty, we take the approach of performing multiple optimization techniques simultaneously. By searching in the combined solution space of multiple optimization techniques, a broader view of the problem leads to the overall better optimization result. This research takes this approach on two problems, namely, simultaneous technology mapping and cell

placement, and simultaneous gate sizing and threshold voltage assignment.

Modern processors have multiple working modes, which trade off between power consumption and performance, or to maintain certain performance level in a power-efficient way. As a result, the design of a circuit needs to accommodate different scenarios, such as different supply voltage settings. This research deals with this multi-scenario optimization problem with Lagrangian relaxation technique. Multiple scenarios are taken care of simultaneously through the balance by Lagrangian multipliers. Similarly, multiple objective and constraints are simultaneously dealt with by Lagrangian relaxation. This research proposed a new method to calculate the subgradients of the Lagrangian function, and solve the Lagrangian dual problem more effectively.

Multi-core architecture also poses new problems and challenges to design automation. For example, multiple cores on the same chip may have identical design in some part, while differ from each other in the rest. In the case of buffer insertion, the identical part have to be carefully optimized for all the cores with different environmental parameters. This problem has much higher complexity compared to buffer insertion on single cores. This research proposes an algorithm that optimizes the buffering solution for multiple cores simultaneously, based on critical component analysis.

Under the intensifying time-to-market pressure, circuit optimization not only needs to find high quality solutions, but also has to come up with the result fast. Recent advance in general purpose graphics processing unit (GPGPU) technology provides massive parallel computing power. This research turns the complex computation task of circuit optimization into many subtasks processed by parallel threads. The proposed task partitioning and scheduling methods take advantage of the GPU computing power, achieve significant speedup without sacrifice on the solution quality.

To My Parents and Grandma

ACKNOWLEDGMENTS

The past few years have made up one of the most valuable periods in my life. I owe my gratitude to all those people who have made this possible. Foremost, I would like to thank my advisor Prof. Jiang Hu. I have been amazingly fortunate to have an advisor who led my way into VLSI design automation field, gave me the freedom to explore various problems, and at the same time, persistently provided me with invaluable academic advice on research. His deep insight, patient guidance, and continuous support has helped me overcome many difficulties and finish this dissertation.

I am very thankful to my doctoral committee members, Prof. Donald Friesen, Prof. Jiang Hu, Dr. Gi-Joon Nam, Prof. Weiping Shi, and Prof. Jun Zou. I really appreciate their invaluable assistance, advice, and support to my research and dissertation.

I am grateful to Prof. Jiang Hu, Prof. Sunil Khatri, Prof. Peng Li, and Prof. Xi Zhang. It has been a very rewarding experience taking their courses and discussing various research topics with them.

Last, my most profound gratitude is expressed to my parents and grandma. My parents' unconditional love accompanies me through all hard times. My grandma's intellectual inspiration has always been with me, and gives me strength and belief in myself. This dissertation is dedicated to them.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

A. Motivation and Major Contributions

In the past several decades Very Large Scale Integration (VLSI) technology has been scaling down and now moves into nanometer regime. The complexity of the circuit keeps growing. Besides increasing number of gates/transistors in circuits, more advanced technologies, such as multi-core architecture and adaptive working modes, pose new challenges in design closure. VLSI design automation becomes more critical in the design flow. Efficient algorithms are desired for the design of high performance, low power consumption, and low cost chips.

Many optimization problems in VLSI design automation are NP-complete and non-convex. These difficult problems are normally solved in two major approaches: continuous optimization for numerical solutions and combinatorial optimization for discrete solutions.

Continuous optimization methods have advantage in runtime efficiency and theoretical optimality under certain conditions. However, due to the limited number of cells in the standard cell library and the discrete nature of circuit implementation, continuous optimization methods have to make significant approximation or use heuristics in problem formulation and solution legalization. The solution quality is largely compromised by various kinds of approximation. Moreover, because most problems in VLSI design practice are non-convex, theoretical optimality conditions

---

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

do not hold. Instead, the tendency to be trapped into local optimum qualifies them more as a greedy approach.

This research takes the combinatorial optimization approach. Among existing work in this approach, a simple discrete method is greedy algorithm. Because it relies on local optimal view in the solution space, globally good solutions are hardly reached. Due to its simplicity, however, it has fast runtime. To the other end, simulated annealing introduces randomness into solution search, and is able to find the optimal solution given infinite runtime. But, in practice the runtime is prohibitive to obtain decent solutions. A more systematical approach is dynamic programming. For circuit optimization in tree topology, dynamic programming can come up with optimal solutions in polynomial time. However, many discrete optimization problems in design automation is formulated on directed acyclic graph (DAG) topology. Conventional dynamic programming encounters significant difficulty when applied to directed acyclic graph (DAG), due to the inconsistency and interference between solutions on reconvergent paths in a DAG. Thus, good solutions can not be guaranteed on DAGs by conventional dynamic programming methods. This research proposes a systematic solution search guided by a global view of the solution space. The key idea of the proposal is joint relaxation and restriction (JRR), which is similar in spirit to mathematical relaxation techniques. Here, the relaxation and restriction together provides a global view, and iteratively improves the solution.

One of the challenges not taken care of well in traditional circuit design automation is sequential optimization in the design flow. As shown in Fig. 1, traditionally, the design flow is carried out in a sequence of stages from high level synthesis, logic synthesis, technology mapping, floor planning, cell placement, gate sizing, voltage assignment, routing, to interconnect optimization, and so on. Early stages are performed with some assumption/estimation on the result from later stages. If the design

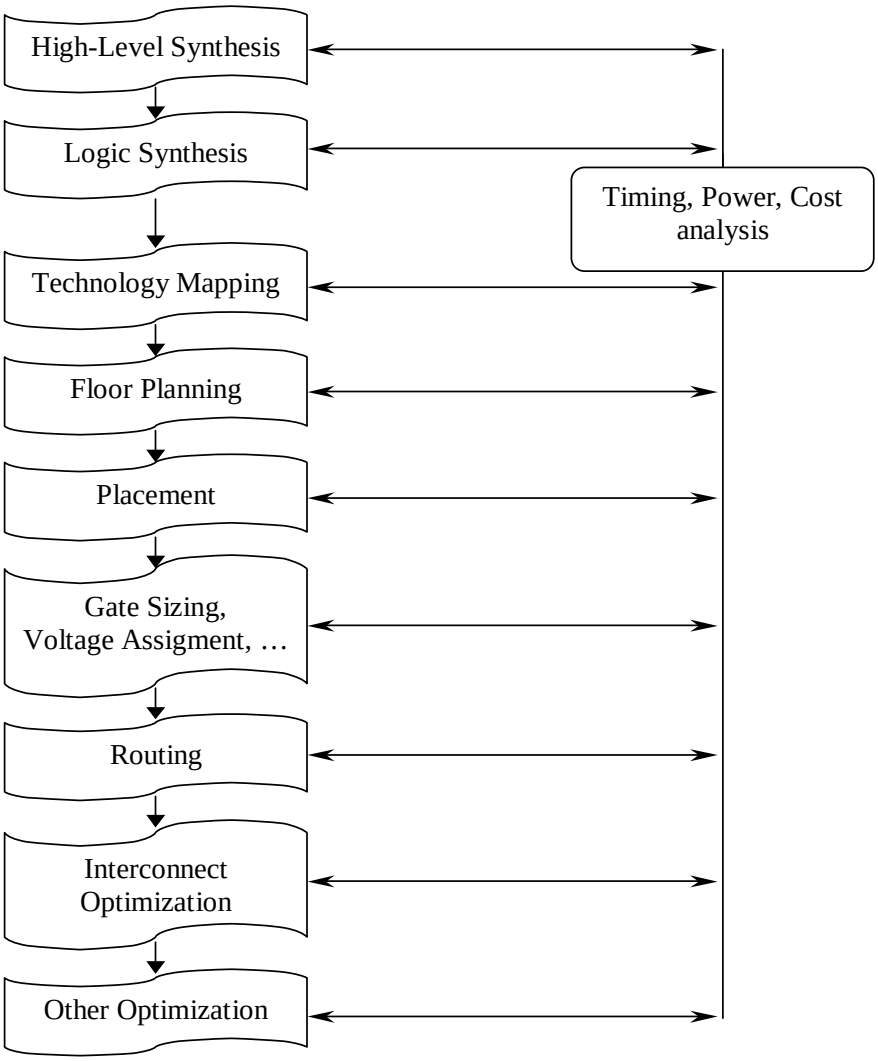


Fig. 1. Design flow

requirement is not met in one of these stages, the procedure turns back to an earlier stage with the assumption based on the result from current stages (the later stages). The problem with this sequential procedure is the limit due to the inaccurate local view on a part of the whole design flow. It does not ensure convergence or good overall solutions, since different later stage results lead to different earlier stage solution during the backtracking, thus oscillation among poor solutions may occur. Evidently, separately considering different stages suffers from the local view of individual opti-

mization stages.

This research overcomes the limitation of the sequential optimization with simultaneous optimization on multiple traditional optimization stages. According to the techniques in multiple stages, different types of optimization options are seamlessly integrated together to form the combined optimization option set for each element in the circuit. Consequently, the solution search is performed in a combined solution space. This way, multiple traditional optimization stages are performed in a true simultaneous manner, which eliminates the oscillation between suboptimal solutions. Regarding its impact on the runtime, this research manages to keep the overall complexity linear to the square of the number of combined options on each element, and linear to the total number of elements in the circuit. This scheme is applied on two problems, namely, simultaneous technology mapping and cell placement, and simultaneous gate sizing and threshold voltage assignment.

New challenges in VLSI circuit optimization also come from the increasingly complex circuit design. Nowadays, micro-processors often consist of multiple cores and run in multiple modes. To make a design optimal for multiple cores/modes is much more difficult than for single core/mode. This research proposes multi-scenario optimization methods for circuit design automation problems, which always consider multiple scenarios simultaneously on each step during the optimization. For multi-scenario gate sizing and threshold voltage assignment problem, Lagrangian relaxation is employed to accommodate multiple modes in one design. A new Lagrangian dual problem solving technique is introduced to achieve higher solution quality. For multi-core buffer insertion problem, a critical component analysis based method is created to handle different environmental parameters across different cores.

Under the intensening time-to-market pressure, circuit optimization not only needs to find high quality solutions, but also has to come up with the result fast.

Recent advance in general purpose graphics processing unit (GPGPU) technology provides massive parallel computing power. This research turns the complex computation task of circuit optimization into many subtasks processed by parallel threads. The proposed task partitioning and scheduling methods take advantage of the GPU computing power, achieve significant speedup without sacrifice on the solution quality. This parallel computing scheme is reflected in the fast gate sizing problem.

The **major contributions** in this research are summarized briefly as follows. Specific problems they are applied on in this research are listed accordingly:

- Joint Relaxation and Restriction (JRR) for efficient systematic solution search on DAGs in circuit optimization - a generic approach, applied on the following problems
  - The problem of simultaneous gate sizing and V_t assignment
  - The problem of simultaneous technology mapping and cell placement

- Multi-Technique Multi-Objective simultaneous circuit optimization by option integration and improved Lagrangian dual problem solving with systematic subgradient calculation, applied on the following problems
  - The problem of simultaneous gate sizing and V_t assignment
  - The problem of simultaneous technology mapping and cell placement

- Multi-Scenario Multi-Core circuit optimization by critical component analysis and Lagrangian relaxation, applied on the following problem
  - The problem of multi-core buffer insertion

- Parallel computing scheme for circuit optimization by GPU-oriented task partition and scheduling, applied on the following problem
  - The problem of fast gate sizing and V_t assignment

B. Applications on Circuit Design Automation Problems

The rest of this dissertation is organized according to the problems investigated in this research, which are solved using the proposed techniques briefly introduced in the previous section.

Chapter II presents algorithms for simultaneous gate sizing and V_t assignment. Gate sizing and threshold voltage (V_t) assignment are popular techniques for circuit timing and power optimization. Existing methods, by and large, are either sensitivity-driven heuristics or based on discretizing continuous optimization solutions. Sensitivity-driven heuristics are easily trapped in local optimum and the discretization may be subject to remarkable errors. Compared to continuous optimization based methods, a combinatorial approach has three main advantages. First, it can be easily applied with look-up table models for delay and power, which are the de facto standard models in industrial designs. In contrast, continuous optimization requires modification to the look-up table by data fitting [2, 3] or selection heuristic, which incurs inaccuracy in delay and power calculation. This is true in the whole continuous problem formulation, which takes a discrete problem and approximates it with a continuous mathematical programming problem. Second, the solution of continuous optimization has to be rounded to obtain settings that exist in a cell library. The rounding is subject to errors and the error can be significant if gate configurations are highly discrete [4]. Moreover, it is very difficult for continuous optimization to handle different pMOS/nMOS size ratios in cell library based designs. Third, combinatorial optimization is easier to utilize parallel computing to gain more significant speedup compared with continuous optimization. In this work, a systematic combinatorial approach for simultaneous gate sizing and V_t assignment is proposed. The core idea of this approach is Joint Relaxation and Restriction (JRR), which em-

ploys consistency relaxation and coupled bi-directional solution search. The process of joint relaxation and restriction is conducted iteratively to systematically improve solutions. The proposed algorithm is compared with a state-of-the-art previous work on benchmark circuits. The results from the algorithm can lead to about 22% less power dissipation subject to the same timing constraints.

Chapter III presents algorithms for simultaneous technology mapping and cell placement. Technology mapping and placement have significant impact on the delays in standard cell based very large scale integrated (VLSI) circuits. Traditionally, these steps are applied separately to optimize delays, possibly since efficient algorithms that allow the simultaneous exploration of the mapping and placement solution spaces are unknown. In fact, there is a cyclic dependency between these two steps. Timing driven technology mapping needs the wire length information determined by placement, while placement needs the result of technology mapping to see the cells to be placed. The placement done after technology mapping may suggest a new mapping solution other than the one before it. Thus, performing these two steps repeatedly in a sequence can turn into a procedure wandering among different mapping and placement solutions without convergence. Instead of performing mapping and placement separately, this work proposes an exact polynomial time algorithm for delay-optimal placement of a tree and extend the same to simultaneous technology mapping and placement for optimal delay in the tree. For delay optimization in directed acyclic graphs (DAGs), the algorithm is extended by employing Lagrangian relaxation technique, which assesses the timing criticality of paths beyond a tree. Experimental results on benchmark circuits in a 70 nm technology show that the proposed algorithms improve timing significantly with remarkably less run-times compared to a competitive approach of iterative conventional timing driven mapping and multi-level placement.

Chapter IV presents algorithms for buffer insertion in multi-core designs. Recently, microprocessor industry is headed in the direction of multi-core designs in order to continue the chip performance improvement. This work investigates buffer insertion, which is a critical timing optimization technique, in the context of an industrial multi-core processor design methodology. Different from the conventional formulation, buffer insertion in this case requires a single solution to accommodate multiple different scenarios. If the conventional buffer insertion is performed for each scenario separately, there may be different solutions corresponding to these scenarios. A naive approach is to judiciously select a solution from one scenario and apply it to all the scenarios. However, a good solution for one scenario may be a poor one for another. This work proposes algorithmic techniques to solve these multi-scenario buffer insertion problems. Compared to the naive approach, the proposed algorithm can improve slack by $102ps$ on average for max-slack solutions. For min-cost solutions, the algorithm causes no timing violation while the naive approach results in 35% timing violations. Moreover, the computation speed of our algorithm is faster.

Chapter V presents algorithms for fast gate sizing and V_t assignment by GPU-based parallelization. The progress of GPU (Graphics Processing Unit) technology opens a new avenue for boosting computing power. This work is an attempt to exploit the parallel computing power in GPUs for massive acceleration in VLSI circuit optimization. This work proposes GPU-based parallel computing techniques and apply them on the solution of simultaneous gate sizing and threshold voltage assignment problem, which is often employed in practice for performance and power optimization. These techniques are aimed to fully utilize the benefits of GPU through efficient task scheduling and memory organization. Compared to conventional sequential computation, the proposed techniques can provide up to 56× speedup without any sacrifice on solution quality.

CHAPTER II

SIMULTANEOUS GATE SIZING AND V_T ASSIGNMENT

A. Introduction

Gate/transistor sizing is a classic technique for optimizing circuit timing and power dissipation. Continuous gate/transistor sizing is formulated as a geometric programming problem in [5] and is solved by Lagrangian relaxation in [6]. The work of [7] employs a randomized search method for discrete gate sizing. The method in [8] applies backtracking on general networks in gate sizing. Recently, a continuous solution guided dynamic programming algorithm [4] is proposed. When leakage power becomes prominent, people start to use gates with different threshold voltage (V_t) levels in order to trade timing slack for leakage power reduction [9, 10, 11]. Due to the similarity between them, gate/transistor sizing and V_t assignment are often conducted simultaneously [1, 12, 13, 14, 15, 16, 17]. Among these previous works, [9] and [15] are greedy or sensitivity driven heuristics. In [10, 16], continuous optimization is performed and then the results are rounded to obtain discrete solutions. The work of [17] exploits parallelism in discrete V_t assignment and continuous gate sizing. In [14], it is found that linear programming based optimization often results in discrete V_t assignment solutions and therefore the rounding can be skipped. However, such self-snapping is guaranteed only in certain scenarios, instead of general cases. A combinatorial algorithm for transistor sizing and V_t assignment is introduced in [12]. However, this algorithm is restricted to tree topologies. The work of [1] is an iterative method. In each iteration, timing slack is allocated to each gate based on sensitivity guided linear programming and then an implementation is selected for each gate such that the allocated slack is traded for power reduction.

We propose a new combinatorial algorithm for simultaneous gate sizing and threshold voltage assignment. Compared to continuous optimization based methods [10, 16], a combinatorial approach has two main advantages. First, it can be easily applied with look-up table gate model, which is the de facto standard in most industrial designs, especially ASIC designs. In contrast, continuous optimization requires modification to the look-up table by data fitting [2, 3] which causes inaccuracy in delay and power estimation. Second, a continuous optimization solution has to be discretized to obtain the V_t assignment and gate size that exist in a cell library. The discretization is subject to errors and the error can be significant if gate configurations are highly discrete [4]. Moreover, it is difficult for continuous optimization to handle different pMOS/nMOS size ratios in library based designs.

Our algorithm is in the same spirit as Dynamic Programming (DP). DP is a systematic combinatorial optimization approach. A well-known example is Ginneken-Lillis buffer insertion algorithm [18], which propagates solutions from leaf nodes of an interconnect tree toward the root and finds the maximal slack solution in quadratic runtime. In general, a tree topology allows DP to reach the optimal solution elegantly. The main challenge of applying DP-like systematic solution search in gate sizing and V_t assignment is that the underlying topology is typically a DAG (Directed Acyclic Graph) instead of a tree, where solutions are often merged at path reconvergence. Such merging requires that the histories of the solutions have to be consistent with each other. Either maintaining or tracing back all history information entails large computation and memory overhead. Due to this difficulty, there is no DP-like or other systematic combinatorial optimization algorithm for gate sizing and V_t assignment, to the best of our knowledge. In this work, we propose a new method of Joint Relaxation and Restriction (JRR), which enables DP-like solution search for gate sizing and V_t assignment. JRR combines consistency relaxation and coupled bi-directional solution

search to systematically improve the solution from a starting point. To improve initial optimization, starting point search is realized with iterative joint relaxation and restriction.

These new techniques distinguish our algorithm from the previous combinatorial methods [9, 12, 15]. Compared to the sensitivity driven heuristics [9, 15], our algorithm is more systematic and therefore can lead to improved solution quality. Our algorithm can be directly applied on DAG topology as opposed to tree topology in [12]. Experiments are performed on ISCAS85, ITC99, and IWLS 2005 benchmark circuits to compare our algorithm with a state-of-the-art previous work [1]. The results indicate that on average our algorithm yields 22% more power reduction under the same timing constraints.

B. Preliminaries

A combinational logic circuit can be described by a DAG $G(V, E)$, where V is a set of nodes, each of which represents a logic gate, and each edge $(v_i, v_j) \in E$ indicates the wire connection between node v_i and v_j. Note that the edge is directed and logic signals are propagated from v_i to v_j. Every gate $v_i \in V$ has multiple implementation options, each of which consists of a size w_i and a V_t level u_i for v_i. The simultaneous gate sizing and V_t assignment problem is to select a size $w_i \in W_i$ and a V_t level $u_i \in U_i$ for every individual gate v_i, so that the total power is minimized subject to timing constraints, i.e., no negative timing slack. A specific choice of size and V_t options for a gate is called a solution for the gate. The solution for the whole circuit is composed of a set of gate solutions, each of which is for an individual gate in the circuit. For the clarity of presentation, we will first describe our algorithm for only timing optimization, which aims to maximize the timing slack of the circuit. After

that, we will show how to extend the algorithm to minimize power under timing constraint. Then, we present a runtime-quality tradeoff method for high runtime efficiency.

Main notations used in this paper are summarized below:

v_i	the ith node/gate.
(v_i, v_j)	interconnect between gate v_i and v_j.
$I(G)$	the set of entrance gates, whose fanins are all primary inputs of circuit G.
$\mathcal{V}$	the set of all the multi-fanin gates in circuit G.
W_i	the set of all possible sizes of gate v_i.
U_i	the set of all possible V_t level of gate v_i.
v_i^k	the kth size and V_t option for gate v_i.
$v_i^k : v_{fout(v_i)}^h$	the kth option of v_i with the gates on its fanout implemented by combined options $v_{fanout(v_i)}^h$.
χ_i	the set of solutions at v_i, $\{(v_i^k : v_{fanout(v_i)}^h)\}$.
X_i	the set of implementation options of gate v_i.
$c(v_i^k)$	input capacitance of the kth option for gate v_i.
$r(v_i^k)$	output resistance of the kth option for gate v_i.
$D(v_i, v_j)$	gate and wire delay: $d(v_i) + d(v_i, v_j)$.
$a(v_i)$	maximum arrival time at v_i's fanins.
$q(v_i)$	minimum required arrival time at v_i's fanins.
$(v_i^k)[c, q]$	the pair of c and q at v_i for its option k.
$s(v_i)$	time slack $q(v_i) - a(v_i)$.
$p(v_i)$	dynamic and leakage power on gate v_i.

We consider two types of power dissipation in this work. One is dynamic power, which can be calculated by: $P_{dynamic} = \frac{1}{2}\alpha V_{DD}^2 f_{clk} C$, where α is the switching factor, f_{clk} is the clock frequency and C is the load capacitance due to gates and wires. The other is leakage power $P_{leakage} = V_{DD} I_{off}$, where the off current I_{off} for a gate of certain size and V_t level is usually obtained in cell characterization and provided along with a cell library. Short circuit power is relatively small and neglected in this work. However, it is straightforward to consider short circuit power in our algorithm.

The size of a gate affects its delay, its input capacitance, dynamic and leakage power dissipation. The V_t level of a gate affects its delay and leakage power. In our

algorithm, the delay can be estimated using either the Elmore delay model or more accurate models like those employed in commercial timing analyzers. For example, if the size and V_t for both v_i and v_j are decided, the delay from v_i to v_j can be estimated as follows. If the wire between v_i to v_j is neglected, the delay can be obtained based on the lookup table for v_i. If the wire is considered, the wire delay can be computed using higher-order model such as RICE [19]. The wire capacitance load to v_i can be evaluated by the effective capacitance technique [20]. We assume that the arrival time at all primary inputs and the required arrival time at all primary outputs are given. The timing criticality of each node $v_i \in V$ is indicated by its slack $s(v_i) = q(v_i) - a(v_i)$, where q and a denote the required arrival time and the arrival time, respectively. The overall timing performance of a circuit is characterized by the minimum slack among all nodes.

C. Timing Optimization

In this section, we describe our simultaneous gate sizing and V_t assignment algorithm for timing optimization. Note that all gates can be simply assigned with the lowest available V_t level if timing is optimized without considering power. However, we still include V_t assignment in the presentation in order to be consistent with the algorithm description in Section D, which introduces the algorithm for timing-constrained power optimization. For a given cell library, choosing a size and V_t level for a gate is equivalent to selecting a gate type in the library that does the same logic operation. Below is the formulation of the timing optimization problem.

Timing Optimization: Given a netlist of combinational logic circuit $G(V, E)$, arrival times at its primary inputs, required arrival times at its primary outputs, and a cell library, select an implementation for each gate to maximize the circuit slack,

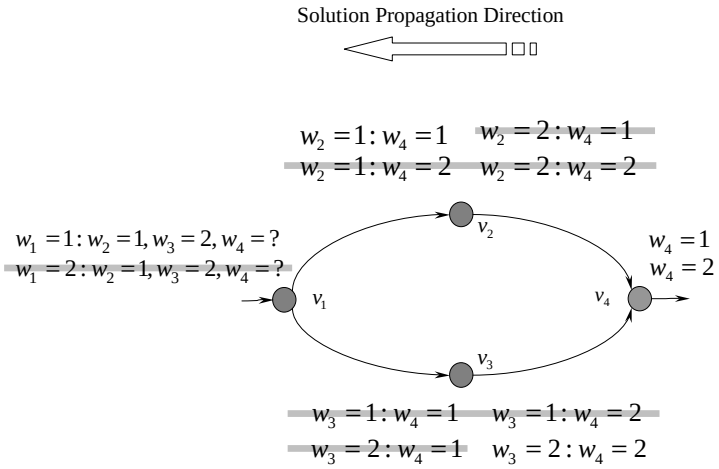


Fig. 2. Solution search from v_4 to v_1. Solution prunings are performed at v_2 and v_3. At node v_1, when solution $(w_2 = 1, w_4 = 1)$ from v_2 is merged with solution $(w_3 = 2, w_4 = 2)$ from v_3, they are based on different solutions at v_4 and therefore not consistent with each other.

i.e.,

$$\begin{aligned} \max \quad & \min_{v_j \in V} s(v_j) & \\ \text{s.t.} \quad & w_i \in W_i, & \forall v_i \in V \\ & u_i \in U_i, & \forall v_i \in V \end{aligned}$$

1. Difficulty of DAG Optimization and the Main Ideas

On a DAG, it is very difficult to perform systematic yet efficient solution search, like dynamic programming, mainly due to reconvergence paths. This can be illustrated by a simple example in Fig. 2, where each node (gate) has two size options. Like dynamic programming, the solution search can be performed backward, i.e., from output toward input. At node v_4, there are two solutions. When they are propagated to v_2, there are four possible solutions. To make the search efficient, unpromising solutions are pruned out. Otherwise, the search is nothing but brute-force and would

cause very slow runtime as well as very poor scalability. At node v_2, let us assume that only solution $(w_2 = 1, w_4 = 1)$ is retained and the others are pruned. Similarly, only solution $(w_3 = 2, w_4 = 2)$ is kept at node v_3. Next, solutions from v_2 and v_3 should be merged at node v_1. However, the solution from v_2 is based on $w_4 = 1$ and the solution from v_3 is according to $w_4 = 2$, i.e., their histories are not consistent with each other. Therefore, they cannot be merged. If the search proceeds forward from input to output, the same problem still exists.

