UNIVERSITY OF CALIFORNIA, SAN DIEGO

Static Timing Analysis in VLSI Design

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Shuo Zhou

Committee in charge:

Professor Chung-Kuan Cheng, Chair
Professor Paul M. Chau
Professor Fan Chung Graham
Professor Ronald Graham
Professor Tajana Rosing

2006

.

The dissertation of Shuo Zhou is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2006

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Chung-Kuan Cheng for his encourage, support, and insightful guidance. I learned from him not only the knowledge of the field but also the wisdoms of life. I wish to thank my dissertation committee members, Professor Ronald Graham, Professor Tajana Simunic Rosing, Professor Fan Chung Graham and Professor Paul Chau for their time and efforts. I would like to express my appreciation to Mike Hutton at Altera for technical discussions and his advices and reviews of my papers.

I am grateful to all the graduate students in the UCSD VLSI CAD group for making the group a fun place to work. Special thanks to Zhengyong Zhu, Bo Yao, Hongyu Chen, Jianhua Liu, Rui Shi, He Peng, Haikun Zhu, Yuanfang Hu, Yi Zhu, Ling Zhang, Renshen Wang, and Wanping Zhang. In particular, I would like to thank Hongyu Chen, Yi Zhu and Yuanfang Hu for technical discussions.

I am deeply thankful to Michael Jackson and Alan Lam at Synopsys, and Rick Pier at Mentor Graphics. They provided me valuable intern opportunities and exposed me to a different field that complements my research work in school.

I would like to thank my parents and brother for their support and care. In particular, I own a lot of thanks to my father for his guidance on my career and constant encourage ever since my childhood. My final thanks go to my husband, Shengrong Lin, for his love, care and patience. My Ph.D. study could not have been completed without him.

The research presented in this dissertation was supported by grants from the California MICRO program and Altera Corporation. Their support is greatly acknowledged.

The text of chapter III, in full, is a reprint of the material as it appears in Proc. Intl. Conf. on Computer-Aided Design 2005, Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, Mike Hutton, Truman Collins, Sridhar Srinivasan, Nanchi Chou, and Peter Suaris, "Improving the Efficiency of Static Timing

Analysis with False Paths", ICCAD 2005. The text of chapter IV, in full, is a reprint of the material as it appears in Proc. Asia and South Pacific Design Automation Conf. 2006, Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, and Mike Hutton, "Efficient Static Timing Analysis Using a Unified Framework for False Paths and Multi-Cycle Paths", ASP-DAC 2006. The dissertation author was the primary researcher and author and the co-authors listed in these publications directed and supervised the research which forms the basis for this chapter.

VITA

1976	Born, Beijing, China
1999	B.S. in Computer Science and Technology Tsinghua University, Beijing, China
2001	M.S. in Computer Science and Technology Tsinghua University, Beijing, China
2006	Ph.D. in Computer Science University of California, San Diego

PUBLICATIONS

S. Zhou and B. Yao and H. Chen and Y. Zhu and C.-K. Cheng and M. Hutton, Efficient Static Timing Analysis Using a Unified Framework for False Paths and Multi-Cycle Paths, ASP-DAC 2006, pp. 73-78

S. Zhou and B. Yao and H. Chen and Y. Zhu and C.-K. Cheng and M. Hutton and et al., Improving the efficiency of Static Timing Analysis with False Paths, ICCAD 2005, pp. 527-531

S. Zhou, B, Yao, J. Liu, C.-K. Cheng, Integrated Algorithmic Logical and Physical Design of Integer Multiplier, ASP-DAC 2005, pp. 1014- 1017

J. Liu, S. Zhou, H. Zhu, C.-K Cheng,An Algorithmic Approach for Generic Parallel Adders, ICCAD 2003, pp. 734-740

FIELDS OF STUDY

Major Field: Computer Science
Studies in VLSI CAD.
Professor Chung-Kuan Cheng

ABSTRACT OF THE DISSERTATION

Static Timing Analysis in VLSI Design

by

Shuo Zhou
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Chung-Kuan Cheng, Chair

The increasing complexity of digital designs and the requirement of timing measurements in various design stages make static timing analysis critical. Each design stage utilizes static timing analysis to evaluate the system performance, and then optimizes the design accordingly. An accurate and efficient timing analysis package is crucial for the success of the whole design process. We studied three important problems in static timing analysis: false paths, multi-cycle paths, and hierarchical timing analysis.

Static timing analysis should deal with false paths and multi-cycle paths to produce accurate timings. Previous published works focused on dealing with false paths in static timing analysis. There are several approaches, such as labeling algorithm and node-splitting approach, using tags to label and eliminate false path timings. A large number of tags need to be created and propagated. Thus, the efficiency is deteriorated. For hierarchical timing analysis, timing analysis iteratively performed on flatten circuits suffers from low efficiency because of increasing design complexity.

Inspired by the gap between challenges and current sub-optimal solutions, we proposed three new techniques:

1. A two-direction propagation is proposed which minimizes the number of tagged false path timings using a biclique covering approach. We proved that all the non-false paths are covered and all the false paths are removed. A polynomial heuristic to perform the biclique covering minimization in minimal degree order is also introduced.

2. A framework unified processing false paths and multi-cycle paths is proposed which expands the tag-based approach for false paths to cover multi-cycle paths. Following the biclique covering approach for false paths, we devise time shifting for multi-cycle paths to improve the efficiency. We prove that the unified framework produces accurate timings.

3. An abstract timing model reduction technique is proposed for hierarchical timing analysis. We introduce an iterative biclique-star replacement technique to minimize the abstract timing model, thus improving the efficiency.

Combined the techniques proposed above, we provide an accurate and efficient static timing analysis package, which can be used in hierarchical design methodology.

I Introduction

I.A Introduction

Efficient chip analysis becomes essential in nowadays circuit design due to the increasing design complexity and rapid technology change. Moore's law predicts that the number of transistors integrated on a die would be doubled every 18 to 14 months. With exponential growth in the number of transistor per chip, the design complexity keeps increasing. Meanwhile, the rapid technology change shortens product life cycles and makes time-to-market a critical issue. As a result, efficient chip analysis in early design stage is essential for reducing the number of design iterations in spite of increasing complexity.

Timing analysis is crucial for high performance digital circuit design as a part of chip analysis. Traditionally, high-performance integrated circuits are characterized by the clock frequency at which they operate. The timing analysis algorithm gauges the ability of a circuit operating at the specified speed. Two methods are used for timing analysis: dynamic simulation and static timing analysis. Dynamic simulation methods perform circuit level simulation, which suffers from high complexity when the circuit is large. Compared with dynamic simulation, static timing analysis is input vector independent. By performing worst case analysis, the analysis complexity is linear. Therefore, in early design stages static timing analysis is the widely used mechanism for timing measurement.

Motivated by the gap between challenges of increasing design complexity and limited accuracy and efficiency, our dissertation focused on three problems

in static timing analysis: false paths, multi-cycle paths and hierarchical timing analysis.

The remainder of the chapter is organized as follows. Section I.B introduces the spectrum of chip analysis. Section I.C introduces static timing analysis in VLSI design flow. Section I.D formulates the problems of false paths, multi-cycle paths, and hierarchical timing analysis. Section I.E is the dissertation organization.

I.B Spectrum of Chip Analysis

Chip analysis is performed on various levels to verifies circuit function and measures circuit timing. Fig.I.1 illustrates the analysis spectrum. Ordered from high-level to low-level, the simulations become more accurate, but they also become progressively more complex and time consuming.

Function Analysis Function level analysis includes behavior simulation and RTL simulation. The behavior of circuit module is modeled and tested. Gate delays and interconnect delays are ignored.

Logic Analysis Logic level analysis is used to verify both circuit functions and circuit timings. The circuit is extracted into models on various levels. Based on the models, analysis are performed on various levels including static timing analysis, gate-level simulation and switch-level simulation. Static timing analysis algorithms compute circuit timing without considering circuit functions. Gate-level simulator treats Boolean gates as black box which models both function and delay information. Switch-level simulator models transistors as switches. It is more accurate than gate-level simulation.

Circuit Analysis Circuit level analysis is used to simulate the electrical behavior of circuit based on circuit theories. The simulator requires models of transistors, describing their nonlinear voltage and current characteristics.

Fabrication Process Simulation Process simulation is used to simulate major

process steps in modern semiconductor fabrication technology based on physical models for diffusion, implantation, oxidation, silicidation and epitaxy.

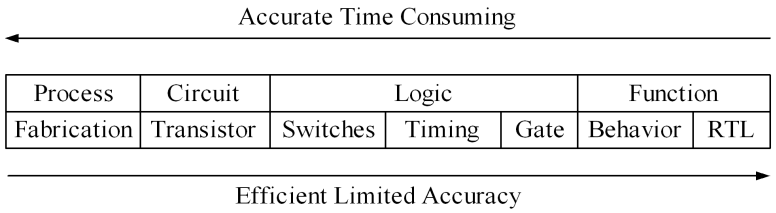


Figure I.1 Spectrum of Chip Analysis

I.C Static Timing Analysis in Design Flow

Static timing analysis algorithms evaluate circuit timing with linear complexity. Compared with dynamic simulation, static timing analysis is input vector independent. The circuit is formulated into a directed acyclic graph. On the graph, the worst case circuit timings, that is, the longest and shortest timing propagation paths in the circuit, are computed and verified with specified clock cycle. As a result, the analysis complexity is linear with respect to the number of edges in the timing graph.

Static timing analysis plays a vital role in nowadays design flow. Besides longest and shortest timing propagation paths, static timing analysis can be used to compute arrival times, required arrival times and slacks at all points in the circuit, which are useful inputs for circuit optimizations. Therefore, each design stage, from floorplaning, logic synthesis, to placement and routing, utilizes static timing analysis to evaluate system performance, and then optimizes the design accordingly (Fig.I.2). For example, in performance-driven placement, the static timer reports critical paths, i.e., the paths violating the expected timing constraints, and slacks of non-critical paths to the placement step. Based on the timing reports, the placement optimization would optimize critical paths while sacrificing non-critical paths. Most of contemporary EDA companies develop static timing engines, such

as Prime Time [2] of Synopsys and incremental common timing engine (CTE) [1] of Cadence. These timing engines are used across the entire synthesis/place-and-route flow.

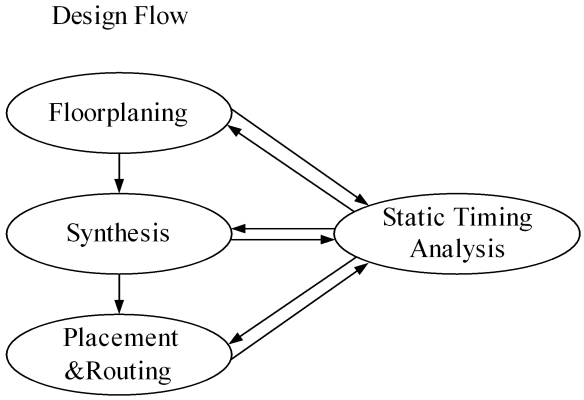


Figure I.2 Static Timing Analysis in the Design Flow

In this scenario, a successful design process relies on the accuracy and efficiency of static timing analysis. Traditional static timing analysis has limited accuracy due to the use of simplified delay models. Because the function of the circuit is not considered, static timing analysis may include false paths, which are paths not logically existing in the circuit. As a result, the optimization may wrongly focus on false paths and miss the real critical path. In general, conservative timing analysis leads to over-design, which increases the product cost. Meanwhile, as part of the inner loop of the optimization, the efficiency of static timing analysis becomes crucial. The improvement on accuracy should not deteriorate the efficiency.

I.D Problem Formulation

This section introduces the problems in static timing analysis we studied.

I.D.1 The Problem of False Paths

Static timing analysis should process false paths to produce accurate timings. A path which never be activated by any input vector is called *false path*. A false path example is shown in Fig.I.3. The path $B- > 1- > 2- > 3- > X_2$ is a false path. In order to allow the signal go through the path from input B to output X_2, the side inputs of the gates along the path should have non-dominant values, i.e., 0 for OR gates and 1 for AND gates. As a result, for OR gate 1 the logic value of input C should be 0, and for AND gate 3, the logic value of input C to be 1. Since there is a conflict, the signal can not go through path $B- > 1- > 2- > 3- > X_2$ to reach the output X_2 under any input vector. As a result, the path will never be activated.

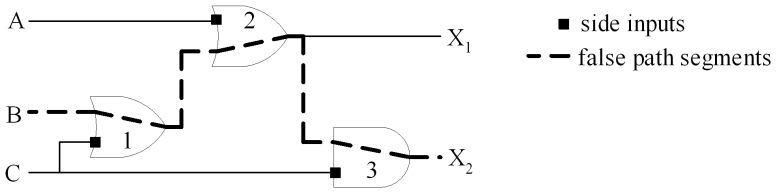


Figure I.3 False Path

False path problem has been studied extensively and a number of algorithms have been proposed to detect and eliminate false paths. The false problem was first addressed in [4], and the NP-completeness of the problem was proved in [32]. The criterions of false paths were presented in [7, 8, 11–13, 31]. A number of algorithms have been proposed to detect and eliminate false paths from timing analysis. The satisfiability based heuristic algorithms were proposed in [7, 8, 22]. Based on the idea of data/control separation on primary inputs proposed in [45], an approximation scheme was proposed in [29], which controls the size of SAT problems.

We focused on the problem of timing analysis with specified false paths, which was first formulated and analyzed in [3].

Timing Analysis with Specified False Paths Problem Given the timing graph

of a circuit and a set of false paths, compute signal arrival times, required times and slacks at every vertex in the timing graph while disregarding false paths timings.

This problem is useful for several reasons. In many cases, users may know certain paths in circuits are false. Therefore, we should allow user to specify false paths and remove false path timing during analysis. Furthermore, based on a loose criterion defined in [du93], one can prove that delays in circuits will not be underestimate if false paths detected before analysis are removed from consideration during a series of physical optimization steps. Therefore, for repeatedly invoked timing analysis, there is no need to re-determine false paths during optimization.

I.D.2 The Problem of Multi-cycle Paths

A *multi-cycle path* is a path that signals propagate longer than one clock cycle. Fig.I.4 illustrates a multi-cycle path. The paths from flip flop FF_4 to flip flop FF_5 are designed to be two-cycle paths, thus allowing the signals propagate through the slow logic.

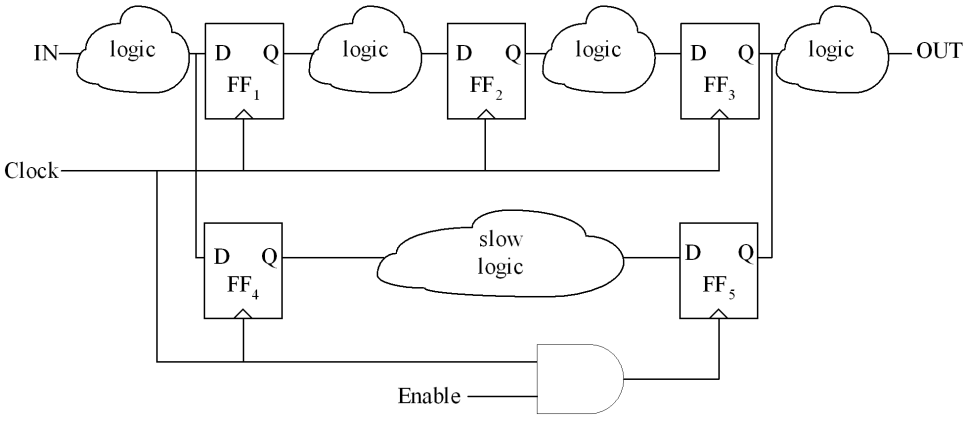


Figure I.4 Multi-cycle Path

There are several published works dealing with multi-cycle paths. In [30], multi-cycle paths are considered as false paths in sequential circuits and ignored in minimum cycle computation. Various techniques were presented to detect and

eliminate multi-cycle false paths using sequential path tracing [17], state encoding [19], and non-false multi-cycle paths traversal [35]. Furthermore, multi-cycle false path elimination is exploited in performance optimization [37, 40].

Similarly as the problem of false paths, we can formulate timing analysis with specified multi-cycle paths as follows.

Timing Analysis with Multi-cycle Paths Given the timing graph of a circuit and a set of multi-cycle paths, compute signal arrival times, required times and slacks at every vertex in the timing graph while constraining multi-cycle paths using multi-cycle required time.

I.D.3 The Problem of Hierarchical Timing Analysis

Hierarchical timing analysis approach can substantially reduce the computational complexity. A design is divided into multiple blocks Fig.I.5.(a) and each block is characterized into a timing model Fig.I.5.(b). For linear delay model, the timing calculation can be separated according to the boundary of the partitions. Assume the timing relations inside each block are fixed. By using pre-calculated timing models in timing analysis, we hide the details inside the blocks, thus reducing the analysis complexity.

Previous works on hierarchical timing analysis focused on characterizing accurate timing model using functional information. Conditional delay matrix was presented to characterize the timing model with timing propagation conditions [44]. Multiple modes of operation were considered in [46, 47]. The idea of data/control separation on primary inputs was proposed in [45] to reduce the characterization complexity. A tighter sensitization criterion was used for false paths elimination [28].

Our research is mainly about abstract timing model reduction. After characterizing a functional block into a abstract timing model, the analysis complexity is linear to the number of edges in the abstract timing model for timing propagation. Therefore, our objective is to minimize the number of edges in the

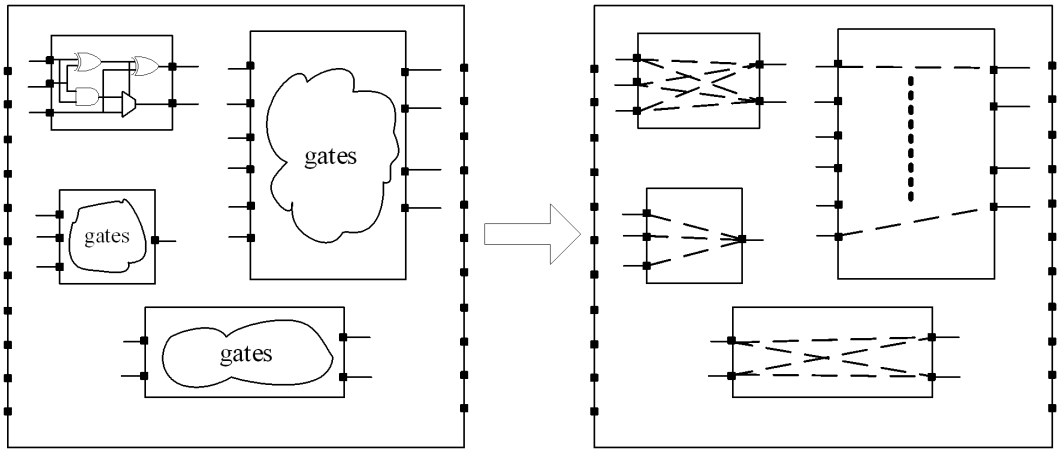


Figure I.5 Hierarchical Timing Analysis

timing model.

Abstract Timing Model Reduction Given a hierarchical block, characterize an abstract timing model covering the longest and shortest path delays between any pair of input and output pins of the hierarchical block that minimizes the number of edges in the timing model.

I.E Dissertation Organization

The remainder of the dissertation is organized as follows.

Chapter II reviews static timing analysis process, including the timing graph, timing constraints, delay models, the arrival and required arrival time propagation. The previous works on false paths and multi-cycle paths are also introduced.

In chapter III, we introduce efficient timing analysis dealing with false paths. We present a two-direction propagation technique which minimizes tagged false path timings using a biclique covering approach. We prove that the proposed approach can cover non-false path timings while eliminating false path timings. Finally, the experimental results are presented.

Chapter IV presents a unified framework for false paths and multi-cycle paths. The framework processes false paths and multi-cycle paths by a tag-based approach. We follow the biclique covering approach for false paths to minimize the number of tagged timings. The approach is efficient according to the computational complexity analysis. In the end, the correctness is proved and the experimental results are presented.

In chapter V, we talk about abstract timing model reduction. We introduce a biclique-star replacement technique to minimize the number of edges for timing propagation. Based on the biclique-star replacement technique, an iterative timing model reduction algorithm is presented. We apply the unified framework presented in chapter IV to cover false paths and multi-cycle paths in timing model. Finally, the experimental results are given.

Chapter VI gives conclusions and discusses future works.

II Background and Previous Works

II.A Introduction

In this chapter, we briefly review static timing analysis algorithms and introduce previous works on the problems we studied. We first introduce the concepts evolved in static timing analysis including the timing graph, the interconnect and device model and the setup/hold time constraints. Based on these concepts, we introduce static timing analysis algorithms of the arrival time, required time and slack computation. In the end, we review several previous works on timing analysis with specified false paths and multi-cycle paths, and timing model reduction for hierarchical timing analysis.

II.B Static Timing Analysis Overview

In static timing analysis, digital circuits are transformed into timing graph, and then the delays of the paths are verified with the corresponding timing constraints. We first review the concepts evolved in timing analysis, and then introduce the static timing analysis algorithm.

II.B.1 Timing Graph

The timing graph is a directed acyclic *graph* $G = \{V, E\}$, where V is a set of vertices and E is a set of edges. Each *edge* (u, v) is an ordered pair from vertex u to vertex v. The timing graph is acyclic because cyclic sequential paths are broken into combinational circuit segments align flip flops, and all the segments are folded into one clock cycle.

The input degree of vertex v, $d^-(v)$, is the number of edges ending at v. The output degree of v, $d^+(v)$, is the number of edges starting at v. The *primary input set* $B = \{v|v \in V, d^-(v) = 0\}$ is the set of primary input vertices. The *primary output set* $D = \{v|v \in V, d^+(v) = 0\}$ is the set of primary output vertices.

The primary inputs of graph G correspond to (1) primary inputs of the circuit, (2)outputs of flip flops, and (3) outputs of memories. The primary outputs of graph G correspond to (1) primary outputs of the circuit, (2) inputs of flip flops, and (3) inputs of memories. All other vertices correspond to inputs and outputs of combinational gates in the circuit. For gates in the circuit, there are edges from inputs to outputs. For nets in the circuit, there are edges from pins as drivers to pins as fan-outs.

Fig.II.1 illustrates an example of the timing graph. A circuit is shown in Fig.II.1.(a). In the circuit, there are two primary inputs, i.e., I_0 and I_1, three primary outputs, i.e., Q_0, Q_1, and Q_2, and three flip flops, i.e., FF_0, FF_1 and FF_2. The input and output pins of gates and flip flops are labeled. For example, the inputs and output of XOR gate A are labeled as 1, 2 and 3. Fig.II.1.(b) illustrates the timing graph G. Each vertex represents a pin in the circuit with the same label. For example, vertex 1 represents pin 1. In the timing graph, the primary input set contains five vertices I_0, I_1, 11, 13, and 15, i.e., $B = \{I_0, I_1, 11, 13, 15\}$. The primary output set contains six vertices Q_0, Q_1, Q_2, 10, 12 and 14, i.e., $D = \{Q_0, Q_1, Q_2, 10, 12, 14\}$. For gates except flip flops, there are edges from inputs to outputs. For example, there are edges from the inputs to the output of XOR gate A, i.e., edge (1,3) and (2,3). For nets, there are edges from the vertex

as the driver to the vertices as the fanouts. For example, there are edges from the output of XOR gate A to the input of XOR gate B, i.e., edge (3,8).

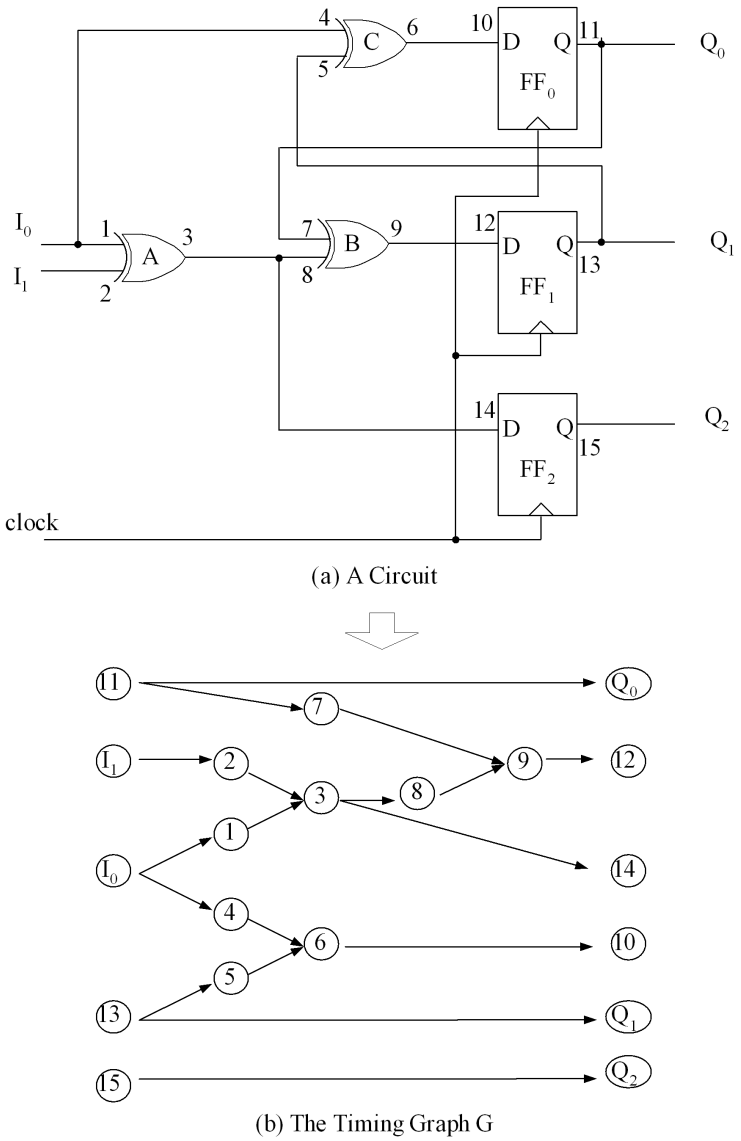


Figure II.1 Timing Graph

A *path* p in graph G is a sequence of vertices and edges. We can represent the path p by only the edges in the path [21]. Each vertex v separates path p into a *head* and a *tail*. Since G is directed acyclic, all the paths in G are simple. A path starting from a vertex in the primary input set B is termed a *prefix path* p^-. A path ending at a vertex in the primary output set D is termed a *suffix path* p^+.

A complete path is both a prefix and a suffix path, which starts from a vertex in the primary input set B, and ends at a vertex in the primary output set D. The *prefix cone* $P^-(v)$ of a vertex v contains all the prefix paths ending at v. The *suffix cone* $P^+(v)$ of v contains all the suffix paths starting at v.

In Fig.II.1, the prefix cone of vertex 3, $P^-(3)$, contains two prefix paths, {(I_1, 2), (2, 3)} and {(I_0, 1), (1, 3)}. The suffix cone of vertex 3, $P^+(3)$, contains two suffix paths, {(3, 8), (8, 9), (9, 12)} and {(3, 14)}. Path {(I_0, 1), (1, 3), (3, 8), (8, 9), (9, 12)} is a complete path from vertex I_0 to vertex 12.

II.B.2 Delay Models

We use interconnect and device models to approximate the delay of gates and nets. The delay of a wire segment, i.e., the delay from a driver pin to a fan-out pin in a net, can be estimated using Elmore model. The delay from an input pin to an output pin of a gate can be evaluated using various device models. In linear delay model, the delay is described with explicit expression including two terms: a fixed intrinsic delay and a propagation delay. The intrinsic delay is independent of the output load. The propagation delay is the portion contributed by the load [9].

The nonlinear delay model uses two-dimension look-up-table to evaluate the gate delay [9]. The index of the first dimension is the output load capacitance, and the index of the second dimension is the input transition times, which is the time period between the 10% and 90% of the waveform.