The path reconvergence problem has two consequences on solution search. First, history consistency check must be performed when solutions from two paths are merged. Second, and more importantly, whether or not a solution is inferior may depend on future solution mergings at reconvergence nodes. This is a key difference from the case of tree, where the pruning at a node can be solely based on its current characteristics.

To avoid the path reconvergence problem, one may consider to optimize individual timing critical paths one after another. However, after optimizing a critical path A, a previously non-critical path B may become critical. Next, if path B is optimized, path A may become critical again. Thus, the path-based optimization may oscillate and is difficult to converge. Iteratively optimizing critical trees can alleviate the problem, but cannot radically solve it.

In this work, we contribute some ideas which form the basis of a systematic and efficient solution search on DAGs. Since the main difficulty is due to the history consistency constraint, we relax this constraint at initial optimization stage and restore it back later. This is in the same flavor as other relaxation techniques, such as Lagrangian relaxation. If the search is performed along only one direction, it is not obvious if a seemingly inferior solution at a node will be useful in future. As a result, it is very difficult to do the solution pruning. To solve this problem, we propose to

perform both backward and forward search, which are carried out one after the other iteratively. The information obtained in the backward search will help the solution pruning in the forward search. Because the coupled bi-directional search after the relaxation stage always maintains the historical consistency throughout the circuit, it enforces the satisfaction of the constrains on solution search. Thus, we call this method Joint Relaxation and Restriction (JRR). The solution found by one iteration of relaxation and restricted bi-directional search may be a local optimum. To cope with this situation, we propose multiple iterations of relaxation and restriction, where each iteration is guided by the solution from the previous iteration, and seeks a better solution in a carefully selected solution set. This approach is thus called Iterative Joint Relaxation and Restriction (IJRR).

An overview of our algorithm framework is outlined in Algorithm 1. The outer loop realizes the Iterative Joint Relaxation and Restriction (IJRR). Each iteration consists of two phases. Phase I is an initial optimization composed by a consistency-relaxation based search and a procedure of consistency restoration. Phase II is an iterative refinement. In each iteration of the refinement, both forward search and backward search are performed. The details of the algorithm are described in Sections 2, 3, and 4.

2. Phase I: Initial Optimization

The initial optimization phase consists of two stages: consistency relaxation and consistency restoration. The two stages are elaborated as follows.

a. Consistency Relaxation

Consistency relaxation is a backward solution search procedure that proceeds from the primary outputs to the primary inputs in reverse topological order. This process

Input : combinational circuit G and cell library L
Output: size and V_t assignment for all gates in G

//Initialize the option sets at multi-fanin gates
$X_i \leftarrow \{$all implementaion options of $v_i\}, \forall v_i \in V$;
repeat
  //PHASE I: Initial Optimization
  $ConsistencyRelaxation(G)$;
  $ConsistencyRestoration(G)$;
  //PHASE II: Iterative Refinement
  $InterativeRefinement(G)$;
  //Update the option sets at multi-fanin gates
  $X_i \leftarrow \{v_i$'s used options in this iteration$\}, \forall v_i \in \mathcal{V}$;
until *no improvement*;

Algorithm 1: $SizeVt\_MaxSlack(G)$

is the same as the one in the buffer insertion [18]. During the search, all options of sizes and V_t levels are enumerated for each gate to generate possible solutions. More precisely, every implementation option in a gate v_i's option set X_i is evaluated. During the first JRR iteration, all the options of each gate are included in its option set, while in later JRR iterations, option sets for multi-fanin gates are updated and option evaluation on multi-fanin gates are confined to the sets. We explain the detail of option sets in Section 4.

To make this paper self-contained, here we briefly describe the backward search procedure. As mentioned in Section B, a solution on a gate is a specific implementation option (size and V_t level) of it. For every solution of a gate, there is a corresponding best solution on each of its fanout gates. Thus, every gate solution is associated with a set of pointers linked to corresponding best downstream gate solutions, so that later on we can trace back all the gate solutions from the best solution at the primary inputs to the primary output. Each solution of gate v_i is characterized by the load capacitance c seen by its upstream gates and the corresponding largest

required arrival time q. (If higher-order delay model is employed, moments are propagated along with the solutions.) All operations of solution propagation are performed according to the solution characterization, which determines the timing at every node. For better computation efficiency, it is crucial to identify and eliminate inferior gate solutions during solution search. A main criterion of inferior gate solutions is based on the c and q values as follows. Solution v_i^k is inferior to v_i^h, if

$$c(v_i^k) \geq c(v_i^h) \text{ and } q(v_i^k) \leq q(v_i^h). \tag{2.1}$$

In situations where there is no confusion, we may represent a gate solution with the implementation options for the gate and its fanout gates, or simply with the $[c, q]$ values.

In order to generate solutions at a gate, solutions at its fanout gates are merged first. When two solutions are merged, the merged solution is characterized by the summation of load capacitances of the two solutions and the smaller required arrival time q of the two solutions. Based on the inferior solution rule given in formula (1), the merging of the solution sets on two different gates, can be performed in linear-time, by excluding inferior merged solutions during the merging. More specifically, the solutions in each of the two solution sets are sorted in ascending order of their c values. At the beginning, two points are linked to the two solutions with minimum c values in each of the two solution sets, respectively. The two pointed solutions are merged to form a merged solution. After that, the pointer linked to the solution with smaller q value (more critical one) between the two solutions moves to the next solution in the same set. This solution merging continues until the end of the sorted solution sets is reached. Once the merged solutions are generated, the gate delays for different implementation options of the gate are added to each merged solution to form the solutions at the gate. Thereafter, more inferior solutions are pruned out by

formula (1).

Although the procedure on each gate in our algorithm is the same as in buffer insertion [18], there are two essential differences. First, the gate sizing and V_t assignment problem faces the reconvergent path issue which often requires history consistency when solutions are merged. Directly applying a dynamic programming algorithm here actually implies a relaxation on the constraint of history consistency. As a result, inconsistent solutions may emerge in the search on DAGs, which needs to be resolved in the next restoration stage.

Second, instead of having the option of not inserting a buffer at each node as in buffer insertion, there is always a gate "inserted" at each node in gate sizing and V_t assignment. This implies a critical property of solutions in our case.

Property 1 *For a specific size w_i and threshold voltage u_i of any gate v_i, there is at most one non-inferior solution at v_i that can be preserved after pruning.*

This property is justified by the fact that all solutions with a specific w_i value at v_i have an identical load c and only the solution with the maximum required arrival time q is not pruned out.

The pseudo code of the consistency relaxation procedure is provided in Algorithm **??**. For each node, the procedure performs three operations: merging the solution sets from its fanouts, which is line 3; applying every option of the node on the merged solutions (adding the gate delay of the node to the merged solutions), which is implemented by lines 5 to 12, (note that only one solution for each implementation option enters the solution set); pruning the solution set at the node is done by line 13.

Fig. 3 illustrates this procedure with a simple example. Solutions are propagated backwards, i.e., in reverse topological order. Solution sets at v_4, v_2 and v_3 have been

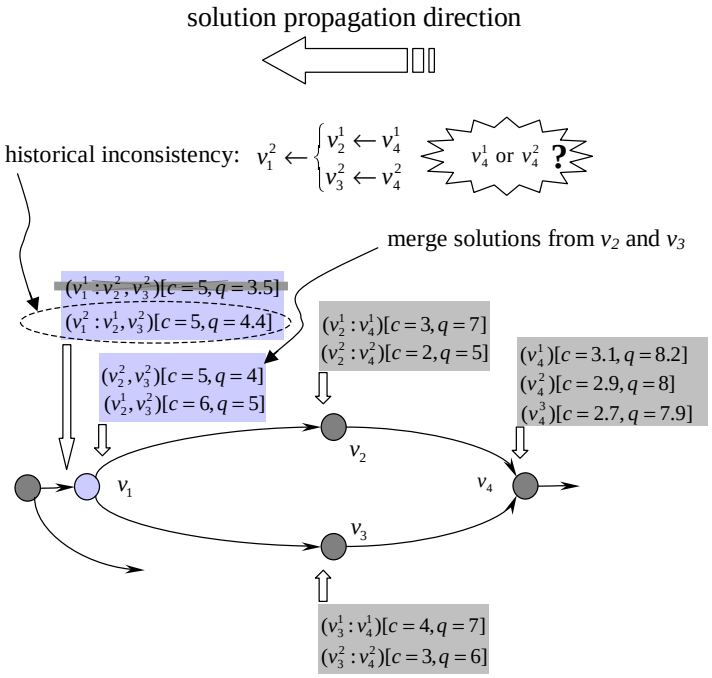


Fig. 3. Multi-fanin node v_4 diverges into two paths in backward direction - reverse topological order, which rejoin at node v_1. Required arrival times are propagated in backward direction - topological order. Merging solutions at the fanout of v_1 yields merged solution set $\{[5,5],[6,6]\}$, which is combined with v_1's options to form its solution set $\{[6,3.5],[5,4.4]\}$. While solution $[6,3.5]$ is inferior and pruned, solution $v_1^2[5,4.4]$ traces back to conflicting ancestor solutions v_4^1 and v_4^2 on v_4.

created. Consider generating solutions at v_1. Operation $merge(v_2, v_3)$ merges two solution sets: $\{[3,7],[2,5]\}$ at v_2 and $\{[4,7],[3,6]\}$ at v_3. The result is the non-inferior merged solutions $\{[5,5],[6,6]\}$ at the fanout of v_1. Adding the gate delay of v_1 to each merged solution yields the solutions at v_1: $\{[6,3.5],[5,4.4]\}$. By pruning rule (2.1), the solution $[6,3.5]$ is pruned due to smaller required arrival time. At the end of *ConsistencyRelaxation*, there may be history inconsistency in solutions. Solution $[5,4.4]$ at v_1 is an example. It is based on downstream solutions v_2^1 and v_3^2 at v_2 and v_3, respectively, which in turn refer to different solutions v_4^1 and v_4^2 at the multi-fanin

Input : combinational circuit G and cell library L
Output: solution sets of size and V_t assignment for all gates in G

```
1  χ_i ← ∅, ∀v_i ∈ V;
2  for v_i ∈ G in reverse topological order do
3      χ_fout(v_i) ← merge(fout(v_i));
4      prune the solution set at v_i by rule in (2.1);
5      for every option v_i^k of v_i in X_i do
6          q(v_i^k) ← −∞ ;
7          for merged solution v^h_fout(v_i) ∈ χ_fout(v_i) do
8              if q(v^h_fout(v_i)) − D(v_i^k, v^h_fout(v_i)) > q(v_i^k) then
9                  q(v_i^k) ← q(v^h_fout(v_i)) − D(v_i^k, v^h_fout(v_i));
10                 ĥ ← h;
11         χ_i ← χ_i ∪ (v_i^k : v^ĥ_fout(v_i)) [c(v_i^k), q(v_i^k)];
12     prune the solution set at v_i by the rule given in Formula (2.1);
```

Algorithm 2: *ConsistencyRelaxation*(G)

node v_4. The conflict needs to be resolved in next stage *ConsistencyRestoration*.

According to Property 1, the size of each solution set is upper bounded by m, the maximum number of implementation options of a gate. Therefore, the runtime of the merging procedure $merge(fanout(v_i))$ is $O(mb)$, where b is the maximum number of fanout among all gates. Applying all implementation options of v_i on merged solutions $\chi_{fanout(v_i)}$ takes $O(m^2b)$ time. Thus, the overall runtime of *ConsistencyRelaxation*(G) is $O(|V|m^2b)$ over all nodes.

b. Consistency Restoration

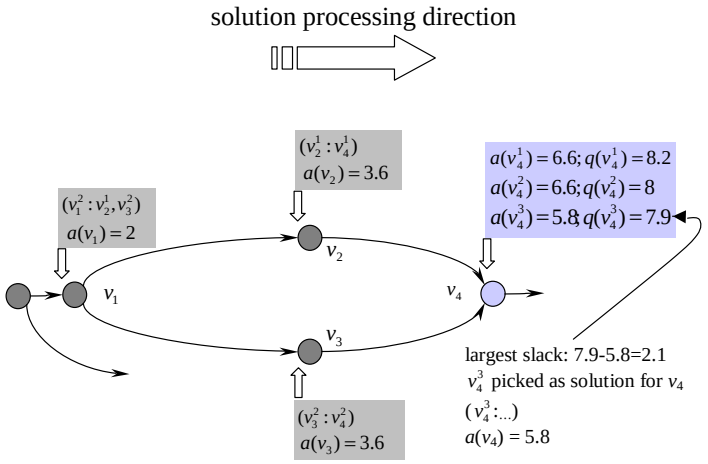
Consistency restoration is a forward search procedure that proceeds from the primary inputs to the primary outputs in topological order. When a node is visited during the search, only one gate implementation option (a size and a V_t level of the gate in library) is selected as the solution of the node. Thus, there is no history inconsistency

when the search is completed.

The information obtained in the previous consistency relaxation stage is used to guide consistency restoration. In particular, a specific gate implementation (size and V_t level) of a node is associated with a unique required arrival time value obtained in the relaxation stage according to Property 1.

Given the arrival time at the primary inputs, the arrival times at all the nodes are updated when the solution is propagated through. At each node, the slack for each of its implementation options is computed. Again, accurately it is the implementation options in each gate's option set that are actually evaluated. In the first JRR iteration, every option set contains all possible implementation options at a gate. The option with the maximum slack is selected as the solution for the gate. Then, the arrival time corresponding to the chosen option is updated as the arrival time at this node, which will be used in computing arrival times at its fanouts.

Fig. 4 illustrates the procedure by resolving the inconsistency that occurs in Fig. 3. Suppose solution $(v_1^2 : v_2^1, v_3^2)[c = 6, q = 4.4]$ is chosen as the solution of v_1, which has arrival time $a(v_1) = 2$. At this point, suppose options v_2^1 and v_3^2 of node v_2 and v_3 have also been chosen as solutions of them, respectively. Their arrival times are shown in the figure. Recall that v_2^1 and v_3^2 are based on different options of v_4 in the previous relaxation stage. Now, when consistency restoration considers the solution for multi-fanin node v_4, all of its three options are evaluated. With related characteristic values known for each option of v_4, all gate and wire delays between v_4 and its fanins can be calculated, which are presented in the table in Fig. 4. Then,



	v_4^1	v_4^2	v_4^3
v_2^1	1.2	3	2.2
v_3^2	3	2	2.2

Fig. 4. Multi-fanin node v_4 is under evaluation in consistency restoration procedure. Arrival time and solutions are propagated forward in topological order. Given the solutions v_2 and v_3 picked, the three options of v_4 have arrival times evaluated. Option v_4^3 is the winner for v_4's solution due to its larger slack.

the arrival time at v_4 for each of its options are calculated as:

$$a(v_4^1) = \max\{3.6 + 1.2, 3.6 + 3\} = 6.6,$$

$$a(v_4^2) = \max\{3.6 + 3, 3.6 + 2\} = 6.6,$$

$$a(v_4^3) = \max\{3.6 + 2.2, 3.6 + 2.2\} = 5.8.$$

The slacks for the options are $s(v_4^1) = 8.2 - 6.6 = 1.6$, $s(v_4^2) = 8.0 - 6.6 = 1.4$, and $s(v_4^3) = 7.9 - 5.8 = 2.1$. Option v_4^3 is chosen as the solution of v_4 due to the largest slack. The corresponding arrival time $a(v_4^3) = 5.8$ is set as the arrival time at v_4 at this point.

The pseudo code of $ConsistencyRestoration(G)$ is given in Algorithm 3. Denote by A the preset arrival time at the primary inputs. Lines 1 to 3 choose the solution

Input : combinational circuit G and cell library L
Output : the solution of size and V_t assignment for all gates in G

1 **for** $v_i \in I(G)$ **do**
2 $\quad solution(v_i) \leftarrow \arg\max_{k \in options(v_i)} q(v_i^k) - A$;
3 $\quad a(v_i) \leftarrow A$;
4 **for** $v_i \in G - I(G)$ *in topological order* **do**
5 $\quad maxSlack \leftarrow -\infty$;
6 $\quad$ **for** *every option* v_i^k *of* v_i *in* X_i **do**
7 $\quad\quad a(v_i^k) \leftarrow \max_{v_j \in fanin(v_i)} \big(a(v_j) + D(v_j, v_i^k)\big)$;
8 $\quad\quad$ **if** $\big(q(v_i^k) - a(v_i^k)\big) > maxSlack$ **then**
9 $\quad\quad\quad maxSlack \leftarrow \big(q(v_i^k) - a(v_i^k)\big)$;
10 $\quad\quad\quad \hat{k} \leftarrow k$;
11 $\quad\quad$ **end**
12 $\quad solution(v_i) \leftarrow v_i^{\hat{k}}$;
13 $\quad a(v_i) \leftarrow a(v_i^{\hat{k}})$;

Algorithm 3: *ConsistencyRestoration*(G)

with the maximum slack for each of the gates that are only driven by primary inputs. Lines 4 to 13 assign a solution to every other gate v_i in the circuit. Lines 6 to 11 calculate the gate and wire delay between every option of v_i and the gates on its fanin, and then compare the slacks of v_i's options. The option with the maximum slack is picked as its solution. The time complexity of *ConsistencyRestoration*(G) is dominated by the loop between line 4 and 13. Clearly, it runs in $O(|V|mb)$ time, where b is the maximum fanin for any gate.

3. Phase II: Iterative Refinement

The solution obtained at the end of phase I is based on the RAT (required arrival time) at each node estimated in the relaxation stage. Because of the relaxation, the estimation may be inaccurate and thus compromise the solution quality. The disadvantage of phase I is compensated by an iterative refinement procedure in phase

II. Each iteration of phase II consists of a backward search followed by a forward search. The backward search inherits the implementations of all *multi-fanin nodes* from the forward search in the previous iteration and keeps them unchanged. At the same time, it finds the non-inferior gate implementations of all *single-fanin nodes*. In the subsequent forward search, the implementations of all multi-fanin nodes are further improved in term of the objective function. Iterating between these two coupled procedures leads to monotonic increasing of slack. History consistency is maintained throughout all refinement iterations.

The necessity of phase II can be demonstrated by a simple example depicted in Fig. 5. After the relaxation stage in phase I, it is likely that G_1 is chosen to be $6\times$ based on $8\times$ size of G_0 while G_4 is set to be $3\times$ based on $6\times$ size of G_0, i.e., there is inconsistency at G_0. The restoration stage of phase I picks a single size for G_0, e.g., $7\times$, in order to balance the timing through G_1 and G_4. Evidently, $7\times$ G_0 presents smaller load to G_1 than $8\times$ G_0, according to which G_1 and G_2 size down to $5\times$ and $3\times$, respectively, at the beginning of phase II. In the following iterations, G_1 and G_2 change to their best sizes $4\times$ and $2\times$, respectively.

Each pass of backward search is initialized with a unique solution obtained from the previous forward search. Then, it traverses the circuit from primary outputs towards primary inputs and generates solutions at each node in a fashion similar to the *ConsistencyRelaxation*. The key difference occurs at the multi-fanin nodes. At each multi-fanin node, a set of possible solutions are generated, but only the one that is the same as the initial solution is further propagated to its fanin gates. This can be illustrated by reusing the example in Fig. 4. Assume that v_4^3 is the unique solution at v_4 obtained from the previous iteration/phase, i.e., v_4^3 is in the initial solution for the backward search. Possible solutions for every implementation of v_4 are generated as in *ConsistencyRelaxation*, but only v_4^3 is further propagated toward v_2 and v_3. By

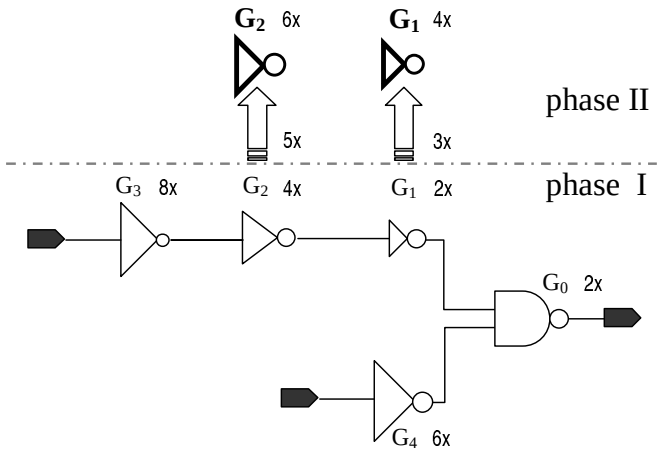


Fig. 5. Suppose after the relaxation stage, gate G_1 and G_2 are $6\times$ and $4\times$, respectively, based on G_0 of size $8\times$. G_3 has a fixed size of $1\times$. Restoration stage chooses the size $7\times$ for G_0 to balance the timing through G_1 and G_4. Then, G_1 needs to be sized down after the restoration stage to account for smaller load, which happens at the beginning of phase II. In phase II, G_1 and G_2 gradually size down to their best sizes $4\times$ and $2\times$, respectively.

doing so, the history consistency is preserved in the backward search. The required arrival times for the other solutions, which are different from the initial solutions, e.g., v_4^1 and v_4^2, will be useful in the subsequent forward search.

The forward search inherits a set of gate solutions at each node from the previous backward search. Note that there is a unique RAT (required arrival time) associated with each implementation of a gate according to Property 1. The forward search decides a single implementation for each gate during a from-PI-to-PO traversal with the same method as the *ConsistencyRestoration* (see Section b).

The pseudo code for iterative refinement, looping between backward search and forward search, is given in Algorithm 4. Lines 2 to 9 present backward search. As shown by line 4, when a multi-fanin node is involved in the merging, only its solution inherited from the previous iteration is used. Lines 10 to 18 presents forward search. As by line 16, the option with maximum slack is chosen as the solution. Obviously, the

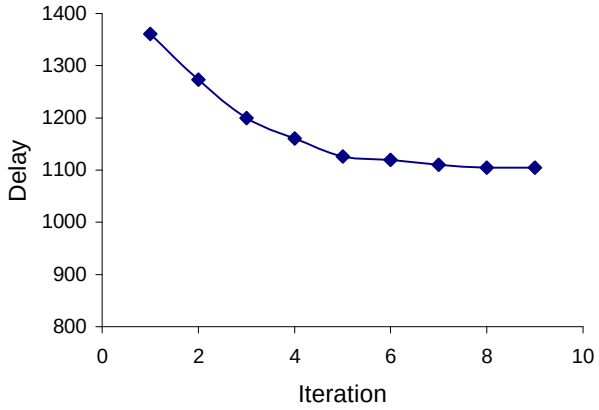


Fig. 6. Circuit delay monotonically decreases by iterations. Delay converges at iteration 8 in ISCAS85 benchmark circuit c2670.

time complexity of each refinement iteration (between line 2 and 18) is still $O(|V|m^2b)$ - the same as $ConsistencyRelaxation$.

Refinement iterations reduce the circuit delay monotonically. This conclusion is easily justified by line 16 in Algorithm 4, since the selection of each new solution always yields a slack equal or higher than previous one. This conclusion ensures the convergence of the refinement iteration, which gradually improves the slack. An example of optimization convergence is shown by the curve in Fig. 6 from an experiment on ISCAS85 benchmark circuit c2670. Based on the experiments on ISCAS85, ITC99, and IWLS 2005 benchmark circuits, the refinement converges within 10 iterations, and there is no correlation observed between the size of the circuits, the variation of cells in the library, and the number of refinement iterations before convergence.

4. Iterative Joint Relaxation and Restriction

Restricted bi-directional search improves the solution starting from the initial one found in relaxation stage. Although the convergence process is systematic, the quality of the converged solution depends on the initial optimization phase, which may come up with an inferior starting solution and lead to a local optimum. We tackle the

Input : combinational circuit G and cell library L
Output : size and V_t assignment for all gates in G

repeat
  $\chi_i \leftarrow \emptyset, \forall v_i \in V$;
  for $v_i \in G$ *in reverse topological order* **do**
    $\chi_{fanout(v_i)} \leftarrow merge(\{\chi_j | v_j \in fanout(v_i)\})$; //for the multi-fanin nodes $\in fout(v_i)$, only their solutions inherited from the previous forward search are involved in this merging;
    for *every option* v_i^k *of* v_i *in* X_i **do**
      $\hat{h} \leftarrow \arg\max_{h \in \chi_{fout(v_i)}} q\left(v^h_{fout(v_i)}\right) - D\left(v_i^k, v^h_{fout(v_i)}\right)$;
      $q(v_i^k) \leftarrow q\left(v^{\hat{h}}_{fout(v_i)}\right) - D\left(v_i^k, v^{\hat{h}}_{fout(v_i)}\right)$;
      $\chi_i \leftarrow \chi_i \cup \left(v_i^k : v^{\hat{h}}_{fout(v_i)}\right)[c(v_i^k), q(v_i^k)]$;
    prune the solution set at v_i by rule in (2.1);
  for $v_i \in I(G)$ **do**
    $solution(v_i) \leftarrow \arg\max_{k \in options(v_i)} q(v_i^k) - A$;
    $a(v_i) \leftarrow A$;
  for $v_i \in G - I(G)$ *in topological order* **do**
    for *every option* v_i^k *of* v_i *in* X_i **do**
      $a(v_i^k) \leftarrow \max_{v_j \in fanin(v_i)} \left(a(v_j) + D(v_j, v_i^k)\right)$;
    $\hat{k} \leftarrow \arg\max_{k \in X_i} \left(q(v_i^k) - a(v_i^k)\right)$;
    $solution(v_i) \leftarrow v_i^{\hat{k}}$;
    $a(v_i) \leftarrow a(v_i^{\hat{k}})$;
until *no improvement*;

Algorithm 4: $IterativeRefinement(G)$

possible inferior starting point issue by strategically searching for better starting points. This is done by conducting multiple iterations of relaxation and restriction.

Recall that during an iterative bi-directional search after one relaxation procedure, the implementation options for multi-fanin gates are constrained to those inherited from previous bi-directional search iteration. However, due to the policy that all options at a multi-fanin gate are evaluated no matter if they are considered or not, solutions at multi-fanin gates are able to systematically improve during re-

stricted iterative search. We call this set of options, which are picked as solutions for a multi-fanin gate during the forward searches in a refinement phase, the used options of the gate in the current JRR iteration. The improvement, which is gained by the used options of multi-fanin gates, not only leads to better solutions, but also provides crucial information about options that can improve the timing of critical paths. This is because forward search always looks for the solution with largest slack at every gate.

Based on the observation above, multiple iterations of relaxation and restriction can leverage the critical path timing information in bi-directional search. Specifically, in the second or later JRR iteration, the option set of every multi-fanin gate is filled with the used options of the gate during the iterative search. For example, in Fig. 5, after phase I and II finish in current JRR iteration, the used options of the multi-fanin gate G_0 are size 7×, 6×, and 5×. These size options replace the former options in the option set X_0 of multi-fanin gate G_0, and are used in the next JRR iteration. This way, each JRR iteration is guided by the previous one for a better starting point. This strategy is reflected in Algorithm 1, where the option sets at multi-fanin gates are updated at the end of each JRR iteration. Correspondingly in Algorithm 2, 3, and 4, only solutions in the option set X_i at each multi-fanin gate are evaluated during relaxation, restoration, and iterative refinement, respectively.

The option set at a multi-fanin gate evolve from one JRR iteration to another. A better starting point given by the previous iteration provides the current iteration with a more accurate timing over the reconvergent paths, thus a better set of used options can be generated and provided to the next iteration. It is clear that this evolution monotonically improves the solution at the end of each JRR iteration. This conclusion simply follows from the fact that each JRR iteration inherits all the best solution from previous JRR iteration, and produce solutions no worse than that in

previous iteration. In practice, according to what we observe in experiments, the number of JRR iterations is usually less than or equal to three. The improvement gained by IJRR is shown in the experiment section for each benchmark circuits.

D. Timing-constrained Power Optimization

The algorithm described in Section C can be extended to simultaneously handle timing and power by Lagrangian relaxation. This section will show how to use the algorithm of Section C to solve the problem of timing-constrained power minimization, which is formally stated as follows.

Timing-Constrained Power Optimization: Given a combinational logic circuit $G(V, E)$, arrival times at its primary inputs, required arrival times at its primary outputs and a cell library, select an implementation option (a size and threshold voltage) for each gate to minimize the power under timing constraints, i.e.,

$$
\begin{aligned}
\min \quad & \sum_{v_i \in V} p(v_i) & \\
\text{s.t.} \quad & q(v_i) \geq a(v_i), & \forall v_i \in I(G) \\
& q(v_i) \geq q(v_j) + D(v_j, v_i), & \forall (v_j, v_i) \in E \\
& w_i \in W_i, & \forall v_i \in V \\
& u_i \in U_i, & \forall v_i \in V
\end{aligned}
$$

where w_i and u_i are the size and the V_t level of gate v_i, respectively.

This constrained optimization problem can be solved using the iterative joint relaxation and restriction method as in the previous sections. However, because of the constraints, each gate solution needs to be characterized by a higher number of variables, which makes the size of solution sets very large (due to low pruning ratio).

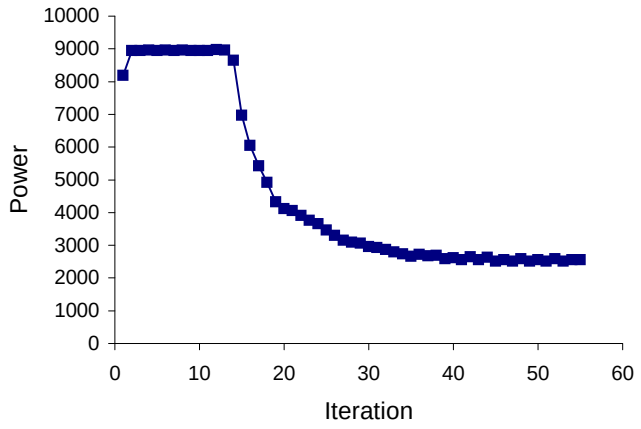


Fig. 7. Convergence of subgradient method used in updating Lagrangian multipliers on ISCAS85 benchmark circuit c2670.

Thus, directly applying IJRR on this problem can be computationally prohibitive. Therefore, we employ Lagrangian relaxation to reduce the dimensionality of the problem. The timing-constraints can be transformed into a part of the cost function with penalty terms for constraints (the Lagrangian function) through Lagrangian relaxation as in [6]. More specifically, we introduce a Lagrangian multiplier μ for each timing constraint. Then, the Lagrangian function is given as:

$$\phi(\mathbf{w}, \mathbf{u}, \mathbf{a}, \mathbf{q}; \mu) = \sum_{v_i \in V} p(v_i) + \sum_{v_i \in I(G)} \mu_{i0}(a_i - q_i) + \sum_{(v_j, v_i) \in E} \mu_{ji}(q_j + D(v_j, v_i) - q_i), \tag{2.2}$$

where $\mathbf{w}$, $\mathbf{u}$, $\mathbf{a}$, and $\mathbf{q}$ are the vectors of gate sizes, gate V_t levels, arrival times, and required arrival times, respectively.

By performing algebraic transforms as in [6], the number of variables is reduced and the Lagrangian function is simplified to

$$\phi(\mathbf{w}, \mathbf{u}; \mu) = \sum_{v_i \in V} p(v_i) + \sum_{(v_j, v_i) \in E} \mu_{ji} D(v_j, v_i). \tag{2.3}$$

Then, the Lagrangian relaxation subproblem with given multiplier values is

formed as:

$$\begin{aligned} \min \quad & \phi(\mathbf{w}, \mathbf{u}; \mu) & \\ \text{s.t.} \quad & w_i \in W_i, & \forall v_i \in V \\ & u_i \in U_i, & \forall v_i \in V \end{aligned}$$

Input : combinational circuit G and cell library L
Output: size and V_t assignment for all gates in G

```
1 repeat
2 |  Update Lagrangian multipliers μ according to static timing analysis on
  |  current solution of G;
3 |  SizeVt_sum(G);
4 until improvement < σ in current iteration (outer loop);
```

Algorithm 5: $SizeVt\_PowerDelay(G)$

By Lagrangian relaxation, the timing-constrained power optimization problem becomes two problems: one is the Lagrangian subproblem and the other is the Lagrangian dual problem, in which the value of the Lagrangian multipliers need to be decided. The Lagrangian dual problem tunes the multipliers to maximize the minimum value of the Lagrangian function enabled by optimal size and V_t options. It can be solved using subgradient method [21], which iteratively updates the values of the multipliers in an outer loop. The Lagrangian subproblem is solved by an algorithm similar to that from Section C in the inner loop. The overall algorithm flow for solving the timing-constrained power optimization problem is outlined in Algorithm 5.

Subgradient method is effective in updating the Lagrangian multipliers. An example of Lagrangian iteration convergence is shown in Fig. 7, in which the power converges after 50 iterations on ISCAS85 benchmark circuit c2670.

The subroutine $SizeVt\_sum(G)$ in Algorithm 5 solves the Lagrangian subproblem like the algorithm from Section C except the difference on the objective function

Input : combinational circuit G and cell library L
Output : size and V_t assignment for all gates in G

$X_i \leftarrow \{\text{all possible options of } v_i\}, \forall v_i \in V$;
repeat
  repeat
    for $v_i \in G$ *in reverse topological order* **do**
      for *every option* v_i^k *of* v_i *in* X_i **do**
        for $v_j \in fout(v_i)$ **do**
          $\hat{h} \leftarrow solution(v_j)$;
          if $fanin(v_j) == 1$ *OR* $refine\_iter == 1$ **then**
            $\hat{h} \leftarrow \arg\min_{h \in X_j} \left(f(v_j^h) + \mu_{ij} D(v_i^k, v_j^h)\right) + p(v_i^k)$;
          end
        $f(v_i^k) \leftarrow \sum_{v_j \in fouts(v_i)} \left(f(v_j^{\hat{h}}) + \mu_{ij} D(v_i^k, v_j^{\hat{h}})\right) + p(v_i^k)$;
        $\chi_i \leftarrow \chi_i \cup (v_i^k : v_{fout(v_i)}^{\hat{h}})[f(v_i^k)]$;
    for $v_i \in I(G)$ **do**
      $solution(v_i) \leftarrow \arg\min_{k \in options(v_i)} f(v_i^k)$;
    for $v_i \in G - I(G)$ *in topological order* **do**
      $solution(v_i) \leftarrow$
      $\arg\min_{k \in X_i} \left(f(v_i^k) + \sum_{v_j \in fanin(v_i)} \left(\mu_{ji} D(v_j, v_i^k) + p(v_j)\right)\right)$;
  until *no improvement in current iteration*;
  $X_i \leftarrow \{v_i\text{'s used options in this iteration}\}, \forall v_i \in \mathcal{V}$;
until *no improvement*;

Algorithm 6: $SizeVt\_sum$

and its computation at each node. When evaluating the objective function $\phi(\mathbf{w}, \mathbf{u}; \mu)$ in the algorithm, we define $f(v_i^k)$ to be the weighted summation of delay and power in the fanout cone covering nodes from v_i to the primary outputs. The minimum summation of delay and power in a fanout cone can be recursively calculated on sub-cones inside it, i.e.,

$$f(v_i^k) = \sum_{v_j \in fanout(v_i)} \min_{h \in options(v_j)} \left(f(v_j^h) + \mu_{ij} D(v_i^k, v_j^h) \right) + p(v_i^k). \tag{2.4}$$

In the backward search, the graph is traversed in reverse topological order. When an option k of node v_i is considered, its downstream solution on v_j - one of its fanout gates - is chosen as

$$\hat{h} = \arg\min_{h \in options(v_j)} \left(f(v_j^h) + \mu_{ij} D(v_i^k, v_j^h) \right) + p(v_i^k), \tag{2.5}$$

to minimize the summation of delay and power.

In the pseudo code of $SizeVt\_sum(G)$ in Algorithm 6, IJRR is implemented by the outer loop, embedded between line 1, 2 and line 18, 19. In each JRR iteration, line 4 to 12 represent backward search. Line 9 selects solutions of a node's fanout gates by Equ (2.5). Line 11 evaluates each implementation option of a gate by Equ (2.4). Note that by condition given in line 8, after the first iteration, the only solution of a multi-fanin node visible to other nodes is the solution inherited from the previous refinement iteration.

Forward search procedure is presented by line 13 to 16 in the pseudo code. At the beginning of forward search, the option of entrance gate v_i with minimum sum of delay and power value $f(v_i^k)$ is chosen as its solution. Line 13 and 14 implement this selection. When any other node is processed, different options of v_i lead to different summation of delay and power on its fanins and in its fanout cone. Again, the option

leading to minimum summation of delay and power is chosen as v_i's solution, i.e.,

$$solution(v_i) = \underset{k \in options(v_i)}{\arg\min} \Big(f(v_i^k) + \sum_{v_j \in fanin(v_i)} (\mu_{ji} D(v_j, v_i^k) + p(v_j)) \Big). \tag{2.6}$$

$SizeVt\_sum$ has the same runtime complexity $O(|V|m^2 b \times num\_iterations)$ as $SizeVt\_MaxSlack$.

E. Solving Lagrangian Dual Problem Effectively

As mentioned in previous section, Lagrangian dual problem asks how to tune the Lagrangian multipliers to maximize the minimal value of the Lagrangian function by optimal gate size and V_t solutions, i.e., to solve the following system.

$$\underset{\mu \geq 0}{\arg\max} \min_{w_i \in W_i, u_i \in U_i, \forall v_i} \phi(\mathbf{w}, \mathbf{u}, \mathbf{a}, \mathbf{q}; \mu), \tag{2.7}$$

where

$$\phi(\mathbf{w}, \mathbf{u}, \mathbf{a}, \mathbf{q}; \mu) = \sum_{v_i \in V} p(v_i) + \sum_{v_i \in I(G)} \mu_{i0}(a_i - q_i) + \sum_{(v_j, v_i) \in E} \mu_{ji}(q_j + D(v_j, v_i) - q_i).$$

In this problem, the independent variables to be decided are the Lagrangian multipliers, which affect the optimal solutions of gate size and v_t levels. Most existing subgradient methods, use only one rough subgradient value to estimate $\frac{\partial \phi}{\partial \mu}$, and they do not take into account of the impact of μ value change on the optimal subproblem solution. Furthermore, they utilize simple multiplier update scheme based on the single subgradient value. Due to these issues, the LR dual problem may be solved with non-trial errors in existing methods.

We propose a new LR dual problem solving method, featuring chain rule in sensitivity computation for $\frac{\partial \phi}{\partial \mu}$, the computation of a spectrum of subgradient, and nonlinear programming problem solving for optimal multiplier values.

The chain rule calculation is carried out when both sides of Equ. (3.2) are differentiated. The partial derivative of the Lagrangian function with respect to a multiplier is calculated with approximation as follows.

$$\frac{\partial \phi}{\partial \mu_{ji}} = \frac{\partial p(v_i)}{\partial \mu_{ji}} + \frac{\partial p(v_j)}{\partial \mu_{ji}} + \left(\mu_{ji} \frac{\partial D(v_j, v_i)}{\partial \mu_{ji}} + (q_j + D(v_j, v_i) - q_i) \right),$$

where $\frac{\partial p(v_i)}{\partial \mu_{ji}}$ and $\frac{\partial D(v_j, v_i)}{\partial \mu_{ji}}$ are due to the change of optimal subproblem solution with the change of the Lagrangian multipliers. They can be calculated by measuring the change on $p(v_i)$ and $D(v_j, v_i)$ under certain perturbation on the multiplier, $\Delta\mu$, i.e., $\frac{\partial p(v_i)}{\partial \mu_{ji}} \approx \frac{\Delta \phi}{\Delta \mu_{ji}}$.

The subgradient spectrum is calculated according to a set of $\Delta\mu$ on each specific multiplier, e.g., $\{0.05, 0.10, 0.15, ...\}$. Therefore, a set of $\frac{\Delta \phi}{\Delta \mu_{ji}}$ values can be calculated corresponding to the set of $\Delta\mu_{ji}$.

Based on the spectrum of subgradients, first order and second order derivatives, i.e., gradient and Hessian, can be obtained. Then, a nonlinear programming problem is formed for Equ. (2.7) and solved by sequential quadratic programming.

F. Runtime-quality Tradeoff

Our systematic approach for timing optimization and timing-constrained power minimization involves iterative techniques on different levels, including Lagrangian relaxation iterations, joint relaxation and restriction iterations, and refinement iterations. These iterations proceed until they converge. One observation is that the iterations close to the convergence point normally do not make significant improvement on solution quality, while they make up a large portion of the total number of iterations. For example, in Fig. 7 Lagrangian relaxation on ISCAS85 c2670 circuit does not reduce much power after iteration 35. Tradeoff between algorithm runtime and solution

quality can be realized by setting improvement thresholds to cut off future inefficient iterations.

In this work, we create two improvement thresholds: $0 < \phi_n < 1$ for Lagrangian iterations and $0 < \phi_r < 1$ for JRR iterations. In the outer loop, we trace the 5 latest Lagrangian iterations that produce solutions satisfying timing constraint. If the reduction of circuit power in the 5 iterations is less than $\phi_n \times 100\%$, the iterations is stopped without converging.

Similarly, inefficient JRR iterations can be avoided too. If the chance of improvement with further JRR iteration is not promising, the JRR iterations are stopped. A JRR iteration consists of both phase I and phase II optimization. We predict the chance of improvement JRR by checking the improvement in phase II - iterative improvement - in current JRR iteration. If the improvement on the Lagrangian function in iterative refinement is less than $\phi_r \times 100\%$, the chance of improvement by next JRR iteration is small, thus, the JRR iterations are stopped in current Lagrangian iteration.

The efficiency of the algorithm is also affected by the delay model used in timing. For the ease of presentation, this paper demonstrated our method on Elmore delay model. In fact, our method accommodates more accurate delay and power models. For example, pin-to-pin gate delay can be applied without introducing extra computational cost. More specifically, before the solution merging operation, the pin-to-pin gate delay can be counted into the required arrival time q of each solution. This way, each pin-to-pin delay is correctly counted in the timing. Furthermore, slew can be considered in timing. In this case, solutions are searched from primary inputs to primary outputs in the consistency relaxation stage, so that the slew is propagated along with the solutions. Now, each solution is characterized by one more value - slew, i.e., the characterization of a solution is 3-dimensional. To cope with the computational

complexity due to the increased number of dimensions, approximation schemes can be applied on the characterization values to control the size of solution sets; alternatively, relaxation techniques, such as Lagrangian relaxation, can be employed to reduce dimensionality. This way, the solution quality is preserved and computational complexity is controlled within a reasonable level.

G. Experiment

In order to validate the effectiveness of our algorithm, we compared it with a state-of-the-art previous work [1]. The work of [1] is centered around slack allocation, which is a linear programming guided by power-delay sensitivity. The size and V_t level are selected for each gate such that its allocated slack can be traded for power reduction. We call this method as SA (Slack Allocation) based approach. Potential advantages of our method over SA are as follows. First, in SA both the timing optimization at the beginning and the slack allocation later requires first-order approximation to the circuit delay and power model, while our method uses table lookup model and has no approximation. Second, numerical optimization in SA produces continuous solution for timing optimization, and then, the solution is rounded up to discrete gate sizes, which is subject to significant error. Our combinatorial algorithm does not need rounding up and thus have no such error. Third, sensitivity-based optimization is more likely to be trapped into local optimum than our combinatorial method.

To see the effect of each technique, we obtained results from our algorithm with different parts: applying our method with initial optimization phase in only one JRR iteration to show the result without phase II - iterative refinement, applying the two-phase algorithm in one JRR iteration to show the result without iterative relaxation and restriction, applying our method with both phases in multiple JRR iterations to

Table I. Comparison on power (μW) and CPU runtime - RT (seconds). All solutions satisfy timing constraints.

			SA [1]		Phase I of JRR		JRR		IJRR		IJRR w/ thresholds	
	Circuit	#gates	power	RT	power	RT	power	RT	power	RT	power	RT
ISCAS85	c432	289	304	2	283	2	269	5	240	11	247	8
	c499	539	617	2	655	5	649	9	496	24	506	14
	c880	340	364	2	359	3	332	5	284	16	284	8
	c1355	579	635	2	783	5	747	10	524	31	524	14
	c1908	722	833	4	871	6	801	15	668	35	681	20
	c2670	1082	969	7	807	9	760	18	733	50	733	24
	c3540	1208	1440	8	1624	12	1512	26	1188	69	1188	36
	c5315	2440	2596	21	2350	24	2099	51	2005	125	2005	68
	c6288	2310	4427	9	4619	21	4358	49	4358	111	4384	64
	c7552	3115	3153	38	2660	32	2528	71	2455	167	2455	94
ITC99	b03	101	93	1	69	1	64	1	61	4	64	2
	b09	105	107	1	104	1	95	2	81	3	83	2
	b10	147	166	1	163	1	152	3	129	6	132	4
	b11	448	560	2	572	5	518	10	444	25	456	14
	b12	827	911	3	763	8	694	15	676	42	696	24
	b14	5524	6674	200	5479	97	5332	194	5215	426	5215	258
	b15	5340	8212	142	7792	85	7457	159	6844	385	6844	228
	b20	10590	13046	745	11262	247	10620	409	10215	1021	10215	434
IWLS05	des_area	3255	3610	38	2979	45	2896	86	2866	203	2875	96
	mem_ctrl	8929	10359	438	7995	194	7867	311	7812	728	7812	348
	spi	2317	2561	20	1977	33	1968	64	1955	160	1955	94
	usb_funct	11848	13060	528	9547	231	9519	348	9507	847	9602	474
	aes_core	15692	17835	1312	15256	400	13934	583	13904	1424	14001	928
	systemcdes	2517	2650	34	2071	32	2054	67	2036	162	2036	76
	tv80	5303	6593	168	5374	86	5169	148	5085	355	5085	180
	ac97_ctrl	9672	10008	192	8908	121	7101	223	6995	602	6995	346
	Average		4299	150	3666	65	3442	110	3337	270	3349	148
	Norm.		1.0	1.0	0.85	0.44	0.80	0.73	0.776	1.79	0.778	0.98

show the full power of the algorithm, and applying the iterative JRR method with improvement thresholds to test runtime-quality tradeoff. In the experiment, timing and power are concurrently optimized with the formulation of minimizing total power, including dynamic and leakage power, subject to timing constraints. We also carried out experiment to verify the efficiency of our method for timing optimization. Both the SA algorithm [1] and our algorithm were implemented in C++. The experiment was conducted on a Windows machine with 2.6GHz Intel core 2 duo CPU and 2GB memory.

The algorithms were tested on ISCAS85, ITC99, and IWLS 2005 benchmark circuits. The circuits are synthesized by SIS [22] and placed by mPL [23] before the optimization. The cell library is based on $70nm$ technology. Each gate has 4 V_t levels and 7 size options (1x, 2x, 4x, 8x, 16x, 24x, 32x), i.e., each gate has 28 implementations. The V_{DD} is set to $0.9V$. For the convenience of algorithm implementation, the Elmore delay model and analytical models for dynamic and

Table II. Comparison on circuit delay (ps) and CPU runtime (seconds).

		SA [1]		JRR	
	Circuit	delay	runtime	delay	runtime
ISCAS85	c432	874	0.1	788	0.3
	c499	705	0.1	629	0.6
	c880	650	0.1	566	0.3
	c1355	787	0.2	716	0.3
	c1908	971	0.4	875	0.6
	c2670	1088	0.4	930	0.42
	c3540	1160	0.6	1084	0.9
	c5315	2149	0.7	2027	0.96
	c6288	3548	0.5	3479	0.84
	c7552	1268	1	1103	1.2
ITC99	b03	285	0.1	237	0.12
	b09	182	0.1	160	0.3
	b10	337	0.1	301	0.12
	b11	722	0.1	656	0.24
	b12	680	0.3	608	0.72
	b14	5724	3	5157	4.2
	b15	6463	3	5985	4.2
	b20	7576	11	6888	11.4
IWLS05	des_area	3253	1	2905	1.8
	mem_ctrl	6503	6	5655	8.4
	spi	3048	1	2651	1.2
	usb_funct	8240	8	7103	9.6
	aes_core	3512	16	3193	17.4
	systemcdes	3713	1	3315	1.2
	tv80	4284	3	3758	3.6
	ac97_ctrl	234	5	198	5.4
	Average	2614	2.4	2345	2.9

leakage power [12] were employed for the experiment. Please note that our algorithm can be directly applied with more accurate models. Wire delay was included in our delay estimation.

The main results of timing-constrained power optimization are summarized in Table I. It compares results on power and CPU runtime. Since the results from all these methods can satisfy timing constraints, the timing data is not included in the table. It can be seen that the initial optimization phase of our algorithm results in 15% less power than SA on average. If we run our complete two-phase algorithm in one JRR iteration, the average power reduction increases to 20%. The two-phase algorithm with multiple JRR iterations reduces the power dissipation further by 22.4%. For both of our initial optimization phase and our 2-phase 1-iteration algorithm, the runtime is significantly less than that of SA. Although SA is faster than our methods

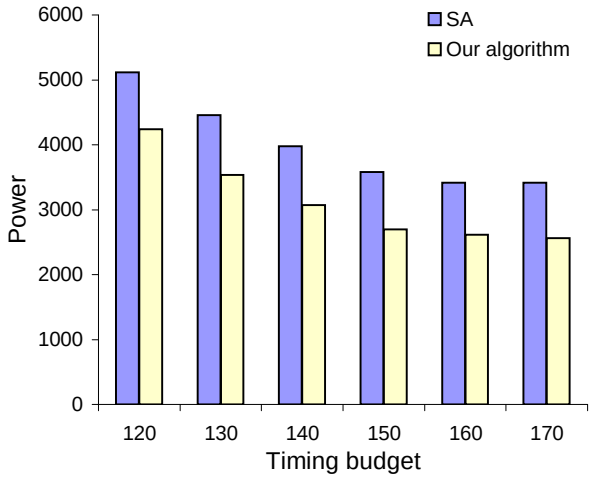


Fig. 8. For different timing budgets, our solutions always yield less power than SA [1] on ISCAS85 benchmark circuit c2670.

on circuits with small size, our methods are much faster than SA on large circuits, because the runtime of our methods is linear to the number of gates in the circuit. For IJRR, the runtime is larger due to multiple iterations. Usually, the number of JRR iterations is no more than 3. The runtime of IJRR is 1.79× of SA on average. A significant speedup (near 2×) to IJRR is obtained by utilizing improvement thresholds for runtime-quality tradeoff. The improvement thresholds are set to $\phi_n = 0.001$ for Lagrangian iterations and $\phi_r = 0.05$ for JRR iterations. Running IJRR with thresholds takes about half of the runtime of IJRR without thresholds, which is comparable to SA and close to single JRR iteration. This improvement on runtime is achieved with negligible difference on solution quality. Specifically, the IJRR with thresholds reduces 22.2% more power than SA, compared to 22.4% by IJRR without thresholds.

The results of timing optimization are summarized in Table II. To obtain the optimal timing solution without any constraint, SA only performs its first part: sensitivity-based delay minimization, while our method applies one JRR iteration without Lagrangian relaxation. The experimental results show that on average our

method outperforms SA by about 270ps with slightly higher runtime.

We also tested the algorithms for different timing constraints on circuit c2670. The results are presented in Fig. 8. When the timing budget increases, the power dissipation decreases. One can see that the power from our algorithm is significantly less than that of SA [1] for all different timing constraints.

H. Conclusion

In this paper, we propose a new algorithm of gate sizing and V_t assignment. Its core idea is Joint Relaxation and Restriction, which employs consistency relaxation and coupled bi-directional solution search. Our approach performs joint relaxation and restriction iteratively. It is a systematic and efficient solution search on DAG. In general, it is a practical approach because it accommodates accurate delay and power models. It can be directly applied with industrial standard cell based designs and its CPU runtime is reasonable. Compared to a state-of-the-art previous work, it leads to about 22% less power dissipation with similar timing performance and CPU runtime.