Given a pair of load C_L and input transition time t_{in}, we find the adjacent pair of indexes with corresponding capacitances C_1, C_2, and transition times t_1, t_2 that bound C_L and t_{in} from both sides, i.e., $C_1 \leq C_L \leq C_2$ and $t_1 \leq t_{in} \leq t_2$ (Fig.II.2). Let $D_i, (1 \leq i \leq 4)$ be the delays at the four points, i.e., $D_1 = D(C_1, t_1), D_2 = D(C_2, t_1), D_3 = D(C_1, t_2), D_4 = D(C_2, t_2)$. Based on $D_i, (1 \leq i \leq 4)$, we approximate delay $D(C_L, t_{in})$ using interpolation. The details of the interpolation can be found in [9].

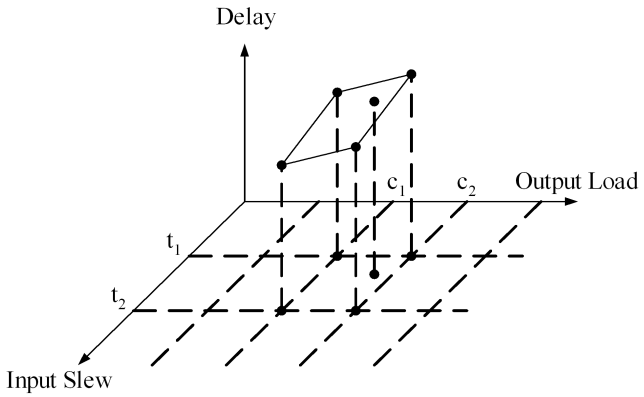


Figure II.2 Two-dimensional Table-lookup Device Model

II.B.3 Timing Constraints

We verify the path delays with *setup time* and *hold time* constraints [26]. The setup time T_s is the minimum time period before the clocking event during which the signal must be stable to be validly captured. The hold time T_h is the minimum time period after the clocking event during which the input signal must be stable to be validly captured (Fig.II.3.(a)). The path delay d_p should satisfy the expressions as follows.

$$d_p < T_{period} - T_s \quad \text{(II.1)}$$

$$d_p > T_h, \quad \text{(II.2)}$$

where T_{period} is the clock cycle.

Fig.II.3.(b) illustrates the setup and hold time constraints on path delays. The signal is launched at the output of flip flop FF_1 by clock i and captured at the input of flip flop FF_2 by clock $i+1$. The longest path should be shorter than one clock cycle minus the setup time. The shortest path should be longer than the hold time. Otherwise, the new value will appear at the input of flip flop FF_2 before FF_2 has chance to catch the previous value from FF_1.

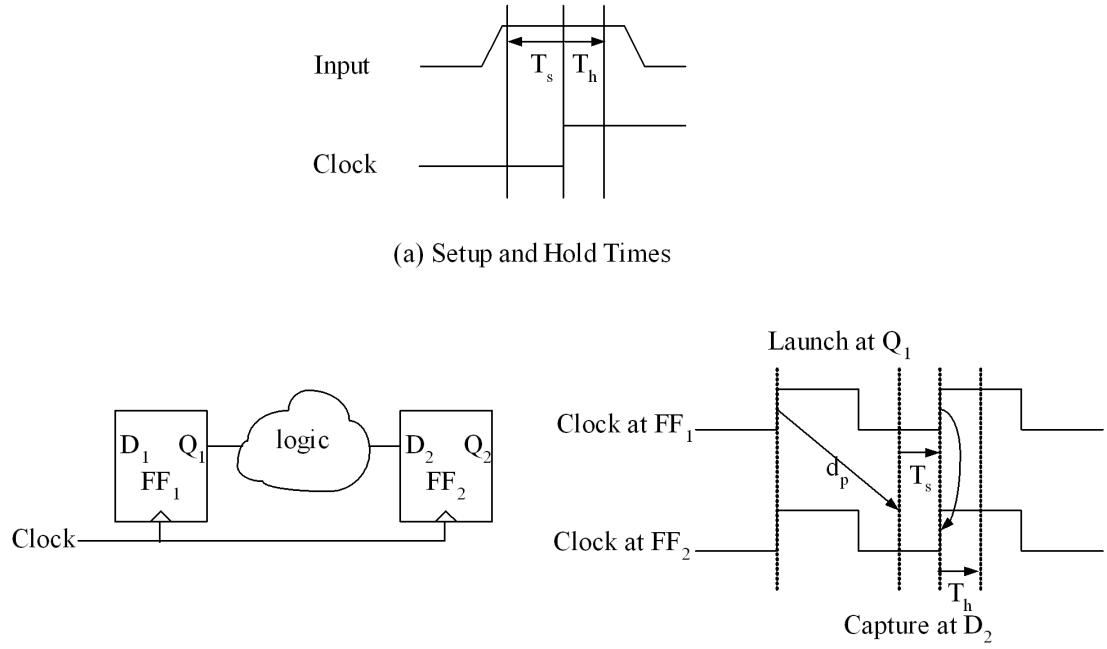


Figure II.3 Setup and Hold Time Constraints on Path Delays

II.B.4 Static Timing Analysis Algorithms

There are two categories of timing analysis techniques: path enumeration techniques and block oriented analysis techniques. Path enumeration starts from a primary output PO in timing graph G and traces back through G until reaching a primary input PI. By doing so, the information of the path from PI to PO is complete. We compute the delays of the gates and nets on the path. As a result, the delay of the whole path equals the sum of all gate and net delays [25,38,41,43]. Since path enumeration techniques enumerate all the paths in the timing graph, it suffers from long run time because the number of paths in the circuit grows exponentially with the size of the circuit. However, by enumerating the paths, it's easier to ignore certain paths in circuits which are never activated.

Block oriented analysis performs longest and shortest path search on the timing graph [20, 24]. It starts from primary inputs PI. At each PI, earliest and latest signal available times are denoted as signal arrival time Arr_{min} and Arr_{max},

respectively. Timing analysis propagates Arr_{min} and Arr_{max} to all the vertices in the timing graph. The earliest and latest arrival times at vertex v are computed as follows:

$$Arr_{min}(v) = \min(Arr_{min}(u) + d(u,v)), \quad \text{(II.3)}$$

$$Arr_{max}(v) = \max(Arr_{max}(u) + d(u,v)), \quad \text{(II.4)}$$

where $d(u,v)$ is the delay of input edge (u,v) of v. Note various interconnect and device models can be used to estimate $d(u,v)$. The Arrival-Time-Propagation algorithm is as follows.

Algorithm: Arrival-Time-Propagation

1. For each vertex v

   If v is primary input add v into $To-Compute$ vertex list;

2. Repeat

   (a) Get v from $To-Compute$ list

   (b) If v is primary input $Arr_{min}(v) = 0, Arr_{max}(v) = 0$;

   (c) else

   For each input edge (u,v)

   A. $Arr_{min}(v) = min(Arr_{min}(v), Arr_{min}(u) + d(u,v))$;

   B. $Arr_{max}(v) = max(Arr_{max}(v), Arr_{max}(u) + d(u,v))$;

   (d) For each output edge (v,w)

   i. $Ready = 1$;

   ii. For each input edge (x,w) of vertex w

   If arrival times of vertex x not available $Ready = 0$;

   iii. If $Ready$ add vertex w into $To-Compute$ list;

   iv. Remove v from $To-Compute$ list;

3. Until $To-compute$ list is empty;

The algorithm calculates arrival times at all vertices in linear time. The computation is performed in topological order, i.e., a vertex v is added into a $To-Compute$ vertex list only when the arrival times of vertices before v are available. By doing so, the arrival times of each vertex is only updated once. Thus, the complexity of timing analysis is linear to the number of edges in the graph. Because the timing graph is acyclic, all the vertices are updated.

The second step involves propagating the required times from primary outputs in a backward pass. At each primary output PO, based on the setup and hold times, the latest required time Req_{max} and earliest required time Req_{min} are as follows:

$$Req_{max} = T_{period} - T_s, \tag{II.5}$$

$$Req_{min} = T_h, \tag{II.6}$$

where T_s is setup time, T_h is hold time, and T_{period} is the clock cycle.

During backward propagation, Req_{max} and Req_{min} at vertex v are computed as follows:

$$Req_{min}(v) = \max(Req_{min}(u) - d(v,u)), \tag{II.7}$$

$$Req_{max}(v) = \min(Req_{max}(u) - d(v,u)), \tag{II.8}$$

where $d(v,u)$ is the delay of output edge (v,u) of v. The Required-Time-Propagation algorithm is as follows.

Algorithm: Required-Time-Propagation

1. For each vertex v

   If v is primary output add v into $To-Compute$ vertex list;

2. Repeat

   (a) Get v from $To-Compute$ list

   (b) If v is primary output $Req_{max} = T_{period} - T_s, Req_{min} = T_h$;

(c) else

For each output edge (v, u)

A. $Req_{min}(v) = max(Req_{min}(v), Req_{min}(u) - d(v, u))$;

B. $Req_{max}(v) = min(Req_{max}(v), Req_{max}(u) - d(v, u)))$;

(d) For each input edge (w, v)

i. $Ready = 1$;

ii. For each output edge (w, x) of vertex w

If required times of vertex x not available $Ready = 0$;

iii. If $Ready$ add vertex w into $To - Compute$ list;

iv. Remove v from $To - Compute$ list;

3. Until $To - compute$ list is empty;

With arrival times and required times, the timing *slack*s at vertex v are defined as follows:

$$slack_{min}(v) = Arr_{min}(v) - Req_{min}(v) \quad \text{(II.9)}$$

$$slack_{max}(v) = Req_{max}(v) - Arr_{max}(v). \quad \text{(II.10)}$$

If $slack_{max}(v)$ is negative, the path violates the setup constraint II.1. If $slack_{min}(v)$ is negative, the path violates the hold constraint II.2.

In Fig.II.4, we use an example to show the timing analysis process. Fig.II.4.(a) illustrates the timing graph with clock cycle $T_{period} = 11$, setup time $T_s = 1$, and hold time $T_h = 5$. The values attached on edges are delays. Fig.II.4.(b) illustrates the earliest and latest arrival times $Arr_{min}(v)/Arr_{max}(v)$ by every vertex v. For example by vertex 5, $Arr_{min}(5) = \min(Arr_{min}(4)+3, Arr_{min}(2)+2) = 2$ and $Arr_{max}(v) = \max(Arr_{max}(4) + 3, Arr_{max}(2) + 2) = 4$.

Fig.II.4.(c) illustrates the earliest and latest arrival times $Req_{min}(v)/Req_{max}(v)$ by every vertex v. For primary output 12, the earliest required time $Req_{min}(12) = T_h = 5$ and the latest required time $Req_{max}(12) = T_{period} - T_s = 10$. For internal

vertex 7, the earliest required time $Req_{min}(7) = \max(Req_{min}(9) - 1, Req_{min}(10) - 2) = 2$, and the latest required time $Req_{max}(7) = \min(Req_{max}(9) - 1, Req_{max}(10) - 2) = 5$.

Fig.II.4.(d) illustrates the slacks $Slack_{min}(v)/Slack_{max}(v)$ by vertices. For example, by vertex 7, $Slack_{min}(7) = Arr_{min}(7) - Req_{min}(7) = 1$ and $Slack_{max}(7)$ = $Req_{max}(7)$ - $Arr_{max}(7)$ = -1. Since $Slack_{max}(7)$ is negative, we trace the negative slack to get the path violating the setup time constraint, i.e., path {(2,6), (6,7), (7,10), (10,11), (11,12)}. Similarly, we can trace the negative $Slack_{min}$s to get the path violating the hold time constraint, i.e., path {(1,4), (4,9), (9,11), (11,12)}.

II.C Previous Works on Timing Analysis Dealing with False Paths and Multi-cycle Paths

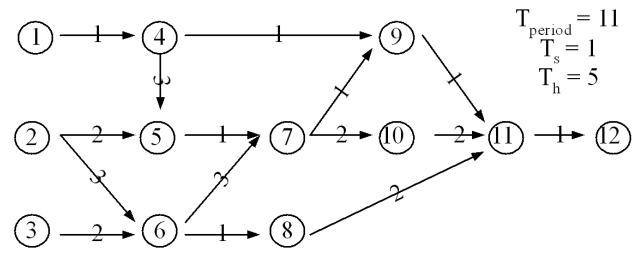
In this section, we first introduce the exceptional rules on timing graph for false paths and multi-cycle paths. Then, we reviews previous works dealing with specified false paths and multi-cycle paths in static timing analysis, i.e., tag-based approach [3, 16] and node-splitting approach [5, 6], and edge-mask based approach [23].

II.C.1 General Rule and Exceptional Rules

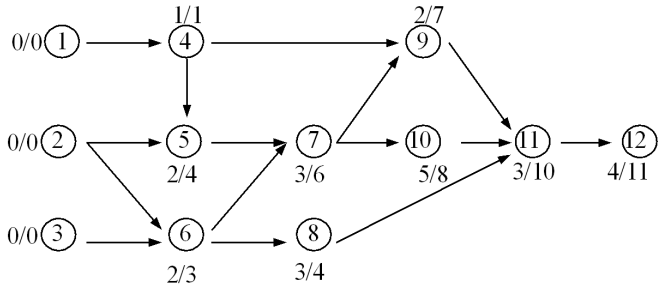
The general rule on graph G is that the complete path p in G satisfies the short-path and long-path delay constraints in equation II.2 and II.1, i.e., $T_h < d_p < T_{period} - T_s$, where d_p is the path delay, T_{period} is the clock cycle, T_h and T_s are hold and setup time constraints.

An exceptional rule r is represented by a subgraph $G_r = \{V_r, E_r\}$, a pair of hold and setup time (h_r, s_r), and a priority p_r.

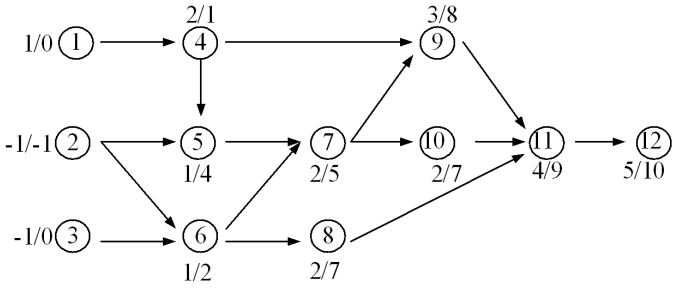
Hold and Setup Time The hold and setup time correspond to the short-path and long-path delay constraints. For a k-cycle subgraph, (h_r, s_r) is the k-



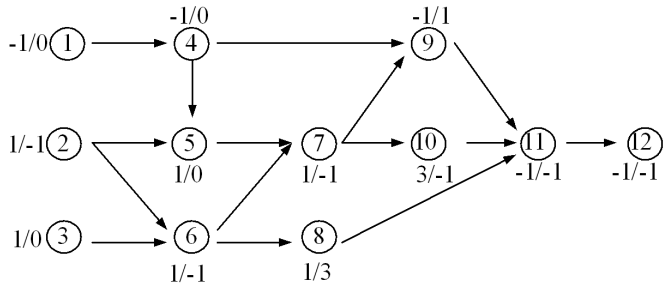
(a) The Timing Graph: Delays Attached on Edges.



(b) Earliest and Latest Arrival Times at Vertices



(c) Earliest and Latest Required Times at Vertices



(d) Slacks for Hold and Setup Times at Vertices

Figure II.4 Static Timing Analysis Algorithm Computing Arrival Times, Required Times and Slacks

cycle hold and setup time defined as follows.

$$h_r = (k-1) \times T_{period} + T_h \quad \text{(II.11)}$$

$$s_r = k \times T_{period} - T_s. \quad \text{(II.12)}$$

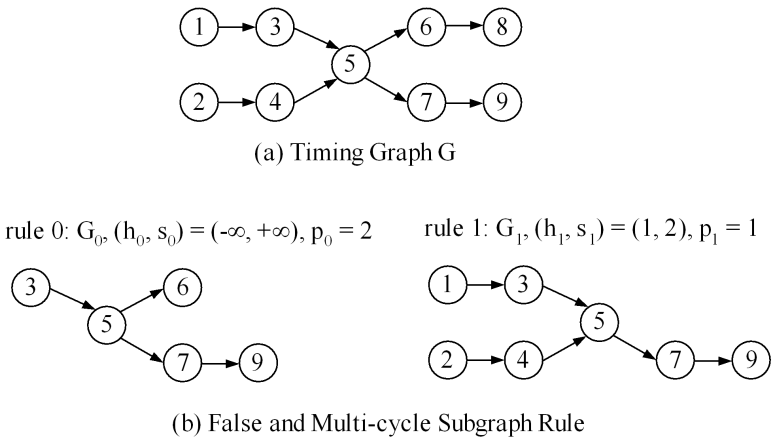
For the false subgraph, the hold time and setup time are unbounded, i.e., -∞ and +∞, respectively.

Subgraph Subgraph G_r describes a set of false paths or multi-cycle paths governed by rule r. The input set of rule r, denoted as B_r, is a set of vertices in G_r which have no input edges in E_r. The output set of rule r, denoted as D_r, is a set of vertices in G_r which have no output edges in E_r. A path in G_r starting from a vertex in B_r is termed a prefix path in G_r. A path in G_r ending at a vertex in D_r is termed a suffix path in G_r. If a path in G_r is both a prefix path and a suffix path in G_r, the path is a complete path in G_r. A path p is a false path or multi-cycle path governed by rule r if the intersection of p and E_r is a complete path in subgraph G_r. All the paths governed by rule r are constrained by an inequality $h_r \leq delay(p) \leq s_r$.

Priority If a path is governed by a set of rules, rule r with the highest priority p_r supersedes others.

Fig.II.5 illustrates a timing graph and two rules represented by a false subgraph G_0 and multi-cycle subgraph G_1. The input set of rule 0 contains vertex 3, i.e., $B_0 = \{3\}$, and the output set of rule 0 contains vertices 6 and 9, i.e., $D_0 = \{6, 9\}$. Path {(1, 3), (3, 5), (5, 6), (6, 8)} is a false path because it contains a complete path {(3, 5), (5, 6)} in subgraph G_0. Another path {(2, 4), (4, 5), (5, 7), (7, 9)} is a 2-cycle path because it belongs to the subgraph G_1. Complete path {(1, 3), (3, 5), (5, 7), (7, 9)} belongs to both G_0 and G_1. Because the priority of rule 0, i.e., $p_0 = 2$, is higher than the priority of rule 1, i.e., $p_1 = 1$, the complete path is a false path.

We formulate rule sets at every vertex and edge in graph G to extract the rules:



(a) Timing Graph G

(b) False and Multi-cycle Subgraph Rule

Figure II.5 False and Multi-cycle Subgraph Rules: False subgraph rule 0 and multi-cycle subgraph rule 1

- Starting Rule set $F(v) = \{r | v \in B_r\}$ contains the rules starting from vertex v;
- Ending Rule set $T(v) = \{r | v \in D_r\}$ contains the rules ending at vertex v;
- Edge Rule set $I(u, v) = \{r | (u, v) \in E_r\}$ contains the rules covering edge (u, v).

In Fig.II.5, vertex 3 has starting rule set $F(3) = \{0\}$, ending rule set $T(3) = \varnothing$. For vertex 9, starting rule set $F(9) = \varnothing$, and ending rule set $T(9) = \{0, 1\}$. The edge rule set of edge (5, 7) contains rules 0 and 1, i.e., $I(5, 7) = \{0, 1\}$.

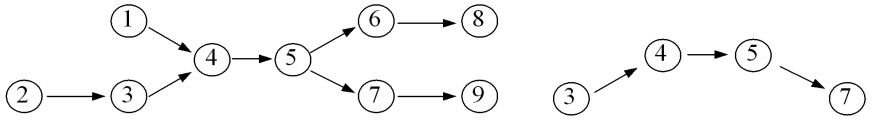
II.C.2 Timing Analysis with Tags

The overall procedure of the tag-based approach is the same as the Arrival-Time-Propagation process except that tags are computed for false path timing during the propagation. Each tag is a rule set of false subgraphs. According to the false path information in the tags, false path timings are distinguished from non-false path timings, and removed after the timing propagation. Rule sets for prefix paths or suffix paths are defined as follows.

Definition II.C.1 ***(Prefix and Suffix Rule Sets)*** *Given a prefix path p^-, the*

prefix rule set *of* p^- *is* $R(p^-) = \{r | p^- \cap E_r$ *is both a prefix path in* G_r *and the tail of* $p^-\}$. *Given a suffix path* p^+, *the* suffix rule set *of* p^+ *is* $R(p^+) = \{r | p^+ \cap E_r$ *is both a suffix path in* G_r *and the head of* $p^+\}$.

Conceptually, the prefix or suffix rule set indicates whether the prefix or suffix path belongs to the false path of a rule. In Fig.II.6, the rule set of prefix path $p_0^- = \{(2,3),(3,4),(4,5)\}$ contains rule 0 because p_0^- is the prefix of the false path {(2, 3), (3, 4), (4, 5), (5, 7), (7, 9)}. Another prefix path $p_1^- = \{(2,3),(3,4),(4,5),(5,6)\}$ exits from the subgraph of rule 0 at vertex 5. Since vertex 5 is not an ending vertex of rule 0, prefix path p_1^- does not belong to the false path of rule 0. Therefore, rule set $R(p_1^-)$ does not contain rule 0.



(a) Timing Graph (b) False Subgraph Rule 0

Figure II.6 Prefix Rule Sets

The following algorithm computes the arrival times and the rule sets at each vertex.

Algorithm Rule-Set-Computation(v)

1. If vertex v is a primary input produce a rule set $R = \varnothing$ at vertex v;
2. else

For each edge (u, v)

For each rule set R at vertex u

(a) $R' = (R \cup F(u)) \cap I(u, v)$;

(b) if($R' \cap T(v) = \varnothing$)

i. $Arr_{max}(v, R') = max(Arr_{max}(v, R'), Arr_{max}(u, R) + d(u, v))$;

ii. $Arr_{min}(v, R') = min(Arr_{min}(v, R'), Arr_{min}(u, R) + d(u, v))$;

Fig.II.7.(a) illustrates timing graph G with two false subgraph rules 0 and 1. In Fig.II.7.(b), we compute the rule sets in a forward propagation. For example, the rule sets at several vertices are as follows:

- At the primary input vertices 1 and 2, the rule sets are $\varnothing$.
- At vertex 4, we propagate the rule set $\varnothing$ of vertex 2 through the edge (2, 4) using the equation $R' = (\varnothing \cup F(2)) \cap I(2,4)$, where starting rule set $F(2) = \{1\}$, and edge rule set $I(2,4) = \{1\}$. We intersect the produced rule set $R' = \{1\}$ with ending rule set $T(4)$, where $T(4) = \varnothing$. Because the intersection is $\varnothing$, we compute the arrival time $Arr_t(4, R')$, where $R' = \{1\}$.
- At vertex 9, we first produce three rule sets $\{0\}$, $\{1\}$, and $\varnothing$ by propagating the rule sets at vertex 8. Because ending rule set $T(9) = \{0, 1\}$, intersection $\{0\} \cap T(9)$ is $\{0\}$, which means vertex 9 is the ending vertex of the false path in rule 0. Therefore, we remove the rule set $\{0\}$ and the false path arrival time $Arr_t(9, \{0\})$. Similarly rule set $\{1\}$ and the false path arrival time $Arr_t(9, \{1\})$ is also removed. Finally only the non-false path arrival time $Arr_t(9, \varnothing)$ is produced at vertex 9.

The correctness of the algorithm is ensured by theorem II.C.1.

Theorem II.C.1 *After a forward Rule-Set-Computation, at each vertex v, the arrival times labelled by prefix rule sets cover the arrival times of prefix paths in prefix cone $P^-(v)$. After a backward Rule-Set-Computation, the arrival times labelled by suffix rule sets cover the arrival times of suffix paths in suffix cone $P^+(v)$.*

II.C.3 Node Splitting Approach

The node-splitting approach follows previous works [27, 39] to remove false paths from the timing graph using node splitting and edge removal [5,6]. On the false path, it splits the vertices with multiple input edges in topological order

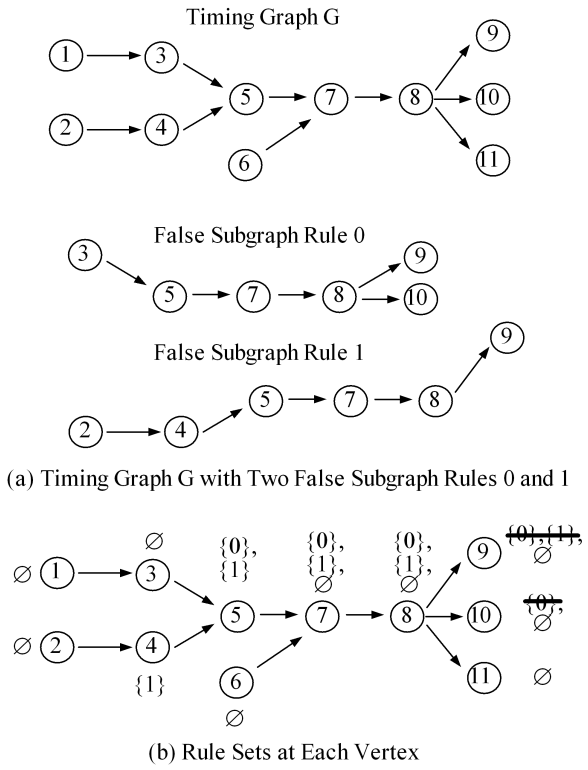


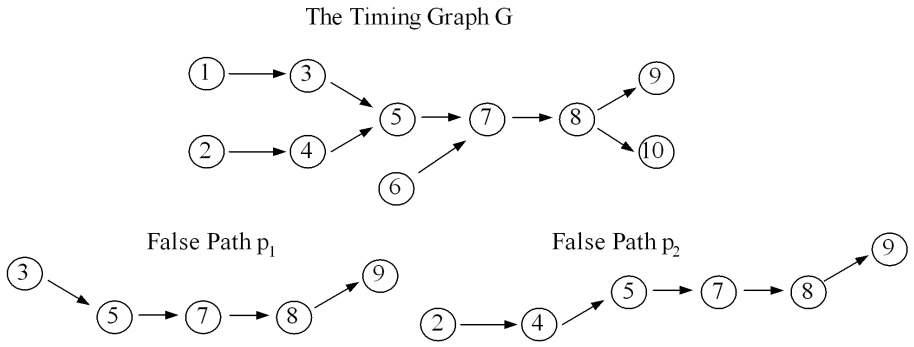
Figure II.7 Rule Set Computation

until arriving at the last vertex with multiple output edges. Then the edges only belonging to false paths are removed.

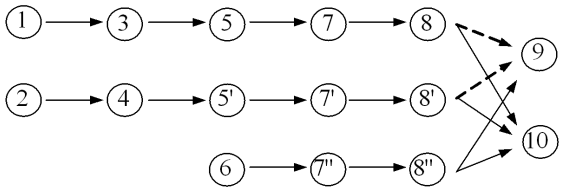
In Fig.II.8.(a), there is a timing graph including two false paths p_1 and p_2. Fig.II.8.(b), illustrates the node splitting in topological order from the primary inputs to the primary outputs.

- Vertex 5 is the first vertex with multiple input edges, which is split into two vertices 5 and 5′.
- Vertex 7 has three input edges, (5, 7), (5′,7), and (6, 7), after vertex 5 is split. Therefore, vertex 7 is split into three new vertices 7, 7′, and 7″.
- Finally vertex 8 is split into three new vertices. The edges (8, 9) and (8′, 9) are removed, because they only belong to false paths.

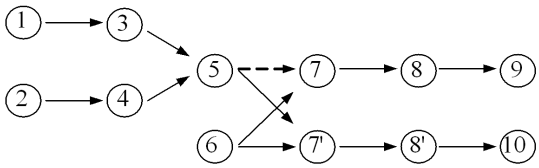
Node splitting can also be performed in backward direction, which splits



(a) Timing Graph G including Two False Paths p_0 and p_1



(b) Node Splitting from Primary Inputs to Primary Outputs



(c) Node Splitting from Primary Outputs to Primary Inputs

Figure II.8 Node Splitting

the vertices with multiple output edges. Fig.II.8 illustrates node splitting proceeding from primary outputs to primary inputs.