CHAPTER III

SIMULTANEOUS TECHNOLOGY MAPPING AND CELL PLACEMENT

A. Introduction

In today's technologies, interconnects contribute to significant portion of the overall delay in VLSI circuits. The trend is likely to continue, or worsen, as the technology scaling continues, since the wire delays do not scale as well as cell delays. The interconnect delay depends on the topology and layer assignment, which is determined by the routing step. This freedom available in the routing phase is often insufficient to optimize the circuit for the required performance. The placement and technology mapping steps also have a great impact on the interconnect delay, since the former decides where the locations of the driver and receivers of a net are and the latter decides which nets exist in the design. Consequently, the algorithms for layout-driven technology mapping, timing-driven placement, and physical synthesis have received attention from CAD researchers over the last several years.

Technology mapping problem minimizing metrics such as total cell area for a directed acyclic graph (DAGs) is known to be NP-hard. For relatively simple structures such as trees, however, the problem can be solved optimally in a polynomial time. The technology mapping algorithm to map individual trees rooted at multi-fanout points or primary outputs in a DAG on to a set of cells in a library was first proposed by Keutzer [24]. The algorithm employs dynamic programming technique and runs in polynomial time in the size of the tree, ensuring optimality for the metrics such as total cell-area. Most of the subsequent work employs the same technique to optimize various cost functions involving area, delay, power possibly subject to constraints, as in [25]. The layout-driven technology mapping was proposed by Pedram *et al.*, where

an initial placement of a subject graph and the assumption about the placement of a match was employed to evaluate wire- and cell-delays to derive a delay-optimized mapped netlist [26]. Obvious limitation of the work is that even for a tree, the placement of the subject graph and that of the mapped netlist can be quite different and that there are multiple placement possibilities for a choice at each node in the tree, whereas only one placement, that of the center of gravity based on the locations of choices at fanins and (unmapped) fanouts, is considered. The limitation was partially eliminated in the subsequent work [27], which solved the problem of simultaneous technology mapping and linear placement of trees in polynomial time. However, the assumption about the placement of the cells in a tree in a single row is not practical, since the cells are allowed to be placed in different rows in 2-dimensional (2-D) area. To overcome this limitation, the subsequent work employed iterative technology decomposition, mapping, and placement [28, 29, 30] to place the primitive gates in a given area, perform mapping with assumptions about the placement of a mapped cell, and then place the mapped netlist or derive the placement of the subject graph from the same for the next iteration. Many industrial tools, which perform physical synthesis, are believed to employ similar iterative mapping and placement schemes to improve the delays locally in parts of the circuit. The limitation of such an approach is that it neither ensures optimality nor guarantees convergence, as a different mapping solution leads to a new placement. Thus, the problem of simultaneous technology mapping and 2-D placement even for trees remains unsolved even today. Recently, Wang *et al.* proposed an iterative mapping scheme [31] employing multipliers, similar to those in Lagrangian relaxation technique, to optimize the area/power under fixed cell-delay model; the wire-delays based on the placement, however, are not considered.

Similar to technology mapping, placement for general graphs to optimize useful objectives is a difficult problem and has been well researched over the last few decades;

see [32] for the recent literature survey. The placement of special structures such as trees, however, can be performed in a polynomial time optimizing certain metrics. For example, Fischer *et al.* presented $O(n \log n)$ algorithm for the optimal placement minimizing the sum of weighted edge-lengths for a tree with n leaves [33]; recent work includes a linear time algorithm to minimize the sum of half-perimeter wirelengths for all nets in a tree[34]. The special case of linear placement for trees is also studied well and several exact polynomial time algorithms exist to minimize total wirelength or the cutwidth; for instance, Yannakakis's algorithm [35] employed in [27] to perform simultaneous mapping and linear placement. However, the problem of delay-optimal placement for trees seem to have received relatively less attention in the published literature, despite the potential usefulness of the solution.

Since the technology mapping and placement have great impact on the overall delays in the circuit, exploring these two spaces simultaneously can result in circuits with better delays than the conventional approach of searching those sequentially, which results in the search in a relatively small solution space. A fundamental contribution of this work is an exact polynomial time, $O(nm^2 f_{max} P_{max}^2)$, algorithm for delay-optimal simultaneous technology mapping and 2-D placement of trees, where n, m, f_{max}, and P_{max} are the number of nodes in the tree, the number of candidate locations in 2-D area, maximum fanin over all the matches at any node, and the maximum number of matches at any node in the tree, respectively. The algorithm is based on the extension of an exact polynomial time, $O(nm^2 f_{max})$, delay-optimal placement algorithm for trees, which is another important contribution. To optimize timing in directed acyclic graphs (DAGs), we propose an iterative algorithm, based on Lagrangian relaxation (LR) technique, which employs the simultaneous technology mapping and placement in the inner loop. The comparison of results on ISCAS'85 benchmarks, with a cell library characterized for a 70 nm technology, due to the algo-

rithm with those due to the conventional iterative delay-oriented mapping in SIS [36] and timing driven placement mPL [37] shows more than 60% slack improvement with 7 times speed-up in runtime, on an average, implying that the proposed algorithms are practical and can be employed to optimize timing during physical synthesis.

The rest of the paper is organized as follows. Section B describes the formal notation employed in this article. Section C presents an algorithm for delay-optimal placement of trees, whereas Section D extends the algorithm to perform delay-optimal simultaneous technology mapping and placement. Section E briefly describes the algorithm based on LR for simultaneous mapping and placement for DAGs. Section G discusses the results due to the algorithms and compares them with those due to the competitive approach, and Section H concludes the paper.

B. Preliminaries

Traditionally, a technology independent Boolean network is first decomposed into a circuit containing only primitives such as two-input NANDs and inverters, which are then mapped on to standard cells in a library during the technology mapping to create a mapped netlist. Subsequently, the placement is carried out on the mapped netlist to assign each cell a location in a given area. The graph theoretic structure underlying either the Boolean network or the technology decomposed circuit or the mapped netlist is a DAG $G(V, E)$, where a node $v \in V$ represents a standard cell in case of mapped netlist or a primitive in case of the technology decomposed circuit. The primary inputs and outputs of the DAG are denoted by $input(G)$ and $output(G)$, respectively. Each directed edge $e(v_i, v_j) \in E$ represents a net whose driver (receiver) is the standard cell represented by v_i (v_j). Each node $v_i \in V$ is associated with the actual (required) arrival time a_i (q_i); the slack for the node is computed as $q_i - a_i$.

The delay between nodes v_i and v_j is denoted by $d(v_i, v_j)$, which comprises the cell delay, $d^{cell}(v_i)$, and the wire delay, $d^{wire}(e(v_i, v_j))$. For a primary input i to the circuit, $d^{cell}(i)$ is simply the actual arrival time of that input. The delay of an input-output path π is denoted by $d(\pi) = \sum_{(v_i,v_j)\in\pi} d(v_i, v_j)$. The slack of the path is computed as $s(\pi) = q - d(\pi)$, where q is the required arrival time at the output of the path. Paths with the minimum slack are critical paths in the circuit.

We introduce a polynomial time algorithm for the delay-optimal placement of a tree in this section and describe its extension to simultaneous mapping and placement in the next. A rooted tree is a tree $T(V_T, E_T)$, with one of its nodes designated as a root. The tree may be a part of a DAG $G(V, E)$, *i.e.*, $V_T \subseteq V, E_T \subseteq E$. The inputs to the tree, also referred to as the leaves, have fixed locations and so does the root of the tree. We want to place the tree in a layout area, which is divided into bins or tiles, similar to those in conventional global placement [37]. Specifically, we want to assign each node $v \in V_T$ a bin (x, y). There are several possible placements leading to different delays, since the wire- and cell-delays are functions of the locations of the driver and the receiver. Among these placements, we want to find the one with the minimum delay. Formally, the problem of delay minimization during tree placement can be stated as follows:

Problem definition B.1 *Given a tree $T(V_T, E_T)$, and a set of candidate locations, Z_i, for each node v_i, minimize*

$$\max_{\pi\in input-root\ \ paths} d(\pi),$$

s.t.

$$(x_i, y_i) \in Z_i, \quad \forall v_i \in V_T.$$

The delay-optimal tree placement problem has optimal substructure, *i.e.*, the

delay-optimal placement for a tree rooted at a node v contains the delay-optimal placements for subtrees rooted at its fanins, since, otherwise we can change the placement for the subtrees to yield delays smaller than that due to the delay-optimal placement for the tree, leading to a contradiction. We exploit this optimal substructure property to come up with a tree placement algorithm based on the dynamic programming.

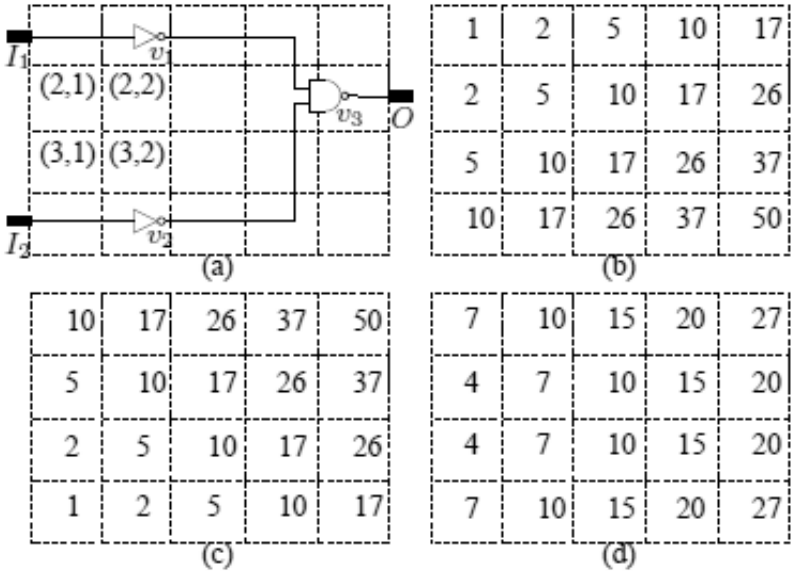


Fig. 9. (a) A tree with fixed i/os I_1, I_2, O and cells v_1, v_2, and v_3, placeable in 4×5 grid. (b) The placement-delay table for v_1, where the entry in bin (i, j) indicates the delay of the subtree rooted at v_1, when v_1 is placed in (i, j). (c) The placement-delay table for v_2. (d) The placement-delay table for v_3, obtained by using the optimal locations for fanins v_1 and v_2.

C. Tree Placement Algorithm

The tree placement algorithm has two phases: first phase of bottom-up solution generation and the second phase of actually choosing a placement from those solutions, given the fixed location of the root. The first phase traverses the tree in a topologi-

cal order and stores the delays due to optimal placements for subtrees rooted at all nodes, assuming that the roots are fixed in all possible candidate locations. It can be explained employing the example in Fig. 9(a), where a tree with fixed locations for inputs I_1, I_2, and an output O is shown. The cells v_1, v_2, and v_3 are to be placed in a 4×5 grid so that delay on any path from I_1 or I_2 to O is minimum. For the sake of illustration, the following assumptions are made: inputs arrive at 0; the cell-delay for v_1, v_2, v_3 is 1; and the wire-delay equals the square of Manhattan distance between nodes, which is same as the Elmore delay model with unit resistance and capacitance per unit wire-length. Consider a location $(3, 1)$ for the cell v_1: the delay for the subtree rooted at v_1 is sum of the arrival time at I_1, $d^{cell}(I_1) = 0$, the wire-delay from I_1 to v_1, $d^{wire}(e(I_1, v_1)) = (|1-3| + |1-1|)^2 = 4$, and the cell delay for v_1, $d^{cell}(v_1) = 1$. Therefore, the optimal delay of the subtree rooted at v_1, when the location of v_1 is fixed at $(3, 1)$, is 5. Similarly, when v_1 is fixed at $(2, 1)$, the optimal delay for the subtree rooted at v_1 is 2, since the wire delay $d^{wire}(e(I_1, v_1)) = (|1-2| + |1-1|)^2 = 1$ and the cell-delay is also 1. There are 20 possible locations for v_1 and for each of those locations, the optimal delays for the subtree rooted at v_1 are shown in Fig. 9(b) depicting a table, referred to as a placement-delay table. Notice that the delay values in bins $(3, 1)$ and $(2, 1)$ are 5 and 2, respectively, as explained before; the delay values in other bins are derived similarly. The placement-delay table for v_2 can be constructed in a similar fashion and is depicted in Fig. 9(c). The tables are constructed for nodes v_1 and v_2 before generating that for v_3, since these nodes occur before v_3 in the topological order. Now, consider the construction of the placement-delay table for v_3. For each position (x, y) for v_3, we consider the optimum location of v_1 and v_2 to compute the delay. Therefore, when v_3 is placed in $(4, 1)$, the location chosen for v_2 is also $(4, 1)$, since that yields the minimum delay of the path from I_2 to v_3, which is 2 (1, optimal delay for the subtree at v_2, when v_2 is fixed at $(4, 1)$, $+ 0^2$, wire-delay, $+$

1, cell-delay for v_3). Similarly, two locations $(3,1)$ and $(2,1)$ for v_1 result in the least path delay of 7. Choosing either of those leads to the same delay, which is minimum for the path from I_1 to v_3, when v_3 itself is placed at $(4,1)$. The overall delay for the subtree rooted at v_3, when it is placed in $(4,1)$ is $max(2,7) = 7$; this is reflected in the bin $(4,1)$ in placement-delay table for v_3, shown in Figure 9(d). Other entries in the table are derived similarly. Thus, each entry at (x,y) location in placement-delay table for a node v corresponds to the optimal delay of the subtree rooted at v, when v itself is fixed at (x,y), and is computed as follows:

$$\begin{aligned} a_v(x,y) &= max_{i \in fanin(v)}\{min_{\forall(x_i,y_i)\text{locations of } i} \\ &\quad \{a_i(x_i,y_i) + d^{wire}(e(i,v)) + d^{cell}(v)\}\} \end{aligned} \tag{3.1}$$

The following proposition states the optimality of the delay values stored in placement-delay table for all nodes.

Proposition 1 *The delay $a_v(x,y)$ is the optimal delay for the placement of the subtree rooted at v, when v is fixed at (x,y).*

Proof 1 *We use induction on the depth of the node. Basis step: depth = 1. In this case, all fanins to the node v are from fixed leaf nodes. If v is also fixed at (x,y), then there is only one possible delay for the subtree rooted at v and therefore, $a_v(x,y)$ is trivially optimal. Induction step: depth $>$ 1. Assume that the proposition is true for all the nodes with depth $< k$. We will prove that it is true for a node with depth k. Consider such a node v, for which $a_v(x,y)$ is given by Eq. (3.1). Suppose $a_v(x,y)$ is not optimal. This implies that there exist some fanin node i, for which $a_i(x_i,y_i)$ is not optimal - a contradiction, since the depth of i is $< k$, because of which $a_i(x_i,y_i)$ is optimal. Therefore, $a_v(x,y)$ must also be optimal.*

After the construction of placement-delay tables, the second phase of the algorithm proceeds, traversing the tree in a reverse topological order to choose the locations for v_3, v_2, and v_1. Since the root node O is fixed in the location $(2, 5)$, the optimal location of v_3, which results in the minimum delay is $(2, 3)$, yielding the delay of 14 (10, $a_{v_3}(2, 3)$, *i.e.,* delay of the subtree rooted at v_3, + 2^2, wire-delay from $(2, 3)$ to $(2, 5)$). The optimal locations of v_1 and v_2, which resulted in the delay of 10 for the subtree rooted at v_3 are $(1, 3)$ and $(4, 3)$, respectively; these can be found out in a constant time by storing additional information along with the placement-delay table. Thus, the optimal placement for the tree is as follows: $v_1(x_{opt}, y_{opt}) = (1, 3)$; $v_2(x_{opt}, y_{opt}) = (4, 3)$; and $v_3(x_{opt}, y_{opt}) = (2, 3)$.

The pseudo-code for the tree placement is shown in Algorithm 7. It processes nodes in the tree in a topological order and for each node v_j, it considers all the possible locations (x_j, y_j). For each of those placements, it finds out the placement for each fanin resulting in the minimum delay. This operation requires $O(m \times |fanin(v_j)|)$ time, since for each node, we store the arrival times, $a_v(x, y)$, indexed by location (x, y) and these represent the optimal delays for the placement of the subtree rooted at v, when v itself is placed at (x, y). Considering the minimum arrival times from the fanins, the arrival times for the delay-optimal placements of the subtree rooted at v_j are computed and stored by indexing on the locations (x_j, y_j). Other auxiliary information such as the optimal locations of fanins for each placement of v_j is also stored so that the delay-optimal placement can be created, employing reverse topological traversal, after all the nodes are processed. The amount of memory required to store the optimal delay values and other auxiliary information for an entire tree is $O(nmf_{max})$, for the tree containing n nodes, each with m placement possibilities, and maximum fanin of f_{max}. The time complexity of the algorithm is $O(nm^2 f_{max})$, since it is dominated by the search for the optimal-delay placement for each fanin of

1: **for all** v_j in V_T in topological order **do**
2: **for all** tiles (x_j, y_j) in candidate locations set of v_j **do**
3: **for all** fanins v_i of node v_j **do**
4: Choose (x_i, y_i), the location for v_i, which yields the minimum value for delay $d(v_i, v_j) + a(v_i)$.
5: **end for**
6: Update arrival time:
$a_{v_j}(x_j, y_j) = \max_{v_i \in fanin(v_j)}(d(v_i, v_j) + a(v_i))$
7: Record corresponding optimal fanin locations:
$\forall v_i \in fanin(v_j)$, $l_{opt}(v_i, v_j, x_j, y_j) = (x_i, y_i)$
8: **end for**
9: **end for**
10: **for all** v_j in V_T in reverse topological order **do**
11: **if** v_j != root(T) **then**
12: f = fanout(v_j)
13: placement(v_j) = $l_{opt}(v_j, f, x_f, y_f)$
14: **end if**
15: **end for**

Algorithm 7: $PlaceTree(T)$

a given node.

Proposition 2 *The tree placement procedure shown in Algorithm 7 returns optimal-delay placement.*

Proof 2 *During the topological traversal, $l_{opt}(i, v, x, y)$ is populated and it stores the delay-optimal locations for fanins i for all possible locations (x, y) of all nodes $v \in V_T$. Considering the location of the root, which is fixed, the reverse topological traversal, assigns the optimal locations to all nodes from those stored in $l_{opt}(i, v, x, y)$ based on the location of their fanouts.*

Even though we explained the tree placement algorithm employing constant and Elmore delay models for cell- and wire-delays, respectively, the algorithm ensures delay-optimality with other delay models also. For instance, asymptotic waveform evaluation (AWE) can be employed to compute wire-delays and without any changes, the algorithm still ensures the optimality. Similarly, the load-dependent cell-delay models can be used, with slight changes in the computation of delays, without affecting the optimality.

D. Delay-optimal Simultaneous Technology Mapping and Placement for Trees

Delay-optimal tree placement algorithm presented in the previous section can be extended to perform simultaneous technology mapping and placement. Traditionally, technology mapping transforms a Boolean network containing primitive gates such as 2-input NANDs and inverters into an implementation based on the set of cells in a library and is carried out in two steps: matching and covering. For conventional delay oriented technology mapping employing load-dependent delay model [36], the matching phase processes each node in a topological order and stores a piecewise linear load-delay curve corresponding to mapping solutions due to non-inferior

1: **for all** nodes v_j in topological order **do**
2: **for all** matches g_j corresponding to cells in the library **do**
3: **for all** bins $(x_j, y_j) \in \mathcal{Z}_j$, set of candidate locations, **do**
4: **for all** fanins i of pattern g_j matched at node v_j **do**
5: Choose (g_i, x_i, y_i) that gives the minimum value of delay $d(v_i, v_j) + a(v_i)$.
6: **end for**
7: Update arrival time:

$$a_{v_j}(g_j, x_j, y_j) = \max_{i \in fanin(g_j)} (d(v_i, v_j) + a(v_i))$$

and record corresponding solutions of all its fanins:

$\{(g_i, x_i, y_i) | i \in fanin(g_j)\}$

8: **end for**
9: **end for**
10: **end for**

Algorithm 8: $MatchPlaceTree(T)$

matches, found either by structural or Boolean techniques, at that node. In the covering phase, the mapping solution is generated by a reverse topological traversal, by selecting the minimum delay matches for given loads. For trees, this algorithm results in delay-optimal solution, ignoring the wire-delays based on placement. To account for placement-based wire-delays, the approaches in the literature such as [26, 29, 28] either assume that the match is placed at some location or iterate between the mapping, placement, and technology decomposition steps. Obviously, these approaches do not claim delay-optimality considering the wire-delays based on the actual placement, even for trees.

To overcome the limitations of the previous approaches, we propose a simultaneous mapping and placement algorithm, which returns the delay-optimal mapped netlist and its placement in a polynomial time for a tree. The algorithm relies on the matching step to store both the mapping choices and their delay-optimal placements, whereas the covering phase, which is same as that in the traditional algorithm, generates a mapping solution with a reverse topological traversal by selecting the delay-optimal choices. Since all the mapping choices and their delay-optimal placements are considered, the final mapping and placement solution is optimal. The novelty of the algorithm lies in its polynomial time and space complexities, despite storing the delay-optimal placements for all the mapping solutions. The algorithm makes the same assumption, as in previous section, that the locations of the inputs and output of a tree are fixed beforehand. The inputs to the tree are either the primary inputs or outputs from the multi-fanout roots of other trees in the DAG; the output is either a primary output or serves as an input to other trees.

The pseudo-code for the matching step is shown in Algorithm 8. Similar to that in conventional approaches, it processes nodes in the tree in a topological order. For each node v_j, it considers all possible matches corresponding to the cells in the li-

brary. For each match g_j, it considers all possible placements (x_j, y_j) in $\mathcal{Z}_j$ and for each of those, it finds out the optimal-delay due to the mapping solution and the placement for each fanin (line 5 in the pseudo-code). This search for optimal delay value at each node requires $O(mP_{max})$ time, since for each node, v_j, we store optimal delay values $a_{v_j}(g_j, x_j, y_j)$ indexed by a match g_j and its placement (x_j, y_j) (line 7). The auxiliary information about the matches at the fanins and their locations is also indexed similarly and is employed during the covering phase to actually build the mapped netlist and its placement. The amount of memory required to store the optimal delay values and other auxiliary information for entire tree is $O(nmf_{max}P_{max})$, since there are n nodes with P_{max} possible matches and m placement possibilities for those matches. The time-complexity of the matching is dominated by the search for the optimal delay value choice and its location at the fanin of a match, placed at all possible locations, for a node. Since there are n nodes with P_{max} matches at most, each of which has m placement possibilities and have f_{max} fanins at most, the time complexity is $O(nm^2 f_{max}P_{max}^2)$.

E. Handling DAGs by Lagrangian Relaxation

A circuit represented by a DAG may contain multi-fanout nodes. Cells on different fanouts of a gate affect each other on timing, since the load capacitance to the multi-fanout node include the capacitance of all fanout cells. As a result, dynamic programming, which deals with single fanout without properly incorporating the interactive effect between different fanouts, can hardly find the overall best solution on the fanout cone. This limits the application of dynamic programming to delay-optimal mapping and placement on DAGs. This issue is illustrated by a simple example in Fig. 10.

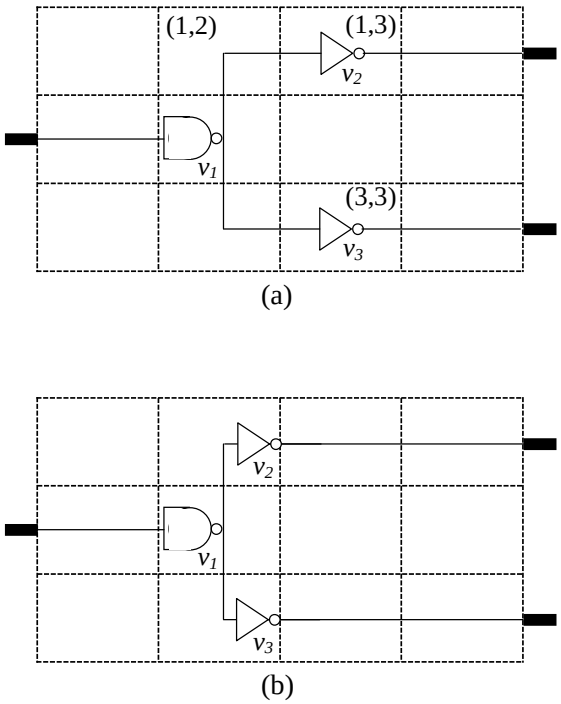


Fig. 10. (a) Two cells (v_2 and v_3) driven by a multi-fanout cell (v_1) placed on a 3×4 grid without consideration of interactive effect between multiple fanouts. I/Os and the multi-fanout cell v_1 are fixed. (b) Optimal placement of v_2 and v_3, considering the load affected by both fanouts of v_1.

Consider the placement of a NAND gate and two INV gates it drives in a 3×4 grid in Fig. 10(a). The primary input I, two outputs O_1, O_2, and cell v_1 are fixed at the locations shown in the figure. Here, the cell delay is load-dependent. In this example, we use the Elmore model for both the cell delay and wire delay, where the delay is linear to the load capacitance it drives and its resistance. For the sake of clear presentation, we assume a unit length wire, every cell, and every I/O pin has unit resistance and unit capacitance. In this case, if we still use the tree placement algorithm in Section C to come up with the delay table for all placements of each node in a topological order traversal, the delays at v_2 and v_3 are considered independently from each other, which does not completely reflect the load-dependent cell delay. The

independent delay calculation leads to an independent placement of v_2 and v_3 as in Fig. 10(a). Cells v_1 and v_2 are uniformly spaced over the upper path, so are v_1 and v_3 over the lower path. This placement is optimal for either the upper path or the lower path, individually. However, this placement is not optimal for all the three cells, because the load v_1 drives is doubled due to multi-fanout. The best placement of v_2 and v_3 are shown in Fig. 10(b), in which v_2 and v_3 are closer to v_1 to compensate its larger load. One may argue that this issue can be resolved by estimating the overall load when considering the solution at one fanout. Unfortunately, this is not true. No matter how much the estimated load on the upper path is, v_3 still needs to be placed on the middle point between v_1 and O_2, because of the quadratic relation between wire delay and wire length on the lower linear path. The same happens to v_2's placement.