Conceptually, each newly created vertex corresponds to a rule set. For example, on the timing graph in Fig.II.8.(a), we consider false path p_1 as rule 0, false path p_2 as rule 1, and compute the rule sets. At vertex 5, rule sets are $\{0\}$ and $\{1\}$ which corresponds to new vertices 5 and $5'$ in Fig II.8.(b), respectively.

The node splitting approach identifies the optimal set of vertices for splitting. For example in Fig.II.8.(c), vertex 5 is not split because the false paths in rule 0 and rule 1 share the common suffix path $\{(5, 7), (7, 8), (8, 9)\}$ starting at

vertex 5. As a result, removing the edge (5, 7) can remove both two false paths.

II.C.4 Multi-clock Domain Analysis with Edge-Masks

An edge-mask data structure is proposed to deal with specified false paths and multi-cycle paths in timing analysis [23]. The false paths and multi-cycle paths are seen to function as multi-clock domain cases. Each edge in the netlist is attached an edge-mask, which represents the reachability between multi-clock domains in the graph. By using edge-masks to filter standard analysis traversals in each abstract clock domain, unnecessary computation time is reduced.

II.D Abstract Timing Model Reduction

In this section, we first introduce the terminologies including the abstract timing model of a hierarchical block, the bipartite timing model and the delay matrix of a bipartite timing model. Then, we introduce previous works on timing model reduction.

II.D.1 Terminology

The timing graph of a hierarchical block H is a weighted graph, denoted as G_H. The weight of each edge (i, j), denoted as edge delay $d_{i,j}$, is the corresponding gate or interconnect delay estimated based on the linear delay model. The delay of a path from input i to output j, denoted as $d_{p_{i,j}}$, is the total delay of edges on the path. The shortest path delay from input i to output j in G_H, denoted as $d^{min}_{H_{i,j}}$, is the minimum of all path delays $d_{p_{i,j}}$ in G_H. The longest path delay from input i to output j, denoted as $d^{max}_{H_{i,j}}$, is the maximum of all path delays $d_{p_{i,j}}$ in G_H.

The timing model of a hierarchical block H is a weighted graph G_M, which has the same input set B and output set D as timing graph G_H. The shortest and longest path delays from input i to output j in G_M are equal to $d^{min}_{H_{i,j}}$ and $d^{max}_{H_{i,j}}$ in G_H. Note the internal vertices and edges in timing model G_M may or

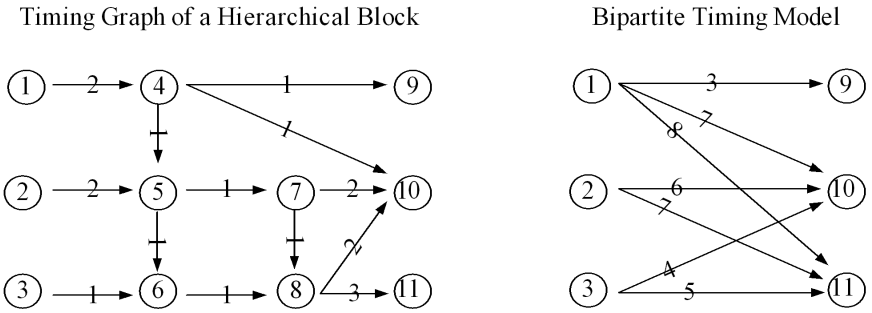
may not be the same as those in timing graph G_H. A bipartite maximum delay model, denoted as G_M^{max}, is a timing model, in which any vertex is either an input or an output. On each edge (i, j) in G_M^{max} the attached edge delay $d_{i,j}$ is equal to the longest path delay $d_{H_{i,j}}^{max}$. A bipartite minimum delay model, denoted as G_M^{min}, is a timing model, in which any vertex is either an input or an output. On each edge (i, j) in G_M^{min} the attached edge delay $d_{i,j}$ is equal to the shortest path delay $d_{H_{i,j}}^{min}$.

Fig.II.9.(a) illustrates a timing graph G_H of a hierarchical block and the corresponding bipartite maximum delay model G_M^{max}. The input set contains three inputs, i.e., $B = \{1, 2, 3\}$. The output set contains three outputs, i.e., $D = \{9, 10, 11\}$. On edges between connected inputs and outputs the longest path delays are attached. For example, the longest path from input 1 to output 10 is $\{(1, 4), (4, 5), (5, 7), (7, 8), (8, 10)\}$ which has delay 7. Thus, the delay attached on edge (1,10) is 7.

We formulate a delay matrix $M(G_M^{max})$ based on the edge delays in bipartite maximum delay model G_M^{max}. The number of rows, denoted as r, is the number of inputs, i.e., $r = |B|$. The number of columns, denoted as c, is the number of outputs, $c = |D|$. The element on the ith row the jth column, denoted as $m_{i,j}$, is (1) edge delay $d_{i,j}$ if edge $(i, j) \in E$, or (2) ∞ if input i disconnects with output j. The input delay vector of input i, denoted as I_i, is a set of elements on the ith row in matrix M, i.e., $I_i = \{m_{i,j} | j \in [1..c]\}$. The output delay vector of output j, denoted as O_j, is a set of elements on the jth column in matrix M,i.e., $O_j = \{m_{i,j} | i \in [1..r]\}$. Similarly, we can formulate the delay matrix for the bipartite minimum delay model.

The delay matrix of the bipartite maximum delay model in Fig.II.9.(a) is shown in Fig.II.9.(b). Each row in the matrix contains the edge delays from one input to all the connected outputs. For example, the first row contains edge delays from input 1 to outputs 9, 10, and 11, i.e., $d_{1,9} = 3$, $d_{1,10} = 7$ and $d_{1,11} = 8$. If the input is disconnected with an output, the delay is set to ∞. For example, input 2

is disconnected with output 9, thus, the element on the 2nd row the 1st column is set to ∞.



(a) Bipartite Timing Model based on a Timing Graph: Delays Attached on Edges

Inputs \ Outputs	9	10	11
1	3	7	8
2	∞	6	7
3	∞	4	5

(b) Delay Matrix

Figure II.9 Bipartite Timing Model and Delay Matrix

A biclique is a complete bipartite graph $G_c = \{B_c, D_c, E_c\}$, i.e., $\forall$ input i in input set B_c is connected with $\forall$ output j in output set D_c, i.e., $E_c = \{(i,j)|i \in B_c, j \in D_c\}$. Fig.II.10 illustrates a biclique and the corresponding delay matrix. Each pair of input and output is connected. In the delay matrix, there is no disconnected symbol ∞.

A star with a center vertex s is a weighted graph $G_s = \{B_s, D_s, s, E_s\}$, where B_s is the input set, D_s is the output set, s is the vertex at the center, and E_s is a set of edges from inputs to s and from s to outputs. Each edge has a weight. The weights of edges (i,s) and (s,j) are denoted as $d_{i,s}$ and $d_{s,j}$, respectively.

Fig.II.11 illustrates a star. The vertex s at the center connects with all inputs and outputs. On each edge, a weight is attached, i.e., edge delay $d_{i,s}$ or $d_{s,j}$. For example, edge delay $d_{1,s}$ of edge $(1,s)$ is 1.

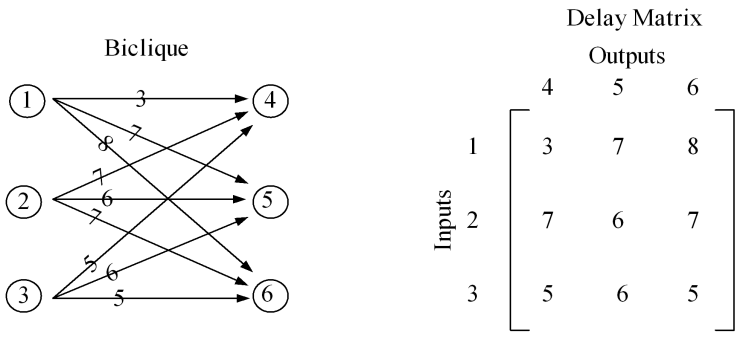


Figure II.10 Biclique and the Delay Matrix

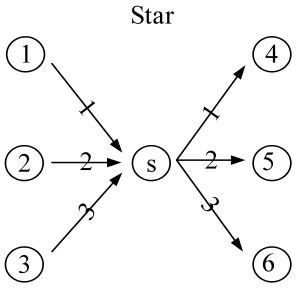


Figure II.11 Star

II.D.2 Timing Model Reduction

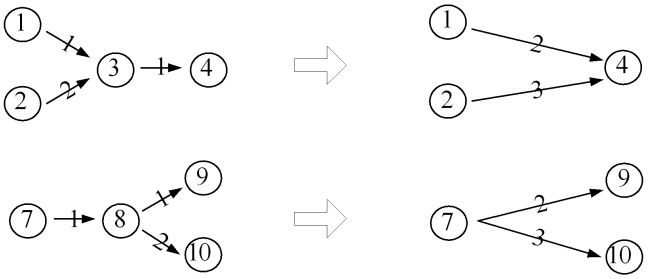
The previous published works on timing model reduction can be categorized into two groups: (1) reducing edges in timing graph G_H, and (2) constructing minimized model based on bipartite timing model G_M.

The reduction based on G_H starts from the original timing graph G_H of the block, and iteratively reduces the number of edges in the graph using graph transformations ([33, 42]):

1. Simplify a path into one edge and attach the path delay on the edge (Fig.II.12.(a));
2. Merge edges sharing common input and output into one edge and attach the dominant delay on the edge (Fig.II.12.(b)).

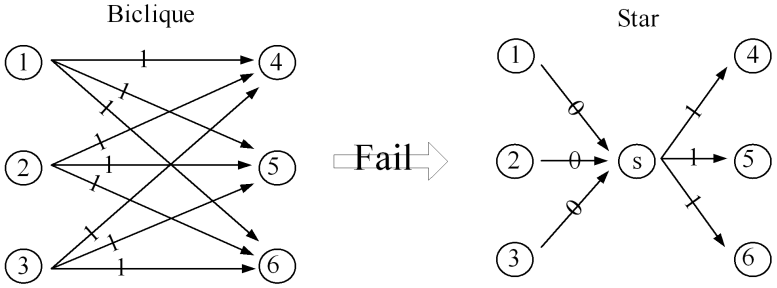
The transformation is a greedy heuristic, which may not always produce the optimal solution. For example in Fig.II.12.(c), the heuristic fails to get the optimal.

Another category of methods tried to represent delay metrics into an abstract timing model and minimized the number edges in the abstract timing model. An optimal realization of a distance matrix problem is formulated as constructing a graph that preserves shortest-path distances while minimizing the total sum of edge weights [10, 18]. In [14], the problem is studied on the case in which a collection of weighted graphs for compression are given. For the graph with unit edge weights, it's shown that the best compression can be achieved by replacing cliques by stars (Fig.II.12.(c)). The un-weighted bipartite graph compression is studied in [15]. The hardness of the graph compression problem is studied for the unit edge weights case [14].



(a) Merge Serial Edges on a Path

(b) Merge Edges Sharing Common Input and Output



(c) Greedy Merging Edges Fails to Transform Biclique to Star

Figure II.12 Timing Model Reduction Based on Original Timing Graph

III Timing Analysis with False Paths

III.A Introduction

In this chapter, we introduce a two-direction propagation approach to improve the efficiency of timing analysis with false paths. The tagged timings for false paths are collected when timing information can be shared. By doing so, we reduce the number tagged timings for timing propagation, thus improving the efficiency.

When false paths are dealt with in timing analysis, a large number of tags or vertices need to be created and propagated, thus, the analysis efficiency is deteriorated. Although there are techniques which reduce the number of timings with tags, the reduction depends on the timing values [16]. Because the timing values may change during the circuit optimization, the reduction is performed together with timing analysis, which induces a large run time penalty in the whole optimization process. The node splitting approach has the ability to identify the optimal set of nodes for splitting. However, it's not clear how to minimize the number of newly created tags when one vertex is split.

We propose a two-direction propagation technique which minimizes tags created at each vertex through a biclique covering approach. We follow the algorithm in [3] to compute prefix and suffix rule sets in two directions. By matching the prefix and suffix rule sets, we can achieve all non-false paths through the ver-

tex. We cover the non-false paths using a biclique covering approach, and collect the prefix rule sets into rule collections. By doing so, the number of tagged false path timings is reduced. A formal proof ensures that we can propagate and keep merging the rule collections at every vertex in topological order, such that all the non-false paths are covered and all the false paths are removed.

Since computing the minimum biclique is NP complete [34, 36], we use a polynomial heuristic to perform the biclique covering minimization in minimal degree order. Because the minimization does not depend on the values of the timings, but depends on the false path specifications, the minimization is only incurred once as a preprocessing step. However, the benefit occurs repeatedly when timing analysis is called during the optimization.

We follow the experiments in [3] to test the proposed approach on the timing graph of a mesh. We also perform experiments on a set of industry test cases. For the industry test cases, comparing with the number of prefix rule sets, the number of the tags is reduced by 99%. Comparing with the node splitting approach, the improvement ratio is up to 48%. The run time of the minimization is only 380 seconds for the case with 533,224 nets.

The remainder of this paper is organized as follows. In section III.B, we use an example to explain the motivation of minimizing the number of tags. Section III.C introduces the two-direction propagation flow and the biclique covering approach. We prove that the algorithm can remove the false paths with minimized number of tags. The experimental results are presented in Section III.D.

III.B Motivation

We observe the tag-based algorithms and find out that the number of tags can be reduced when the timing information can be shared. For example in Fig.III.1.(a), there are three rules: (1) rule 0 containing false paths from vertex 1, through vertices v and u, to vertices 5 and 6, (2) rule 1 containing a false path

from vertex 2, through vertices v and u, to vertex 6, and (3) rule 2 containing false paths from vertex 3, through vertices v and u, to vertices 6 and 7.

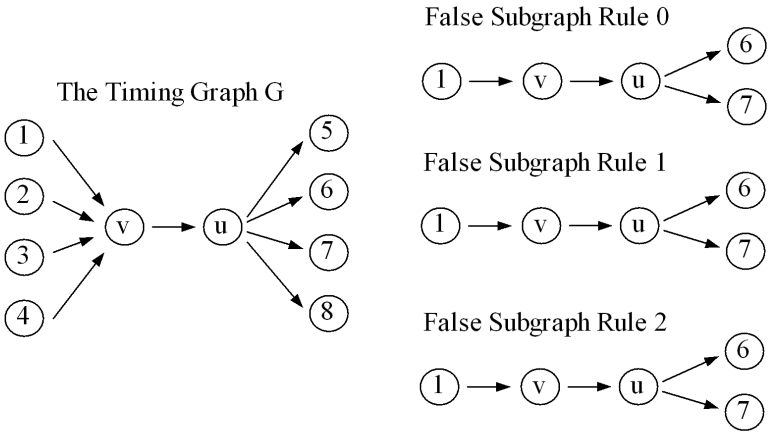
Suppose we follow the tag-based approach in [3] to scan from inputs toward outputs as shown in Fig.III.1.(b). We need four rule sets {0}, {1}, {2}, and $\varnothing$ at vertex v to represent 4 distinct arrival times from inputs. If we follow the node splitting approach in [6], we need to split vertex v into four vertices v_0, v_1, v_2, and v_3. Fig.III.1 (c) illustrates the non-false paths after node splitting and edge removal. We can merge vertex v_1 with vertices v_0 and v_2 as shown in Fig.III.1.(d) and still cover the time information of non-false paths from vertex 2 because the timing information on the common tails can be shared, i.e., 1) the non-false paths from vertex 2 to vertices 7 and 8 share common tails $\{(v,u),(u,5)\}$ and $\{(v,u),(u,8)\}$ with the non-false path from vertex 1 to vertices 7 and 8, and 2) the non-false paths from vertices 2 to vertices 5 and 8 share common tails $\{(v,u),(u,7)\}$ and $\{(v,u),(u,8)\}$ with the non-false paths from vertex 3 to vertices 5 and 8. Therefore, we reduce the number of split vertices from 4 to 3.

The example indicates some hints for the rule set minimization:

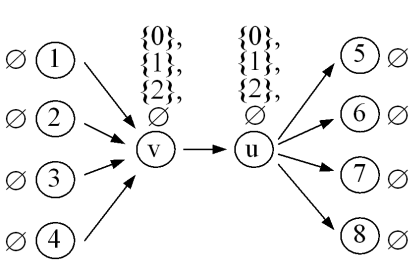
- The timing information labeled by rule sets can be shared when non-false paths contain common prefix or suffix paths.
- The rule set minimization requires information of complete non-false paths, while the forward sweeping or backward sweeping only provides information of prefix or suffix paths.

III.C Rule Collection Minimization

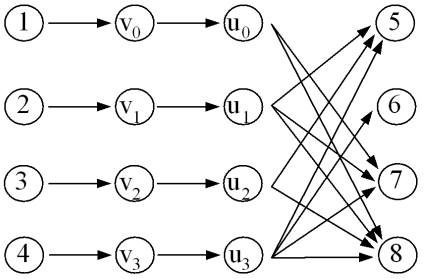
We propose a two-direction propagation to minimize the number of tags. We firstly use a backward propagation to produce rule sets for required arrival times. Then, we perform another forward propagation to produce rule sets for arrival times. When the timing information can be shared, we use a biclique covering approach to collect prefix rule sets into rule collections. Each distinct



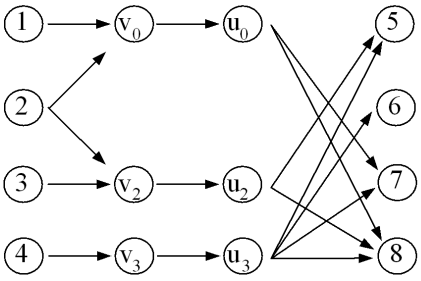
(a) Timing Graph G with Three False Subgraph Rules 0, 1 and 2



(b) Rule Sets at Each Vertex: 4 rule sets at vertex v



(c) Node Splitting: 4 vertices created at vertex v



(d) 3 Vertices are Enough to Cover Non-false Paths through Vertex v.

Figure III.1 Merging Rule Sets

arrival time needs a tag, which is a rule collection. Therefore, by minimizing the rule collections, we can reduce the number of tags of distinct arrival times, thus improving the efficiency of timing analysis.

In this section, we first define the rule collection and give the rule collection minimization algorithm. Then, we prove that the algorithm can remove the false paths without removing the non-false paths. The definition of the rule collection is as follows.

Definition III.C.1 *(**Rule Collection**) A rule collection at vertex v is a set of rule sets of prefix paths which ends at v, i.e. $\Re(v) \subseteq \{R(p^-)|p^- \in P^-(v)\}$, where $P^-(v)$ is the prefix cone at v.*

III.C.1 Main Flow of Rule Collection Minimization

The overall minimization flow is a two-direction propagation: a backward sweeping to compute suffix rule sets and another forward sweeping to minimize the rule collections. At each vertex, we compare the prefix and suffix rule sets to construct the non-false paths. After the construction, we use a biclique covering approach to cover non-false path timing with rule collections. Since each rule collection is the tag of a distinct arrival time, the primary object is to minimize the number of rule collections.

Main flow

1. Produce suffix rule sets by a backward sweeping;
2. For each vertex v in topological order
   (a) If vertex v is a primary inputs then produce an initial rule collection $\{\varnothing\}$;
   (b) else
      i. for each edge (u, v)
         for each $\Re(u)$ Rule-Collection-Propagation $(\Re(u), u, v)$;

ii. Rule-Collection-Minimization(v);

The Rule-Collection-Propagation ($\Re(u), u, v$) of step 2).b).i) computes the rule collections at vertex v by propagating the rule collections of vertex u, where edge (u, v) is an input of v. We will introduce the details of the routine in section III.C.4.

The Rule-Collection-Minimization(v) of step 2).b).ii) minimizes the rule collection in three steps: (1) performing intersections between rule collections and suffix rule sets, (2) constructing bipartite graph based on intersection results, and (3) covering edges in the bipartite graph with minimized number of bicliques. These three steps are introduced in section III.C.2 and III.C.3.

Algorithm: Rule-Collection-Minimization(v)

1. For each rule collection $\Re(v)$ and suffix rule set $R(p^+)$

   Produce $Intersect(\Re(v), R(p^+)) = \{R(p^-)_i \cap R(p^+) | R(p^-)_i \in \Re(v)\}$;

2. Construct bipartite(v) based on the intersections;

3. Biclique-Covering(v);

III.C.2 Intersection of Prefix Rule Collections with Suffix Rule Sets

We match prefix and suffix rule sets to achieve non-false paths through each vertex. Theorem II.C.1 ensures that the prefix and suffix rule sets at each vertex cover all prefix and suffix paths in the prefix and suffix cone. As a result, by matching false path information in the prefix and suffix rule sets, we can achieve all non-false paths while removing false paths.

We use Rule-Set-Computation algorithm to compute the suffix rule sets at every vertex v [3]. For the case in Fig.III.1, the rule collections and suffix rule sets at vertex v are in the first column and first row of Table III.1. The suffix rule sets, {0}, {0, 1, 2}, {2}, and $\varnothing$ correspond to the suffix paths, {(v, u), $(u$,5)}, {(v, u), $(u$,6)}, {(v, u), $(u$,7)}, and {(v, u), $(u$,8)}. The prefix rule sets, {0}, {1},

{2}, and ∅ correspond to the prefix paths, {(1,v)}, {(2,v)}, {(3,v)}, and {(4,v)}. Without minimization, each prefix rule set produces a rule collection.

Table III.1 Intersections of Rule Collections and Suffix Rule Sets at Vertex v

		$R(p^+)$			
		{0}	{0,1,2}	{2}	∅
$\{R(p^-)\}$	{{0}}	{{0}}	{{0}}	{∅}	{∅}
	{{1}}	{∅}	{{1}}	{∅}	{∅}
	{{2}}	{∅}	{{2}}	{{2}}	{∅}
	{∅}	{∅}	{∅}	{∅}	{∅}

We intersect the prefix and suffix rule sets to gather the non-false paths through the vertex. Conceptually, the $R(p^-) \cap R(p^+) = \varnothing$ means that the concatenation of the prefix path p^- and the suffix path p^+ is a non-false path.

Definition III.C.2 ***(Intersection of rule collection and rule set)*** *The intersection between a rule collection $\Re(v)$ and a suffix rule set $R(p^+)$ intersects each prefix rule set in $\Re(v)$ with $R(p^+)$, i.e., $Intersect(\Re(v), R(p^+)) = \{R(p^-)_i \cap R(p^+) | R(p^-)_i \in \Re(v)\}$.*

Table III.1 shows the intersections at vertex v. For example, the second row contains the intersections of the rule collection $\{R(p^-)\} = \{\{0\}\}$ with various suffix rule sets. The intersection with $R(p^+) = \{0, 1, 2\}$ is $\{\{0\}\}$, which means the concatenation is a false path governed by rule 0. The intersection with $R(p^+) = \{2\}$ is $\{\varnothing\}$, which means the concatenation is a non-false path.

III.C.3 Bipartite Graph and Biclique Covering

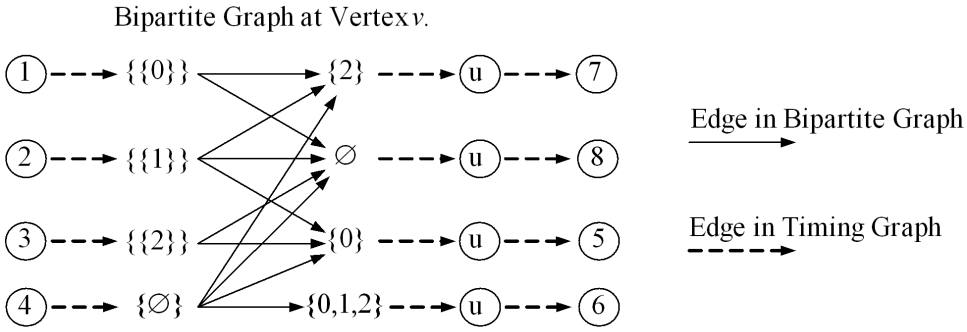
Based on the intersections, we construct a bipartite graph to cover the non-false paths through the vertex, and produce minimum biclique covering on the bipartite graph. By doing so, we can cover non-false paths with minimized number of rule collection tags.

The bipartite graph is constructed in two steps as follows.

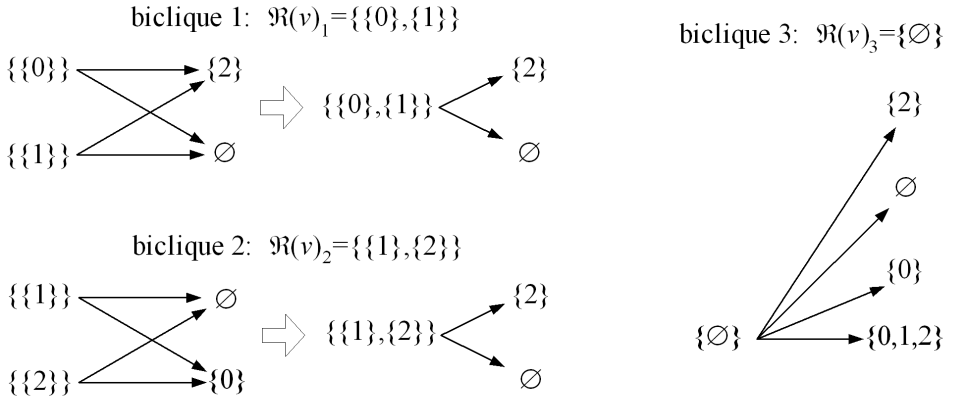
- Add the prefix and suffix rule sets into the bipartite graph.

- If $Intersection(\Re(v), R(p^+))$ only contains $\varnothing$, add an edge from $\Re(v)$ to $R(p^+)$ to represent the non-false path.

Fig.III.2.(a) illustrates the bipartite graph at vertex v based on the intersections in Table III.1. There is an edge from $\{R(p^-)\} = \{\{0\}\}$ to $R(p^+) = \{2\}$ because the intersection $\{0\} \cap \{2\}$ is $\varnothing$. The edge represents the concatenation of prefix path $\{(1, v)\}$ and suffix path $\{(v, u), (u, 7)\}$, which is a non-false path.



(a) Bipartite Graph at Vertex v



(b) Bicliques and Rule Collections at Vertex v

Figure III.2 Bipartite Graph and Rule Collections at Vertex v

We cover the edges in the bipartite graph by a set of bicliques, and collect the prefix rule sets in each biclique into one rule collection. By doing so, we cover the complete paths through the vertex by rule collections, thus reducing the number of the tags. The biclique covering algorithm is as follows.

Algorithm: Biclique-Covering(v)

1. Initialize the biclique set as $B = \varnothing$;
2. For each rule collection $\Re(v)$ in minimum degree order
   (a) Enlarge biclique b in the biclique set B to cover edges $\{(\Re(v), R(p^+)_i)\}$ from $\Re(v)$, where $R(p^+)_i \in b$;
   (b) If there are edges from $\Re(v)$ not covered
      i. Produce a new biclique containing $\Re(v)$, the edges $\{(\Re(v), R(p^+)_i)\}$s from $\Re(v)$, and the suffix paths $R(p^+)_i$s of the edges;
      ii. Add the new biclique to the biclique set B;
3. For each biclique $b \in B$

   New $\Re(v) = \cup\Re(v)_i$, where $\Re(v)_i \in b$;

Since computing the minimum biclique covering on a general bipartite graph is NP complete, optimal solution cannot be obtained in polynomial time unless P=NP. Therefore, we use a minimal degree approach to cover the edges from every prefix rule set. For each prefix rule collection $\Re(v)$, we try to add $\Re(v)$ and the edges from $\Re(v)$ to a smaller biclique b. If b remains a biclique, the edges can be covered by b. We try to cover all the edges from $\Re(v)$ by a set of smaller bicliques. If some edges can not be covered, a new biclique is produced to contain $\Re(v)$ and the edges from $\Re(v)$. Finally, for each biclique b, the newly created rule collection is the union of all the rule collections in b.