To overcome the difficulty, we propose a method based on Lagrangian relaxation (LR): it applies the simultaneous tree mapping and placement to minimize delays weighted by Lagrangian multipliers iteratively; the algorithm stops, if there is no significant improvement in the slack. The whole circuit delay is broken into timing constraints on every timing arc in nodal form. Then, the weighted delay is expressed in the form of timing arc delay summation. Timing points are at the inputs of the gates. Each timing arc, connecting two timing points, spans from the input of a cell to the input a cell on its fanout. For example, there are two timing arcs covering gate v_1 in Fig. 10(a): one is from the input of v_1 to the input of v_2; the other is from the input of v_1 to the input of v_3. The basic idea behind the LR approach is to use weights (Lagrangian multipliers) to put different focus on different parts of timing. We will explain how the weights (Lagrangian multipliers) encode the mutual effect between multiple fanouts of a cell into our problem with more details on the LR method next.

Let $PO(G)$ be the set of primary outputs in G, and $PI(G)$ be the primary inputs

in G. The mapping and placement problem in a general circuit is then formulated as:

DAG Mapping and Placement: Given the net list of a decomposed circuit as a DAG $G(V, E)$, a set of candidate locations $\mathcal{Z}_i$ for each gate in the circuit, and a given cell library $\mathcal{B}$, perform technology mapping and cell placement of the circuit to maximize the circuit slack.

$$
\begin{array}{lll}
\min & -s & \\
\text{s.t.} & q_i - a_i \geq s, & \forall v_i \in PO(G), \\
& a_j \geq a_i + D_{ij}, & \forall v_j \in V \cup PO(G), \forall v_i \in input(v_j), \\
& (x_i, y_i) \in \mathcal{Z}_i, & \forall v_i \in V, \\
& v_i \in g, & \forall v_i \in V, \exists g \in \mathcal{B}.
\end{array}
$$

Notice that a_i at $v_i \in PI(G)$ and q_j at $v_j \in PO(G)$ are constants given by the problem.

A non-negative Lagrangian multiplier is introduced for each constraint on arrival time - the second constraint above. The Lagrangian function is a summation of the objective and weighted timing constraints:

$$
\begin{aligned}
L_\lambda(s, \mathbf{a}) = -s + & \sum_{v_i \in PO(G)} \lambda_{i0}(s + a_i - q_i) \\
& + \sum_{v_j \in V - PI(G)} \sum_{v_i \in input(v_j)} \lambda_{ij}(a_i + D_{ij} - a_j)
\end{aligned} \tag{3.2}
$$

Then, the Lagrangian relaxation dual problem with given multiplier values is

expressed by:

$$\begin{aligned} \min \quad & L_\lambda(s, \mathbf{a}) \\ \text{s.t.} \quad & (x_i, y_i) \in \mathcal{Z}_i, && \forall v_i \in V, \\ & v_i \in g, && \forall v_i \in V, \exists g \in \mathcal{B}. \end{aligned}$$

As shown in [38], the problem can be simplified by eliminating the arrival times in the Lagrangian function according to the Kuhn-Tucker conditions [21].

$$L_\lambda(s, \mathbf{a}) = \sum_{v_i \in PO(G)} \lambda_{i0} q_i + \sum_{v_j \in V - PI(G)} \sum_{v_i \in input(v_j)} \lambda_{ij} D_{ij}. \tag{3.3}$$

In our Lagrangian relaxation framework, there are two problems to solve. The first one is the Lagrangian subproblem solved in each Lagrangian iteration, which is to minimize $L_\lambda(s, \mathbf{a})$ in Equ (3.3) with specific multiplier values. The other problem is the Lagrangian dual problem, which updates the multipliers at the end of each Lagrangian iteration to maximize the minimum value of $L_\lambda(s, \mathbf{a})$ with optimal mapping and placement solutions.

The Lagrangian subproblem is solved using our combinatorial algorithm of simultaneous mapping and placement in Section D. The same method is employed here, except the cost function used to evaluation each mapping and placement option is different - instead of minimizing the arrival time, choose the options to reduce the summation of delays. Specifically, line 5 in Algorithm 8 changes to use the following formula.

$$L_\lambda(v_i) + \lambda_{ij} D_{ij},$$

where v_i's mapping and placement solutions are under consideration for the minimum cost function value at v_j.

The Lagrangian dual problem is solved by sub-gradient ([21]) method. The mul-

tipliers are updated employing sub-gradients [21], following the static timing analysis on the mapping and placement solution in the current iteration. Basically, timing arcs that are more critical are updated with larger multipliers. This way, more attention is focused on the critical parts in the circuit to reduce the overall delay.

The rational of Lagrangian multipliers explains why they help resolving the difficulty caused by multi-fanout in DAGs. Use the same example in Fig. 10. As mentioned before, cell v_1 is covered by two Lagrangian multipliers - one for (v_1, v_2); one for (v_1, v_3). The weight on v_1's cell delay is the summation of the two multipliers, thus is higher than the weight on v_2 or v_3. As a result, in order to minimize the total weighted sum of delays, it is better to reduce the load of v_1 with the cost of increasing v_2 or v_3's load. Consequently, the dynamic programming applied on each of the two fanouts of v_1 would put v_2 and v_3 closer to v_1, specifically in bins $(1, 2)$ and $(3, 2)$. Therefore, by LR the best overall solutions can be found.

The time complexity of our algorithm is dominated by the number of iterations in LR and the matching phase, whose complexity is same as that of $MatchPlaceTree(T)$ in the previous section, since the simultaneous mapping/placement is carried out on individual trees in the DAG.

F. Handling Placement Density Constraint

To this point, our algorithms ignores the possibility of over crowded areas during cell placement. Although cell overlapping is unlikely to happen when re-placement is performed with carefully selected candidate locations for each cell in the whole under-utilized placable area, this over-crowding issue still needs to be taken care of, because a violation of non-overlapping constraint may result in unexpected timing penalty in following legalization stage, which resolves cell overlapping. Therefore, it is better

to deal with the overlapping risk early during our cell placement by controlling the placement density in small tiles, each of which is composed of multiple bins, and all of which together form the whole placement area. We take this approach and enforce the density constraint on small tiles in our cell placement.

Suppose the whole placement area is divided into many small tiles, the kth of which is denoted by $\mathcal{Y}_k$. Let the upper bound of the tile density be γ, i.e., $\frac{\sum_{(x_i,y_i)\in\mathcal{Y}_k}|v_i|}{|\mathcal{Y}_k|} \leq \gamma$ should hold, where $|v_i|$ and $|\mathcal{Y}_k|$ represents the area of the ith cell and the kth tile, respectively. Then, the formulation of our simultaneous mapping and placement problem can be updated as follows.

Density-Constrained DAG Mapping and Placement: Given the net list of a decomposed circuit as a DAG $G(V, E)$, a set of candidate locations $\mathcal{Z}_i$ for each gate in the circuit, a tile density constraint γ, and a given cell library $\mathcal{B}$, perform technology mapping and cell placement of the circuit to maximize the circuit slack.

$$
\begin{aligned}
\min \quad & -s & & \\
\text{s.t.} \quad & q_i - a_i \geq s, & & \forall v_i \in PO(G), \\
& a_i \geq a_j + D_{ji}, & & \forall v_i \in V \cup PO(G), \forall v_j \in input(v_i), \\
& (x_i, y_i) \in \mathcal{Z}_i, & & \forall v_i \in V, \\
& v_i \in g, & & \forall v_i \in V, \exists g \in \mathcal{B}, \\
& \frac{\sum_{(x_i,y_i)\in\mathcal{Y}_k}|v_i|}{|\mathcal{Y}_k|} \leq \gamma, & & \forall \mathcal{Y}_k.
\end{aligned}
$$

To solve this problem with extra density constraint on tiles, we employ Lagrangian relaxation again. Similar to how we deal with arrival time constraints, we turn the density constraint into a penalty term in the Lagrangian function (the cost function). Each density constraint on a specific tile $\mathcal{Y}_k$ is assigned with a Lagrangian multiplier μ_k. Thus, the Lagrangian function becomes

$$L_\lambda(s, \mathbf{a}) = \sum_{v_i \in PO(G)} \lambda_{i0} q_i + \sum_{v_j \in V - PI(G)} \sum_{v_i \in input(v_j)} \lambda_{ij} D_{ij} + \sum_{\mathcal{Y}_k} \mu_k \left(\frac{\sum_{(x_i, y_i) \in \mathcal{Y}_k} |v_i|}{|\mathcal{Y}_k|} - \gamma \right). \quad (3.4)$$

In each Lagrangian iteration, the subproblem of minimizing the Lagrangian function is solved using our combinatorial mapping and placement algorithm. The only difference induced by this subproblem is the cost function value in the characterization of each solution during the solution search. Specifically, to perform the task here Algorithm 8 is modified on line 5 using the following formula.

$$L_\lambda(v_i) + \lambda_{ij} D_{ij} + \mu_k \frac{|v_j|}{|\mathcal{Y}_k|},$$

where $\mathcal{Y}_k$ is the tile where the current candidate location of v_j resides, i.e., $(x_j, y_j) \in \mathcal{Y}_k$.

The Lagrangian dual problem is also solved by updating the multipliers using subgradient method. Besides the multipliers for timing constraints updated according to criticality on different timing arcs, the multipliers for tile placement density are updated to impose higher cost on tiles that are too crowded, thus, in succeeding subproblem solving iteration the cells are pushed away from over-crowded tiles to tiles with lower density. This can be viewed as an analog to a flow driven by the difference of potential (multiplier) at different spots (tiles).

Table III. Comparison of conventional delay oriented mapping followed by timing driven placement with proposed approaches employing only tree placement, simultaneous tree mapping and placement, and Lagrangian relaxation (LR) with simultaneous mapping and placement. The maximum path delay and the minimum slack are in *ps*; CPU time is in seconds; total wire length, cell area are normalized with respect to the corresponding quantities due to the conventional approach.

	Conventional			Simul tree mapping & placement					LR /w simul mapping & placement				
Circuit	Delay	Slack	CPU	Delay	Slack	CPU	Wire	Area	Delay	Slack	CPU	Wire	Area
C432	1091	59	148	932	218	2	0.83	1.03	921	229	47	0.99	0.98
C499	1043	57	254	933	167	2	1.01	1.13	925	175	31	1.12	1.09
C880	989	11	140	803	197	1	0.92	1.02	788	212	29	0.95	1.00
C1355	1240	60	193	1099	201	1	0.94	1.01	1029	271	35	0.95	1.002
C1908	1465	85	290	1221	329	2	0.92	0.96	1203	347	39	0.96	0.97
C2670	1229	71	564	1039	261	6	1.03	1.07	1020	280	42	1.01	1.00
C3540	1760	90	637	1672	178	43	1.00	1.08	1593	257	395	1.07	0.98
C5315	2011	89	1101	1820	280	12	1.03	0.99	1799	301	102	1.02	1.00
C6288	5191	159	1118	5169	181	14	1.00	0.81	5148	202	69	0.99	1.007
C7552	1465	85	2555	1416	134	12	1.08	1.04	1307	243	165	1.06	1.008
Ave.	1748	77	700	1610	215	9.5			1573	251	95		
Norm.	1	1	1	0.92	2.8	0.014	1.02	0.99	0.90	3.26	0.136	1.01	1.003
B14	3790	150	2025	3574	366	51	0.9	1.03	3533	407	259	1.01	1.03
B15	4185	325	1302	3792	718	268	1.01	1.02	3549	961	587	1.02	0.98
B20	4857	343	7154	4296	904	232	1.11	1.00	4281	919	862	1.05	0.99
Ave.	4277	818	3493	3887	1988	183			3788	2287	569		
Norm.	1	1	1	0.884	2.43	0.05	1.001	1.012	0.881	2.80	0.163	1.02	0.998

G. Experimental Results

The algorithms described in this paper are implemented in a C++ program on Windows platform with 3.0 GHz Pentium IV processor. To evaluate the efficacy of the algorithms, the experiments are run on the set of ISCAS'85 combinational circuits and selected ITC'99 benchmarks with a standard cell library characterized employing 70 nm technology parameters [39]. Typical cell utilization is around 50% for each of the benchmarks, which is normally the case for average synthesizable blocks in high performance microprocessor circuits. The results due to the following three iterative approaches, whose goal is to maximize the worst case slack, are compared:

- **Conventional:** In this case, each iteration performs conventional delay oriented technology mapping followed by timing driven placement. The technology mapping algorithm is similar to that in [36] and considers the wire-delays

based on placements, whereas the timing driven placement is implemented by incorporating timing aware net weighting technique [40] with mPL6 [37].

- **Simultaneous delay-optimal tree mapping and placement:** In each iteration, timing critical trees are optimized by simultaneous mapping and placement algorithm discussed in Section D.

- **LR with simultaneous tree mapping and placement:** In each iteration, timing critical cones are optimized by the LR-based extension of simultaneous tree mapping and placement to DAGs, presented in Section E.

The stopping criterion for all the approaches is less than $10ps$ slack improvement in consecutive iterations. The results due to all the approaches are shown in Table III. As compared to the conventional approach, Lagrangian relaxation based algorithm improves the average slacks and maximum delays by $64 \sim 69\%$ and $11 \sim 14\%$, respectively, with 7 times speed-up in the run-time. Similarly, tree based simultaneous mapping and placement leads to $59 \sim 62\%$ and $7 \sim 13\%$ improvements in the slacks and delays, respectively, with approximately 2 orders of magnitude small run-times. The improvement in runtimes over the conventional approach comes from the absence of timing-driven net-weighting and the placement of whole circuit. Moreover, the conventional approach is likely to be more susceptible for divergence than tree placement or simultaneous tree mapping and placement. Even in case of LR approach, after the first iteration, we allow the placement of the cells within only certain radius, which although reduces the placement search space, still allows the complete exploration of the mapping space and ensures placement stability. The improvements highlight the fact that the simultaneous exploration of the mapping and placement spaces can lead to the timing convergence not only faster but also with better quality than exploring the mapping and the placement spaces separately, as in the conven-

tional approach. One can observe that the proposed methods have limited impact on wirelength and cell area, although these are not included in the problem formulation. The results due to employing only tree placement to improve timing show that it increases wirelength and cell area marginally, but still improves the slacks considerably. This shows that employing simultaneous mapping and placement may be a better approach than applying delay oriented mapping and placement separately, since the technology mapping which considers the wire-delays based on placement is sensitive to the placement of the subject graph and considering only center of gravity placements for the matches, as opposed to all possible placements in simultaneous mapping and placement approaches, limits the optimization scope.

H. Conclusion

In this paper, we proposed polynomial time algorithms for delay-optimal placement as well as simultaneous technology mapping and placement for trees. We extended the simultaneous mapping and placement algorithm to DAGs and placement density constraints using Lagrangian relaxation technique. Compared to the conventional iterative mapping and timing driven placement approach, our methods improve the slacks by more than 60%, with 7 times or greater speed-up, and have negligible impact on total wirelength and cell area.

CHAPTER IV

BUFFER INSERTION IN MULTI-CORE DESIGNS

A. Introduction

When VLSI feature size shrinks to nanometer regime, chip power density approaches its fundamental limit. Shackled by the tight power constraint, performance gain from the frequency increase is diminishing. This fact forced microprocessor companies to make a strategic move - pursuing multi-core designs. Nowadays, multi-core designs become common in almost all kinds of processor applications: servers, desktops and laptops.

Since multi-core is an architectural approach, most of related research works are naturally focused on architecture level. In this paper, we will show that multi-core designs sometimes also implicates circuit level issues and discuss a such problem. In specific, we will investigate how to perform buffer insertion in the context of an industrial multi-core processor design methodology. Buffer insertion is a powerful technique for interconnect performance optimization. In traditional designs, buffer insertion solutions are often found by using van Ginneken's [41] or Lillis' [42] algorithm. Given an interconnect tree and candidate buffer locations on it, van Ginneken's algorithm [41] propagates a set of partial solutions from the leaf nodes toward the source and eventually finds the best timing solution at the source. Lillis made an important extension [42] that can deliver a set of solutions with different timing-cost tradeoff. This allows people to find the minimum cost (or power) solution subject to timing constraints. In both algorithms, inferior partial solutions are pruned during the propagation so that the computation runtime is reasonable.

In an industrial multi-core design methodology, the design of the cores and chip

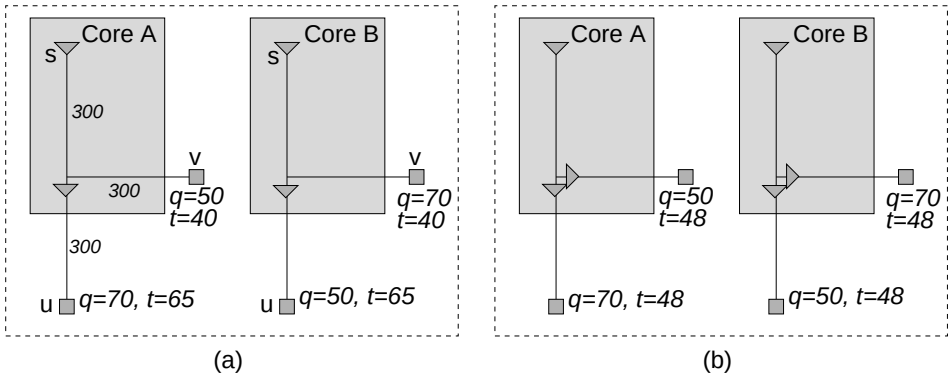


Fig. 11. The required arrival time and delay are denoted by q and t, respectively. The length of each wire segment is $300\mu m$.

integration are performed at about the same time. An interconnect net may have multiple instances for different cores. In Figure 11(a), the net has one instance for core A and another instance for core B. For each net, the interconnect inside the cores should be identical for each instance since these cores have to be the same and even a small difference may lead to large change through ripple effect. However, the interconnect outside any core may vary as it is difficult for chip integration to enforce core-based regularity to a large extent outside the cores. For example, in Figure 11, the required arrival time q at sink u is $70ps$ for core A and $50ps$ for core B. Since these cores have to be identical, the buffer insertion solutions inside the cores should be identical for all these instances. Therefore, we need to find a single buffer insertion solution that accommodates different scenarios outside the cores. This is a key difference from the conventional buffer insertion problem.

Please note that the multi-core processor design is different from the case of IP-core design where the knowledge on the prospective applications is limited. In contrast, in a microprocessor company, designers of the cores work side-by-side with chip integration team and therefore know the out-core environment, at least approxi-

mately. The often-aggressive performance goal in microprocessor designs requires that such knowledge is utilized for performance improvement rather than being neglected. Using certain interface, such as buffers at the boundary of cores, can decouple the designs of in-core and out-core portions and therefore makes the problem much easier to solve. However, such interface or boundary buffers may result in large area/power overhead if they are deployed without scrutiny.

For the multi-scenario buffer insertion problem in multi-core processor designs, a naive approach is to run Ginneken-Lillis algorithm separately on each instance and then pick one eventually shared by all instances. However, the algorithm run on one instance may prune a partial solution which is preferred in another instance. It is also likely that the solution sets at the sources of these instances have no overlap. If we pick the optimal solution for one instance and apply it to the other instances, timing violation may occur in the other instances. For example, in Figure 11(a), the minimum cost solution satisfying timing constraint for core A is to insert a buffer between the Steiner node and sink u. However, this solution causes negative slack of $-15ps$ at sink u for core B. A single solution that satisfies the timing constraints for both core A and core B is to insert buffers at both branches as in Figure 11(b). This solution is pruned out when Ginneken-Lillis algorithm is performed for either core A or core B separately.

In this work, we make significant extensions to Ginneken-Lillis algorithm such that a single buffering solution can be found to accommodate difference scenarios for different cores. In this paper, we focus on the following differences among instances: (1) required arrival time (RAT) for sinks outside cores; (2) sink capacitance for sinks outside cores; (3) arrival time (AT) for drivers outside cores; (4) driver resistance for drivers outside cores. Please note that the arrival time at the driver is not important for conventional buffer insertion problems as changing AT does not affect the relative

timing criticality among all candidate solutions. For multi-core designs, different arrival time implies different timing criticalities among the instances even when they share the same delay and the same sink RATs. In our work, we deal with the overall *net* solutions which are constituted by *instance* solutions instead of handling the instance solutions separately. The dimension of net solution space is much higher than that of instance solution space. In general, the computation complexity grows rapidly with the number of dimensions. We propose several techniques to reduce the dimension of solution space with limited degradation to solution quality.

To the best of our knowledge, this is the first work on the multi-scenario buffer insertion problem in multi-core processor designs. Compared to the naive approach that applies one instance solution to all of the instances, our algorithm can improve slack by $102ps$ on average for max-slack solutions. For the formulation of minimizing cost (buffer area, buffer capacitance or buffer power) subject to timing constraints, our algorithm causes no timing violation while the naive approach results in 35% timing violations. Moreover, the computation speed of our algorithm is faster.

B. Traditional Buffering

Traditional buffering problem is solved with Ginneken-Lillis style algorithms. Given the layout of a Steiner tree with candidate buffering locations, Ginneken-Lillis algorithm propagates candidate solutions from the sinks towards the source.

Each node v_i is associated with a solution set, $S(v_i)$, which includes candidate solutions propagated there. Each candidate solution is characterized by a 3-tuple $(c(v_i), q(v_i), w(v_i))$, where the value of $c(v_i)$ denotes the downstream load capacitance, $q(v_i)$ represents the required arrival time, and $w(v_i)$ is the cost for the solution. At each node, a candidate solution is formed with a combination of child solutions

(solutions of immediate downstream nodes) and the buffer choice at node v_i. The cost, $w(v_i)$ is the summation of child node costs and the cost of the buffer at node v_i. The RAT at node v_i is given by decreasing the minimum child solution RAT by wire and buffer delay at node v_i, i.e.,

$$q(v_i) = \min_{v_j \in \text{children}(v_i)} q(v_j) - \text{elmore}(wire_i) - \text{delay}(buffer_i), \tag{4.1}$$

where children(v_i) is the set of child nodes of node v_i, and the buffer delay is zero when no buffer is inserted at node v_i, i.e., delay($\emptyset$) = 0.

Solutions that lead to worse source RAT and cost than other solutions are inferior solutions. In order to prevent the solution set size from growing exponentially, inferior solutions are kept from entering new solution sets and pruned from existing solution sets.

A basic inferior solution detection rule [42] is as follows.

Property 2 *Given two solutions $s = (c, q, w)$ and $s' = (c', q', w')$ in a node's solution set, s is inferior to s' if the following condition holds:*

$$w \geq w', \quad c \geq c', \quad \textit{and } q \leq q'. \tag{4.2}$$

By implementing this rule with an efficient data structure, Lillis' algorithm [42] limits the solution set to a pseudo-polynomial size.

C. Multi-scenario Buffer Insertion

In buffering for multi-core designs, a net may have multiple instances, each of which corresponds to one core. For example, in Fig. 11, the net has two instances, one for core A and the other for core B. In the general case, some parts of the net are inside cores and the other parts are outside cores. In Fig. 11, sink u and v are outside cores

and the source s is inside cores. In reality, it is also likely that the source is outside cores. Since the cores are usually identical, the in-core part of the net is the same in all instances. However, the out-core part may vary from one instance to another, depending on the design of outside cores. In this work, we consider the following differences among the out-core parts of different instances:

1. Required arrival time (RAT) of sinks outside cores.
2. Capacitance of sinks outside cores.
3. Arrival time (AT) of source outside cores.
4. Driver resistance of source outside cores.

In reality, the out-core topology may vary from one instance to another. In this paper, we focus on the above four difference and will study the topology difference in future work.

Because the cores are identical, buffering solutions for all instances have to be the same, at least for in-core part. However, traditional buffering algorithms such as Ginneken-Lillis algorithm [41, 42] are applicable to only individual instances. If they are carried out on each instance separately, it is difficult to ensure that all instances share the same buffering solution. Sometimes, the solution sets at the sources of different instances have no overlap and consequently it is impossible to find a common solution among these sets. A naive method is to perform Ginneken-Lillis algorithm on one instance and apply the solution of this instance to all the other instances. However, a good solution for one instance may be poor for other instances due to the difference on out-core part.

In multi-core designs, the buffering problem is: how to find a single solution that can accommodate all instances with differences? We consider two common problem

formulations stated as follows.

Max-slack problem:*Find a single buffering solution for all instances of a net such that the minimum slack among all instances is maximized.*

Min-cost subject to timing constraint problem: *Find a single buffering solution for all instances of a net such that the total buffering cost is minimized while the timing constraints of all instances are satisfied.*

We define **critical slack** as the minimum slack among all instances of a net. Therefore, the max-slack problem is to maximize the critical slack of a net. In this work, we use buffer capacitance as the buffer cost. Alternatively, the buffer cost can be defined as buffer area or buffer power without affecting the algorithms.

D. The Algorithm

1. Algorithm Overview

Our algorithm also propagates candidate solutions from sinks toward sources like the dynamic programming in Ginneken-Lillis algorithm. A key idea for solving the multi-core buffering problem is to propagate the same buffering solutions among all instances. In other words, we propagate net solutions. If there are h cores, a **net solution** consists of h identical buffering solutions, one for each instance. The solutions in conventional buffering can be treated as **instance solutions**. Therefore, we can also say that a net solution is composed by multiple identical instance solutions. For example, Fig 11(a) indicates one net solution composed of two instance solutions.

Although the instance solutions of a net solution are identical, they may have different RATs and/or different load capacitances due to the differences on the out-core parts. Therefore, a net solution is characterized in $2h + 1$ dimensional space for an h-core design: h dimensions for load capacitances, another h dimensions for RATs

and the other dimension for the buffering cost. For example, a solution at node v_i is characterized by

$$(c(v_i, \phi_1), q(v_i, \phi_1), ..., c(v_i, \phi_h), q(v_i, \phi_h), w(v_i))$$

where $c(v_i, \phi_j)$ and $q(v_i, \phi_j)$ are the load capacitance and RAT of node v_i in instance $\phi_j, j \in \{1, 2, ..., h\}$, respectively.

The framework of our algorithm is similar as Ginneken-Lillis algorithm except that we propagate net solutions instead of instance solutions. It is very difficult to perform pruning for the net solutions since their dimension is significantly higher than that of conventional buffering. Consequently, the algorithm on $2h + 1$ dimensional solution space can be very slow. A main focus and contribution of our work is to represent the $2h + 1$ dimensional problem by a 3-dimensional problem, which well preserves solution quality with a complexity similar to conventional buffering. Such transform is achieved through the concept of critical component.

Definition 1 *The critical component of a solution at node v_i in multi-core buffering problem is a 3-tuple,*

$$s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i)), \tag{4.3}$$

where $\hat{w}(v_i) = w(v_i)$ is the cost and $\hat{q}(v_i)$ is the minimum RAT over all instances, i.e., $\hat{q}(v_i) = \min_{k=1}^{h} q(v_i, \phi_k)$. The first element, $\hat{c}(v_i)$ is a capacitance value extracted in different ways under different conditions.

In Section 2, we introduce the algorithm for a relatively simple case where only the sink RATs are different among instances. The algorithm for this case is still optimal. In Section 3, we discuss more general and more difficult cases where each sink has different load capacitances and different RATs in different instances. The differences on source arrival time and driver resistance are addressed in Section 4.

2. Cases with Only Sink RAT Differences

In this section, we introduce an algorithm for a multi-core buffering problem where only sink RAT may be different among instances. Then, without loss of generality, we can assume that the arrival times (ATs) at the sources of all instances are zero [1]. In this case, the slack of a instance ϕ_k is equal to the RAT $q(v_0, \phi_k)$ at the source node v_0. In addition, the critical slack is equal to the RAT of the critical component at the source node v_0. Then, the two multi-core buffering problems formulated in Section C can be restated as max-slack problem:

$$\text{maximize: } \min_{k=1}^{h} q(v_0, \phi_k) = \hat{q}(v_0), \tag{4.4}$$

and min-cost subject to timing constraint problem:

$$\begin{aligned} \min \quad & w(v_0) = \hat{w}(v_0), \\ \text{s.t.} \quad & \min_{k=1}^{h} q(v_0, \phi_k) = \hat{q}(v_0) \geq 0. \end{aligned} \tag{4.5}$$

Next, we introduce the notion of complementary solution to assist the presentation of properties and algorithms. A complementary solution at node v_i, denoted by $u(v_i)$, is a solution that contains buffer choices at all nodes in the net other than those in the subtree rooted at node v_i. In the example shown in Fig. 12, a partial solution at node v_5, $s(v_5)$ is composed of all the buffer choices at nodes outside the shaded area. A complementary solution at node v_5, $u(v_5)$ consists of all the buffer choices at nodes in the shaded area. Obviously, with a pair of partial and complementary solution, $s(v_i)$ and $u(v_i)$ at node v_i, $u(v_i) \cup s(v_i)$ forms an overall solution, which contains buffer choices for all nodes in the net. Denote by $U(v_i)$ the set of all possible complementary solutions at node v_i. Complementary solutions build up

[1] The difference of arrival time at the sources will be discussed in Section 4.4.

a bridge between partial and overall solutions, which enables some general rules to identify inferior solutions.

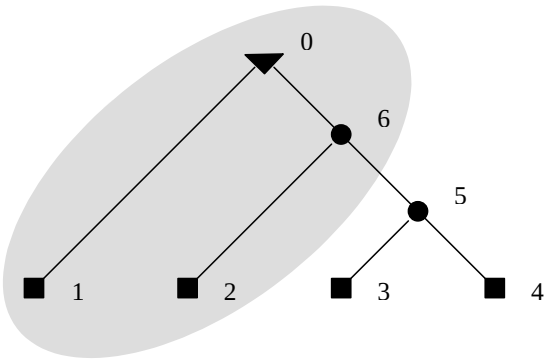


Fig. 12. Complementary solutions

Property 3 *Given two partial solutions $s(v_i)$ and $s'(v_i)$ at some node v_i, $s(v_i)$ is inferior to $s'(v_i)$ if the following condition holds:*

$\forall u(v_i) \in U(v_i)$,

$u(v_i) \cup s(v_i)$ *is an overall solution inferior to* $u(v_i) \cup s'(v_i)$.

Based on above general properties of inferior solutions, we develop more specific properties for multi-core nets. The case of equal sink capacitance is investigated in the rest of this section.

Definition 2 *An iso-cap net is a multi-core net, such that each of its sinks has equal capacitance over all instances. In another word,* $\forall k \in \{1, ..., h\}, C(v_i, \phi_k) = C(v_i)$ *holds for each sink node* v_i.

Also, we define iso-cap solution at node v_i to be a solution with equal downstream capacitances across all instances, i.e., $\forall k \in \{1, ..., h\}, c(v_i, \phi_k) = c(v_i)$. It is obvious that every solution at any node in an iso-cap net is an iso-cap solution.

For solutions in iso-cap nets, we set the first element of their critical component to be the unique downstream capacitance across all instances, i.e., for a solution at node v_i,

$$\hat{c}(v_i) = c(v_i, \phi_k), \text{ for any } k \in \{1, ..., h\}. \tag{4.6}$$

The following property of inferior solutions in iso-cap nets is based on this critical component assignment.

Property 4 *Given two partial solutions,*

$$s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i))$$
$$\textit{and } s'(v_i) = (\hat{c}'(v_i), \hat{q}'(v_i), \hat{w}'(v_i))$$

at node v_i in an iso-cap net, $s(v_i)$ is inferior to $s'(v_i)$ if the following condition holds:

$$\hat{w}(v_i) \geq \hat{w}'(v_i), \hat{c}(v_i) \geq \hat{c}'(v_i), \textit{and } \hat{q}(v_i) \leq \hat{q}'(v_i). \tag{4.7}$$

Proof 3 *See appendix.*

Based on property 4, a minor modification to Ginneken-Lillis algorithm is needed to accommodate buffer insertion in iso-cap nets. At the beginning, the critical components of all sinks are extracted. Then the solutions are propagated from sinks to the source with critical component $(\hat{c}, \hat{q}, \hat{w})$ being used as (c, q, w) of each solution in conventional buffering algorithm. This way, the algorithm reduces the computation complexity to single core problem by performing combination and pruning in

3-dimensional solution space, while the algorithm guarantees the optimal overall net solution of iso-cap net.

An example of two merging solution sets in a net with 2 cores is shown in Fig. 13. The box next to each node contains the solutions at that node. Each solution is expressed in the form of $(c(\phi_1), q(\phi_1), c(\phi_2), q(\phi_2), w)$. In this small example, $s'(v_1)$ and $s(v_1)$ merges with the solution at node v_2 to form solutions $s'(v_3)$ and $s(v_3)$ at node v_3, respectively. The two solutions at node v_1 is not inferior to each other by traditional pruning. However, with their critical components, $(0.3, 2.5, 2.5)$ and $(0.5, 2.1, 3.1)$, $s'(v_1)$ is determined to be inferior to $s(v_1)$. As a result, $s'(v_3)$ is inferior to $s(v_3)$ by critical components as predicted by property 4. In this example, $s'(v_3)$ happens to be inferior to $s(v_3)$ by traditional pruning too.

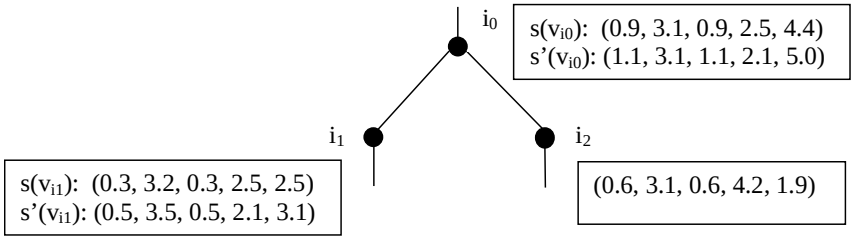


Fig. 13. A part of a net with equal sink capacitance over two instances.

3. Cases with Different Sink Cap and RAT

In this section, we first introduce the notion of iso-cap frontline, and then use it to categorize nodes in the net and process them with different techniques.

The *frontline* at any moment during solution propagation is composed of processed nodes whose parent nodes have not been processed yet. Fig. 14 is an example in a very small net with two cores. Each solution set is shown with a box next to its corresponding node. An node without a box next to it indicates empty solution set for it (not processed yet). Each solution is expressed in the form of

$(c(\phi_1), q(\phi_1), c(\phi_2), q(\phi_2), w)$. The example gives a snapshot of solution sets in the middle of solution propagation. Node v_5 and v_6 have been processed, while their parent nodes v_7 and v_8 have not yet. Therefore, node v_5 and v_6 constitute the frontline at the moment.

In nets with different sink capacitances across different instances, property 4 does not apply directly. However, this does not mean that the dimension of solutions should rise to $2h + 1$. We have two techniques to reduce solution dimension in this case. Recall that a solution at node v_i with buffer inserted at it must be an iso-cap solution, though this is not the only way for a solution to be iso-cap. If all solutions at the frontline nodes are iso-cap solutions, then the part of net above the propagation frontline can be treated as an iso-cap net. We solve this part with help similar to property 4 as in iso-cap nets. For the part of net below the front-line, we use a heuristic to extract critical components of solutions, thus reduce solution space to 3-dimension while well preserving the quality of solution.

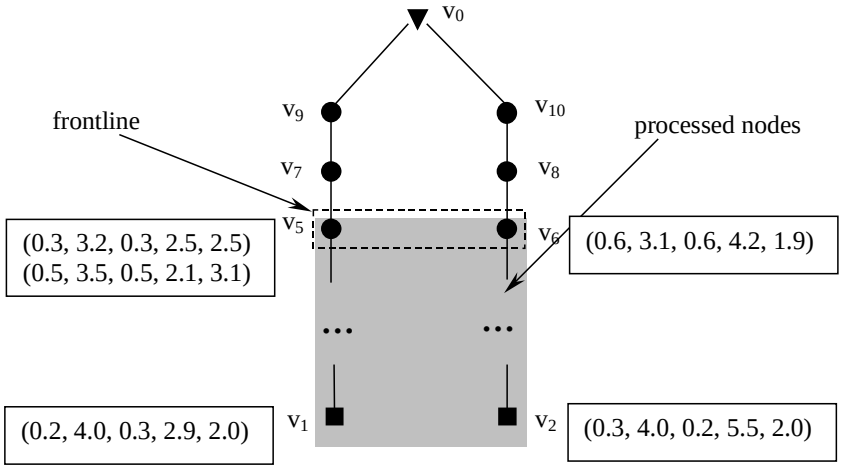


Fig. 14. A net with different sink capacitance over two instances.

We call a solution set an *iso-cap solution set* if all solutions in it are iso-cap. If all frontline solution sets are iso-cap, then the frontline is iso-cap. The part of the

net composed of the nodes on the iso-cap frontline and all their upstream nodes is an *iso-cap sub-net*. An iso-cap sub-net has similar property as iso-cap net. All solutions in iso-cap sub-net are iso-cap. Again, for these solutions we set the first element of their critical components to their unique downstream capacitance, i.e.,

$$\hat{c} = c(\phi_k), \text{ with any } k \in \{1, ..., h\}. \tag{4.8}$$

Property 5 *Given two partial solutions,*

$$s(v_i) = (\hat{c}(v_i), \hat{q}(v_i), \hat{w}(v_i))$$
$$\textit{and } s'(v_i) = (\hat{c}'(v_i), \hat{q}'(v_i), \hat{w}'(v_i))$$

at node v_i in an iso-cap sub-net, $s(v_i)$ is inferior to $s'(v_i)$ if the following condition holds:

$$\hat{w}(v_i) \geq \hat{w}'(v_i), \hat{c}(v_i) \geq \hat{c}'(v_i), \textit{and } \hat{q}(v_i) \leq \hat{q}'(v_i). \tag{4.9}$$

Proof 4 *The proof is similar to that of property 4.*

During the solution propagation procedure, once the frontline is detected to be iso-cap, we use property 5 to solve the iso-cap sub-net as in an iso-cap net. For the downstream nodes below the iso-cap frontline, solutions are represented with critical components extracted by another dimension reduction method.

For nodes below the iso-cap frontline, it is very difficult to determine if one solution is strictly inferior to another only based on its min-RAT ($\hat{q}$), cost ($\hat{w}$), and load capacitances. Because the solutions are not iso-cap, an instance with a small RAT can have a low load capacitance; further more, solutions at other nodes also have various load capacitance over different instances. Therefore, when two solutions

are propagated upstream and are combined with solutions from other nodes, their relative order in min-RAT may change along the way. One way to compensate this variance is to incorporate the prediction of future capacitance difference into critical component extraction. But it is computationally expensive and ineffective. Therefore, we choose a simple yet effective method. Considering the worst case, in each solution the maximum load capacitance over all instances is used as an estimation of the load for critical component, i.e., the first element of critical component is set as

$$\hat{c} = \max_{k=1}^{h} c(\phi_h). \tag{4.10}$$

For example, node v_1 in Fig. 14 has its critical component as $(0.3, 2.9, 2.0)$. Extracting this critical component and using it as (c, q, w) of a solution in conventional buffering, our algorithm propagates solutions from the sinks to the iso-cap frontline in a slightly different style from Gennekin-Lillis algorithm, which is introduced next.

```
1: if status[v_i] has never been updated then
2:    if ∀v_j ∈ children(v_i), status[v_j] = isocap then
3:       status[v_i] ← isocap
4:       update_isocap(parent(v_i))
5:    end if
6: end if
```

Algorithm 9: Procedure:$update\_isocap(v_i)$

There is one more issue to resolve: an efficient method to detect iso-cap frontline during the propagation procedure. The iso-cap frontline is better to be detected as early as possible, and we do not want to run the detection over and over during solution propagation. Our method runs in linear time and ensures the detection of iso-cap frontline at the earliest moment. The basic idea is as follows. When a new solution

```
1: if status[v_i] has never been updated by update_nonisocap then
2:     status[v_i] ← nonisocap
3:     for all v_j ∈ children(v_i) do
4:         update_nonisocap(v_j)
5:     end for
6: end if
```

Algorithm 10: Procedure:*update_nonisocap*(v_i)

set is found to be iso-cap for the first time, it updates the status of the corresponding node and its upstream nodes (if applicable) to *isocap*. If a newly created solution set is not iso-cap, the status of the corresponding node and all nodes in the subtree rooted at its parent node are set to *nonisocap* (if they have not been processed yet). Current frontline is detected to be iso-cap if the root's status becomes *isocap* at any moment. Algorithm 9 and 10 give the outlines of status updating procedures called each time a new solution set is created. Initially, the status of all nodes in the net are set to *nonisocap*. Their status are updated during the propagation procedure. In order to ensure the earliest detection of iso-cap frontline, only those *nonisocap* parents of frontline nodes can be the next node to be processed. This is a key difference of iso-cap frontline detection from traditional propagation procedure.

In the example shown in Fig. 14, after node v_5's iso-cap solution set is created, it invokes procedure *update_isocap*(v_5) and keeps from propagating solutions towards node v_7, waiting for possible signal of source iso-cap. If, as in this example, node v_6's solution set is created and found to be iso-cap, it invokes procedure *update_isocap*(v_6). As the result of the update procedures, the source's status would become *isocap*; thus, the iso-cap frontline is detected at node v_5 and v_6. This way, all upstream nodes above the frontline are enabled to take advantage of iso-cap sub-net. If, in another case, node

v_6, v_8 and v_{10}'s solution sets are created to be not iso-cap, then the *update_nonisocap* procedure updates their status while they propagate solutions upstream. Until the source push the *nonisocap* status all the way down to node v_7, solution set at node v_7 is created, and solutions are propagated upstream to the source and form overall solutions.

We also use another criterion to prune solution sets further. For each buffer, there is a limit on the load capacitance it can drive [43]. Denote by max_cap the maximum load capacitance any buffer can drive. Then, any solution with its critical capacitance $\hat{c} > max\_cap$ is pruned.

Now, with all parts of the algorithm introduced, we assemble them to show how critical component buffering algorithm works on each node. The process is outlined in Algorithm 11.

4. Handling Source Difference

In the multi-core buffering problem, the signal arrival time (AT) at the source may be different for different instances. Consequently, one buffering solution may result in different slacks at different instances, even if these instances share the same sink cap, same sink RAT and the same driver resistance. It is difficult to directly consider the effect of source AT in a bottom-up dynamic programming. Our approach is to shift the source AT and all sink RAT of each instance such that the source AT of every instance is aligned to zero. For each instance, if we shift its source AT and all of its RAT by the same amount, the slack, which is the difference between AT and RAT, is not affected. Therefore, the problem after the AT alignment is equivalent to the original problem. We construct an equivalent dual problem as follows:

Given a net of h instances, the source ATs ($g(v_0, \phi_k)$) and sink RATs of each

1: **if** $status[v_0] = isocap$ **then**
2: Process node v_i by critical component $(\hat{c}, \hat{q}, \hat{w})$ as in Lillis' algorithm
3: **else**
4: Generate solutions by combining child solutions and inserting buffers, with critical components $(\hat{c}, \hat{q}, \hat{w})$.
5: **for all** $s \in S(v_i)$ **do**
6: Update $(c(\phi_1), q(\phi_1), ..., c(\phi_h), q(\phi_h), w)$ of s
7: Extract $(\hat{c}, \hat{q}, \hat{w})$ of s with $\hat{c}$ given by equation (4.10)
8: **end for**
9: Prune $S(v_i)$ by property 4
10: Add wire to each solution in $S(v_i)$
11: Prune $S(v_i)$ by property 4 and max_cap
12: **if** $S(v_i)$ is iso-cap **then**
13: $status[v_i] \leftarrow isocap$
14: $update\_isocap(parent(v_i))$
15: **else**
16: $status[v_i] \leftarrow nonisocap$
17: $update\_nonisocap(parent(v_i))$
18: **end if**
19: **end if**

Algorithm 11: $node\_solution(v_i)$

instance ϕ_k are shifted as follows:

$$g(v_0, \phi_k) = 0,$$
$$\forall v_i \in \{\text{v|v is a sink node}\},\ q(v_i, \phi_k) = q(v_i, \phi_k) - g(v_0, \phi_k). \tag{4.11}$$

Similarly, the driver resistance may be different in different instances. As a result, a partial net solution inferior to another in the cores may become superior to the later net solution when propagated to the source, if the source is outside the cores. Solution inferiority reversion like this does not happen with small driver resistance difference. If the difference between driver resistances in different instances is large, we insert a buffer at the source to make the resistance at the driver identical, which only induces a minor decrease of slack.

E. Experimental Results

1. Experiment Setup

The multi-core buffering algorithms are implemented with C++ code and tested on 200 nets based on industrial designs. The number of sinks varies from 2 to 36. The number of candidate buffer locations for each instance of each net is up to 300. The distribution of number of sinks is shown in Table IV.

We consider 4-core designs so that there are 4 instances for each net. For each sink, its capacitances at different instances vary by at most $\pm 10\%$ around a center value. The sink RATs may vary by up to $\pm 10\%$ of max source-sink delay. The variations of driver resistance is about $\pm 5\%$ of a center value. The buffer library contains 5 buffers. The driving resistance of buffers varies between 45Ω and 120Ω, and the buffer input capacitance is from 6.27 fF to 12.15 fF.

All interconnect delays at buffers and wires are calculated according to Elmore

Table IV. Sink distribution of the testcases.

Number of sinks	$[1, 5)$	$[5, 20)$	$[20, 35)$	$[35, 50)$
Number of nets	35	98	38	29

Table V. Max-slack solution results for 200 nets.

	Instance Based Buffering	Our Algorithm
Avg slack improvement/net (ps)	0 (baseline)	102.08
Avg slack improvement/instance (ps)	0 (baseline)	77.69
Total buffer capacitance (fF)	2503.66	2514.65
Total CPU time (s)	6408	849

delay model. Our algorithms can be extended for accurate delay models like in [44]. The total cost of a net is measured based on the total input capacitance of all inserted buffers.

To the best of our knowledge, there is no previous work on the multi-core buffering problem. We compared our algorithm with instance based buffering (**IBB**). As mentioned previously, IBB basically solves a traditional buffering problem on each individual instance. It selects the instance whose best solution (for whichever objective) is the most timing-critical among all the instances, and applies this best solution to all the other instances.

2. Max-slack Solution

In the max-slack problem formulation, the objective is to maximize the critical slack, which is the minimum slack among all instances of a net. For each net, we compute the critical slack improvement from our algorithm over IBB, which is the critical slack from our algorithm minus the critical slack from IBB. Fig. 15 shows the histogram of the critical slack improvement of all nets. For most of the nets, the slack improvement from our algorithm is at least 20 ps. The average slack improvement is 102 ps. The maximum source-sink delay in these nets is usually less than 1100ps. Therefore, the average delay reduction is about 9%.

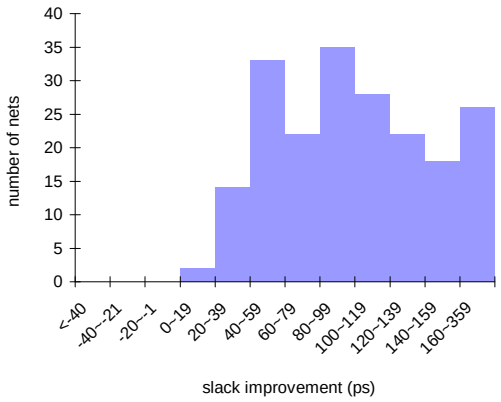


Fig. 15. Histogram of critical slack improvement from our algorithm over IBB for the max-slack problem.

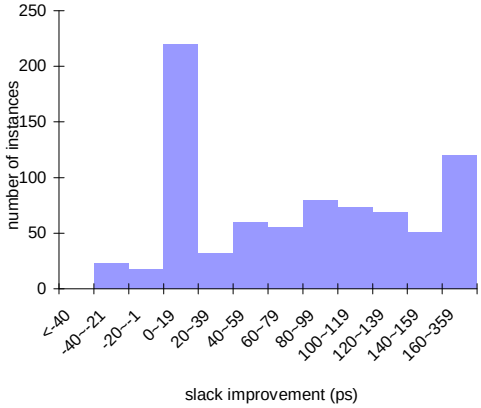


Fig. 16. Histogram of instance slack improvement from our algorithm over IBB for the max-slack problem.

In order to obtain more insight, we compared the slack of individual instances in addition to the critical slack of each net. Fig. 16 shows the histogram of the instance slack improvement from our algorithm over IBB. Most of the time, our algorithm results in remarkable improvement over IBB. Occasionally, the improvement is a small negative value. This is because that our algorithm attempts to provide good solutions for all instances of a net while IBB is usually focused only on one instance for a net. Although IBB may occasionally yield a good solution to a specific instance, the solution is often poor for the other instances. For those instances with poor

Table VI. Min-cost solution results for 200 nets.

	Instance Based Buffering	Our Algorithm
Total # of critical timing violations (ps)	155 (77.50% of 200 nets)	0
Total # of instance timing violations (ps)	282 (35.25% of 800 instances)	0
Total buffer capacitance (fF)	2312.12	2314.77
Total CPU time (s)	6412	851

IBB solutions, the improvement from our algorithm can be very large. This is why the histogram of instance slack improvement in Fig. 16 is spread out more than the critical slack improvement in Fig. 15.

Table V summarizes the average slack improvement, total cost and computation time of the two algorithms. Our algorithm has slightly (0.6%) higher total cost, in term of buffer capacitance, than IBB. The computation complexity of our algorithm is similar to that of Lillis' algorithm and does not grow with the number of cores. In contrast, IBB basically calls Lillis' algorithm h times for h cores. Therefore, our algorithm runs faster than IBB and is much more scalable with respect to the number of cores.

3. Min-cost Solution

Min-cost solutions are from the formulation that minimizes the total buffer cost subject to timing constraints. In other words, a min-cost solution should have non-negative slack. Otherwise, it has timing violation.

Fig. 17 presents the distribution of critical slack, which is the minimum slack among all instances for a net, from both our algorithm and IBB. It is clear that our algorithm can ensure non-negative slacks while IBB cause many timing violations. IBB picks the solution from an instance and applies it to the other instances. Although such a solution is feasible for one instance, there is no guarantee it is feasible for the other instances. The distributions of slacks for individual instances are shown in

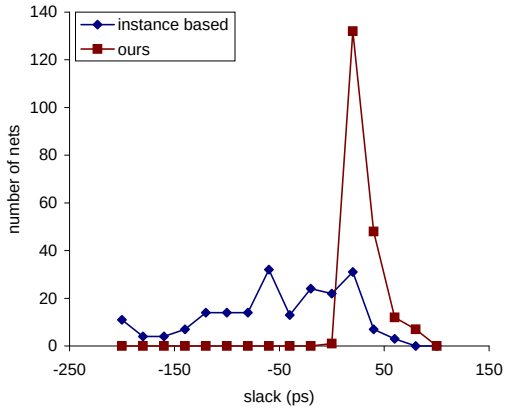


Fig. 17. Histograms of critical slacks for min-cost problem.

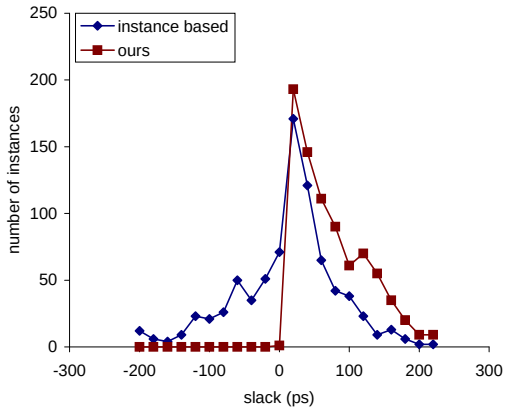


Fig. 18. Histograms of instance slacks for min-cost problem.

Fig. 18. Again, all instance slacks from our algorithm are non-negative while IBB causes many timing violations.

The number of timing violations, total buffer capacitance and CPU time for min-cost solutions are listed in Table VI. The total buffer capacitance from our algorithm is slightly larger than that from IBB, but the difference is negligible.

F. Conclusion and Future Work

This paper proposes algorithms for multi-scenario buffer insertion in an industrial multi-core processor design methodlogy. Experiment results show our algorithm significantly outperforms an extension to conventional buffering in terms of both slack quality and computation speed. In future, we will extend the multi-scenario buffer insertion to handle topology differences among out-core interconnect. Further, we will incorporate various speedup techniques, such as those in [45], into our algorithm framework.