For example in Fig.III.2.(b), we first produce two bicliques to cover the edges from prefix rule collections $\{\{0\}\}$ and $\{\{2\}\}$. Then we cover the edges from prefix rule collection $\{\{1\}\}$ by enlarging the two smaller bicliques into bicliques 1 and 2. Finally, we use biclique 3 to cover the edges from the prefix rule collection $\{\varnothing\}$.

Each biclique covers the non-false paths represented by the edges. For example, biclique 1 contains four edges, from $\{\{0\}\}$ to $\{2\}$, from $\{\{0\}\}$ to $\varnothing$, from

{{1}} to {2}, and from {{1}} to ∅. Therefore, rule collection $\Re(v)_1$ covers four non-false paths represented by the edges, i.e., {(1, v), (v, u), (u, 7)}, {(1, v), (v, u), (u, 8)}, {(2, v), (v, u), (u, 7)} and {(2, v), (v, u), (u, 8)}.

Theorem III.C.1 *The time complexity of the heuristic Biclique-Covering(v) algorithm is O(k^3), where k is the number edges in false path specifications.*

Proof: According to [6], the number of prefix and suffix rule sets at vertex v is O(k). For each prefix rule set $R(v)$, we try to cover the edges from $R(v)$ using existing bicliques. Because the number of existing bicliques is O(k), and the number suffix rule set in each existing biclique is O(k), the run time of covering edges from one rule set $R(v)$ is O(k^2). As a result, the time complexity to cover all the prefix rule sets is O(k^3).□

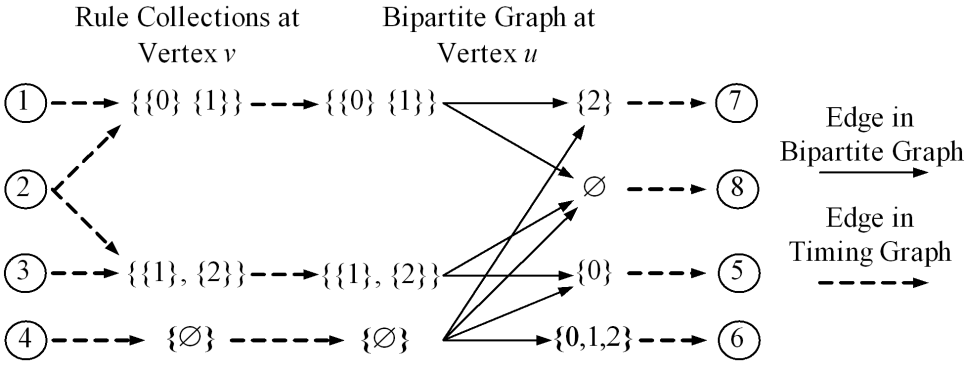
III.C.4 Rule Collection Propagation

We propagate every rule collection $\Re(v)$ to vertex u, where there is an edge (v, u) from vertex v to u.

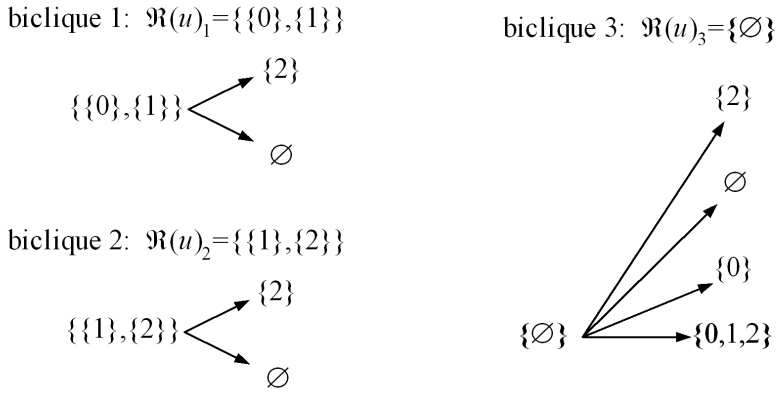
Algorithm: Rule-Collection-Propagation ($\Re(v), v, u$)

1. FalseEnd = 0; $\Re(u) = \varnothing$;
2. For each prefix rule set $R \in \Re(v)$
   (a) $R' = (R \cup F(v)) \cap I(v, u)$;
   (b) if $(R' \cap T(u)) = \varnothing$ then $\Re(u) = \Re(u) \cup \{R'\}$;
   (c) else FalseEnd = 1;
3. if not FalseEnd, and $\Re(u)$ not empty then add $\Re(u)$ into vertex u's rule collection list;

Each prefix rule set in the rule collection is propagated by the equation similar as the equation in the Rule-Set-Computation algorithm. After propagation, we perform biclique covering at vertex u as shown in Fig.III.3.



(a) Bipartite Graph at Vertex u



(b) Bicliques and Rule Collections at Vertex v

Figure III.3 Bipartite Graph and Rule Collections at Vertex u

In Fig.III.4, we summarize the tags and new vertices produced by our approach. Compared with the rule set computation and the node splitting approach in Fig.III.1, our approach reduces the number of tags at vertices v and u from 4 to 3.

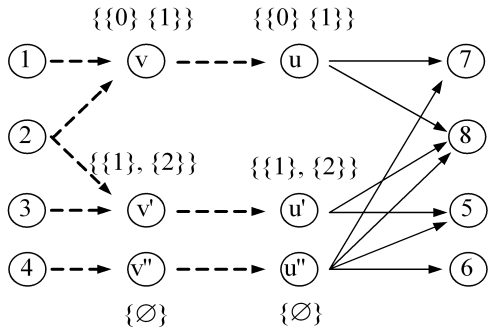
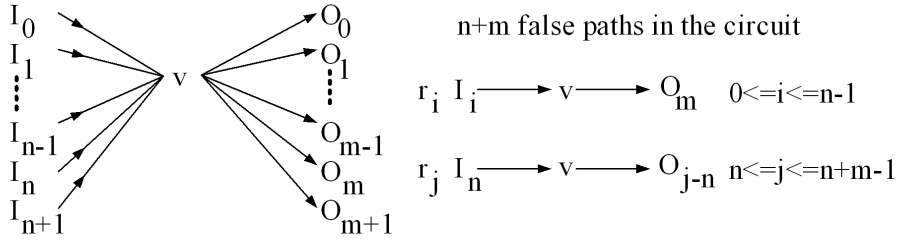


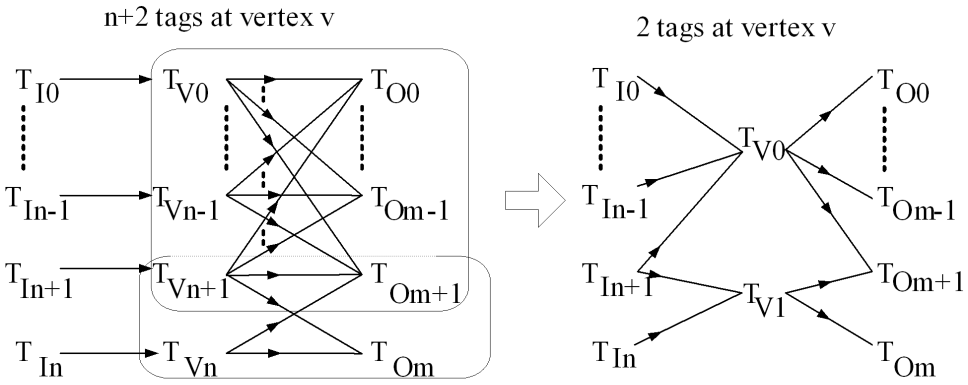
Figure III.4 Rule Collections and New Vertices after Minimization

III.C.5 Theoretical Improvement Ratio

In theory, the proposed method can reduce the number of tags [3] or the number of split nodes [6] by another order of magnitude. For example in Fig.III.5.(a), a circuit has n+2 inputs and m+2 outputs connected through vertex v. There are n+m false paths: n paths from inputs $I_0, I_1, \ldots, I_{n-1}$ through vertex v to output O_m and m paths from input I_n though vertex v to outputs $O_0, O_1, \ldots, O_{m-1}$. Suppose we follow the algorithm of [3] to scan from inputs toward outputs. We need n+2 tags at vertex v to represent n+2 distinct arrival times from inputs. For biclique covering approach, we collect at vertex v all distinct arrival times from inputs as labels $T_{v0}, T_{v1}, \ldots, T_{vn+1}$ and all distinct required arrival times from output as labels $T_{O0}, T_{O1}, \ldots, T_{om+1}$. As shown in Fig.III.5.(b), we construct an edge between T_{vi} and T_{Oj} to match the arrival time and required arrival time for a non-false path. After the construction, we have two bicliques as illustrated by two rectangles in Fig.III.5.(b). Since the timing information in each clique can be shared, we use two tags to represent the two cliques. Therefore, at vertex v the number of tags is reduced from n+2 to 2.



(a) Timing Graph with False Paths: rules r_i, $i \in [0..n\text{-}1]$ share common tail (v, O_m), and rules r_j, $j \in [n..n+m\text{-}1]$ share common head (I_n, v)



(b) Merge Tags when Timing Information can be Shared: biclique covering approach merges n+2 tags into two tags at vertex v.

Figure III.5 Theoretical Analysis of Rule Collection Minimization

III.C.6 Correctness

We prove that the rule collections produced by our minimization cover all the non-false paths in timing graph G.

Theorem III.C.2 *At each vertex v, the rule collections produced by the biclique covering approach cover all the non-false paths through vertex v.*

Proof :We prove the theorem by induction on vertices in topological order. If vertex v is a primary input vertex, lemma 4.1 guarantees that all the prefix and suffix paths are covered by the prefix and suffix rule sets. Therefore, all the paths through vertex v, which are the concatenations of prefix and suffix paths, are covered by the $Intersect(\Re(v), R(p^+))$s. By containing edges corresponding to $Intersect(\Re(v), R(p^+)) = \{\varnothing\}$, the bipartite graph covers all the non-false paths. Because every edge in the bipartite graph is covered by at least one biclique, the rule collections produced based on the bicliques cover the non-false paths. If vertex v is not a primary input vertex, assume the statement is true for all the input vertices of vertex v. We show that the statement is true at vertex v.

The proof goes as follows. We show that the bipartite graph at v contains edges for true paths through v. Since the bicliques produced by the Biclique-Covering (v) algorithm cover all the edges in the bipartite graph, we can conclude that the produced rule collections cover all the true paths through v.

Let us denote a true path through vertex v as p. p also passes through one of vertex v's input vertex, say vertex u. Therefore, p contains three parts, i.e., prefix path p_u^- ending at u, edge (u, v), and suffix path p_v^+ starting from v. By our induction assumption, true path p through vertex u are covered by a rule collection at u, say $\Re(u)$. After Rule-Collection-Propagation$(\Re(u), u, v)$, $\Re(u)$ is propagated into $\Re(v)$. We show that: (i) $Intersect(\Re(v), T(v)) = \{\varnothing\}$, and (ii) $Intersect(\Re(v), R(p_v^+)) = \varnothing$. From (i) we can conclude that the bipartite graph at v contains $\Re(v)$, and from (ii) we conclude that there is an edge from $\Re(v)$ to $R(p_v^+)$ for path p.

(i)According to the assumption at vertex u, p is covered by an edge in the bipartite graph at u from $\Re(u)$ to $R(p_u^+)$, the suffix rule set $R(p_u^+)$ satisfies:

$$R(p_u^+) \cap F(u) = \varnothing. \tag{III.1}$$

Equation (1) is valid because if $R(p_u^+) \cap F(u)$ is not $\varnothing$ the bipartite graph at vertex u will not contains $R(p_u^+)$.

The intersection $Intersect(\Re(u), R(p_u^+))$ satisfies:

$$Intersect(\Re(u), R(p_u^+)) = \{\varnothing\}, \tag{III.2}$$

where p_u^+ is the suffix path starting at vertex u.

We compute the suffix rule set $R(p_u^+)$ by the expression as follows:

$$R(p_u^+) = (R(p_v^+) \cup T(v)) \cap I(u, v). \tag{III.3}$$

From equation (1) and (3), we produce expressions as follows:

$$R(p_v^+) \cap I(u, v) \cap F(u) = \varnothing \tag{III.4}$$

$$T(v) \cap I(u, v) \cap F(u) = \varnothing. \tag{III.5}$$

Based on equation (3) we compute $Intersect(\Re(u), R(p_u^+))$ in equation (2) as follows.

$$\begin{aligned} & Intersect(\Re(u), R(p_u^+)) \\ = \; & \{R \cap (R(p_v^+) \cup T(v)) \cap I(u, v) | R \in \Re(u)\}. \end{aligned} \tag{III.6}$$

According to equation (2), equation (6) is $\{\varnothing\}$ and implies:

$$R \cap R(p_v^+) \cap I(u, v) = \varnothing \tag{III.7}$$

$$R \cap T(v) \cap I(u, v) = \varnothing, \tag{III.8}$$

where $R \in \Re(u)$.

To compute $Intersect(\Re(v), T(v))$, we first compute $\Re(v)$ by the expression as follows.

$$\Re(v) = \{(R \cup F(u)) \cap I(u, v) | R \in \Re(u)\}. \tag{III.9}$$

Based on equation (9),

$$Intersect(\Re(v), T(v)) = \{(R \cup F(u)) \cap I(u,v) \cap T(v) | R \in \Re(u)\}. \quad \text{(III.10)}$$

Because of equations (8) and (5), equation (10) is $\{\varnothing\}$.

(ii)We compute the intersection of $\Re(v)$ and $R(p_v^+)$ as follows.

$$\begin{aligned} & Intersect(\Re(v), R(p_v^+)) \\ = \; & \{(R \cup F(u)) \cap I(u,v) \cap R(p_v^+) | R \in \Re(u)\} \\ = \; & \{(R \cap I(u,v) \cap R(p_v^+)) \cup \\ & (F(u) \cap I(u,v) \cap R(p_v^+)) | R \in \Re(u)\} \end{aligned} \quad \text{(III.11)}$$

Because of equations (7) and (4), equation (11) is $\{\varnothing\}$, which produces an edge from $\Re(v)$ to $R(p_v^+)$.

From (i) and (ii), the bipartite graph of vertex v contains an edge covering the true path p. Therefore, the statement is true for the non-primary vertices. This concludes the proof.□

III.D Experimental Results

We test the proposed approach on both artificial test cases and industry test cases. The algorithm is implemented in C and run on a Pentium 4 Linux machine.

We first follow the experiments in [3], which randomly create false subgraphs on a 100×100 mesh. We produce 14 test cases with each case including various number of false subgraphs. The largest test case includes 140 false subgraphs. The average number of edges in each false subgraph is 6000.

We compare the number of rule collections produced by biclique covering approach with the number of prefix rule sets produced by the Rule Set Computation algorithm [3]. Table III.2 shows the experimental results. The first column is the number of the rules in the test case. The second column and the third column

Table III.2 Tag Minimization on 100×100 Mesh

#rules	$\#R(v)$	$\#\Re(v)$	$\%r$	runtime(sec)
10	8655	4247	50.93	3
20	21575	6735	68.78	5
30	45048	8899	80.25	12
40	63492	8332	86.88	15
50	105284	12721	87.92	28
60	88527	8114	90.83	23
70	87469	8276	90.54	24
80	129689	9139	92.95	40
90	125834	7442	94.09	38
100	151339	8919	94.11	52
110	127040	7918	93.77	47
120	173213	9496	94.52	66
130	189574	13641	92.80	74
140	179127	12032	93.28	80

–$R(v)$:Prefix rule set
–$\Re(v)$:Rule collection (proposed approach)
–Reduction $r = (\#R(v) - \#\Re(v))/\#R(v)$

contain the numbers of the prefix rule sets and the rule collections. The reduction ratio = (#prefix rule sets - #rule collections)/ (#prefix rule sets). The maximum improvement ratio is 94.52% for the case including 66 rules. The run-time of the minimization increases when the number of rules in the case increases. For the largest case including 140 rules, the CPU time is only needs 80 seconds.

We also test our algorithm on a set of industry test cases. The experiment results are in Table III.3. The second column and the third column contain the numbers of the nets and the rules in the test case. The smallest circuit contains 27,555 nets with 28 rules. The largest circuit contains 533,224 nets with 2264 rules. The number of prefix rule sets and the number of the new vertices produced by the node slitting approach [6] are in the fourth and fifth columns. The node splitting approach creates fewer vertices because it optimizes the number of vertices need to be split. The sixth column shows the number of rule collections. Comparing with the prefix rule sets, the largest improvement ratio is 99.01% for the case *atmlcore*. Comparing with the node splitting approach, our approach reduces the number of

Table III.3 Tag Minimization on Industry Test Cases

cases	#nets/#rules	$\#R(v)$	$\#V$	$\#\Re(v)$	$\%r^1$	$\%r^2$
tdl	27,555/28	131	7	7	94.66	0
cq_mod	38,535/5681	47,836	24,484	12,628	73.60	48.42
pm25c	325,582/1995	1,524,260	308,755	218,602	85.66	29.20
atmlcore	533,224/2247	1,953,034	22,860	19,278	99.01	15.67

–$R(v)$:Prefix rule set
–V:Vertices created by node splitting
–$\Re(v)$:Rule collection (proposed approach)
–Reduction $r^1 = (\#R(v) - \#\Re(v))/\#R(v)$
–Reduction $r^2 = (\#V - \#\Re(v))/\#V$

Table III.4 Tag Minimization Run Time

cases	runtime (sec)		
	rule set	node splitting	rule collection
tdl	1	1	1
cq_mod	12	17	17
pm25c	40	44	45
atmlcore	120	155	170

the tags of case *cq_mod* by 48.42%.

We also compare the run time of rule collection minimization algorithm with the rule times of rule set computation algorithm [3] and node splitting approach [6]. The run times in the Table III.4 do not include the CPU time for loading the test cases. For the largest case containing 533,224 nets, the proposed minimization algorithm needs 170 seconds, which is more than 120 and 155 seconds required by the other two approaches. However, as a preprocessing step only incurring once, the run time is affordable.

III.E Acknowledgement

This chapter, in full, is a reprint of the material as it appears in Proc. Intl. Conf. on Computer-Aided Design 2005, Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, Mike Hutton, Truman Collins, Sridhar Srinivasan, Nanchi Chou, and Peter Suaris, "Improving the Efficiency of Static Timing Analysis with

False Paths", ICCAD 2005. The dissertation author was the primary researcher and author and the co-authors listed in these publications directed and supervised the research which forms the basis for this chapter.

IV Unified Framework Dealing with False Paths and Multi-Cycle Paths

IV.A Introduction

In this chapter, we present a framework to unify the process of false paths and multi-cycle paths as exceptional rules, and then minimize the number of rule sets to improve the efficiency.

There are several published works dealing with multi-cycle paths in timing analysis. Some previous works considered multi-cycle paths as false paths in sequential circuits, then ignored multi-cycle paths in minimum cycle times computation [17, 19, 30, 35]. An edge-mask data structure was proposed to deal with both false paths and multi-cycle paths in timing analysis [23]. The false paths and multi-cycle paths are seen to function as multi-clock domain cases. Each edge in the netlist is attached an edge-mask, which represents the reachability between multi-clock domains in the graph. By using edge-masks to filter standard analysis traversals in each abstract clock domain, unnecessary computation time is reduced. However, it's not clear how to reduce the number of abstract clock domains to improve the efficiency.

The contributions of this chapter are as follows.

- We expand rule sets in tag-based approach to cover both false paths and

multi-cycle paths. We allow priority for the rules. When there is a conflict among the rules in a rule set, the rule with the highest priority dominates.

- We devise time shifting to align the short-path and long-path delay constraints for multi-cycle paths. By doing so, we can merge the distinguished timings, and collect different rule sets into one rule collection. For example, when a 2-cycle path and a 3-cycle path converge at a vertex, we can shift the arrival time of the 2-cycle path by one cycle, and merge the timing information.

- We adopt the biclique covering approach in [48] to minimize the number of rule collections at every vertex. We use theorems to show that timing analysis algorithms can produce correct slacks using rule collection tags.

- The cost of the rule collection minimization is only incurred at the initializing stage, thus not contributing to the CPU time of the optimization program.

The remainder of this chapter is organized as follows. Sections IV.B presents the unified framework. In section IV.C, we introduce the minimization algorithm. The experimental results are presented in Section IV.D.

IV.B Unified Framework Processing False Paths and Multi-cycle Paths

We expand the tag-based approach to unify the process of false paths and multi-cycle paths. We follow the previous work in [3] to compute rule set tags for distinct timings. However, we expand the subgraphs of exceptional rules, such that after the expansion, for each rule r, $D_r \subseteq D$ and $B_r \subseteq B$, where D_r and B_r are the input and output sets of rule r, D and B are the input and output sets of timing graph G. As a result, all exceptional rules are covered in the rule set tags at each vertex from the input to the output, thus, the process of the false path and multi-cycle paths rules are unified. The special case in which the false paths are

covered by the multi-cycle paths is discussed in section IV.C.7. In this section, we first introduce the subgraph expansion. Then, we present the unified framework and show the correctness.

IV.B.1 Subgraph Expansion

We expand subgraphs to primary inputs and outputs. As a result, both false path rules and multi-cycle path rules start from primary inputs and end at primary outputs, thus being unified. The subgraph G_r of rule r is expand if it satisfies one of the following two conditions.

Expand-Condition:

1. $\exists$ vertex $u \in D_r$ and $u \notin D$, where D is the output set of graph G and D_r is the output set of rule r.
2. $\exists$ vertex $u \in B_r$ and $u \notin B$, where B is the input set of graph G and B_r is the input set of rule r.

Algorithm: Subgraph-Expansion(r)

Repeat

For each non-primary vertex v in input set B_r

i. Starting rule set $F(v) = F(v) - \{r\}$, $B_r = B_r - \{v\}$;

ii. For each edge (u,v)

$$I(u,v) = I(u,v) \cup \{r\},\ F(u) = F(u) \cup \{r\},\ B_r = B_r \cup \{u\};$$

Until input set B_r no change

Repeat

For each non-primary vertex v in output set D_r

i. Ending rule set $T(v) = T(v) - \{r\}$, $D_r = D_r - \{v\}$;

ii. For each edge (v,u)

$$I(v,u) = I(v,u) \cup \{r\}, T(u) = T(u) \cup \{r\}, D_r = D_r \cup \{u\};$$

Until output set D_r no change

If rule r ends at non-primary vertices, i.e., $\exists$ vertex $u \in D_r$ and $u \notin D$, where D is the output set of graph G and D_r is the output set of r, we distribute r to edge rule sets $I(u,v)$ and ending rule sets $T(o)$, where (u,v) and o are the edge and primary output in suffix cone $P^+(u)$ of u. We perform similar distribution in the prefix cone $P^+(u)$ if the rule starts from non-primary vertices u.

An example is shown in Fig.IV.1. There are two rules in graph G, multi-cycle subgraph rule 0 and false subgraph rule 1. We expand false subgraph G_1 by including vertices 1, 2, 7, 8, and edges (1,3), (2,3), (5,7), (5,8). After the expansion, all the false paths and multi-cycle paths are complete paths in graph G. Thus, the process of false paths and multi-cycle paths can be unified at primary outputs.

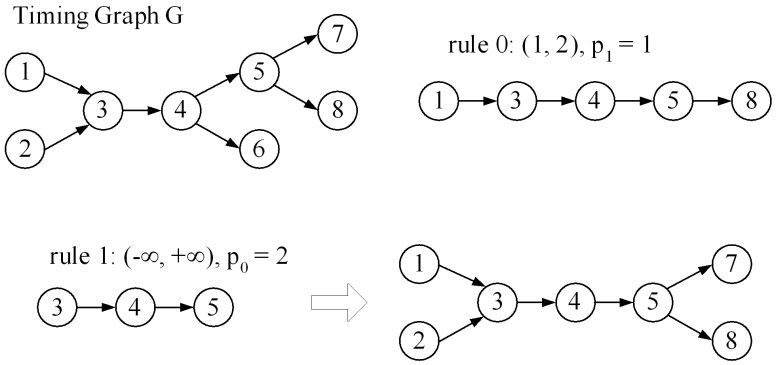


Figure IV.1 Subgraph Expansion: False subgraph of rule 0 is expanded.

After Subgraph-Expansion(r), starting rule set $F(v)$ and ending rule set $T(v)$ have several properties as follows.

Property:

1. If vertex $v \notin B$, $F(v) = \varnothing$, where B is input set of graph G.
2. If vertex $v \notin D$, $T(v) = \varnothing$, where D is output set of graph G.
3. For $\forall$ vertex v, $F(v) \cap T(v) = \varnothing$.

We use Lemma IV.B.1 to show that the expansion does not change whether a path is governed by rule r.

Lemma IV.B.1 *A path p is governed by rule r after Subgraph-Expansion(r) if and only if p is governed by r before the expansion.*

Proof: Denote the original subgraph of rule r as G_r, and the subgraph after the expansion as $G_{r'}$.

After the expansion, a path p governed by rule r is a path in $G_{r'}$. Because paths in $G_{r'}$ contain complete paths in G_r, p is governed by rule r before the expansion.

If p is governed by rule r before the expansion, p contains 1) prefix path $p^-(v)$ ending at vertex v, where v belongs to input set B_r, 2) complete path $p^{'}$ in G_r, and 3) suffix path $p^+(u)$ starting from vertex u, where u belongs to output set D_r. After Subgraph-Expansion(r), $p^-(v)$ plus $p^{'}$ plus $p^+(u)$ belongs to subgraph $G_{r'}$. Thus, p is governed by rule r after the expansion. □

IV.B.2 Rule Sets Based Unified Framework

We modify the rule set computation to deal with both false and multi-cycle paths. The subgraph expansion will be a preprocessing step to unify the exceptional rules. After the expansion, at non-primary vertices only rule set propagation is needed. The false path timing removal and the multi-cycle path slack computation are performed at primary outputs according to the rule in the rule set with highest priority.

Algorithm: Unified-Rule-Set-Computation(v)

If vertex v is a primary input

Rule set $R = F(v)$;

else For each edge (u, v)

For each rule set R at vertex u

i. $R' = R \cap I(u, v)$;

ii. if v is primary output

if $R' \cap T(v)$ is not dominated by false subgraph rule

A. $Arr_{min}(v, R') = min(Arr_{min}(v, R'), Arr_{min}(u, R) + d(u, v))$;

B. $Arr_{max}(v, R') = max(Arr_{max}(v, R'), Arr_{max}(u, R) + d(u, v))$;

iii. else

A. $Arr_{min}(v, R') = min(Arr_{min}(v, R'), Arr_{min}(u, R) + d(u, v))$;

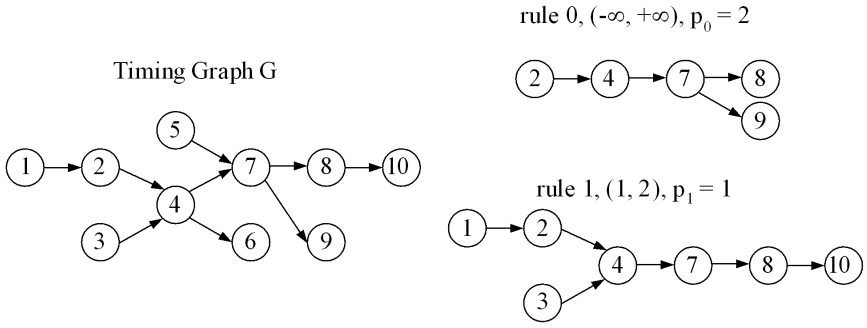
B. $Arr_{max}(v, R') = max(Arr_{max}(v, R'), Arr_{max}(u, R) + d(u, v))$;

The rule set at primary input v is $F(v)$. If v is non-primary input, in step i, the intersection $R \cap I(u, v)$ means that only rules containing edge (u, v) remain in the rule set. In step ii at primary output v, if the intersection $R' \cap T(v)$ is dominated by false path rules, we eliminate false path arrival times by producing no rule set. If v is not primary output, we propagate the arrival time with rule set R' to v.

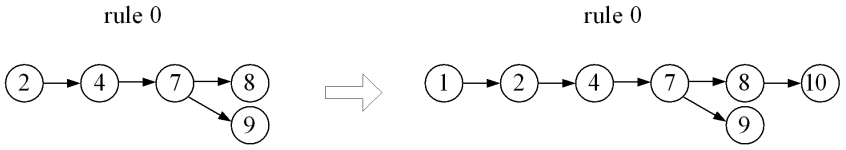
Fig.IV.2.(a) illustrates a graph with a false subgraph rule 0 and a multi-cycle subgraph rule 1. In Fig.IV.2.(b), we expand false subgraph of rule 0. After the expansion, the input set contains vertex 1, and the output set contains vertices 9 and 10. In Fig.IV.2.(c), we compute rule sets in a forward propagation. For example, the rule sets at several vertices are as follows:

- At primary input vertex 1, the rule set is {0, 1}. At primary input 3, the rule set is {1}
- At vertex 2, we propagate the rule set {0, 1} of vertex 1 through the edge (1, 2) using the equation $R = \{0, 1\} \cap I(1, 2)$.
- At vertex 4, we produce two rule sets. Rule set {0,1} is propagated from vertex 1, and rule set {1} is propagated from vertex 3.
- At vertex 6, because edge rule set $I(4, 6) = \varnothing$, the rule sets of vertex 4 becomes $\varnothing$ after intersection with $I(4, 6)$.

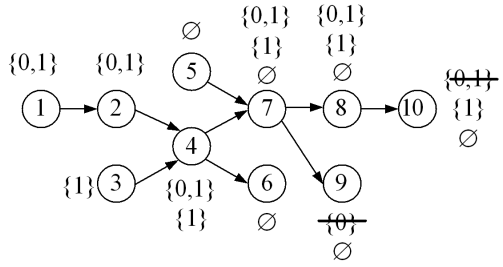
- At primary output vertex 9, we produce two rule sets $\{0\}$ and $\{\varnothing\}$. Because ending rule set $T(9) = \{0\}$, intersection $\{0, 1\} \cap T(9)$ is dominated by false path rule 0. Therefore, we remove the rule set $\{0\}$ and the false path arrival time $Arr(8, \{0\})$. Similarly at vertex 10, rule set $\{0, 1\}$ and the false path arrival time $Arr(10, \{0, 1\})$ are also removed. The arrival times $Arr(10, \{1\})$ and $Arr(10, \varnothing)$ are 2-cycle path and 1-cycle path timings, respectively.



(a) Timing Graph G, False Subgraph Rule 0 and 2-cycle Subgraph Rule 1



(b) Expand False Subgraph of Rule 0



(c) Prefix Rule Sets at Vertices

Figure IV.2 Unified Rule Set Computation

The required time and slack computation is similar to that in II.B except that the backward sweeping backtracks the forward rule set propagations, and computes the required time for each rule set using multi-cycle hold and setup

times. As a result, the multi-cycle path slacks are covered.

The process computes latest and earliest required times Req_{max} and Req_{min} at vertex v. The hold and setup times (h_r, s_r) of rule r are short-path and long-path delay constraints.

Algorithm: Unified-Required-Time-Propagation(v)

1. If vertex v is a primary output

   For each rule set R at v

   $Req_{max}(v, R) = s_r, Req_{min} = h_r$, where r is the rule in R with the highest priority.

2. else

   For each rule set R at vertex v

   For each edge (v, u)

   (a) if R is forward propagated to R' at u

   $$Req_{min}(v, R) = max(Req_{min}(v, R), Req_{min}(u, R') - d(v, u));$$
   $$Req_{max}(v, R) = min(Req_{max}(v, R), Req_{max}(u, R') - d(v, u));$$

We use the graph and rules in Fig.IV.2 as an example to show the process of required time and slack computation.

- At primary outputs, for each rule set R, the required time is the hold and setup time of rule $r \in R$ with highest priority. For example in Fig.IV.2.(c), at vertex 10, the required time tagged by rule set $\{1\}$ is 2-cycle.

- For non-primary vertices, if the arrival time with tag $R(v)$ is forward propagated to rule set $R(u)$, the required time with tag $R(u)$ is backward propagated to $R(v)$, i.e., $Req_{max}(v, R(v)) = min(Req_{max}(u, R(u)) - d(v, u), Req_{max}(v, R(v)))$. For example in Fig.IV.2.(c), at vertex 4, the required time of rule set $\{1\}$, $Req_{max}(4, \{1\})$, is backward propagated from required times $Req_{max}(7, \{1\})$ at vertex 7 and $Req_{max}(6, \varnothing)$ at vertex 6, i.e., $Req_{max}(4, \{1\}) = min(Req_{max}(7, \{1\}) - d(4, 7), Req_{max}(6, \varnothing) - d(4, 6))$.

- The slack with each rule set tag is the required time minus the arrival time, i.e., $Slack_{max}(v, R(v)) = Req_{max}(v, R(v)) - Arr_{max}(v, R(v))$. The slack of the vertex is the minimum of all the slacks with rule set tags, i.e., $Slack_{max}(v) = min(Slack_{max}(v, R(v)))$.

Now, we show the correctness of the proposed framework. Theorem II.C.1 in Section II.C shows that Rule-Set-Computation in [3] ensures all the false path arrival times removed. When we include multi-cycle paths and allow priority for various rules, we need to show that both false and multi-cycle path information are covered by prefix and suffix rule sets as follows.

Theorem IV.B.1 *The false path and multi-cycle path rules are covered by prefix rule set $R(p^-(v))$ if the subgraphs of the rules contain prefix path $p^-(v)$. The false path and multi-cycle path rules are covered by suffix rule set $R(p^+(v))$ if the subgraphs of the rules contain suffix path $p^+(v)$.*

Proof:

We prove by induction on vertices in topological order. If vertex v is a primary input, $R(p^-(v)) = F(v)$. If vertex v belongs to subgraph G_r, $r \in F(v)$, thus $r \in R(p^-(v))$. Therefore, the statement is true. If vertex v is not a primary input, assume the statement is true for all the input vertices of v. We show the statement is true at vertex v.

Suppose prefix path $p^-(v)$ is a prefix path $p^-(u)$ at vertex u plus an edge (u, v). According to the Unified-Rule-Set-Computation algorithm, we propagate $R(p^-(u))$ into rule set $R(p^-(v)) = R(p^-(u)) \cap I(u, v)$ at vertex v, where $I(u, v)$ is the rule set of edge (u, v). If subgraph G_r of rule r contains prefix path $p^-(v)$, G_r contains prefix path $p^-(u)$ and edge (u, v) because $p^-(v)$ is $p^-(u)$ plus (u, v). By our induction assumption, rule r is covered by $R(p^-(u))$. According the edge rule set definition, $r \in I(u, v)$. Thus, $r \in R(p^-(v)) = R(p^-(u)) \cap R(p^-(v))$. Therefore, the statement is true for non-primary input vertex. Similarly, we can prove that the false path and multi-cycle path rules are covered by suffix rule sets. This

concludes the proof. □

An example illustrating theorem IV.B.1 is that at vertex 7 in Fig.IV.2, the arrival times of prefix paths {(1, 2), (2, 4), (4, 7)}, {(3, 4), (4, 7)} and {(5, 7)} are covered by the arrival times with tags {0, 1}, {1}, and ∅, respectively.

IV.C Rule Collection Minimization

We follow the two-direction propagation in [48] to minimize the number of tagged timings with a biclique covering approach. In order to cover multi-cycle paths in the minimization, we devise *time shifting* to align different hold and setup times, and collect rule sets into rule collections. The definition of the rule collection follows Definition III.C.1.

In this section, we first use an example to show the basic idea of the time shifting. Then, we propose the minimization algorithm and show that the produced slacks are correct.

IV.C.1 Time Shifting Example

Although the rule sets in a rule collection may contain rules with different hold and setup times, we can use time shifting to align.

Fig.IV.3.(a) illustrates a timing graph including a 2-cycle path {(1, 3), (3, 4)} and a 3-cycle path {(2, 3), (3, 4)}. At vertex 3, there are two distinct arrival times with rule sets {0} and {1}. However, if we shift the arrival time of prefix path {(1, 3)} forward by 1 cycle, we can merge two arrival times and use the same hold and setup time (2, 3) for both paths. As a result, we can collect two rule sets into one rule collection.

The example indicates that we can align the hold and setup times of various multi-cycle paths by time shifting, and then merge the timings.

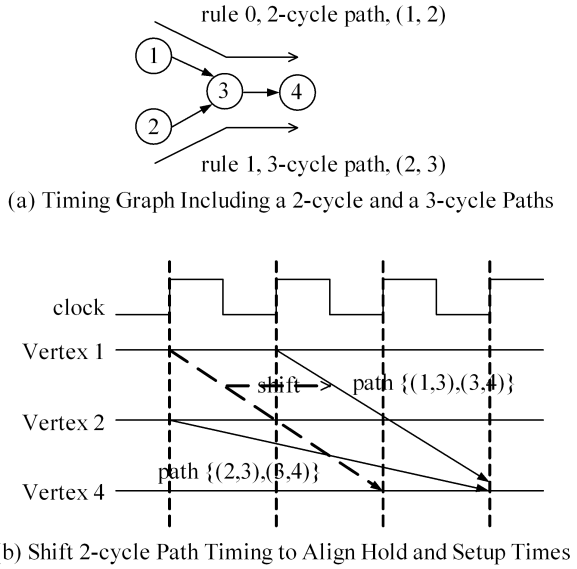


Figure IV.3 Collect Rule Sets Using Time Shifting

IV.C.2 Main Flow of Rule Collection Minimization

The main flow follows the two-direction propagation in Section III.C [48]. However, to include multi-cycle paths, we use the Unified-Rule-Set-Computation algorithm to compute the rule sets. In the Unified-Rule-Collection algorithm, we construct the non-false paths attached with hold and setup times, and then perform biclique covering with time shifting.

Main flow

1. Produce suffix rule sets using Unified-Rule-Set-Computation backward;
2. For each vertex v in topological order
   (a) If vertex v is a primary inputs
   Initial rule collection $\Re(v) = \{F(v)\}$;
   (b) else
      i. for each edge (u, v)
      for each $\Re(u)$ Unified-Rule-Collection-Propagation($\Re(u), u, v$);
      ii. Unified-Rule-Collection-Minimization(v);

The Unified-Rule-Collection-Propagation $(\Re(u), u, v)$ of step 2).b).i) computes the rule collections at vertex v by propagating the rule collections of vertex u, where edge (u, v) is an input of v. We will introduce the details of the routine in section IV.C.5. The Unified-Rule-Collection-Minimization(v) of step 2).b).ii) minimizes the rule collection as follows.

Algorithm: Unified-Rule-Collection-Minimization(v)

1. For each rule collection $\Re(v)$ and suffix rule set $R(p^+)$

   Produce $Intersect(\Re(v), R(p^+)) = \{R(p^-)_i \cap R(p^+) | R(p^-)_i \in \Re(v)\}$;

2. Bipartite-Graph-Construction based on the intersections;

3. Unified-Biclique-Covering(v);

IV.C.3 Intersection of Rule Collection with Suffix Rule Set and Bipartite Graph

At each vertex v, we perform intersections of prefix rule collections with suffix rule sets to gather the non-false paths through v, and then construct a bipartite graph to cover all the non-false paths.

We follow the Definition III.C.2 to intersect a rule collection with a suffix rule set, i.e. $Intersect(\Re(v), R(p^+)) = \{R(p^-)_i \cap R(p^+) | R(p^-)_i \in \Re(v)\}$. If rule r belongs to intersection $R(p^-) \cap R(p^+)$, the path of prefix path p^- plus suffix path p^+ is governed by rule r. When there are various rules in the intersection, the rule with the highest priority is dominant. If the intersection is not dominated by a false subgraph rule, the prefix path plus the suffix path produces a non-false path. For various $R(p^-)_i \cap R(p^+) \in Intersect(\Re(v), R(p^+))$, if $\forall R(p^-)_i \cap R(p^+)$ is not dominated by false subgraph rules, the set of prefix paths p_i^-s plus the suffix path p^+ are non-false paths.

Fig.IV.4 illustrates a timing graph including two false subgraph rules 1 and 2, and one 2-cycle rule 3. In Table IV.1, we show the rule collections, suffix rule sets and the intersections at vertex 4. The suffix rule sets are shown in the

first row, i.e., {3}, {1, 3}, and {2, 3}, which correspond to the suffix paths, {(4, 5), (5, 7)}, {(4, 5), (5, 8)}, and {(4, 5), (5, 6)}. The prefix rule sets are shown in the first column, i.e., {1}, {2}, and {3}, which correspond to the prefix paths, {(1, 4)}, {(3, 4)}, and {(2, 4)}. Without minimization, each prefix rule set produces a rule collection. There are 9 intersections of 3 rule collections and 3 suffix rule sets in Table IV.1. Each intersection corresponds to a path, which is a prefix path plus a suffix path. The rule in the intersection governs the path. For example, the intersection of rule collection {{3}} and rule set {1, 3} is $Intersect(\{\{3\}\}, \{1, 3\}) = \{\{3\}\}$. This intersection corresponds to the path {(2, 4),(4, 5),(5, 8)}, which is the prefix path {(2, 4)} plus the suffix path {(4, 5),(5, 8)}. Because the intersection contains rule 3, the path is a 2-cycle path governed by rule 3.

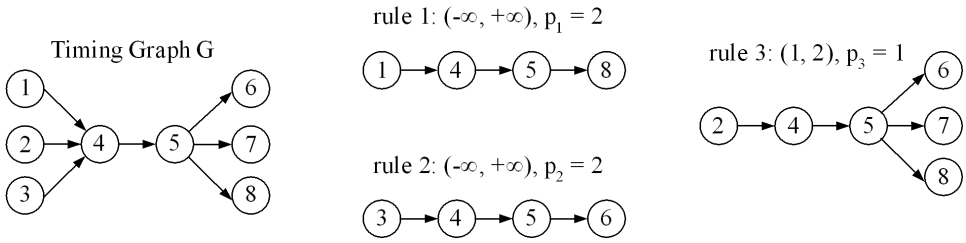


Figure IV.4 Timing Graph with Two False Subgraph Rules 1 and 2, and a 2-cycle Subgraph Rule 3

Table IV.1 Intersections of Rule Collections and Suffix Rule Sets at Vertex 4

		$R(p^+)$		
		{3}	{1,3}	{2,3}
$\{\Re(p^-)\}$	{{1}}	{∅}	{{1}}	{∅}
	{{2}}	{∅}	{∅}	{{2}}
	{{3}}	{{3}}	{{3}}	{{3}}

Based on the intersections, we construct a bipartite graph to cover non-false paths through the vertex.

Algorithm: Bipartite-Graph-Construction(v))

1. Add all suffix rule sets $R(p^+)$ to bipartite graph B(v);
2. For each rule collection $\Re(v)$

(a) Add $Re(v)$ to B(v)

(b) For each suffix rule set $R(p^+)$

i. If $\forall R(p^-) \cap R(p^+) \in Intersection(\Re(v), R(p^+))$ is not dominated by non-false subgraph rule r

Add edge from $\Re(v)$ to $R(p^+)$;

ii. Attach the hold and setup time of rule r on the edge, where $r \in R$ with highest priority and $R \in Intersection(\Re(v), R(p^+))$.

Fig.IV.5.(a) illustrates the bipartite graph at vertex 4 based on the intersections in Table IV.1. We produce an edge from rule collection $\{\{3\}\}$ to rule set {2,3} because the intersection $Intersection(\{\{3\}\}, \{2,3\}) = \{\{3\}\}$ only contains 2-cycle subgraph rule 3. The edge represents the 2-cycle path {(2, 4),(4, 5), (5, 6)}. The hold and setup time of rule 3, i.e., (1,2), is attached on the edge. We produce no edge from rule collection $\{\{2\}\}$ to rule set {2,3} because $Intersection(\{\{2\}\}, \{2,3\}) = \{\{2\}\}$ and rule set {2} in $Intersection(\{\{2\}\}, \{2,3\})$ is dominated by false subgraph rule 2. The intersection represents false path {(3, 4),(4, 5),(5, 6)}.

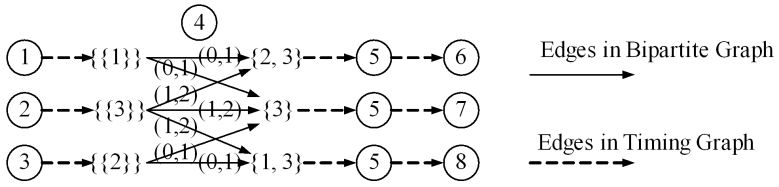


Figure IV.5 Bipartite Graph at Vertex 4: Each edge represents a non-false path with hold and setup time attached.

IV.C.4 Time Shifting and Biclique Covering

We devise time shifting to cover multi-cycle paths in the biclique covering approach [48]. We cover the edges in the bipartite graph by a set of bicliques, and collect the prefix rule sets in each biclique into one rule collection. Since all the complete paths covered by one biclique should have aligned hold and setup times

for slack computation, we devise time shifting to align different hold and setup times.

Definition IV.C.1 *(**Time Shifting**): On rule collection $\Re(v)$, time shifting plus ΔT cycles on (i) arrival time with tag $\Re(v)$, and (ii) the hold and setup time of every rule $r \in R, R \in \Re(v)$. The rule collection after time shifting is denoted as $\Re(v)^{+\Delta T}$.*

Note that the hold time and setup time of a false path remains the same.

We use time shifting to align hold and setup time in a biclique, and the union of rule collections in the biclique is the new rule collection based on the biclique. A biclique b is aligned if for $\forall$ suffix rule set $R(p^+) \in b$, all the edges to $R(p^+)$ are attached with the same hold and setup time. A biclique b can be aligned by time shifting if there are a set of ΔTs such that after time shifting on each rule collection $\Re(v)_i \in b$ by ΔT_i, and updating the hold and setup times of the edges, biclique b becomes aligned.

Definition IV.C.2 *(**Rule Collection on Biclique**): For aligned biclique b, rule collection $\Re(v) = \cup \Re(v)_i^{\Delta T_i}$, where $\Re(v)_i \in b$, and the hold and setup time of $r \in R, R \in \Re(v)$ is shifted by ΔT_i.*

The biclique covering on the bipartite graph covers the edges by a set of bicliques, i.e., the complete bipartite subgraphs, and aligns each biclique by time shifting.

Algorithm: Unified-Biclique-Covering(v)

1. Initialize the biclique set as $B = \varnothing$;
2. For each rule collection $\Re(v)$ in minimum degree order
   (a) For each biclique $b \in B$
      i. If for $\forall R(p^+)_i \in b$, $\exists$ edge $(\Re(v), R(p^+)_i)$
         Enlarge b to include all the edges $\{(\Re(v), R(p^+)_i) | R(p^+)_i \in b\}$;

ii. If b can not be aligned by time shifting;

Recover b by removing edges from $\Re(v)$;

(b) If there are edges from $\Re(v)$ not covered by bicliques in B

i. Recover bicliques b including edges from $\Re(v)$;

ii. Produce a new biclique containing $\Re(v)$, the edges from $\Re(v)$, and the suffix rule sets $R(p^+)_i$ of the edges;

iii. Add the new biclique to the biclique set B;

3. For each biclique $b \in B$

New $\Re(v) = \cup\Re(v)_i^{\Delta T_i}$, where $\Re(v)_i \in b$, the hold and setup time of $r \in R, R \in \Re(v)$ is shifted by ΔT_i;

Since computing the minimum biclique covering on a general bipartite graph is NP complete, optimal solution cannot be obtained in polynomial time unless P=NP [34, 36]. Therefore, we use a minimal degree approach to cover the edges from every prefix rule set. For each prefix rule collection $\Re(v)$, we try to add $\Re(v)$ and the edges from $\Re(v)$ to a smaller biclique b. We use time shifting to align the hold and setup time in b. If b can be aligned, the edges can be covered by b. We try to cover all the edges from $\Re(v)$ by a set of smaller bicliques. If some edges can not be covered, we produce a new biclique including $\Re(v)$ and the edges from $\Re(v)$. Finally, for each biclique b, the newly created rule collection is the union of all the rule collections in b with the time shifting ΔT, i.e., $\Re(v)' = \cup\Re(v)_i^{\Delta T_i}, \Re(v)_i \in b$.

For example in Fig.IV.6, we first produce two bicliques to cover the edges from prefix rule collections $\{\{1\}\}$ and $\{\{2\}\}$. Then we cover the edges from prefix rule collection $\{\{3\}\}$ by enlarging the two smaller bicliques into bicliques 1 and 2. For biclique 1, we shift the hold and setup time of edges from rule collection $\{\{1\}\}$ by 1 cycle, i.e. (0, 1)+1 = (1, 2). As a result, the hold and setup times of biclique 1 are aligned. Similarly, we align the hold and setup times of biclique 2. As a result, we cover the bipartite graph by two bicliques, and produce two rule collections, i.e., $\Re'_1 = \{\{1\}^{+1}, \{3\}\}$ and $\Re'_2 = \{\{2\}^{+1}, \{3\}\}$.

We can go back to Fig.IV.5 to check the path coverage. Since biclique 1 covers four edges from rule collections {{1}} and {{3}} to suffix rule sets {2, 3} and {3}, rule collection $\Re'_1$ covers four paths represented by the edges, i.e., {(1, 4), (4, 5), (5, 6)}, {(1, 4), (4, 5), (5, 7)}, {(2, 4), (4, 5), (5, 6)} and {(2, 4), (4, 5), (5, 7)}.

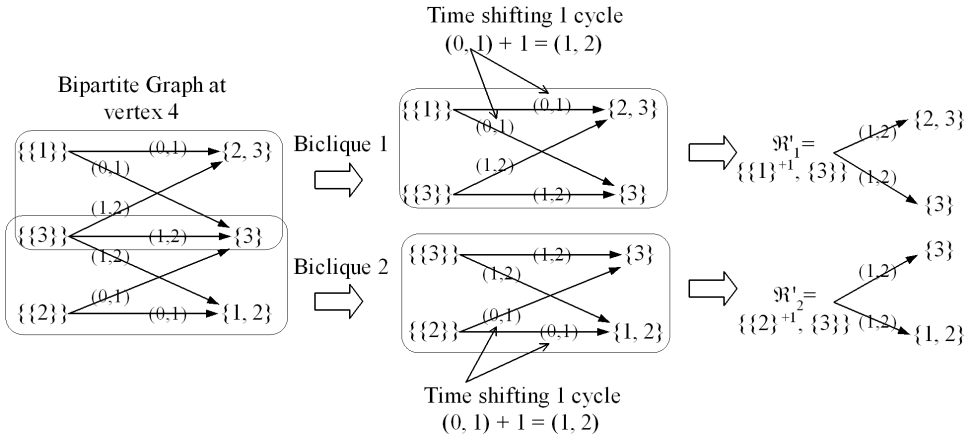


Figure IV.6 Bipartite Covering at Vertex 4: Produce a rule collection based on each biclique.

Theorem IV.C.1 *The time complexity of the heuristic Biclique-Covering(v) algorithm is O(k^3), where k is the number edges in false path specifications.*

Proof: According to [6], the number of prefix and suffix rule sets at vertex v is O(k). For each prefix rule set $\Re(v)$, we try to cover the edges from $\Re(v)$ using existing bicliques. Because the number of existing bicliques is O(k), and the number suffix rule set in each existing biclique is O(k), the run time of covering edges from one rule collection $\Re(v)$ is O(k^2).

The time shifting on a biclique b aligns the hold and setup time of edges to each suffix rule set $R(p^+) \in b$. Because b is aligned before adding the edge from $\Re(v)$, for each $R(p^+) \in b$, the run time of time shifting is O(1). As a result, the time shifting on all suffix rule sets in b requires O(k) time.

Therefore, the time complexity to cover edges from all the rule collections is O(k^3).□

IV.C.5 Rule Collection Propagation

We propagate rule collection $\Re(v)$ to vertex u through edge (v, u). Each prefix rule set in the rule collection is propagated by the equation similar as the equation in the Unified-Rule-Set-Computation algorithm.

Algorithm: Unified-Rule-Collection-Propagation($\Re(v), v, u$)

1. FalseEnd = 0; $\Re(u) = \varnothing$;
2. For each prefix rule set $R \in \Re(v)$
   (a) $R' = R \cap I(v, u)$;
   (b) if u is primary output
      i. if $R' \cap T(u)$ is dominated by non-false subgraph rule
         $\Re(u) = \Re(u) \cup \{R'\}$;
      ii. else FalseEnd = 1;
   (c) else $\Re(u) = \Re(u) \cup \{R'\}$;
3. If not FalseEnd add $\Re(u)$ into vertex u's rule collection list;

After rule collection propagation, we perform intersections of rule collections with suffix rule sets and construct bipartite graph at vertex 5. Table IV.2 shows the intersections and Fig.IV.7 illustrates the bipartite graph.

Table IV.2 Intersections of Rule Collections and Suffix Rule Sets at Vertex 5

		$R(p^+)$		
		{3}	{1,3}	{2,3}
$\{\Re(p^-)\}$	$\{\{1\}^{+1}, \{3\}\}$	$\{\varnothing^{+1}, \{3\}\}$	$\{\{1\}^{+1}, \{3\}\}$	$\{\varnothing^{+1}, \{3\}\}$
	$\{\{2\}^{+1}, \{3\}\}$	$\{\varnothing^{+1}, \{3\}\}$	$\{\varnothing^{+1}, \{3\}\}$	$\{\{2\}^{+1}, \{3\}\}$

For each $Intersect(\Re(v), R(p^+))$, if $\exists R(p_i^-) \cap R(p^+) \in Intersect(\Re(v), R(p^+))$ is dominated by a false subgraph rule, we produce no edge from $\Re(v)$ to $R(p^+)$ because the prefix path p_i^- plus the suffix path p^+ is a false path. For example, according to Table IV.2, we produce no edge from rule collection $\Re_1 = \{\{1\}^{+1}, \{3\}\}$ to suffix rule set {1, 3} because the $Intersect(\{\{1\}^{+1}, \{3\}\}, \{1, 3\}) = \{\{1\}^{+1}, \{3\}\}$,

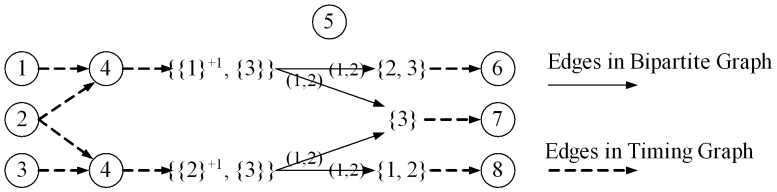


Figure IV.7 Bipartite Covering at Vertex 5

where $\{1\}^{+1}$ is dominated by false subgraph rule 1. This intersection corresponds to false path $\{(1, 4), (4, 5), (5, 8)\}$.

Lemma IV.C.1 ensures that we can get aligned hold and setup time for each edge.

Lemma IV.C.1 *If* $\forall R(p_i^-) \cap R(p^+) \in Intersect(\Re(v), R(p^+))$ *is not dominated by a false subgraph rule, the non-false path rules dominating various* $R(p_i^-) \cap R(p^+)s$ *have aligned hold and setup times.*

Proof:We prove by induction on vertices in topological order. Denote the bipartite graph at vertex v as $B(v)$, the edge in $B(v)$ from rule collection $\Re(v)$ to $R(p^+)$ as $(\Re(v), R(p^+))$. If v is a primary input, $\Re(v) = \{F(v)\}$. Therefore, the $Intersection(\Re(v), R(p^+)) = \{F(v) \cap R(p^+)\}$. The statement is true because the $Intersection(\Re(v), R(p^+))$ only contains one set. If v is not primary input, assume the statement is true for all the input vertices of v. We show that the statement is true at v.