CHAPTER V

GPU-BASED PARALLELIZATION FOR FAST CIRCUIT OPTIMIZATION

A. Introduction

Fast circuit optimization technique is an increasingly compelling need for chip designs. While the pressure of time-to-market is almost never relieved, design complexity keeps growing along with transistor count. In addition, more and more issues need to be considered – from conventional objectives like performance and power to new concerns like process variability and transistor aging. On the other hand, the advancement of chip technology opens new avenues for boosting computing power. One example is the amazing progress of GPU (Graphics Processing Unit) technology. In the past 5 years, the computing performance of GPU has grown from about the same as CPU to about ten times of CPU in term of GFLOPS [46]. GPU is particularly good at fine-grained parallelism and data-intensive computations. Recently, GPU-based parallel computation has been successfully applied for the speedup of fault simulation [47] and power grid simulation [48].

In this work, we propose GPU-based parallel techniques for simultaneous gate sizing and threshold voltage assignment. Gate sizing is a classic approach for optimizing performance and power of combinational circuits. Threshold voltage (V_t) assignment is a popular technique for reducing leakage power without degrading timing performance. Since both of them essentially imply a certain implementation for a logic gate, it is not difficult to perform them simultaneously. It is conceivable that a simultaneous approach is often superior to a separated one in term of solution quality. We will focus on discrete algorithm because (1) it can be directly applied with cell library based timing and power models; (2) V_t assignment is a highly discrete

problem. Discrete gate sizing and V_t assignment faces two inter-dependent difficulties. First, the underlying topology of a combinational circuit is typically a DAG (Directed Acyclic Graph). The path reconvergence of DAG makes it difficult to carry out systematic solution search like dynamic programming (DP). Second, the size of a combinational circuit can be very large, sometimes with dozens of thousands of gates. As a result, most of existing methods are simple heuristics [7, 9, 15]. Recently, a Joint Relaxation and Restriction (**JRR**) algorithm [49] is proposed to handle the path reconvergence problem and enable a DP-like systematic solution search. Indeed, the systematic search [49] remarkably outperforms its previous work. To address the large problem size, a grid-based parallel gate sizing method is introduced in [50]. Although it can obtain high solution quality with very fast speed, it concurrently uses 20 computers and entails significant network bandwidth. In contrast, GPU-based parallelism is much more cost-effective. The expense of a GPU card is only a few hundreds of dollars and the local parallelism obviously causes no overhead on network traffic.

It is not straightforward to map a conventional sequential algorithm onto GPU computation and achieve desired speedup. In general, parallel computation implies that a large computation task needs to be partitioned to multiple threads. For the partitioning, one needs to decide its granularity levels, balance the computation load and minimize the interactions among different threads. Data and memory management is also very important. One needs to properly allocate the data storage to various parts of the somewhat complex memory system of a GPU. Apart from general parallel computing issues, the characteristics of GPU should be taken into account. For example, the parallelization should be SIMD (Single Instruction Multiple Data) in order to better exploit the advantages of GPU. In this work, we propose task scheduling and memory management techniques for performing gate sizing/V_t assignment on GPU.

To the best of our knowledge, this is the first work on GPU-based combinational circuit optimization. In the experiment, we compared our parallel version of the joint relaxation and restriction algorithm [49] and its original sequential implementation. The results show that our parallelization achieves speedup of up to 56× and 39× on average. At the same time, our techniques can retain the exactly same solution quality as [49]. Such speedup will allow many systematic optimization approaches, which were slow and previously regarded as impractical, to be widely adopted in realistic applications.

B. Algorithm of Simultaneous Gate Sizing and V_t Assignment

We briefly review the simultaneous gate sizing and V_t assignment algorithm proposed in [49], since the parallel techniques proposed here are built upon this algorithm. This algorithm has two phases: relaxation phase and restriction phase. It is called Joint Relaxation and Restriction (**JRR**). Each phase consists of two or multiple circuit traversals. Each traversal is a solution search in the same spirit as dynamic programming. The main structure of the algorithm is outlined in Algorithm 12. For the ease of description, we call a combination of certain size and V_t level as an implementation of a gate (or a node in the circuit graph).

The relaxation phase includes two circuit traversals: history consistency relaxation and history consistency restoration. The history consistency relaxation is a topological order traversal of the given circuit, from its primary inputs to its primary outputs. In the traversal, a set of partial solutions are propagated. Each solution is characterized by its arrival time (a) and resistance (r). A solution is pruned without further propagation if it is inferior on both a and r. This is very similar to dynamic programming based buffer insertion algorithm [51]. However, the topology here is a

```
1 Phase I: Relaxation
2     history consistency relaxation;
3     history consistency restoration;
4 Phase II: Restriction
5     repeat
6         topological order search;
7         reverse topological order search;
8     until improvement < σ in current iteration;
```

Algorithm 12: Outline of the Joint Relaxation and Restriction (JRR) Algorithm

DAG as opposed to a tree in buffer insertion. Therefore, two fanin edges e_1 and e_2 of a node v_i may have a common ancestor node v_j. When solutions from e_1 and e_2 are merged, normally one need to ensure that they are based on the same implementation at v_j. This is called history consistency constraint. When apply DP-like algorithm directly on a DAG, this constraint requires to keep or trace all history information and consequently causes substantial computation or memory overhead. To solve this difficulty, the work of [49] suggests to relax this constraint in the initial traversal, i.e., solutions are allowed to be merged even if they are based on different implementations of their common ancestor nodes. Although the resulting solutions are not legitimate, they provide a lower bound to the a at each node, which is useful for subsequent solution search.

In the second traversal of the relaxation phase, any history inconsistency resulted from the first iteration is solved in a reverse topological order, from the primary outputs to the primary inputs. When a node is visited in the traversal, only one

implementation is selected for the node. Hence, no history inconsistency should exist after the traversal is completed. The implementation selection at each node is to maximize the timing slack at the node. The slack can be easily estimated by the required arrival time (q), which is propagated along with the traversal, and the a obtained in the previous traversal.

The solution at the end of the relaxation phase can be further improved. This is because the solution is based on the a obtained in the relaxation, which is not necessarily an accurate one. Due to the relaxation, the a is just a lower bound, which implies optimistic deviations. Such deviations are compensated in the restriction phase. In contrast to relaxation, where certain constraints are dropped, restriction imposes additional constraints to a problem. Both relaxation and restriction are for the purpose of making solution search easy. A restricted search provides a pessimistic bound to the optimal solution. Using pessimistic bounds in the second phase can conceivably compensate the optimistic deviation of the relaxation phase.

The restriction phase consists of multiple circuit traversals with one reverse topological order traversal following each topological order traversal. Each topological order traversal generates a set of candidate solutions at each node, with certain restrictions. Each reverse topological order traversal selects only one solution in the same way as the history consistency restoration traversal in the relaxation phase. A topological order traversal starts with an initial solution inherited from the previous traversal. The candidate solution generation is also similar to DP-based buffer insertion algorithm. The restriction is that only those candidate solutions based on the initial solution is propagated at every multi-fanout node. For example, at a multi-fanout node v_i, candidate solutions are generated according to its implementations, but only the candidates which are based on the initial solution of v_i are propagated toward its child nodes. By doing so, the history consistency can be maintained

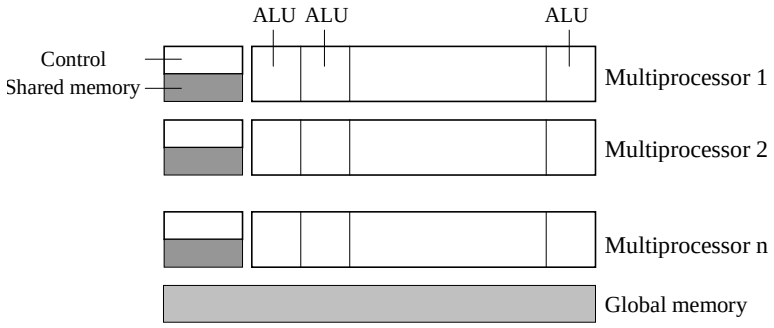


Fig. 19. A generic GPU hardware architecture.

throughout the traversal. The candidate solutions which are not based on the initial solution are useful for the subsequent reverse topological order traversal.

The description so far is for the formulation of maximizing timing slack. If power and/or other objectives are considered simultaneously, one can apply the Lagrangian relaxation technique [6] together with the algorithm of Joint Relaxation and Restriction.

C. GPU-based Parallelization

1. Background on GPU

A GPU is usually composed of an array of multiprocessors and each multiprocessor consists of multiple processing units (or ALUs), as shown in Fig. 19. Each ALU is associated with a set of local registers and all the ALUs of a multiprocessor share a control unit and some shared memory. There is basically no synchronization mechanism among different multiprocessors. A typical GPU may have over one hundred ALUs. GPU is designed for SIMD (Single Instruction Multiple Data) parallelism. As such, an ideal usage of GPU is to execute identical instructions on a large volume of data, each element of which is processed by one thread.

The software program applied on GPU is called kernel function, which is executed

on multiple ALUs in the basic unit of thread. The threads are organized in a two-level hierarchy. Multiple threads form a warp and a set of warps constitute a block. All warps of the same block run on the same multiprocessor. This organization is for convenience of sharing memory and synchronization among thread executions. The thread blocks are often organized in a 3-dimensional grid, just as threads in themselves are. The global memory for a GPU is usually a DRAM off the GPU chip but on the same board as GPU. The latency of global memory access can be very high, due to the small cache size. Similarly, loading the kernel function onto GPU also takes a long time. In order to improve the efficiency of GPU usage, one needs to load data infrequently and make the ALUs dominate the overall program runtime.

2. Two-level Task Scheduling

Exploring GPU-based parallelism for gate sizing and V_t assignment is motivated by the observation that the algorithm repeats a few identical computations on many different gates. When evaluating the effect of an implementation (a specific gate and V_t level) for a gate, we compute its corresponding delay, AT/RAT, and power. These a few computations are repeated for many gates, sometimes hundreds of thousands of gates, and for multiple iterations [49]. Evidently, such repetitive computations on a large number of objects fit very well with SIMD parallelism.

We propose a two-level task scheduling method which allocates the computations to multiple threads. At the first level, the computations for each gate is allocated to a set of thread blocks. The algorithm, software and hardware unit of this level are gate, thread block and multiprocessor, respectively. Since different implementations of a gate share the same context (fanin and fanout characteristics), such allocation matches the shared context information with the shared memory for each thread block. At the second level, the evaluation of each gate implementation is assigned

to a set of threads. The algorithm, software and hardware unit of this level are gate implementation, thread and ALU, respectively. For each gate, the evaluations for all of its implementation options are independent of each other. Hence, it is convenient to parallelize them on multiple threads. The two-level task scheduling will be described in details in the subsequent sections.

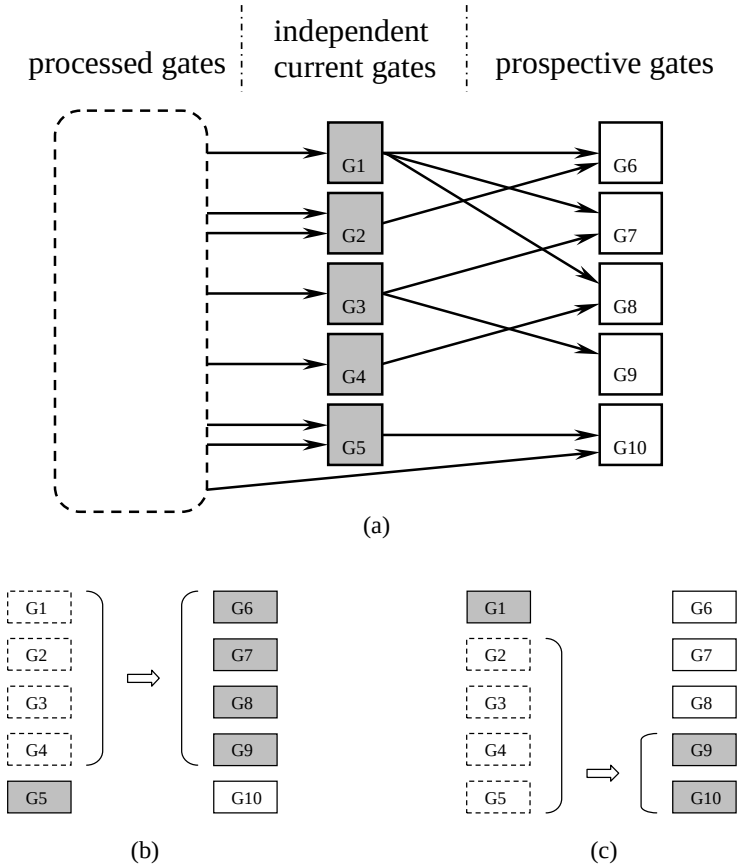


Fig. 20. Processed gates: dashed rectangles; independent current gates: grey rectangles; prospective gates: solid white rectangles. For the scenario in (a), one can choose at most 4 independent current gates for the parallel processing. In (b): if G1, G2, G3 and G4 are selected, we may choose another 4 gates in G5, G6, G7, G8 and G9 for the next parallel processing. In (c): if G2, G3, G4 and G5 are selected, only 3 independent current gates G1, G9 and G10 are available for the next parallel processing.

3. Gate-level Task Scheduling

We describe the gate-level task scheduling in the context of topological order traversal of the circuit. The techniques for reverse topological order traversal are almost the same. In a topological order traversal, a gate is a *processed gate* if the computation of delay/power for all of its implementations is completed. A gate is called a *current gate* if all of its fanin gates are processed gates. We term a gate as a *prospective gate* when all of its fanin gates are either current gates or processed gates. In GPU-based parallel computing, multiple current gates can be processed at the same time because there is no inter-dependency among their computations. A set of current gates are called *independent current gates* (ICG), since there is no computational interdependency among them. Due to the restriction of GPU computing bandwidth, the number of current gates which can be processed at the same time is limited. A critical problem here is how to select a subset of independent current gates for parallel processing. This subset is designated as *concurrent gate group*, which has a maximum allowed size.

The way of forming a concurrent gate group may greatly affect the efficiency of utilizing the GPU-based parallelism. This can be illustrated by the example in Fig. 20. In Fig. 20, the processed gates, independent current gates and prospective gates are represented by dashed, grey and white rectangles, respectively. If the maximum group size is 4, there could be at least two different ways of forming a concurrent gate group for the scenario of Fig. 20(a). In Fig. 20(b), $\{G1, G2, G3, G4\}$ are selected to be the concurrent gate group. After they are processed, any four gates among $\{G5, G6, G7, G8, G9\}$ may become the next concurrent gate group. Alternatively, we can choose $\{G2, G3, G4, G5\}$ as in Fig. 20(c). However, after $\{G2, G3, G4, G5\}$ are processed, we can include at most three gates $\{G1, G9, G10\}$ for the next concurrent

gate group since a fanin gate for $\{G6, G7, G8\}$ has not been processed yet. The selection of concurrent gates in Fig. 20(c) is inferior to that in Fig. 20(b) since Fig. 20(c) cannot fully utilize the bandwidth of concurrent group size 4.

The problem of finding concurrent gate group among a set of independent current gates can be formulated as a max-throughput problem, which maximizes the minimum size of all concurrent gate groups. The max-throughput problem is very difficult to solve. Therefore, we will focus on a reduced problem: max-succeeding-group. Given a set of independent current gates, the max-succeeding-group problem asks to choose a subset of them as the concurrent gate group such that the size of the succeeding independent gate group is maximized. We show in the appendix that the max-succeeding-group problem is NP-complete.

Input : *current_grp*, *prospective_grp*

Output: *concurrent_grp*

1 $concurrent\_grp \leftarrow \emptyset$;
2 **repeat**
3 $\quad prop\_gate \leftarrow \text{gate}(min\_ICGfanin)$;
4 $\quad prospective\_grp \leftarrow prospective\_grp - prop\_gate$;
5 $\quad concurrent\_grp \leftarrow concurrent\_grp \bigcup (\text{inputs}(prop\_gate) \bigcap current\_grp)$;
6 $\quad current\_grp \leftarrow current\_grp - \text{inputs}(prop\_gate) + prop\_gate$;
7 $\quad$ update_ICGfanin (outputs(inputs($prop\_gate$)));
8 **until** $|concurrent\_grp| \geq concurrent\_budget$;

Algorithm 13: Concurrent Gate Group Selection

Since the max-succeeding-group problem is NP-complete, we propose a linear-time heuristic to solve it. This heuristic iteratively examines the prospective gates

and puts a few independent current gates into the concurrent gate group. For each prospective gate, we check its fanin gates which are independent current gate. The number of such fanin gates is called ICG (independent current gate) fanin size. In each iteration, the prospective gate with the minimum ICG fanin size is selected. Then all of its ICG fanin gates are put into the concurrent gate group. After this, the selected prospective gate will no longer be considered in subsequent iterations. At the same time, the selected ICG fanin gates are not counted in the ICG fanin size of the remaining prospective gates.

In the example of Fig. 20, the prospective gate with the minimum ICG fanin size is $G9$. When it is selected, gate $G3$ is put into the concurrent gate group. Then, the ICG fanin size of $G7$ becomes 1, which is the minimum. This requires that gate $G1$ is put into the concurrent gate group. Next, any two of $G2$, $G4$ and $G5$ can be selected to form the concurrent gate group of size 4.

Here is the rationale behind the heuristic. The maximum allowed size of concurrent gate group can be treated as a budget. The goal is to maximize the number of succeeding ICGs. If a prospective gate has a small ICG fanin size, selecting its ICG fanin gates can increase the number of succeeding ICGs with the minimum usage of concurrent gate group budget.

This heuristic is performed on CPU once. The result, which is the gate-level scheduling, is saved since the same schedule is employed repeatedly in the traversals of the JRR algorithm (see Section B). The pseudo code for the concurrent gate selection heuristic is given in Algorithm 13. The minimum ICG fanin size is updated each time an ICG fanin size is updated, so the computation time is dominated by fanin size updating. If the maximum fanin size among all gates is F_i, each gate can be updated on its ICG fanin size for at most F_i times. Thus, the time complexity of this heuristic is $O(|V|F_i)$, where V denotes the set of nodes in the circuit.

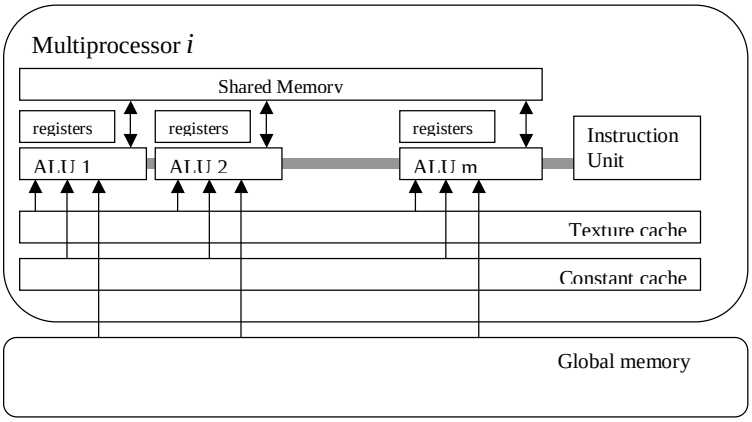


Fig. 21. A multiprocessor with on-chip shared memory. Device memory is connected to the processors through on-chip caches.

4. Parallelization for Gate Implementation Evaluation

When processing a gate, we need to evaluate all of its implementations. It is not difficult to see that the evaluations for different implementations of the same gate are independent of each other. Therefore, we can allocate these evaluations into multiple threads without worrying interdependency. These evaluations include a few common computations: the timing and power estimation for each implementation. Different implementations of the same gate also share some common data such as the parasitics of the fanin and fanout. According to this observation, the evaluations of implementations for the same gate are assigned to the same thread block and the same multiprocessor. Since all the ALUs of the same multiprocessor have access to a fast on-chip shared memory (see Fig. 21), the shared data can be saved in the shared memory to reduce memory access time.

In order to facilitate simultaneous memory access, the shared memory is usually divided into equal-sized banks. Memory requests on addresses that fall in different banks are able to be fulfilled simultaneously, while requested addresses falling in the same bank cause a bank conflict and the conflicting requests have to be serialized. In

order to reduce the chance of bank conflict and avoid the cost of serialized access, we store all information of a gate in the same bank and separate it from that of other gates in other banks.

GPU global memory often has memory coalescing mechanism for improved access efficiency. In order to take advantage of the memory coalescing, we save gate information of sibling nodes in the circuit adjacent to each other in the global memory whenever possible. The benefit is that global memory requests from different threads in a warp have a greater chance to be coalesced and the access latency is thereby reduced.

GPU global memory contains a constant memory, which is a read-only region. This constant memory can be accessed by all ALUs through a constant cache (see Fig. 21), which approximately halves the access latency if there is a hit in the cache. We save cell library data, which are constant, in the constant memory so that the data access time can be largely reduced.

Loading the kernel function to GPU can be very time consuming. Sometimes, the loading time is comparable with time of all computations and memory access for one gate. Therefore, it is highly desirable to reduce the number of calls to the kernel function. Since the computation operations for all gates are the same, we load the computation instructions only once and apply them to all of the gates in the circuit. This is made possible by the fine-grained parallel threads and the pre-computed gate-level task schedule (see Section 3.3). In other words, the gate-level task schedule is computed before the circuit optimization and saved in the GPU global memory. Once the kernel function is called, the optimization follows the schedule saved on GPU and no kernel reloading is needed.

To reduce the idle time of the multiprocessors during memory access, we assign multiple thread blocks to a multiprocessor for concurrent execution. This arrange-

ment has positive impact on the performance, because memory access takes a large portion of the total execution time of the kernel function.

Table VII. Comparison on power (μW) and runtime (seconds). All solutions satisfy timing constraints.

		SA [1]		Sequential JRR [49]		Parallel JRR	
Circuit	#gates	power	runtime	power	runtime	runtime	speedup
c432	289	703	1.7	701	3.25	0.317	10×
c499	539	1669	4.9	1590	6.27	0.295	21×
c880	340	1817	5.1	1050	3.61	0.328	11×
c1355	579	1385	3.3	1076	7.36	0.218	34×
c1908	722	2502	10.7	2296	9.20	0.327	28×
c2670	1082	3412	18.6	2509	15.70	0.376	42×
c3540	1208	4645	22.3	3830	21.30	0.515	41×
c5315	2440	8406	26.8	5023	64.88	1.156	56×
c6288	2342	13685	19.2	12356	53.47	1.295	41×
c7552	3115	9510	46.1	5949	67.44	1.595	42×
Average		4773	15.87	3638	25.25	0.64	39×
Norm.		1.0		0.76			

D. Experimental Results

In our experiment, the testcases are the ISCAS85 combinational circuits. They are synthesized by SIS [22] and their global placement is obtained by mPL [23]. The global placement is for the sake of including wire delay in the timing analysis. The cell library is based on $70nm$ technology. Each gate has 6 different sizes and 4 V_t levels so that it has 24 options of implementation.

In order to test the runtime speedup of our parallel techniques, we compare our parallel version of the JRR algorithm [49] with its original sequential implementation. Regarding solution quality, we compare the results with another previous work [1] in addition to ensuring that the parallel JRR solutions are identical with those of sequential JRR. The method of [1] starts with gate sizing that maximizes timing

slack. Then, the slack is allocated to each gate using a linear programming guided by delay/power sensitivity. The slack allocated to each gate is further converted to power reduction by greedily choosing a gate implementation. We call this as Slack Allocation (SA) based method. The problem formulation for all these methods is to minimize total power dissipation subject to timing constraints. The power dissipation here includes both dynamic and leakage power. The timing evaluation accounts for both gate and wire delay. Since we do not have lookup table based timing and power information for the cell library, we use analytical model for power [49] and the Elmore model for delay computation. However, the JRR algorithm and our parallel techniques can be easily applied with lookup table based models.

The SA method and sequential JRR algorithm are implemented in C++. The parallel JRR implementation includes two parts: one part is in C++ and runs on the host CPU; the other part runs on the GPU through CUDA. CUDA (Compute Unified Device Architecture) is a parallel programming model and interface developed by NVIDIA [52]. The major components of the parallel programming model and software environment include thread groups, shared memory and thread synchronization. The experiment was performed on a Windows XP based machine with an Intel core 2 duo CPU of 2.66GHz and 2GB memory. The GPU is NVIDIA GeForce 9800GT, which has 14 multiprocessors and each multiprocessor has 8 ALUs. The GPU card has 512MB off-chip memory. We set the maximum number of gates being parallel processed to 4. Therefore, at most 96 gate implementations are evaluated at once.

The main results are listed in Table VII. Since all of these methods can satisfy the timing constraints, timing results are not included in the table. The solution quality can be evaluated by the results of power dissipation. One can see that JRR can reduce power by about 24% on average when compared to SA [1]. The parallel JRR achieves exactly the same power as the sequential JRR. Our parallel techniques

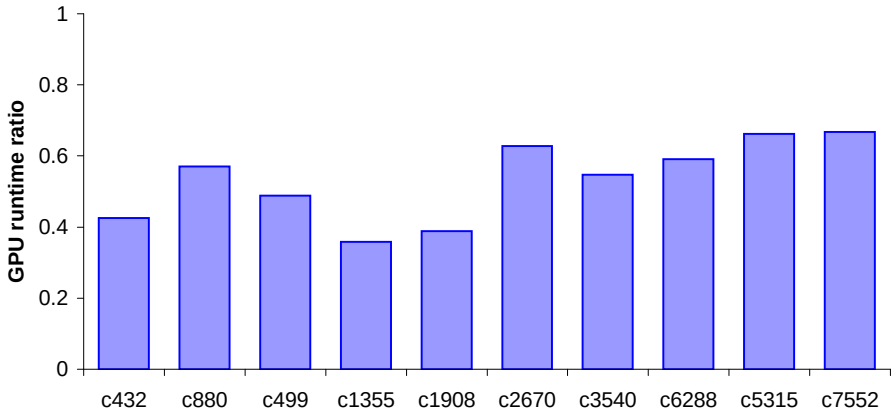


Fig. 22. The ratio of GPU runtime over overall runtime.