Suppose $\forall R(p_i^-) \cap R(p^+) \in Intersect(\Re(v), R(p^+))$ is not dominated by a false subgraph rule, that is, $\forall R(p_v^-) \cap R(p_v^+) \in Intersect(\Re(v), R(p_v^+))$ is not dominated by false path rules. We want to prove the rules dominating various $R(p_v^-) \cap R(p_v^+)$ have aligned hold and setup time.

Suppose the rule collection after tag minimization is the union $\Re(v) = \cup\Re(v)_i^{\Delta T_i}$, and $\Re(v)_i^{\Delta T_i}$ is propagated from rule collection $\Re(u)$ at v's input vertex

u. Thus,

$$\begin{aligned} \Re(v)_i^{\Delta T_i} &= \{(R(p_u^-) \cap I(u,v))^{\Delta T_i} | R(p_u^-) \in \Re(u)\} \\ Intersect(\Re(v)_i^{\Delta T_i}, R(p_v^+)) &= \{(R(p_u^-) \cap I(u,v) \cap R(p_v^+))^{\Delta T_i} | R(p_u^-) \in \Re(u)\} \\ &= \{(R(p_u^-) \cap R(p_u^+))^{\Delta T_i} | R(p_u^-) \in \Re(u)\} \\ &= Intersect(\Re(u)^{\Delta T_i}, R(p_u^+)). \end{aligned} \quad \text{(IV.1)}$$

Equation IV.1 is valid because $R(p_v^+)$ is backward propagated into suffix rule set at u, i.e., $R(p_u^+) = R(p_v^+) \cap I(u,v)$.

Therefore, $\forall R(p_u^-) \cap R(p_u^+) \in Intersect(\Re(u), R(p_u^+))$ is not dominated by false path rules. From the induction assumption, rules dominating $(R(p_u^-) \cap R(p_u^+))^{\Delta T_i} \in Intersect(\Re(u), R(p_u^+))$ have aligned hold and setup time (h, s). Therefore, after time shifting ΔT_i, the rules dominating various $R(p_v^-) \cap R(p_v^+)$ in $Intersect(\Re(v)_i^{\Delta T_i}, R(p_v^+))$ have aligned hold and setup time

According to the Unified-Biclique-Covering algorithm, we keep the biclique b aligned when adding rule collections into b. Therefore, after union $\Re(v) = \cup \Re(v)_i^{\Delta T_i}$, the rules dominating $\forall R(p_v^-) \cap R(p_v^+) \in Intersect(\Re(v), R(p_v^+))$ have aligned hold and setup time. Therefore, for non-primary input v, the statement is true. This concludes our proof. □

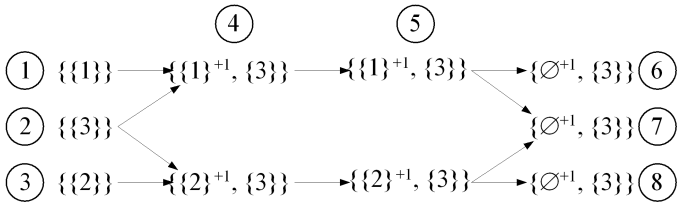


Figure IV.8 Rule Collections and New Vertices after Minimization: $\varnothing^{+1}$s at vertices 6 and 8 are propagated from $\{1\}^{+1}$ and $\{2\}^{+1}$ at vertex 5, respectively; $\varnothing^{+1}$ at vertex 7 is propagated from $\{1\}^{+1}$ and $\{2\}^{+1}$ at vertex 5.

After we perform biclique covering on the bipartite graph at vertex 5, we propagate the rule collections to vertices 6, 7 and 8. Fig.IV.8 summarizes the rule collections at every vertex after rule collection minimization. At vertices 4 and 5,

the number of tags is reduced from 3 to 2.

IV.C.6 Timing Analysis with Rule Collections

We compute the arrival time, required time and slack for each rule collection by forward and backward sweepings similar as the timing analysis process based on rule sets. The only difference is that for rule collections with time shifting, $\Re(v)^{+\Delta T}$, we forward and backward shift the arrival times and required times by ΔT. In Fig.IV.6, at vertex 4, for the rule collection $\Re'_1 = \{\{1\}^{+1}, \{3\}\}$, the arrival time of prefix path $\{(1, 4)\}$ is shifted by 1 cycle and merged with the arrival time of prefix path $\{(2, 4)\}$. When the required time labeled by $\Re'_1$ is backward propagated to vertex 1, we shift back the required time by 1 cycle.

Now, we use Theorem IV.C.2 to show that exceptional rules governing paths through each vertex are covered by the intersections after the tag minimization at the vertex.

Theorem IV.C.2 *Non-false exceptional rules governing paths through vertex v are covered by the intersections of prefix rule collections and suffix rule sets after rule collection minimization at v.*

Proof: We prove the theorem by induction on vertices in topological order. If v is a primary input, path p through v is v plus suffix path $p^+(v)$, rule collection $\Re(v)$ at primary input v is $\{F(v)\}$, and intersection $Intersect(\Re(v), R(p^+)) = \{F(v) \cap R(p^+)\}$. If p is governed by non-false path rule r, G_r contains vertex v and suffix path $p^+(v)$. According to the definition of rule set $F(v)$, $r \in F(v)$. According to Theorem IV.B.1, $r \in R(p^+)$. Therefore, rule r is covered by $F(v) \cap R(p^+) \in Intersect(\Re(v), R(p^+))$. Because r is non-false path rules, $Intersect(\Re(v), R(p^+))$ is not removed at v, thus, rule r is covered by $Intersect(\Re(v), R(p^+))$ at v. If v is not primary input, assume the statement is true for all the input vertices of v. We show that the statement is true at v.

Suppose path p passes through v's input u. Therefore, p contains three parts, i.e., prefix path p_u^- ending at u, edge (u, v), and suffix path p_v^+ starting from

v. If p is governed by non-false path rule r, according to our induction assumption, rule r is covered by intersection $Intersect(\Re(u), R(p_u^+))$ at vertex u. Suppose rule collection $\Re(u)$ is propagated to rule collection $\Re(v)$ vertex v, we show that rule r is covered by intersection $Intersect(\Re(v)', R(p_v^+))$ at vertex v, where $\Re(v) \subseteq \Re(v)'$.

The proof goes as follows. $\Re(v)$ is propagated from $\Re(u)$, i.e., $\Re(v) = \{R(p_u^-) \cap I(u,v) | R(p_u^-) \in \Re(u)\}$. $R(p_u^+)$ is backward propagated from $R(p_v^+)$, i.e., $R(p_u^+) = R(p_v^+) \cap I(u,v)$. Therefore,

$$\begin{aligned} Intersect(\Re(v), R(p_v^+)) &= \{R(p_u^-) \cap I(u,v) \cap R(p_v^+) | R(p_u^-) \in \Re(u)\} \\ &= \{R(p_u^-) \cap R(p_u^+) | R(p_u^-) \in \Re(u)\} \\ &= Intersect(\Re(u), R(p_u^+)) \end{aligned} \quad \text{(IV.2)}$$

Since $Intersect(\Re(v), R(p_v^+)) = Intersect(\Re(u), R(p_u^+))$, r is covered by $Intersect(\Re(v), R(p_v^+))$. Rule sets in $Intersect(\Re(u), R(p_u^+))$ are not dominated by false path rules according to the biclique covering at vertex u. Therefore, rule sets in $Intersect(\Re(v), R(p_v^+))$ are not dominated by false path rules. According to Lemma IV.C.1, we produce an edge in the bipartite graph at vertex v with the aligned setup and hold time attached. According to the biclique covering algorithm, after rule collection minimization, rule collection $\Re(v)$ is a sub-set of rule collection $\Re(v)'$. Therefore, r is covered by $Intersect(\Re(v)', R(p_v^+))$. The statement is true for the non-primary inputs. This concludes the proof.□

IV.C.7 Special Cases of False Subgraph Rules

We can further reduce the number of tagged timings if there are special false path rules satisfying conditions as follows.

Not-Expand-Condition: For ∀ path p governed by rule r, if p is also governed by multi-cycle rule r', the priority of r is higher than the priority of r', i.e., $p_r > p_{r'}$.

If false subgraph rule r satisfies the Not-Expand-Condition, we do not expand r and modify the Unified-Rule-Set-Computation algorithm to remove false

path timing before reaching the primary outputs.

Unified-Rule-Set-Computation^{Modified}(v)

If vertex v is a primary input

If $F(v) \cap T(v) = \varnothing$ Rule set $R = F(v)$;

else For each edge (u, v)

For each rule set R at vertex u

i. $R' = (R \cap I(u, v)) \cup F(v)$;

ii. if v is primary output

if $R' \cap T(v)$ is not dominated by false subgraph rule

A. $Arr_{min}(v, R') = min(Arr_{min}(v, R'), Arr_{min}(u, R) + d(u, v))$;

B. $Arr_{max}(v, R') = max(Arr_{max}(v, R'), Arr_{max}(u, R) + d(u, v))$;

iii. else

if $R' \cap T(v) = \varnothing$

A. $Arr_{min}(v, R') = min(Arr_{min}(v, R'), Arr_{min}(u, R) + d(u, v))$;

B. $Arr_{max}(v, R') = max(Arr_{max}(v, R'), Arr_{max}(u, R) + d(u, v))$;

In step i, the union with rule set $F(v)$ means false path rules starting from vertex v are included in the rule set. For each vertex v before primary output, intersection $R \cap T(v) \neq \varnothing$ indicates a false subgraph rule r ending at v. According to the Not-Expand-Condition, all the paths governed by r is dominated by false path rules. Therefore, we remove the false path timing.

After modification, we reduce the number of tagged-timings by removing false path timings before primary outputs. Since only false path timing is removed, the slacks computation based on rule collections remains correct.

IV.D Experimental Results

We test the proposed approach on both artificial test cases and industry test cases. The algorithm is implemented in C and run on a Pentium 4 Linux

machine.

We first follow the experiments in [3], which randomly create false subgraphs and multi-cycle subgraphs on a 100×100 mesh. The average number of edges in each false subgraph is 6000. Each test case contains from 9 to 104 rules including 30% false subgraph rules. The hold and setup times of multi-cycle paths are in the range from 2-cycle to 4-cycle.

We compare the number of rule collections produced by biclique covering approach with the number of prefix rule sets produced by the Unified-Rule-Set-Computation algorithm in [3] when there are only false paths. Table IV.3 shows the experimental results. The first column is the number of the rules in the test case. The second column and the third column contain the numbers of the prefix rule sets and the rule collections. The reduction ratio = (#prefix rule sets - #rule collections)/ (#prefix rule sets). For the 5 test cases in Table IV.3, the average improvement ratio is 31.22%. The run-time of the minimization increases when the number of rules in the case increases. For the largest case including 104 rules, the CPU time is only 87 seconds.

Table IV.3 Tag Minimization on 100×100 Mesh

#rules	$\#R(v)$	$\#\Re(v)$	$\%r$	runtime(sec)
9	9129	8281	9.29	2
34	77102	49321	36.03	19
69	137581	89987	34.59	44
88	176384	97124	44.94	61
104	209718	145484	30.63	87
average			31.10	

–$R(v)$:Prefix rule set
–$\Re(v)$:Rule collection
–Reduction $r = (\#R(v) - \#\Re(v))/\#R(v)$

We also test our algorithm on a set of industry test cases. The experiment results are in Table IV.4. The second column and the third column contain the numbers of the nets and rules in the test case. The largest circuit contains 533,224 nets with 2 false subgraph rules and 2262 multi-cycle subgraph rules. We use the

Table IV.4 Tag Minimization on Industry Test Cases

cases	#nets	#rules		$\#R(v)$	$\#\Re(v)$	$\%r$	runtime (sec)
		False	Multi-cycle				
tdl	27,555	1	27	158	67	57.59	1
cq_mod	38,535	2517	3181	217,456	14,972	93.11	22
pm25c	325,582	7	2574	1,781,400	101,238	94.32	106
atmlcore	533,224	2	2262	2,411,892	159,451	93.39	383

–$R(v)$:Prefix rule set; $\Re(v)$:Rule collection
–Reduction $r = (\#R(v) - \#\Re(v))/\#R(v)$

Table IV.5 Run Time of Static Timing Analysis Using Rule Collection Tags

	STA Runtime (sec)		
	Use $R(v)$ Tag	Use $\Re(v)$ Tag	% reduction
tdl	1.2	1.2	0
cq_mod	4.2	2	52.38
pm25c	55.33	28.5	48.49
atmlcore	40.33	19.5	51.65

–Reduction of STA runtime = (STA Runtime with $R(v)$ Tag
- STA Runtime with $\Re(v)$ Tag)/STA Runtime with $R(v)$ Tag

Unified-Rule-Set-Computation algorithm to produce prefix rule sets, and minimize rule sets into rule collections. The average improvement ratio is 84.60%. For the largest case *atmlcore*, the runtime of minimization is only 383 seconds, including the CPU time for loading the test cases, mapping the rules on the graph, and minimizing the rule collections.

Table IV.5 also shows runtimes of static timing analysis (STA). If STA uses prefix rule sets to deal with false paths and multi-cycle paths, for the largest circuit *atmlcore*, the runtime is 40.33 seconds. If rule collections are used in STA, the runtime is reduced to 19.5 seconds. Though the reduction is only 20.83 seconds for performing STA once, the reduction ratio is 51.65%. If timing analysis is repeatedly called during performance driven optimization, for example 100 times, the reduction on STA runtime would be 2083 seconds, which is larger than the runtime cost 40.33 seconds for the preprocessing minimization.

IV.E Acknowledgement

This chapter, in full, is a reprint of the material as it appears in Proc. Asia and South Pacific Design Automation Conf. 2006, Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, and Mike Hutton, "Efficient Static Timing Analysis Using a Unified Framework for False Paths and Multi-Cycle Paths", ASP-DAC 2006. The dissertation author was the primary researcher and author and the co-authors listed in these publications directed and supervised the research which forms the basis for this chapter.

V Timing Model Reduction for Hierarchical Timing Analysis

V.A Introduction

In this chapter, we propose an abstract timing model reduction algorithm for hierarchical timing analysis using a biclique-star Replacement technique. By applying tag-based approach, we expand the timing model to cover false paths and multi-cycle paths.

The complexity of hierarchical timing analysis is linear to the number of edges in abstract timing models. In hierarchical timing analysis, a design is divided into multiple blocks and each block is characterized into an abstract timing model. For linear delay model, the timing calculation can be separated according to the boundary of the partitions. Assume the timing relations inside each block are fixed. By minimizing the number of edges for timing propagation in the pre-calculated timing models, we can reduce the analysis complexity.

The previous published works on timing model reduction can be categorized into two groups. The reduction based on the timing graph of the block iteratively reduces the number of edges using graph transformations [33, 42]. However, the transformation is a greedy heuristic, which may not always produce the optimal solution. Another category of methods tried to represent delay metrics in the abstract timing model with fewest edges. An optimal realization of a distance matrix problem is formulated as constructing a graph that preserves shortest-path

distances while minimizing the total sum of edge weights [10, 18]. A clique-star replacement technique is proposed for the graph with unit edge delays [14,15]. The clique is replaced by the star by 1) inserting a Steiner vertex at the center and 2) assigning 12 delay to each edge. However, if the graph has general edge delays, the clique may not be replaced by star due to infeasible edge delays. We are unaware reports that can identify cliques with feasible edge delays for the star replacement in the graph with general edge delays.

We proposed a biclique-star replacement technique to replace bicliques with general edge delays by stars. Based on the proposed technique, we develop a timing model reduction algorithm which minimizes the number of edges in the abstract timing model. Our contributions are as follows.

- We replace a biclique by a star when edge delays from various inputs can be matched to a common delay pattern. The star is constructe based on the delay pattern. By doing so, we cover multiple edges from each input by one edge with a delay offset attached, thus reducing the number of edges. We allow don't-care edges in the delay pattern, thus maximizing the number of edges reduced.

- We present a timing model reduction algorithm which searches bicliques containing delay patterns in the timing model, and replaces the bicliques by stars. The timing model reduction is iteratively performed to maximize the reduction.

- We expand timing models to cover false paths and multi-cycle paths. The tag-based approach in chapter IV is utilized to construct timing models. By attaching tags on vertices in the timing model, false paths and multi-cycle paths are covered. The timing model reduction technique remains valid for the timing model covering false paths and multi-cycle paths.

The remainder of this chapter is organized as follows. In section V.B, we introduce the biclique-star replacement technique. Section V.C presents the timing

model reduction algorithm. Section V.D expands the timing model to cover false paths and multi-cycle paths. The experimental results are presented in Section V.E.

V.B Biclique-Star Replacement

In this section, we propose a biclique-star replacement technique which replaces bicliques with general edge delays by stars. Intuitively, a biclique with unit edge delays can be replaced by a star, such that the edge delays are covered with fewer edges. However, if a biclique contains general edge delays, the delays may not coincide to be covered by a star. We match edge delays from various inputs to a common delay pattern and construct the star based on the pattern. By doing so, we cover multiple edges from one input by one edge, thus reducing the number of edges.

V.B.1 Replacement Covering All Edge Delays

In this section, we propose to replace a biclique by a star and cover all the edge delays in the biclique by the star. We define the edge delay coverage and the biclique-star replacement first. After that, based on the observation on an example, we introduce the technique matching edge delays to a pattern and replacing a biclique by a star.

Definition V.B.1 *(**Edge Delay Coverage**) Edge (i, j) in biclique G_c is covered in a star G_s if $d_{i,s} + d_{s,j} = d_{i,j}$, where $d_{i,j}$, $d_{i,s}$ and $d_{s,j}$ are edge delays in G_c and G_s.*

Definition V.B.2 *(**Biclique-star replacement**) A biclique-star replacement replaces biclique G_c by a star G_s such that*

1. *$B_s = B_c$, $D_s = D_c$, where B_s and D_s are input and output sets of G_s, B_c and D_c are input and output sets of G_s;*

2. all edges in biclique G_c are covered in G_s.

The reduction ratio is the number of edges in G_c over the number of edges in G_s, i.e.,

$$r = (r \times c)/(r + c), \tag{V.1}$$

where r and c are the number of inputs and outputs in G_c.

One observation on biclique-star replacement is that the delays from various input delay vectors are a pattern plus various offsets. For example, Fig.V.1.(a) illustrates a biclique-star replacement. Each edge in the biclique is covered by a two-edge path in the star. For example, the edge (1,4) in the biclique is covered by the path $\{(1, s), (s, 4)\}$ in the star because $d_{1,4} = d_{1,s} + d_{s,4} = 2$. The delay matrix is shown in Fig.V.1.(b). We find out that input delay vector 1 is $0+\{2, 3, 4\}$, input delay vector 2 is $1 + \{2, 3, 4\}$, and input delay vector 3 is $2 + \{2, 3, 4\}$. Thus, the vector $\{2, 3, 4\}$ is the common pattern shared by three input delay vectors.

The hint of the example is that we can replace a biclique by a star as far as the input delay vectors share a common pattern. To identify the pattern in the input delay vectors, we define a vector subtraction operation as follows.

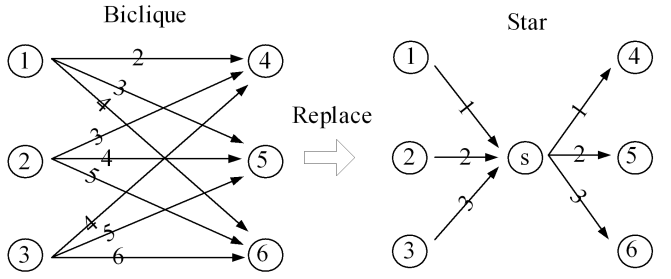
Definition V.B.3 *(**Vector Subtraction**) Given a delay matrix M of a biclique G_c, a vector subtraction between two input delay vectors I_a and I_b, denoted as $Sub(I_a, I_b)$, performs subtractions*

$$\delta_j^{I_a,I_b} = m_{a,j} - m_{b,j}, \tag{V.2}$$

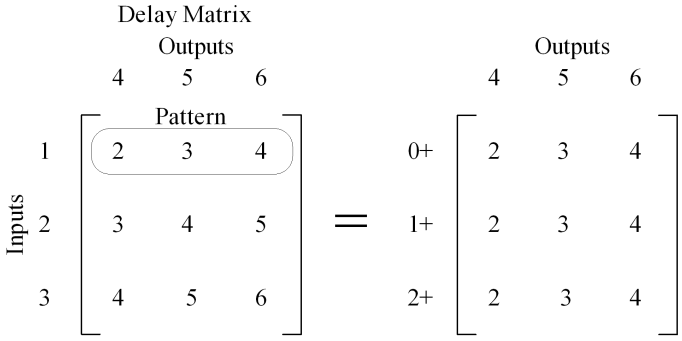
where $m_{a,j} \in I_a$ and $m_{b,j} \in I_b$ and returns a distance vector $V^{I_a,I_b} = \{\delta_j^{I_a,I_b} | j \in [1..c]\}$, where c is the number of column in M.

Definition V.B.4 *(**Pattern of input delay vectors**)*

1 *Two input delay vectors I_a and I_b in delay matrix of biclique G_c share a pattern vector, denoted as $I_a \parallel I_b$, if in the distance vector V^{I_a,I_b} after vector subtraction $Sub(I_a, I_b)$, all $\delta_j^{I_a,I_b}$s in distance vector V^{I_a,I_b} are equal. The value is termed δ^{I_a,I_b}.*



(a) Replace a Biclique by a Star



(b) Input Vectors in the Delay Matrix Share a Common Pattern

Figure V.1 Biclique-Star Replacement

2 *If* $\forall$ *two input delay vectors* I_a *and* I_b *in delay matrix* M *share a pattern vector, i.e.,* $I_a \parallel I_b$*, the biclique* G_c *contains an input pattern vector, denoted as* $\parallel G_c$*.*

The pattern relation between two input delay vectors has properties as follows.

Symmetric If $I_a \parallel I_b$, then $I_b \parallel I_a$, where I_a and I_b are input delay vectors.

Transitive If $I_a \parallel I_b$ and $I_b \parallel I_c$, then $I_a \parallel I_c$,where I_a, I_b and I_c are input delay vectors.

The biclique in Fig.V.1.(a) is an example of $\parallel G_c$. In the delay matrix in Fig.V.1.(b), input delay vectors $I_1 = \{2, 3, 4\}$, $I_2 = \{3, 4, 5\}$, and $I_3 = \{4, 5, 6\}$.

1. $I_2 \parallel I_1$ because after vector subtraction $Sub(I_2, I_1)$, distance vector $V^{I_2,I_1} = \{1, 1, 1\}$, in which all δs are equal to $\delta^{I_2,I_1} = 1$.

2. $I_3 \parallel I_2$ because after vector subtraction $Sub(I_3, I_2)$, distance vector $V^{I_3,I_2} = \{1, 1, 1\}$, in which all δs are equal to $\delta^{I_3,I_2} = 1$.

3. According to the transitive property, from 1 and 2, we have $I_3 \parallel I_1$.

4. According to the symmetric property, any two input delay vectors share a vector patter. Thus, biclique G_c contains an input pattern vector, i.e., $\parallel G_c$

Lemma V.B.1 *Given delay matrix* M *of biclique* G_c*, the complexity to evaluate the pattern vector in* G_c *is* $O(r \times c)$*, where* r *and* c *are the numbers of rows and columns in* M*.*

Proof: We randomly choose one input delay vector I_a and evaluate the pattern relations with all other input delay vectors. Each vector subtraction $Sub(I_a, I_i)$ with input delay vector I_i requires q subtractions. Thus, for r input delay vectors, the complexity is $O(r \times c)$.

The evaluation returns two kind of results:

1. If an input delay vector I_i does not share the pattern with I_a, we conclude that G_c contains no input pattern vector.

2. If we get $I_i \parallel I_a$ for and input delay vector I_i, according to transitive property of pattern relation, $\forall$ input delay vector I_b satisfying $I_i \parallel I_b$ with input delay vector I_i. Therefore, we conclude $\parallel G_c$.

Therefore, for both results, we need no further computation. The complexity of the evaluation is $O(r \times c)$. $\square$

Theorem V.B.1 *Biclique G_c can be replaced by a star if G_c contains an input pattern vector,i.e., $\parallel G_c$.*

Proof: The proof goes as follows. We first construct a star G_s for biclique G_c. Then, we prove that in the produced star G_s all edges in G_c covered.

Algorithm: Biclique-Star-Replacement(G_c)

1. Star $G_s = \{B_s, D_s, s, E_s\}$, where input set $B_s = B_c$, output set $D_s = D_c$, and edge set $E_s = \{(i,s)|i \in B_s\} \cup \{(s,j)|j \in D_s\}$;

2. Pick input $0 \in B_s$ and assign $d_{0,s} = 0$, $d_{s,j} = d_{0,j}$ for edge $(s,j) \in E_s$;

3. For each input $i \in B_s$

   $d_{i,s} = d_{i,0} - d_{0,0}$, where $d_{i,0}$ and $d_{0,0}$ are delay of edges $(i,0)$ and $(0,0)$ in G_c;

We prove delays of all edges in G_c are covered in star G_s. For output edge $(0,j)$ of input 0, according to step 2 in the Star-of-Biclique algorithm, $d_{0,s} + d_{s,j} = 0 + d_{0,j} = d_{0,j}$. Therefore, delays of all output edges from input 0 are covered.

For output edge (i,j) from $\forall i \in B_s$,

$$\begin{aligned} d_{i,s} + d_{s,j} &= d_{i,0} - d_{0,0} + d_{0,j} && \text{(V.3)} \\ &= d_{i,j} - d_{0,j} + d_{0,j} && \text{(V.4)} \\ &= d_{i,j}. \end{aligned}$$

Equation V.3 is valid because of step 3 in the Star-of-Biclique algorithm. Because vector I_0 share a pattern with vector I_i, $\delta_j^{I_i,I_0} = \delta_0^{I_i,I_0}$ for $\forall j \in D_c$. Thus, equation V.4 is valid. Therefore, delays of all the output edges from i are covered. As a result, G_s covers G_c. □

Fig.V.2.(a) illustrates a biclique and the delay matrix. In Fig.V.2.(b), we performs vector subtractions $Sub(I_2, I_1)$ and $Sub(I_3, I_1)$ and get the distance vectors $V^{I_2,I_1} = \{1, 1, 1\}$ and $V^{I_3,I_1} = \{2, 2, 2\}$. Since the δs in each distance vector are equal, the biclique contains an input pattern vector. In Fig.V.2.(c), we construct a star for the biclique. We first cover input delay vector I_1 by setting edge delays $d_{1,s} = 0$, $d_{s,4} = d_{1,4} = 2$, $d_{s,5} = d_{1,5} = 3$, and $d_{s,6} = d_{1,6} = 4$. Then, we cover input delay vectors I_2 and I_3 by setting edge delays $d_{2,s} = \delta^{I_2,I_1} = 1$ and $d_{3,s} = \delta^{I_3,I_1} = 2$. As a result, all the edges in the bicliques are covered in the star and the number of edge is reduced from 9 to 6.

V.B.2 Replacement Allowing Don't Care Edges

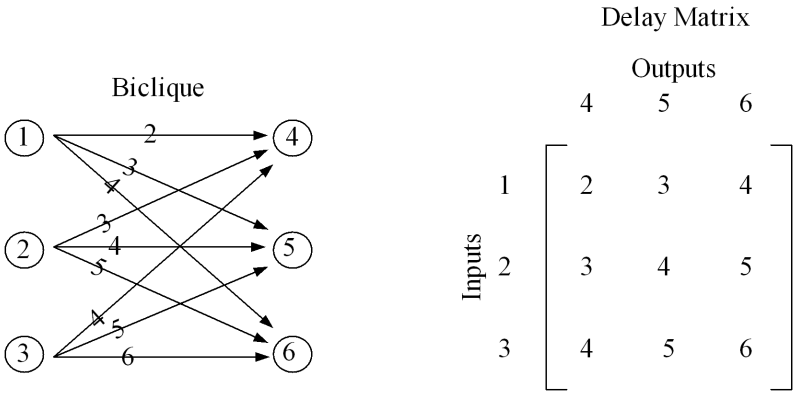
We allow don't-care edges thus generalizing biclique-star replacement to all bicliques. After the replacement, the edge delays in the biclique may or may not be covered in the star. However, as far as the number of edges covered is more than the number of edges used in the star, the replacement is beneficial.