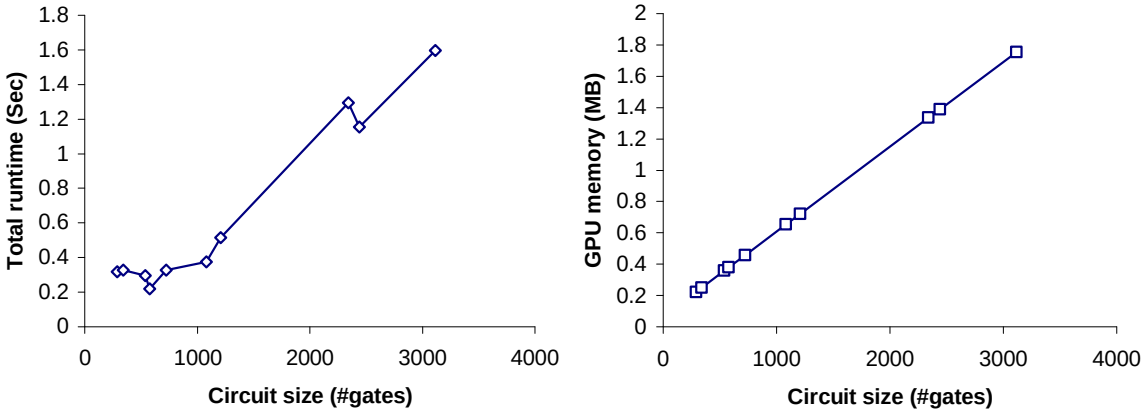


Fig. 23. Runtime and GPU memory scalability.

provide runtime speedup from 10× to 56×. One can see that the speedup tends to be more significant when the circuit size grows. One of the reasons is that both small and large circuits have similar overhead for setup.

In Fig. 22, we depict the ratio between the GPU runtime and the total runtime. The ratio is mostly between 0.4 and 0.6 among all circuits. Usually, larger circuits have higher GPU runtime percentage. This may be due to the higher parallel efficiency of larger circuits.

In Fig. 23, the total runtime and GPU memory usage versus circuit size are

plotted to show the runtime and memory scalability of our techniques. The main trend of the runtime curve indicates a linear dependence on circuit size. There are a few non-monotone parts in the curve which can be explained by the fact that the runtime depends on not only the circuit size but also circuit topology. The memory curve exhibits a strong linear relationship with circuit size. At least for ISCAS85 benchmark circuits, we can conclude that our techniques scale well on both runtime and memory.

E. Conclusions and Future Work

It has long been a challenge to optimize combinational circuit in a systematic yet fast manner due to its topological reconvergence and large size. A recent progress [49] suggests an effective solution to the reconvergence problem. This work addresses the large problem size by exploiting GPU-based parallelism. The proposed parallel techniques are integrated with the state-of-the-art gate sizing and V_t assignment algorithm [49]. These techniques and the integration effectively solves the challenge of combinational circuit optimization. A circuit with thousands of gates can be optimized with high quality in less than 2 seconds. The parallel techniques provide up to 56× runtime speedup. They also show an appealing trend that the speedup is more significant on large circuits. In future, we will test these techniques on larger circuits, for example, circuit with hundreds of thousands of gates.

CHAPTER VI

CONCLUSIONS

This research addresses several critical issues in VLSI circuit design automation problems, and achieves significant runtime speedup on the proposed algorithms.

A number of combinatorial optimization problems in circuit optimization are NP-hard due to the Directed Acyclic Graph (DAG) topology of circuit RTLs. These problems pose significant difficulty for circuit design closure. The proposed systematic solution search scheme, called Joint Relaxation and Restriction (JRR), takes this challenge and manages to produce quality solutions in polynomial time regardless of the complexity of this problem. Experiments for a typical problem solved with JRR shows 24% average improvement on solution quality compared to another state-of-art algorithm.

The multi-scenario optimization and high-dimension solution complexity are coped with Lagrangian relaxation approach. A novel Lagrangian dual problem solving method is proposed to effectively update the Lagrangian multipliers. Instead of ignoring the effect of multiplier on the optimal subproblem solution as in most existing methods, the proposed scheme uses chain rule in sub-gradient calculation, which captures more complete impact of the multiplier change. With a spectrum of sub-gradients, better multipliers are obtained by solving a non-linear programming problem.

Performing multiple stages in the design flow simultaneously avoids oscillation between sub-optimal solutions. In this research, the simultaneous optimization technique is realized by combining different solution options in different design stages. It is applied on two specific problems, namely, simultaneous technology mapping and placement, and simultaneous gate sizing and threshold voltage assignment. Experi-

ments show 10% improvement in solution quality compared to sequential optimization approach.

This research also takes advantage of the fast-growing GPU parallel computing power. After proving the optimal parallelization of the circuit optimization algorithm is NP-complete, an efficient task partition and scheduling algorithm is proposed. For optimization on smaller circuits, experiments show average speedup of 38× over the sequential version. The speedup tends to increase while the circuit size increases and the number of processing units on the GPU grows. Since the parallelization scheme is targeted on a fairly generic circuit optimization algorithm, the proposed parallel computing scheme is generic for a variety of problems.

Because this research aims at some common critical issues in VLSI design automation, the methods/algorithms are not restricted to the problems shown as examples in this dissertation. Other problems in the design flow can benefit from the proposed methods. Specific adjustment on the methods can be done according to problem-specific details in the formulation.

REFERENCES

[1] D. Nguyen, A. Davare, M. Orshansky, D. Chinnery, B. Thompson, and K. Keutzer, "Minimization of dynamic and static power through joint assignement of threshold voltages and sizing optimization," in *Proc. ISLPED*, 2003, pp. 158–162.

[2] K. Kasamsetty, M. Ketkar, and S.S. Sapatnekar, "A new class of convex functions for delay modeling and its application to the transistor sizing problem [cmos gates]," *IEEE Trans. CAD*, vol. 19, no. 7, pp. 779–788, 2000.

[3] S. Roy, W. Chen, and C.C. Chen, "Convexfit: An optimal minimum-error convex fitting and smoothing algorithm with application to gate-sizing," in *Proc. ICCAD*, 2005, pp. 196–204.

[4] S. Hu, M. Ketkar, and J. Hu, "Gate sizing for cell library based designs," in *Proc. DAC*, 2007, pp. 847–852.

[5] J. Fishburn and A. Dunlop, "TILOS: A Posynomial Programming Approach to Transistor Sizing," in *Proc. ICCAD*, 1985, pp. 326–328.

[6] C. Chen, C. C. N. Chu, and D. F. Wong, "Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation," *IEEE Trans. CAD*, vol. 18, no. 7, pp. 1014–1025, 1999.

[7] O. Coudert, "Gate sizing for constrained delay/power/area optimization," *IEEE Trans. VLSI*, vol. 5, no. 4, pp. 465–472, Dec. 1997.

[8] P. K. Chan, "Algorithms for library-specific sizing of combinational logic," in *Proc. of DAC*, 1990, pp. 353–356.

[9] L. Wei, Z. Chen, K. Roy, and V. De, "Design and optimization of dual threshold circuits for low voltage low power application," *IEEE Trans. VLSI*, vol. 7, no. 1, pp. 16–24, 1999.

[10] V. Sundararajan and K.K. Parhi, "Low power synthesis of dual threshold voltage CMOS circuits," in *Proc. ISLPED*, 1999, pp. 139–144.

[11] V. Khandelwal, A. Davoodi, and A. Srivastava, "Simultaneous Vt selection and assignment for leakage optimization," *IEEE Trans. VLSI*, vol. 13, no. 6, pp. 762–765, 2005.

[12] M. Ketkar and S. S. Sapatnekar, "Standby power optimization via transistor sizing and dual threshold voltage assignment," in *Proc. ICCAD*, 2002, pp. 375–378.

[13] D. G. Chinnery and K. Keutzer, "Linear programming for sizing, vth and vdd assignment," in *Proc. of ISLPED*, 2005, pp. 149–154.

[14] S. Shah, A. Srivastava, D. Sharma, D. Sylvester, D. Balaauw, and V. Zolotov, "Discrete Vt assignment and gate sizing using a self-snapping continuous formulation," in *Proc. ICCAD*, 2005, pp. 704–710.

[15] S. Sirichotiyakul, T. Edwards, C. Oh, J. Zuo, A. Dharchoudhury, R. Panda, and D. Blaauw, "Stand-by power minimization through simultaneous threshold voltage selection and circuit sizing," in *Prod. DAC*, 1999, pp. 436–441.

[16] H. Chou, Y. Wang, and C. Chen, "Fast and effective gate-sizing with multiple-Vt assignment using generalized Lagrangian relaxation," in *Proc. ASPDAC*, 2005, pp. 381–386.

[17] T.-H. Wu, L. Xie, and A. Davoodi, "A parallel and randomized algorithm for large-scale dual-Vt assignment and continuous gate sizing," in *Proc. ISLPED*, 2008, pp. 45–50.

[18] 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*, pp. 437–447, 1996.

[19] C.L. Ratzlaff, N. Gopal, and L.T. Pillage, "RICE: Rapid interconnect circuit evaluator," in *Proc. DAC*, 1999, pp. 555–560.

[20] J. Qian, S. Pullela, and L.T. Pillage, "Modeling the effective capacitance of RC interconnect," *IEEE Trans. CAD*, vol. 13, no. 12, pp. 1526–1535, Dec. 1994.

[21] M. Bazaraa, H. Sherali, and C. Shetty, *Nonlinear Programming: Theory and Algorithms*, Wiley, New York, NY, 2nd edition, 2003.

[22] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Snagiovanni, "SIS: A system for sequential circuit synthesis," in *Memorandum no. M92/41, ERL, UCB*, May 1992.

[23] *CPMO-constrained placement by multilevel optimization: Multilevel Optimization with Nonconvex Nonlinear Programming for Large Scale Circuit Placement with Complex Constraints*, http://ballade.cs.ucla.edu/cpmo. Computer Science Department, UCLA, mpl6 edition, 2007.

[24] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. DAC*, Jun. 1987, pp. 341–347.

[25] K. Chaudhary and M. Pedram, "A near optimal algorithm for technology mapping minimizing area under delay constraints," in *Proc. DAC*, 1992, pp. 492–498.

[26] M. Pedram and N. Bhat, "Layout driven technology mapping," in *Proc. DAC*, 1991, pp. 99–105.

[27] J. Lou, A. H. Salek, and M. Pedram, "An exact solution to simultaneous technology mapping and linear placement problem," in *Proc. ICCAD*, 1997, pp. 671–675.

[28] J. Y. Lin, A. Jagannathan, and J. Cong, "Placement-driven technology mapping for LUT-Based FPGA's," in *Proc. ISFPGA*, 2003, pp. 121–126.

[29] W. Gosti, S. R. Khatri, and A. L. Sangiovanni-Vincentelli, "Addressing timing closure problem by integrating logic optimization and placement," in *Proc. ICCAD*, November 2001, pp. 224–231.

[30] D. Pandini, L. T. Pileggi, and A. J. Strojwas, "Global and local congestion optimization in technology mapping," *IEEE Trans. CAD*, vol. 22, no. 4, pp. 498–505, 2003.

[31] X. Wang and S. Burns, "Technology mapping using a fixed delay and variable area-power model," in *Proc. IWLS*, Jun. 2007, pp. 171–177.

[32] J. Cong, J. Shinnerl, M. Xie, T. Kong, and X. Yuan, "Large-scale circuit placement," *ACM Trans. on Design Automation of Electronic Systems*, vol. 10, no. 2, pp. 1–42, 2005.

[33] M. Fischer and M. Paterson, "Optimal tree layout (preliminary version)," in *Proc. STOC*, 1980, pp. 177–189.

[34] S. Chatterjee, Z. Wei, A. Mischenko, and R. Brayton, "A linear time algorithm for optimum tree placement," in *Proc. IWLS*, Jun. 2007, pp. 92–98.

[35] M. Yannakakis, "A polynomial algorithm for the min-cut linear arrangement of trees," *Journal of the Association for Computing Machinery*, vol. 32, no. 4, pp. 950–988, Oct. 1985.

[36] E. M. Sentovich, "SIS: A system for sequential circuit synthesis," Memorandum No. UCB/ERL M92/41, May 1992.

[37] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force-directed method for circuit placement," in *Proc. ISPD*, Apr. 2005, pp. 185–192.

[38] H. Chen, H. Zhou, F. Y. Young, D. F. Wong, H. H. Yang, and N. Sherwani, "Integrated floorplanning and interconnect planning," in *Proc. ICCAD*, 1999, pp. 354–357.

[39] "Berkeley predictive technology model," `http://www-device.eecs.berkeley.edu/~ptm/download.html2005`.

[40] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proc. ISFPGA*, Feb. 2000, pp. 203–213.

[41] 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.

[42] C. K. Cheng J. Lillis 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, March 1996.

[43] B. Liu I. I. Mandoiu C. J. Alpert, A. B. Kahng and A. Z. Zelikovsky, "Minimum buffered routing with bounded capacitive load for slew rate and reliability control," *IEEE Transactions on Computer-Aided Design*, vol. 22, no. 3, pp. 241–253, March 2003.

[44] A. Devgan C. J. Alpert 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.

[45] C. J. Alpert J. Hu Z. Li, C. N. Sze and W. Shi, "Making fast buffer insertion even faster via approximation techniques," in *Proc. Asia and South Pacific Design Automation Conference*, 2005, pp. 13–18.

[46] N. Govindaraju M. Harris J. Krger A. Lefohn T. Purcell J. Owens, D. Luebke, "A survey of general-purpose computation on graphics hardware," in *Proc. of Eurographics*, Aug. 2005, pp. 21–51.

[47] K. Gulati and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proc. ACM/IEEE DAC*, Jun. 2008, pp. 822–827.

[48] Z. Feng and P. Li, "Multigrid on GPU: Tackling Power Grid Analysis on Parallel SIMT Platforms," in *Proc. ACM/IEEE ICCAD*, Nov. 2008, pp. 647–654.

[49] Y. Liu and J. Hu, "A new algorithm for simultaneous gate sizing and threshold voltage assignment," in *Proc. of ACM International Symposium on Physical Design*, Mar. 2009, pp. 27–34.

[50] T. Wu and A. Davoodi, "Pars: Fast and near-optimal grid-based cell sizing for library-based design," in *Proc. ACM/IEEE ICCAD*, Nov. 2008, pp. 107–111.

[51] L.P.P.P. van Ginneken, "Buffer Placement in Distributed RC-Tree Networks for minimal Elmore Delay," in *Proc. IEEE International Symposium on Circuits and Systems*, May 1990, pp. 865–868.

[52] "Nvidia cuda homepage," in *http://www.nvidia.com/object/cude_home.html*, 2009.

APPENDIX A

PROOF OF PROPERTY 4 IN CHAPTER IV

Proof 5 *According to property 2, in order to prove that $s(v_i)$ is inferior to $s'(v_i)$, we just need to show that with any complementary solution $u(v_i)$, overall solution $s(v_i) \cup u(v_i)$ is inferior to $s'(v_i) \cup u(v_i)$.*

We prove this by induction on the steps of solution propagation from current node to the source.

Basic step: at the current node v_i, following inferiority relation holds on $s(v_i)$ and $s'(v_i)$.

$$\hat{w}(v_i) \geq \hat{w}'(v_i), \hat{c}(v_i) \geq \hat{c}'(v_i), \text{ and } \hat{q}(v_i) \leq \hat{q}'(v_i)$$

Induction step: for any two consecutive nodes v_j and its parent v_k on the path from v_i to the source v_0, if the inferior relation holds on two solutions at v_j, i.e.,

$$\hat{w}(v_j) \geq \hat{w}'(v_j), \hat{c}(v_j) \geq \hat{c}'(v_j), \text{ and } \hat{q}(v_j) \leq \hat{q}'(v_j),$$

then the inferior solution relation is also held on the solutions, $s(v_k)$ and $s'(v_k)$ at v_k, which are derived from $s(v_j)$ and $s'(v_j)$ respectively, i.e.,

$$\hat{w}(v_k) \geq \hat{w}'(v_k), \hat{c}(v_k) \geq \hat{c}'(v_k), \text{ and } \hat{q}(v_k) \leq \hat{q}'(v_k).$$

Apparently, if the induction above stands, we have $\hat{w}(v_0) \geq \hat{w}'(v_0)$, $\hat{c}(v_0) \geq \hat{c}'(v_0)$, and $\hat{q}(v_0) \leq \hat{q}'(v_0)$ at the source v_0, i.e., $s(v_i) \cup u(v_i)$ is inferior to $s'(v_i) \cup u(v_i)$.

The basic step holds immediately from the problem. Now, we prove the induction step. Each solution propagation step from node v_j to its parent v_k is composed of two operations: merging of solutions from the children of v_k, and adding buffer and wire delay on the fanin of v_k.

First, we prove that when $s(v_j)$ and $s'(v_j)$ are merged with a solution $s(v_l)$ from its sibling v_l, the new merged solutions $s(v_k)$ and $s'(v_k)$ formed at their parent v_k still have the relation, $\hat{w}(v_k) \geq \hat{w}'(v_k)$, $\hat{c}(v_k) \geq \hat{c}'(v_k)$, and $\hat{q}(v_k) \leq \hat{q}'(v_k)$. By the definition of iso-cap nets, $\hat{w}(v_k) \geq \hat{w}'(v_k)$ and $\hat{c}(v_k) \geq \hat{c}'(v_k)$ are trivially true. Here, $\hat{q}(v_k) \leq \hat{q}'(v_k)$ is proved by contradictory.

Assume $\hat{q}(v_k) > \hat{q}'(v_k)$ instead. Since merging operation changes each q value in a solution in a non-increasing way, in order for $\hat{q}(v_k)$ to become larger than $\hat{q}'(v_k)$, $\hat{q}'(v_k) < \hat{q}'(v_j)$ must hold.

Without loss of generality, let

$$\hat{q}'(v_k) = q'(v_k, \phi_1). \tag{A.1}$$

The fact $\hat{q}'(v_j) \leq q'(v_j, \phi_1)$ and the above relation $\hat{q}'(v_k) < \hat{q}'(v_j)$ imply $q'(v_k, \phi_1) < q'(v_j, \phi_1)$, which leads to $q(v_l, \phi_1) < q'(v_k, \phi_1)$. Then, we have

$$q(v_k, \phi_1) \leq q(v_l, \phi_1) < q'(v_k, \phi_1). \tag{A.2}$$

Since $\hat{q}(v_k) = \min_r q(v_k, \phi_r)$, we have

$$\hat{q}(v_k) \leq q(v_k, \phi_1). \tag{A.3}$$

From (A.1), (A.2) and (A.3), we get $\hat{q}(v_k) \leq \hat{q}'(v_k)$, which contradicts with the assumption $\hat{q}(v_k) > \hat{q}'(v_k)$.

Therefore, we have proved that $\hat{q}(v_k) \leq \hat{q}'(v_k)$ holds after merging operation.

Next, we show that the relation also holds after adding buffer and wire delay at the fanin of v_k. Since every solution has identical capacitance for all instances, all instances are shifted by the same amount of delay in a solution. Because $\hat{c}(v_k) \geq \hat{c}'(v_k)$, the buffer and wire delay d for $s(v_k)$ is larger than d' for $s'(v_k)$. As a result, $\hat{q}(v_k) - d \leq \hat{q}'(v_k) - d'$ still holds, since $\hat{q}(v_k) \leq \hat{q}'(v_k)$. Thus, after adding buffer and

wire delay, $\hat{w}(v_k) \geq \hat{w}'(v_k)$, $\hat{c}(v_k) \geq \hat{c}'(v_k)$, *and* $\hat{q}(v_k) \leq \hat{q}'(v_k)$ *still hold.*

Therefore, the induction step is proved by showing that merging solutions and adding buffer and wire delay preserve the inferiority relation in each induction step.

APPENDIX B

PROOF OF THE NP-COMPLETENESS OF MAX-SUCCEEDING-GROUP PROBLEM IN CHAPTER V

The NP-completeness of **max-succeeding-group** is proved by reducing **CLIQUE** problem to an auxiliary problem **MIN-EDGECOVER**, which in turn is reduced to **MIN-DEPCOVER**. Then, we show that MIN-DEPCOVER is equivalent to max-succeeding-group.

Before getting into the first part of our proof, we introduce the concept of edge cover. An edge is covered by a node, if the node is its source or sink. The problem MIN-EDGECOVER asks if b nodes from the node set, V in a graph $G(V, E)$ can be selected, such that the number of edges covered by the nodes is at most a.

Lemma 1 MIN-EDGECOVER *is NP-complete.*

Proof 6 *First, MIN-EDGECOVER* $\in$ *NP, because checking if a set of* b *nodes covers at most* a *edges takes linear time.*

Second, MIN-EDGECOVER is NP-hard, because CLIQUE *is polynomial-time reducible to MIN-EDGECOVER, i.e.,* CLIQUE $\leq_P$ *MIN-EDGECOVER. We construct a function to transform a CLIQUE problem to a MIN-EDGECOVER problem. For a problem that asks if a* b*-node complete sub-graph can be found in* $G(V, E)$*, the corresponding MIN-EDGECOVER problem is whether there exists a set of* $|V| - b$ *nodes in* $G(V, E)$ *that covers at most* $|E| - b(b-1)/2$ *edges. If the answer to the first question is true, then the* $|V| - b$ *nodes apart from the* b *nodes in the complete sub-graph found in the first question are nodes that cover the* $|E| - b(b-1)/2$ *edges outside the complete sub-graph. In this case, the answer to the second question is also*

true. On the other hand, if the answer to the second question is true, then at least $b(b-1)/2$ *edges other than the* $|E| - b(b-1)/2$ *edges found in second question are between the rest b nodes. Then, the b nodes make a complete graph. Thus, the answer to the first question is true. Therefore, the answer to each of the two problems above is true if and only if the answer to the other is positive too.*

Due to the simply arithmetic calculation, the transform of the reduction above clearly runs in $O(1)$ *time. Therefore,* $\mathrm{C}LIQUE \leq_P$ *MIN-EDGECOVER.*

By the two steps of reasoning above, it is proved that MIN-EDGECOVER problem is NP-complete.

Next, in the second part of our proof, we show that MIN-EDGECOVER is polynomial-time reducible to the problem MIN-DEPCOVER, which is equivalent to MAX-INDEPSET. Independent graph is a directed graph $G(V, E')$ with two sets of nodes V_1 and V_2 ($V = V_1 \bigcup V_2$, $V_1 \bigcap V_2 = \emptyset$), and every edge $e' \in E'$ originates from a node in V_1 and ends at a node in V_2. A node v_2 in V_2 is said to be dependently covered by a node v_1 in V_1, if there is an edge from v_1 to v_2. MIN-DEPCOVER asks if a set of b nodes can be selected from V_1, so that at most a nodes in V_2 are dependently covered.

Lemma 2 MIN-DEPCOVER *is NP-complete.*

Proof 7 *First, MIN-DEPCOVER* $\in$ *NP, because checking if a set of b nodes in* V_1 *dependently cover at most a nodes in* V_2 *takes polynomial-time.*

Second, we verify that MIN-DEPCOVER is NP-hard by showing MIN-EDGECOVER $\leq_P$ *MIN-DEPCOVER. We transform any given MIN-EDGECOVER problem on* $G(V, E)$ *into a* $\mathrm{M}IN$*-DEPCOVER problem on* $G(V_1 \bigcup V_2, E')$*. Each node in V corresponds to a node in* V_1*, and each edge in E corresponds to a node in* V_2*. An edge* $e' \in E'$ *from node* $v_1 \in V_1$ *to* $v_2 \in V_2$ *exists if and only if the node v corresponding*

to v_1 is adjacent to the edge e corresponding to v_2. If the answer to the first question is true, then there are b nodes in V_1 that dependently-covers at most a nodes in V_2, therefore, the answer to the second question is true too. And vice versa, if the answer to the second question is true, then there are b nodes in V that covers at most a edges corresponding to a nodes in V_2. So, the answer to the first question is true. Therefore, each of the positive answers to the two problems holds if and only if the other one holds.

During the construction of the corresponding MIN-DEPCOVER problem given a MIN-EDGECOVER problem, going through every edge in $G(V, E)$ and its adjacent nodes takes $O(E)$ time. So, the transform above takes polynomial time. Consequently, MIN-EDGECOVER $\leq_P$ MIN-DEPCOVER.

By the two steps of reasoning above, it is proved that MIN-DEPCOVER problem is NP-complete.

In the final part of our proof, we show that max-succeeding-group problem is NP-complete. Before getting into it, we review the concept in max-succeeding-group problem. The decision problem of max-succeeding-group asks if a set of b gates from current gate group can be chosen as concurrent gate group, so that the size of the succeeding independent group is at least a.

Theorem 1 max-succeeding-group *is NP-complete.*

Proof 8 *First, max-succeeding-group $\in$ NP, because it takes polynomial time to check if a set of b independent gates enable at least a prospective gates to become independent gates.*

Second, we verify that max-succeeding-group is NP-hard by showing MIN-DEPCOVER $\leq_P$ max-succeeding-group. It is true simply because max-succeeding-group is equivalent to MIN-DEPCOVER. The situation that at least a gates in the

prospective group are made independent by selecting b gates from the independent current set is the same as the situation that at most $|prospective\_group| - a$ *gates in the prospective group are prevented from becoming independent gates by excluding* $|independent\_current\_group| - b$ *independent gates from entering the concurrent group. The two statements above are sufficient and necessary conditions to each other, since they are equivalent.*

Clearly, above transform takes $O(1)$ *time, therefore, MIN-DEPCOVER* $\leq_P$ *max-succeeding-group.*

By the two steps of reasoning above, it is proved that max-succeeding-group problem is NP-complete.

In fact, the NP-completeness of MAX-THROUGHPUT can be verified by a straightforward polynomial-time transform that reduces max-succeeding-group to MAX-THROUGHPUT.

VITA

Yifang Liu

Department of Electrical and Computer Engineering

Texas A&M University

TAMU 3128

College Station, Texas 77843-3128

Yifang Liu received his B.S. and M.S. degrees in computer science from the University of Electronic Science and Technology of China and from the University of Maryland, Baltimore County respectively. He received his Ph.D. degree in 2010 from Texas A&M University, College Station, Texas. During his doctoral studies, he worked as an intern at IBM, T.J. Watson Research Center, from May 2008 to September 2008, and at AMD CAD Department, from September 2008 to January 2009. His research interests include algorithm design and analysis, applied optimization, high performance computing, and their applications in VLSI circuit design automation.

Yifang Liu is a recipient of a TAMU research scholarship, a finalist of the Best Paper award of ISPD, a flagship conference in VLSI physical design. He is a member of the honor society of Phi Kappa Phi and the computer science honor society of Upsilon Pi Epsilon.

The typist for this dissertation was Yifang Liu.