A biclique can be replaced by star as far as the delay from an input to an output in the star does not dominate the corresponding edge delay in the biclique. The biclique-star replacement allowing don't-care edge is defined as follows.

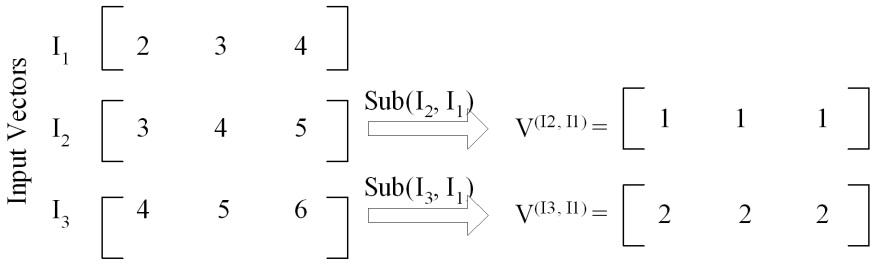
Definition V.B.5 ***(Biclique-star Replacement Allowing Don't Care Edges)*** *Given a maximum delay biclique G_c (II.D.1), a biclique-star replacement allowing don't care edges replaces G_c by a star G_s such that*

1. *$B_s = B_c$, $D_s = D_c$, where B_s and D_s are input and output sets of G_s, B_c and D_c are input and output sets of G_c;*

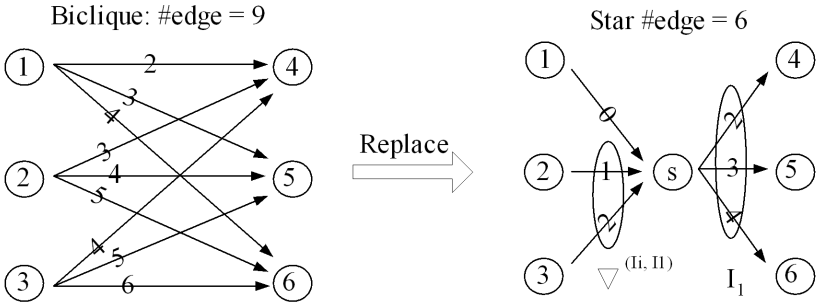
2. *For edge $(i, j) \in E_c$, the delays $d_{i,s}$ and $d_{s,j}$ of edges (i, s) and (s, j) in E_s*



(a) Biclique and Delay Matrix



(b) Biclique Contains an Input Vector Pattern



(c) Replace the Biclique by a Star

Figure V.2 Biclique-Star Replacement Based on Delay Pattern

satisfy

$$d_{i,s} + d_{s,j} \leq d_{i,j}. \tag{V.5}$$

Edge (i, j) *is a don't care edge. The reduction ratio is the number of edges covered in* G_s *over the number of edges in* G_s*. Note an edge is covered in* G_s *only if* $d_{i,s} + d_{s,j} = d_{i,j}$*.*

For a minimum delay biclique (II.D.1), the definition is similar except that the inequality V.5 is changed to

$$d_{i,s} + d_{s,j} \geq d_{i,j}. \tag{V.6}$$

By allowing don't care edges, we can replace a biclique by a star when some sub-vectors of input delay vectors share a delay pattern and all other edges are don't care edges. After replacement, the edge delays in the sub-vectors sharing a pattern are covered. The sub-vectors sharing patterns and sub-vectors of don't care edges are defined as follows.

Definition V.B.6 ***(Pattern of Sub-Vectors)***

Sub-vector Sharing Pattern *Given input delay vectors* I_a *and* I_b*, the sub-vector* $I_b^\delta \subseteq I_b$ *shares a pattern with corresponding sub-vector* $I_a^\delta \subseteq I_a$ *under* δ*, termed* $I_b^\delta \parallel I_a^\delta$*, where* $I_a^\delta = \{d_{a,j} | d_{b,j} \in I_b^\delta\}$*, if for* $\forall d_{b,j} \in I_b^\delta$*,* $d_{b,j} - d_{a,j} = \delta$*, i.e.,* $\forall \delta_j^{I_b,I_a} \in V^{I_b,I_a}$ *equals* δ*, where* V^{I_b,I_a} *is the distance vector.*

Sub-vector of Don't Care Delays *Given input delay vectors* I_a *and* I_b*, a delay* $d_{b,j}$ *is a don't care delay under* δ *if* $\delta_j^{I_b,I_a} > \delta$*. All the don't care delays formulates the sub-vector, termed* $I_b^{\delta *}$*.*

An example of sub-vectors sharing pattern and sub-vectors of don't care delays are in Fig.V.3. Fig.V.3.(a) illustrates a biclique and the corresponding delay matrix. In Fig.V.3.(b), we perform vector subtraction $Sub(I_2, I_1)$ and $Sub(I_3, I_1)$. The distance vector $V^{I_3,I_1} = \{0, 0, 1, 1\}$. Under $\delta = 0$, the sub-vector $I_3^0 = \{2, 3\} \in$

I_3 shares pattern with $I_1^0 = \{2,3\} \in I_1$, and the sub-vector $I_3^{0*} = \{5,6\} \in I_3$ contains don't care delays. Under $\delta = 1$, the sub-vector $I_3^1 = \{5,6\} \in I_3$ shares pattern with $I_1^1 = \{4,5\} \in I_1$, and the sub-vector of don't care delays is empty.

Theorem V.B.2 *When allowing don't care edges, $\forall$ biclique G_c can be replaced by a star G_s.*

Proof: The proof goes as follows. We first modify the Biclique-Star-Replacement algorithm to construct a star allowing don't cares. Then, we show that we can replace the biclique by the star and the replacement satisfies inequality V.5.

Algorithm: Biclique-Star-Replacement-Allowing-Don't-Care(G_c,a)

1. Construct star $G_s = \{B_s, D_s, s, E_s\}$, where input set $B_s = B_c$, output set $D_s = D_c$, and edge set $E_s = \{(i,s)|i \in B_s\} \cup \{(s,j)|j \in D_s\}$;
2. Randomly choose input a and assign $d_{a,s} = 0$, $d_{s,j} = d_{a,j}$ for each edge $(s,j) \in E_s$;
3. For each input $i \in B_s$
   (a) Vector subtraction $Sub(I_a, I_i)$;
   (b) $d_{i,s} = \min\{\delta_j^{I_a,I_i} | \delta_j^{I_a,I_i} \in V^{I_a,I_i}\}$;

Now, we prove that after replacing G_c by G_s, edge delays in G_s satisfy inequality V.5. For output edge (a,j) of input a, according to step 2 in the algorithm, $d_{a,s} + d_{s,j} = 0 + d_{a,j} = d_{a,j}$. Therefore, the delays from input a to outputs j remain the same.

For input i, since we assign the minimum δ in distance vector V^{I_a,I_i} as edge delay $d_{i,s}$, we have inequalities as follows,

$$d_{i,s} + d_{s,j} <= \delta_j^{I_a,I_i} + d_{a,j} \quad \text{(V.7)}$$

$$= d_{i,j} - d_{a,j} + d_{a,j} \quad \text{(V.8)}$$

$$= d_{i,j} \quad \text{(V.9)}$$

Therefore, the delays from input i to outputs j do not dominate the edge delay $d_{i,j}$ in G_c. This concludes the proof. □

We keep the direct edge (i, j) in the timing model to cover the edge delay of each don't care edge when replacing a biclique by a star. An example of the replacement allowing don't care edges is illustrated in Fig.V.3. Fig.V.3.(a) illustrates the biclique and the delay matrix. In Fig.V.3.(b), we get the distance vectors V^{I_2,I_1} and V^{I_3,I_1}. The minimum δs in V^{I_2,I_1} and V^{I_3,I_1} are 1 and 0, respectively. In Fig.V.3.(c), we construct the star by assigning delays in input delay vector I_1, i.e., $\{2, 3, 4, 5\}$, to edges $(s, 4)$, $(s, 5)$, $(s, 6)$ and $(s, 7)$. The minimum $\delta^{I_2,I_1} = 1$ and $\delta^{I_3,I_1} = 1$ are assigned to edges $(2, s)$ and $(3, s)$. We keep the don't care edges (3,6) and (3,7) after the replacement. From the biclique to the star, the number of edge is reduced from 12 to 9.

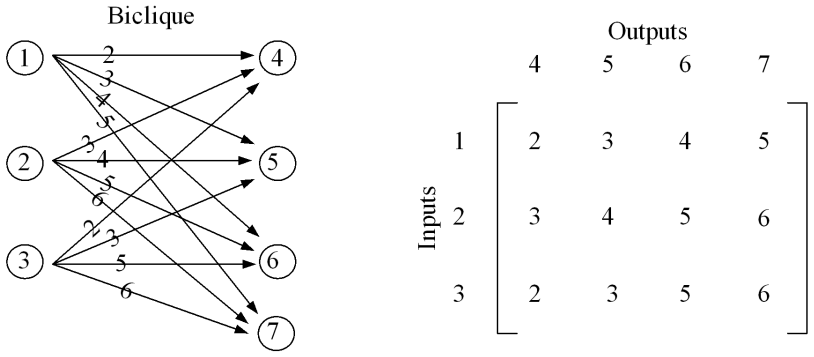
V.C Timing Model Reduction Based on Biclique-Star Replacements

In this section, we search bicliques containing delay patterns in the abstract timing model and minimize the number of edges by replacing bicliques by stars. The problem is equivalent to the minimum biclique covering problem without considering the edge delays, which is NP complete [15, 34, 36]. Therefore, we develop a set of heuristics to solve the problem in polynomial time.

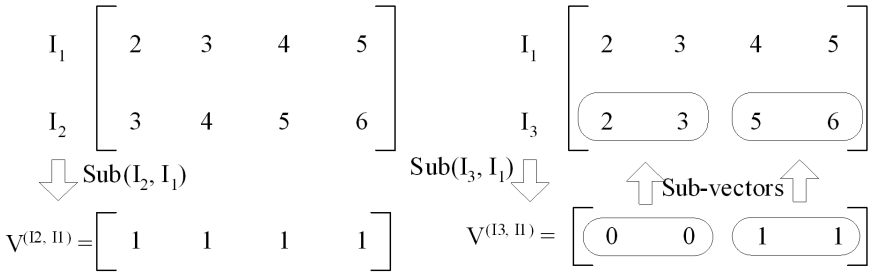
V.C.1 Main Flow of Bipartite Timing Model Reduction

The main flow contains three steps. Firstly, we achieve a set of bicliques in the timing model as the replacement candidates. After that, we evaluate the reduction ratio for each biclique. A high reduction ratio indicates that a large number of edges can be reduced after the replacement. Finally, we choose the biclique with the maximum reduction ratio to replace.

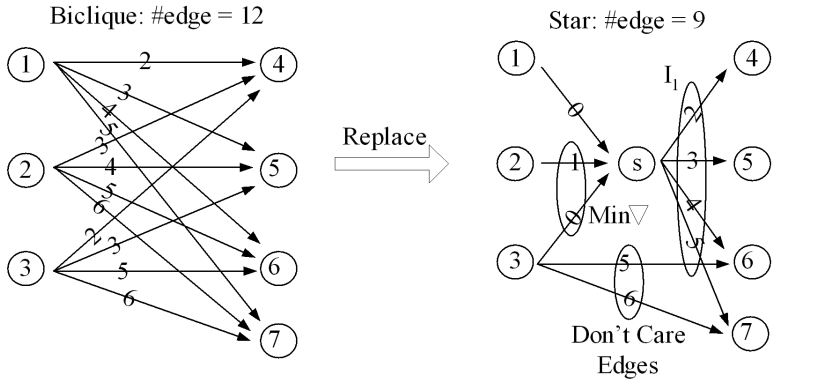
Bipartite Timing Model Reduction(G)



(a) Biclique and Delay Matrix



(b) Sub-vectors Sharing Pattern



(c) Replace the Biclique by a Star with Don't Care Edges

Figure V.3 Biclique-Star Replacement Allowing Don't Care Edges

1. Biclique Pool = Biclique-Search(G);
2. Repeat
   (a) Evaluate the reduction ratio for each biclique in Biclique Pool;
   (b) if max reduction > 1
      i. Replaces G_c with the max reduction by a star;
      ii. Remove G_c from BicliquePool;
3. Until max reduction < 1

The Indentify-Bicliques procedure returns a set of bicliques as the candidates to be replaced. We evaluate all the bicliques in the Biclique Pool, and replace the one with the maximum reduction ratio by a star using the Biclique-Star-Replacement-Allowing-Don't-Care algorithm. We repeat the evaluation and replacement steps until all the reduction ratios are smaller larger than 1, which means the number of edges can not be reduced further.

V.C.2 Biclique Search in Bipartite Timing Model

We search bicliques in the timing model as the replacement candidates and try to maximize the edge reduction. Although any biclique can be replaced by a star by allowing don't care edges, the edge reduction produced by the replacement is different. We devise two rules for biclique search according to reduction ratio defined in equation V.1. The definition is not accurate for some cases, such as the biclique including don't care edges and some edges covered in various bicliques. However, it indicates the reduction potential of the biclique, and thus can be used in the biclique search.

Maximize Biclique Size Bicliques of larger size potentially have higher reduction ratio.

Maximize Edge Coverage We try to cover as many as possible edges with bicliques. If an edge is not covered by any biclique, we need one direct edge

to cover the edge delay in the timing model after minimization. However, if the edge is covered by a biclique with reduction ratio r, after the biclique is replaced by a star, the edge delay is covered by $1/r$ edge. As far as $r > 1$, there are benefits.

Following these two rules, the biclique search algorithm is as follows.

Algorithm: Biclique-Search(G)

1. Biclique Pool = $\varnothing$;
2. Repeat
   (a) Randomly choose edge $(p, q) \in E$ which is not covered by any biclique;
   (b) Input set $B_c = \{p\}$, edge set $E_c = \{(p, j)|(p, j) \in E\}$, output set $D_c = \{j|(p, j) \in E\}$;
   (c) For each input i connected with output q Biclique-Expansion(G, G_c, p, q, i);
   (d) Add biclique G_c to Biclique Pool;
3. Until all edges covered;

In the algorithm, we iteratively construct bicliques starting from uncovered edges thus maximizing the edge coverage. When we construct the biclique for edge (p, q), all the edges (p, j) from input p are added to the biclique first. Then, we expand the biclique to cover edges from other inputs. When performing biclique expansion to input i, we try to cover as many as possible edges from input i and remove as few as possible edges already in the biclique. By doing so, the size of the biclique is maximized. The Biclique-Expansion algorithm is as follows.

Algorithm: Biclique-Expansion(G, G_c, p, q, i)

1. Vector subtraction $Sub(I_i, I_p)$ between input delay vectors I_i and I_p;
2. max = 0;
3. For each $\delta \leq \delta_q^{I_i, I_p}$

(a) Get sub-vector I_i^δ sharing a pattern with $I_p^\delta \in I_p$ under δ;

(b) Get sub-vector $I_i^{\delta*}$ of don't care delays under δ;

(c) current = Added-over-Removed($G_c, I_i^\delta, I_i^{\delta*}, \delta$);

(d) If (current > max)

max = current, max vector = I_i^δ;

4. If max > 0

(a) For each output j in D_c

i. If $d_{i,j} \in I_i^\delta$ Add edge (i, j) to E_c;

ii. else Remove output j and all input edges to j from G_c;

Function: Added-over-Removed($G_c, I_i^\delta, I_i^{\delta*}, \delta$)

1. Added = $|I_i^\delta|$, Removed = 0;

2. For each output j in D_c

(a) If $d_{i,j} \notin I_i^\delta \cup I_i^{\delta*}$

(b) Removed = Removed + #Edges to output j with delays covered in G_c;

3. Return(Added - Removed);

We first perform vector subtraction between input delay vectors I_i and I_p. Then, we use the Added-over-Removed function to evaluate the number of edges covered for each $\delta \leq \delta_q^{I_i,I_p}$ as if we 1) find sub-vectors sharing pattern under δ, i.e., $I_i^\delta \parallel I_p^\delta$, and 2) perform the Biclique-Star-Replacement with $d_{i,s} = \delta$. According to the Biclique-Star-Replacement-Allowing-Don't-Care algorithm, when assigning $d_{i,s} = \delta$, an edge (i, j) is covered if delay $d_{i,j} \in I_i^\delta$, where $I_i^\delta \parallel I_p^\delta$. The edges (i, j) with delays $d_{i,j} \notin I_i^\delta \cup I_i^{\delta*}$ can not be added. Therefore, we need to remove output j to keep the biclique complete. As a result, the edges to j which are originally covered in the biclique are counted as removed edges. After the evaluation, we expand the biclique based on the δ which maximizes the edge coverage. The edge

(i, j) is added if $d_{i,j}$ belongs to the union $I_i^{\delta} \cup I_i^{\delta*}$. Otherwise, the output j and the edges to j are removed. By restricting $\delta_j^{I_i,I_p} \leq \delta_q^{I_i,I_p}$, we ensure that $d_{i,q} \in I_i^{\delta} \cup I_i^{\delta*}$, thus keeping output q and edge (p, q) in the biclique.

Fig.V.5 and Fig.V.6 illustrate a biclique expansion example based on the bipartite timing model in Fig.V.4. We want to construct the maximum biclique covering edge (1,6). The expansion to each input is as follows.

Step 1 Edges from input 1 are included. Thus, edge (1,6) is covered.

Step 2 We expand the biclique to input 2. We first perform the vector subtraction between input delay vectors I_1 and I_2 and get distance delta $V^{I_2,I_1} = \{1, 1, 1, 2, \infty\}$. Then, under $\delta_6^{I_2,I_1} = 1$ we get sub-vector $\{4, 5, 6\}$ in I_2 which shares a pattern with sub-vector $\{3, 4, 5\}$ in I_1. Therefore, the number of edges covered is increased by 3. Edge (2,9) is a don't care edge because $\delta_9^{I_2,I_1} > \delta_6^{I_2,I_1}$. Thus, no edge is removed. Since δ should be smaller than $\delta_6^{I_2,I_1} = 1$ to keep edge (1,6) in the biclique, we don't need to try $\delta_9^{I_2,I_1} = 2$. Thus, we expand biclique to input 2 by adding four edges (2,6),(2,7), (2,8), and (2,9). Among these four edges, three edges, i.e., (2,6), (2,7), and (2,8) are covered.

Step 3 We expand the biclique to input 3. The expansion process is similar as that of input 2. Four edges (3,6), (3,7), (3,8) and (3,9) are added into the biclique, and all these four edges are covered.

Step 4 We expand the biclique to input 4. The distance vector is $V^{I_4,I_1} = \{3, 3, 3, 2\}$. Under $\delta_6^{I_4,I_1} = 3$, we get sub-vector $\{6, 7, 8\}$ in I_4 which shares a pattern with sub-vector $\{3, 4, 5\}$ in I_1. Therefore, the number of edge covered is increased by 3. Edge (4,9) is not a don't care edge because $\delta_9^{I_4,I_1} < \delta_6^{I_4,I_1}$. Since edge (4,9) can not be added into the biclique, we have to remove output 9. Two edges to output 9, i.e., (1,9) and (3,9), originally covered in the biclique are also removed. Therefore, the added-over-removed is 1 for $\delta_6^{I_4,I_1} = 3$. Because $\delta = 2$ is smaller than $\delta_6^{I_4,I_1} = 3$, we need to evaluate the

added-over-removed for $\delta = 2$. The result is also 1. We randomly choose to expand biclique based on $\delta_6^{I_4,I_1} = 3$. Three edges (4,6), (4,7) and (4,8) are added. Output 9 and edges to output 9 are removed.

Step 5 We expand the biclique to input 5. After expansion. three edges (5,6), (5,7), and (5,8) are added into the biclique, and two edges (5,6) and (5,7) in these three are covered.

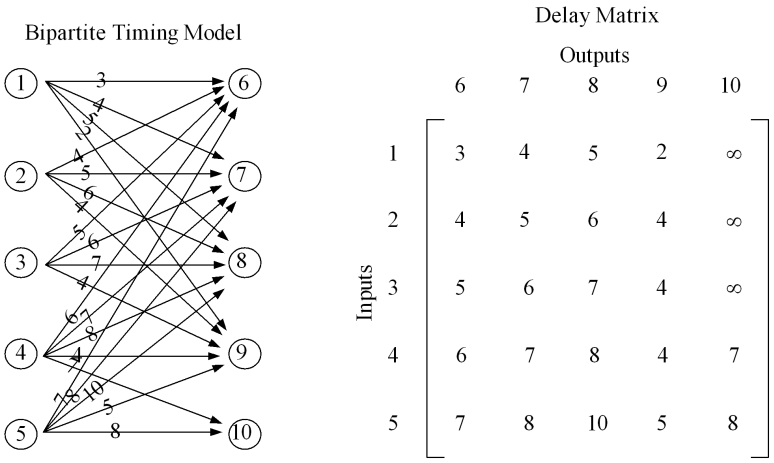


Figure V.4 Bipartite Timing Model and the Delay Matrix Example

We further achieve all the bicliques in the bipartite timing model in Fig.V.4, and illustrate the bicliques in Fig.V.7. For each biclique, we show the edge from which we start the expansion. For example, we expand biclique G_{c1} starting from edge (1,6), and expand biclique G_{c2} from edge (1,9). The solid lines represent edges covered in the bicliques, and the dotted lines represent don't care edges. Each edge in the bipartite timing model is covered by at least one biclique.

V.C.3 Edge Reduction Evaluation

We use reduction ratios to evaluate benefits of replacing bicliques by stars. However, the reduction ratio defined in equation V.1 is not accurate when there are don't care edges in the biclique or some edges are covered by multiple bicliques. A more accurate and general definition of the reduction ratio is as follows.

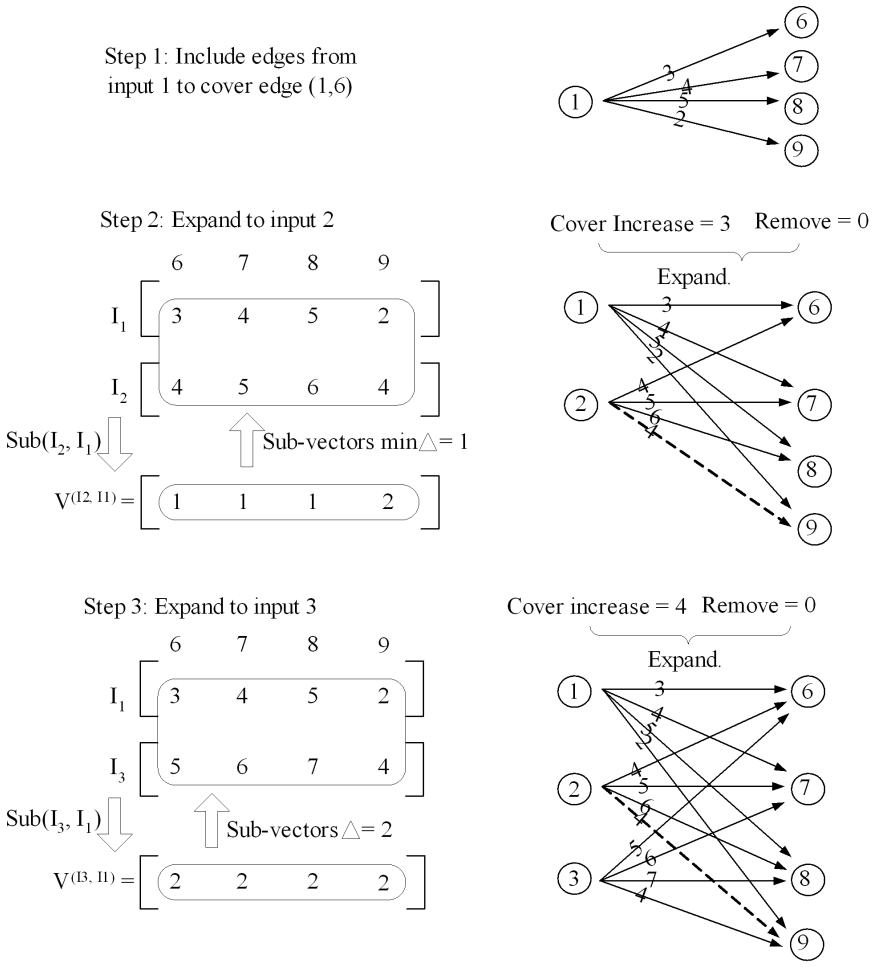


Figure V.5 Biclique Expansion Starting from Edge (1,6) in Bipartite Timing Model (Fig.V.4): Steps 1 to 3.

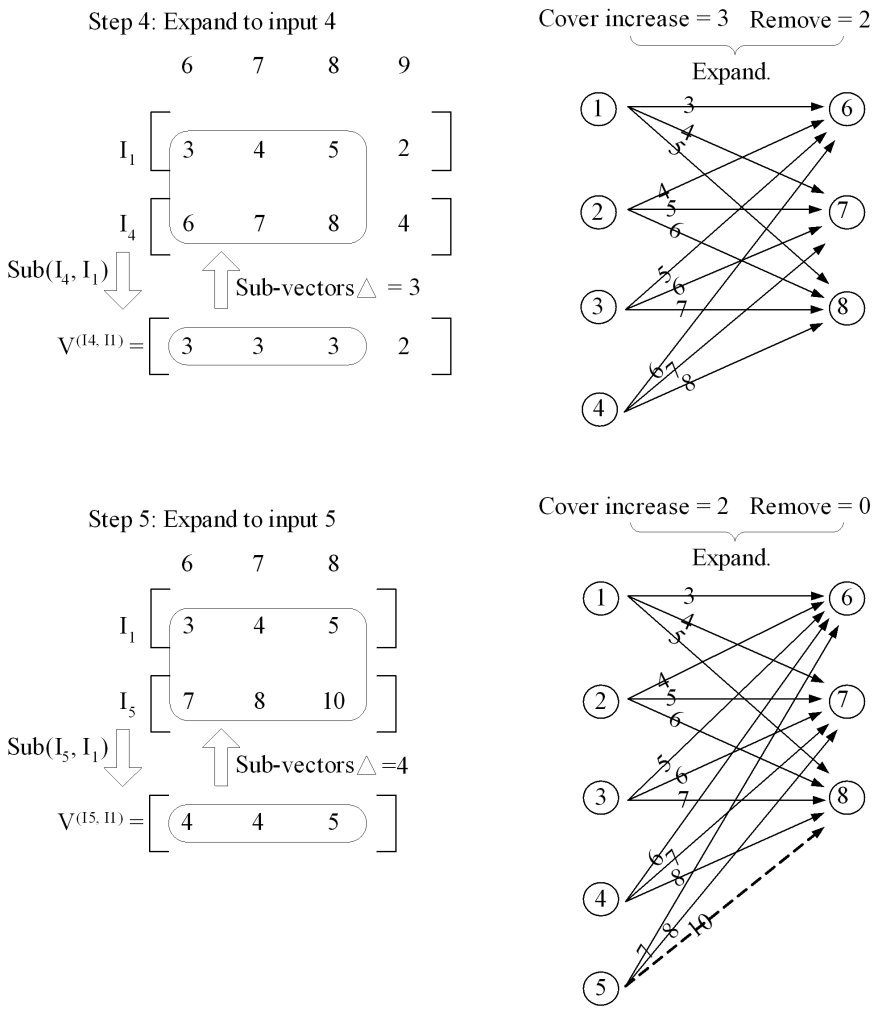


Figure V.6 Biclique Expansion Starting from Edge (1,6) in Bipartite Timing Model (Fig.V.4) : Steps 4 and 5.

Definition V.C.1 *(**Reduction Ratio**) Given a biclique G_c, if G_c can be replaced by a star G_s, the reduction ratio $r = c/(m+n)$, where c is the number of edge delays covered and only covered by G_s, $m+n$ is the number edges in G_s.*

Therefore, when replacing a biclique by a star, we label the edges covered by the star. After the replacement, we re-compute the reduction ratios for the bicliques left in the Biclique Pool. All the edges with labels are not counted.

For example, for the bicliques in Fig.V.7, before any biclique replaced by a star, the reduction ratios are $r(G_{c1}) = 14/(3+5) = 1.75$, $r(G_{c2}) = 13/(4+5) =$

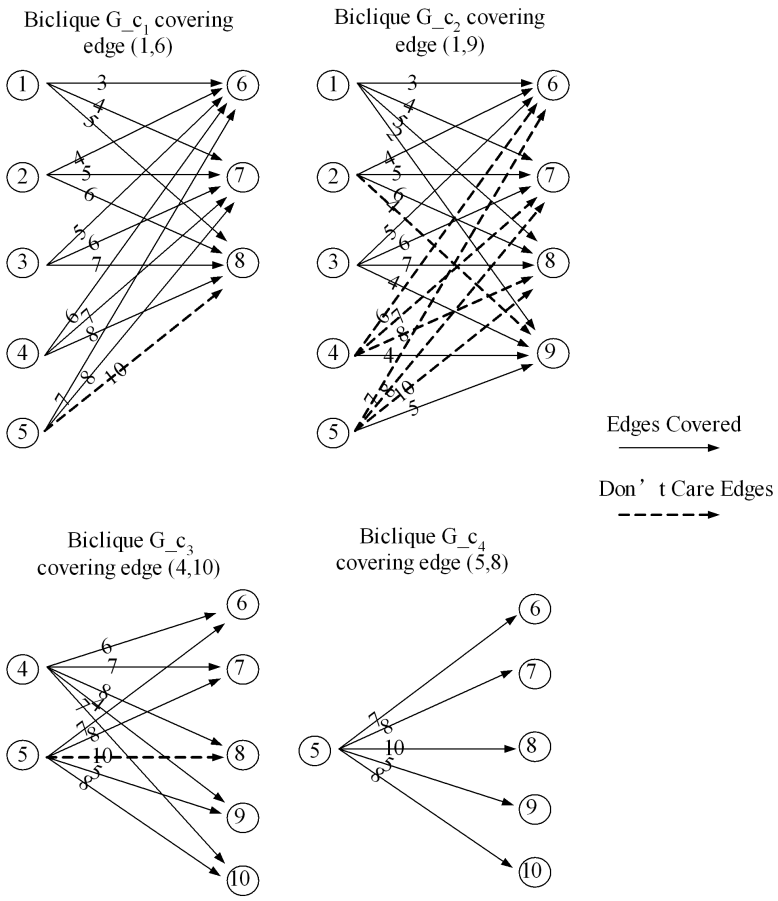


Figure V.7 All Bicliques in Bipartite Timing Model (Fig.V.4)

13/9, $r(G_{c3}) = 9/7$, and $r(G_{c4}) = 5/(5+1) = 5/6$. Because biclique G_{c1} has the maximum reduction ratio, we will replace G_{c1} by a star. After the replacement, we re-evaluate the reduction ratios. $r(G_{c2}) = 4/(4+5) = 4/9$ because edges (1,6), (1,7), (1,8), (2,6), (2,7), (2,8), (3,6), (3,7) and (3,8) covered by G_{c1} are not counted. After similar re-evaluation, $r(G_{c3}) = 4/7$ and $r(G_{c4}) = 1/2$. Therefore, after G_{c1} replaced by the star, the reduction ratios of all biclique are no larger than 1. The timing model after reduction is shown in Fig.V.8. The number of edges in the bipartite timing model is reduced from 22 to 16.

Theorem V.C.1 *The complexity of constructing maximum biclique for edge* (p, q) *is* $O_{pq} = O((d^-(q) \times d^+(p)^2))$*, where* $d^+(p)$ *and* $d^-(q)$ *are output and input degree of input* p *and output* q*;*

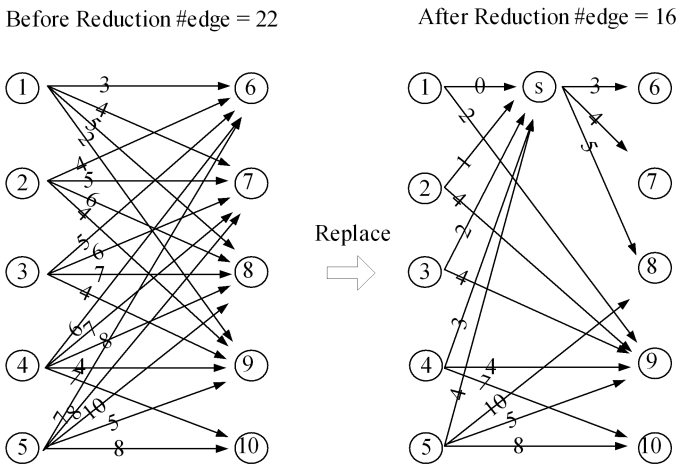


Figure V.8 Bipartite Timing Model (Fig.V.4) Reduction : The number of edges is Reduced from 22 to 16.

The complexity of biclique evaluation $O_e = O(|E| \times k)$, where E is the edge set of the timing model and k is the number of bicliques in Biclique Pool;

The complexity of timing model reduction is $O(\Sigma(O_{pq}) + k \times O_e)$.

Proof:

(1) For each edge (p, q), we expand biclique to input i if i connects with output q which is $d^-(q)$ times expansion. For each expansion, we search sub-vectors in input delay vector I_i using different δs. The number of δs is $d^+(p)$. For each δ, to evaluate the number of edges added over removed, we need another δ times comparison. As a result, the complexity to constructing maximum biclique from edge (p, q) is $d^-(q) \times (d^+(p))^2)$.

(2) To evaluate the reduction ratio of each biclique, we need to check each edge in the biclique and count the number of covered edges. Each biclique in Biclique Pool contains at most $|E|$ number of edges. Thus, the complexity for k bicliques would be $O(|E| \times k)$.

(3) Because we achieve a maximum biclique from each edge, for all edges the complexity is $O(\Sigma(O_{pq}))$. After replacing each biclique in Biclique Pool by a star, we need to update the biclique reduction ratios. Thus, for k biclique, the

complexity is $O(k \times O_e)$. As a result, the complexity of timing model reduction is $O(\Sigma(O_{pq}) + k \times O_e)$. $\square$

V.C.4 Iterative Timing Model Reduction

We iteratively perform the bipartite timing model reduction to reduce the number of edges further. A vertex splitting and a star recover technique are proposed to maintain the timing model a bipartite graph. Based on the bipartite timing model, we can repeat the bipartite timing model reduction process until no improvement, thus maximizing the edge reduction.

After replacing a set of bicliques by stars, the timing model is not bipartite graph any more. The inserted vertices, which are the centers of the stars, partition a part of the timing model into two bipartite graphs. Intuitively, we can repeat the timing model reduction on each bipartite partition and accumulate the results into the whole timing model. However, we may lose the ability to discover larger bicliques crossing multiple bipartite partitions, which makes the reduction low efficient. For example in Fig.V.9, the timing model is partitioned by vertices s_1 and s_2 into several bipartite graphs. The biclique circled in the figure which includes inputs 1,2,3, vertices s_1 and s_2, and output 9, is hard to be discovered.

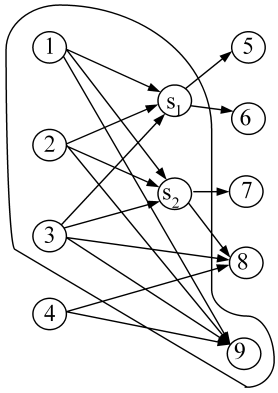


Figure V.9 Bicliques Crossing Multiple Bipartite Partitions

We propose a vertex splitting technique to transform the timing model into a bipartite graph. By doing so, we can discover larger bicliques thus improving

the reduction ratio. We split vertex s of each star G_s into two vertices s and s' as follows.

Algorithm: Vertex-Splitting(G, G_s)

1. Split vertex s into vertices s and s';
2. Input set $B = B \cup \{s'\}$, output set $D = D \cup \{s\}$;
3. For each output j in output set D_s

   $E = E - \{(s, j)\}$ and $E = E \cup \{(s', j)\}$

We add the duplicated vertex s' into the input set and push vertex s into output set D. All the edges originally from s to j are moved to vertex s'.

As the reverse process of the vertex splitting, we recover a star by merging the corresponding vertices s' and s.

Algorithm: Star-Recover(G', s, s')

1. $B = B - \{s'\}$, $D = D - \{s\}$;
2. For each edge (s', j)

   Add edge (s, j);

Fig.V.10 illustrates an example of vertex splitting and star recover. From left to right, we split vertices s_1 and s_2, add s'_1 and s'_2 into the input set, and push s_1 and s_2 into the output set. After the vertex splitting, the timing model G is transformed into a bipartite graph. The star recover is the reverse process, which merges s_1 with s'_1, and s_2 with s'_2. The bipartite graph is recovered into the timing model.

According to the vertex splitting and star recover algorithm, there is a One-to-One mapping between edges before and after the vertex splitting.

Property: One-to-One Mapping: Suppose timing model G is transformed into bipartite graph G' by vertex splitting.

- Edge $(s, j) \in E$ corresponds to edge $(s', j) \in E'$.

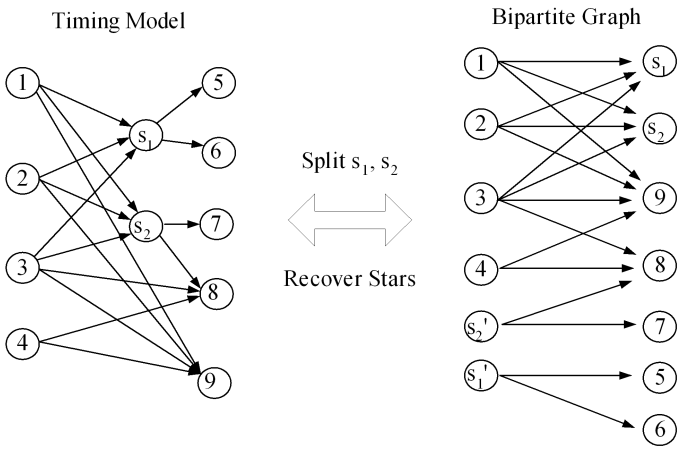


Figure V.10 Vertex Splitting and Star Recover

- Edge $(i, j) \in E$ corresponds to edge $(i, j) \in E'$.
- Edge $(i, s) \in E$ corresponds to edge $(i, s) \in E'$.

Note the map is unique and two-directional.

With vertex splitting and star recover, we can iteratively perform the bipartite timing model reduction to minimize the number of edges in the timing model.

Algorithm: Iterative-Reduction(G)

1. Repeat
   (a) Bipartite-Timing-Model-Reduction(G);
   (b) For all stars G_s Vertex-Splitting(G, G_s)
2. Until no edge reduction
3. For all vertices s and s' Star-Recover(G, s, s')

Theorem V.C.2 *Edge delay $d_{i,j}$ of any connected input i and output j in timing model G is covered by the longest path delay $d'_{i,j}$ from input i to output j in timing model G' after the reduction.*

Proof: We prove by induction on bipartite timing model reduction iterations. Suppose the maximum delay model before reduction is G_0. We label the timing models after each step in the ith iteration as follows.

G_i**:** Models after Timing-Model-Reduction.

G'_i**:** Bipartite graph after Vertex-Splitting.

G''_i**:** Timing model after Star-Recover.

We prove in G''_i, the longest path delay for input-output pair (i, j) equals delay $d_{i,j}$ of edge (i, j) in model G_0.

After the 1st iteration, we get G_1, in which a set of bicliques are replaced by star. According to the Bipartite-Timing-Model-Reduction algorithm, for $\forall$ input-output pair (i, j), $\max(d_{i,s} + d_{s,j}) = d_{i,j}$ where $\{(i, s), (s, j)\}$ is a two-edge path in G_1 and $d_{i,j}$ is delay of edge (i, j) in G_0. Since G''_1 is the same as G_1, the statement holds. Assume the statement holds for all iterations before k. We show the statement true for the kth iteration.

According to the reduction assumption, in G''_{k-1} the longest path delay from input i to output j equals $d_{i,j}$, where $d_{i,j}$ is delay of edge (i, j) in G_0. According to the One-to-One Mapping property of vertex splitting, each edge on the path from i to j corresponds to an edge (s_{p-1}, s_p) in G'_{k-1}.

The model reduction of iteration k is performed on G'_{k-1}. According to the Bipartite-Timing-Model-Reduction algorithm, after model reduction, the longest two-edge path delay from input s_{p-1} to s_p equals delay d_{s_{p-1},s_p} of edge (s_{p-1}, s_p) in G'_{k-1}. Thus, after recovering the stars, in G''_k the longest path delay from input i to output j equals $d_{i,j}$, where $d_{i,j}$ is delay of edge (i, j) in G_0. The statement holds for iteration k. This concludes our proof. $\square$

V.D Timing Model Reduction with False Paths and Multi-cycle Paths

We expand timing model to cover false paths and multi-cycle paths, and keep the proposed model reduction technique valid for the expanded timing model. We first analyze the cases of false paths and multi-cycle paths in hierarchical blocks. Then, we propose to use the tag-based approach IV in the bipartite timing model characterization. The false paths are removed and multi-cycle timings are covered on edge delays. Since the abstract timing model remains in the same format, the model reduction technique is still valid.

The hierarchical blocks may contain false paths and multi-cycle paths. Fig.V.11 illustrates three possible cases.

1. The path is contained in the block, i.e., f_1 in block 2.
2. The path starts in one block and ends in another, i.e., f_2 from block 1 to block 2.
3. The path goes through the block, i.e., f_3 through block 2.

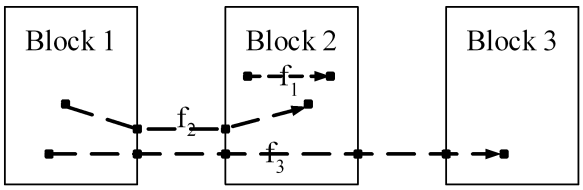


Figure V.11 Hierarchical Blocks Containing False Paths

We follow the tag-based approach to deal with false and multi-cycle paths on the flat circuit, and then construct the abstract timing model based on the tags inside hierarchical blocks. Tags are attached on the inputs and outputs in the timing model for slack computation. After we get the bipartite abstract timing model, we can iteratively apply the model reduction algorithm to do the minimization. The maximum delay model of hierarchical block H is constructed as follows.

Algorithm: Model-Construction(H)

1. $G = \{B, D, E\}$, input set $B = \varnothing$, output set $D = \varnothing$, edge set $E = \varnothing$;
2. For each tag R at input i of block H $B = B \cup \{i_R\}$;
3. For each tag R at output j of block H $D = D \cup \{j_R\}$;
4. For each tag R at input i of block B
   (a) For each tag R' at output j of block B $Arr_{max,R'}(j) = -1$;
   (b) Forward propagates $Arr_{max}(i, R)$ until outputs of B;
   (c) For each tagged arrival time $Arr_{max}(j, R')$ at output j
   If $Arr_{max}(j, R') \neq -1$
      i. $E = E \cup \{(i_R, j_{R'})\}$;
      ii. $d_{i_R, j_{R'}} = Arr_{max}(j, R')$;

For each tag R at input i, we add a tagged input i_R into input set B. Similarly, we add tagged outputs j_R into output set D. We forward propagate tagged timing $Arr_{max}(i, R)$ at the input i to all outputs. The arrival time $Arr_{max}(j, R')$ at output j is the longest path delay from i_R to $j_{R'}$. The time shifting is included when forward propagating the arrival times. After timing propagation, an edge $(i_R, j_{R'})$ with delay $Arr_{max}(j, R')$ is added to edge set E.

V.E Experimental Results

We test the proposed approach on industry test cases. The algorithm is implemented in C and run on a Pentium 4 Linux machine. We construct and minimize the timing models for two circuit blocks. Circuit block 1 contains 8499 inputs, 16885 outputs, 138,360 edges in the timing graph of the block, and 262,491 edges in the bipartite timing model. Circuit block 2 contains 4260 inputs, 103,414 edges in the timing graph of the block, and 7728 edges in the bipartite timing model.

We compare the number of edges in the timing model after reduction with both the number of edges in the timing graph of the block and the number of edges in the bipartite timing model. Two reduction ratios are defined as follows.

$$r_G = (E_G - E_m)/E_G \quad \text{(V.10)}$$

$$r_B = (E_B - E_m)/E_B, \quad \text{(V.11)}$$

where E_G is the number of edges in the timing graph of the block, E_B is the number of edges in the bipartite timing model, and E_m is the number of edges in the timing model after the reduction.

Fig.V.12 illustrates the reduction ratios in the iterative reduction process on block b_1. Each data point represents a biclique-star replacement. The vertical axis shows the reduction ratio r_B. The horizontal axis represents the order of the replacements. The reduction process contains three iterations. In each iteration, bicliques are replace by stars in the order of reduction ratios. In the iterative reduction process, the average reduction ratio of each iteration is decreasing. The total reduction is $r_B = 5.1\%$. According to the curve, we conclude that our heuristic approach can effectively obtain bicliques in the abstract timing model in terms of the reduction ratios, and multiple iterations are necessary to maximize the edge reduction.

We allow error bounds on edge delays. For any connected input i and output j, the error bound is defined as follows.

$$d_{i,j} - d'_{i,j} \leq error\_bound, \quad \text{(V.12)}$$

where $d_{i,j}$ and $d'_{i,j}$ are the longest path delays in timing models before and after the model reduction. In Table V.1, we show the number of edges in the timing model after the reduction, i.e., E_m, and the reduction ratios, i.e., r_G and r_B. For block 1, if the error bound is 0, E_m will be larger than E_G and smaller than E_B because the number of edges is increased when we transform the timing graph into the bipartite timing model. By allowing 0.1ns error bound, the number of edges

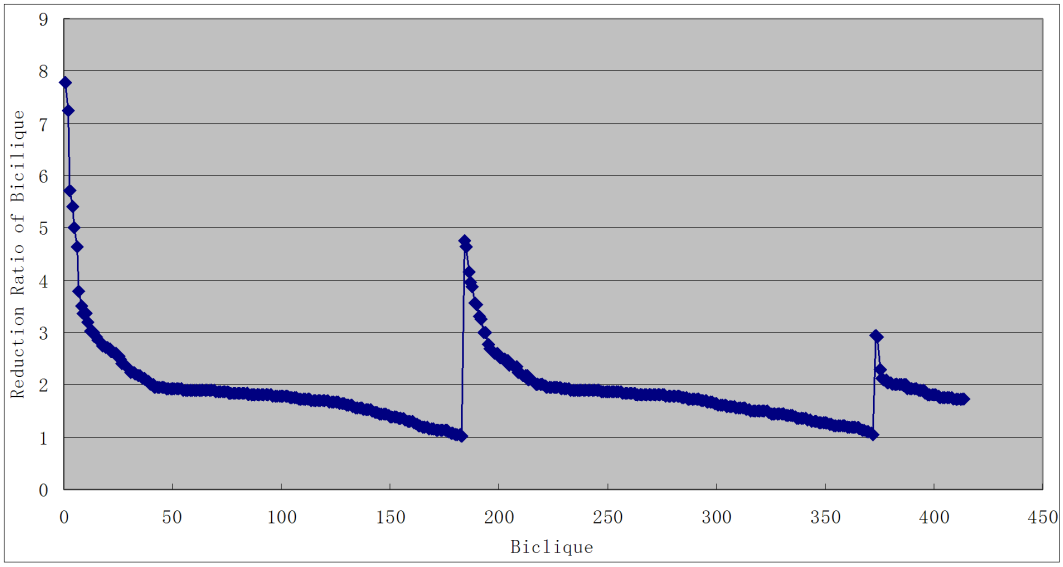


Figure V.12 Reduction Ratios of Replaced Bicliques

is reduced by 69.9% compared with the timing graph and 84.1% compared with bipartite timing model. In block 1, the delay of minimum size buffer, $Buffer \times 1$, is 1.34ns, which is much larger than the error bound. Therefore, we consider the timing model with 0.1ns error bound is acceptable. By further increasing the error bounds, we can reduce more edges. However, the improvement is not substantial and if the error bound is too large, the abstract timing model is not accurate.

Table V.1 Edge Reduction with Error Bounds

Block 1 E_G = 138,360 E_B = 262,491				Block 2 E_G = 103,414 E_B= 465,190			
Error(ns)	E_a	r_G	r_B	Error(ns)	E_a	r_G	r_B
0	249,032	-80.0%	5.1%	0	397,384	-284.3%	14.6%
0.1	41,696	69.9%	84.1%	0.01	49,613	52.0%	89.3%
0.5	39,099	71.7%	85.1%	0.05	35,901	65.3%	92.3%
1.0	36,980	73.3%	85.9%	0.10	29,477	71.5%	93.7%
5.0	36,108	73.9%	86.2%	0.50	22,214	78.5%	95.2%
10.0	35,981	74.0%	86.3%	1.0	21,192	79.5%	95.4%
50.0	36,169	73.9%	86.2%	5.0	20,459	80.2%	95.6%
100.0	36,169	73.9%	86.2%	10.0	20,262	80.4%	95.6%
Buffer × 1 delay = 1.34ns.				Buffer × 1 delay = 0.74ns			

VI Conclusion

VI.A Dissertation Contribution

Static timing analysis plays a vital role in nowadays design flow. The success of the whole process relies on an accurate and efficient static timing analysis package. In this dissertation, we studied the false path, multi-cycle path and hierarchical timing analysis problems which closely relate to the timing analysis accuracy and efficiency.

For false path problem, we present a two-direction propagation technique which uses a biclique covering approach improving the efficiency of static timing analysis. We follow the rule set concept in the previous works to consider the false paths and propose to collect the rule sets into rule collections. By doing so, we effectively reduce the number of the distinguished timings for timing propagations, thus improving the efficiency. We compare the proposed approach with the previous optimization methods. The experimental results verify that the proposed approach significantly reduces the number of the distinguished timings with excellent run time performance. For the optimization process which iteratively invokes timing analysis, the efficiency improvement is substantial.

For multi-cycle path problem, we propose a framework to unify the process of false paths and multi-cycle paths in static timing analysis. Furthermore, we apply the two-direction propagation and the biclique covering on the framework, thus improving the efficiency. Finally, we present theorems to guarantee that our approach produces correct timing information with false paths and multi-cycle

paths considered. The experimental results demonstrate that our minimization is effective.

For hierarchical timing analysis, we focused on abstract timing model reduction, which minimizes the number of edges in the abstract timing model for timing propagations, thus improving the analysis efficiency. We propose a biclique-star replacement technique and develop an iterative timing model reduction algorithm based the biclique-star replacement. By allowing reasonable error bounds, the experiments results show that the proposed algorithm effectively reduces the number of edges in the timing model. Furthermore, by applying tag-based approach, the proposed algorithm can be expanded to cover false paths and multi-cycle paths in timing models.

VI.B Future Works

The future research including following directions.

- Model reconstruction. The unified framework for false paths and multi-cycle paths should deal with false paths and multi-cycle paths dynamically altered during the optimization. Similarly, the timing model reduction algorithm assume the timing relations inside the hierarchical block are fixed, which will not hold when the block is optimized. An intuitive approach which regenerates all the tags or reconstruct the abstract timing model would be low efficient. However, the change on timing graph topology makes the incremental regeneration non trivial.

- Extension to false crosstalk. When signals on two wires transit at the same time, the timing of these two wires affects each other, which is crosstalk. We could consider a timing path between the two wires which the crosstalk goes through. Since crosstalk happens depending on the timing correlation between signals, the timing path for crosstalk between wires maybe false. As a result, we may need to detect and remove false crosstalk during timing

analysis, which leads to high complexity.

- Complexity of timing model reduction. The timing model reduction algorithm suffers from high complexity when the abstract timing model contains a large number of edges. The reason comes from too many bicliques as the reduction candidates. An arbitrary bound on the number of candidates can not ensure the quality. Since the problem is similar as the K-map covering problem in synthesis, we may borrow some classic heuristics to reduce the complexity.

- Non-linear Abstract Timing Model. The abstract timing model we constructed is based on linear delay model. However, in some cases, we may need non-linear delay model to achieve higher accuracy. For example, we may need to construct a two-dimension look-up table for each pair of input and output of the timing model such that we can propagate signal slopes through the hierarchical block. If we want to reduce the number of edges in a non-linear abstract timing model, we need to combine the two-dimension look-up tables attached on edges, which is non-trivial.

Bibliography

[1] Cadence incremental common timing engine, 2005. http://www.cadence.com/products/digital_ic/gps.aspx.

[2] Synopsys static timing analysis, 2006. http://www.synopsys.com/products/analysis/ptsi_ds.html.

[3] K. P Belkhale and A. J. Suess. Timing analysis with known false sub-graphs. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 736–740, 1995.

[4] J. Benkowski, E. Vanden Meersch, L. Claesen, and H. De Man. Efficient algorithms for solving the false path problem in timing verification. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 44–47, 1987.

[5] D Blaauw and T Edwards. Generation of false path free timing graphs for circuit optimization. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 165–170, 1999.

[6] D. Blaauw, R. Panda, and A. Das. Removing user-specified false paths from timing graphs. In *Proc. of the Design Automation Conf.*, pages 270–273, 2000.

[7] H. C. Chen and D. H. C. Du. Path sensitization in critical path problem. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 208–211, 1991.

[8] H. C. Chen and D. H. C. Du. Path sensitization in critical path problem. *IEEE Trans. Computer-Aided Design*, pages 196–207, 1993.

[9] C. K. Cheng, J. Lillis, S. Lin, and N. Chang. *Interconnect Analysis and Synthesis*. John Wiley Sons, first edition, 1999.

[10] F. Chung, M. Garrett, R. Graham, and D. Shallcross. Distance realization problems with applications to internet tomography. http://www.math.ucsd.edu/?fan.

[11] S. Devadas, K. Keutzer, and S. Malik. Computation of floating mode delay in combinational circuits: Theory and algorithm. *IEEE Trans. Computer-Aided Design*, 12:1913–1923, 1993.

[12] S. Devadas, K. Keutzer, S. Malik, and A.Wang. Computation of floating mode delay in combinational circuits: Practice and implementation. *IEEE Trans. Computer-Aided Design*, 12:1924–1936, 1993.

[13] D. H. C. Du, S. H. C. Yen, and S. Ghanta. On the general false path problem in timing analysis. In *Proc. of the Design Automation Conf.*, pages 555–560, 1989.

[14] T. Feder, A. Meyerson, R. Motwani, L. OCallaghan, and R. Panigrahy. Representing graph metrics with fewest edges. In *Proc. of Symp. on Theoretical Aspects of Computer Science*, pages 355–366, 2003.

[15] T. Feder and R. Motwani. Clique partitions, graph compression and speeding up algorithms. In *Proc. of the ACM Symposium on Theory of Computing*, pages 123–133, 1991.

[16] E. Goldberg and A. Saldanha. Timing analysis with implicitly specified false path. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 157–164, 1999.

[17] A. Gupta and D. P. Siewiorek. Automated multi-cycle symbolic timing verification of microprocessor-based designs. In *Proc. of the Design Automation Conf.*, pages 113–119, 1994.

[18] S. L. Hakimi and S. S. Yau. Distance matrix of a graph and its realizability. *Quart. Appl. Math.*, 22:305–317, 1964.

[19] Z. Hasan and M. Ciesielski. Elimination of multi-cycle false paths by state encoding. In *Proc. of the European Design Automation Conf.*, pages 155–159, 1995.

[20] R. B. Hitchcock. Timing verification and timing analysis program. In *Proc. of the Design Automation Conf.*, pages 594–604, 1982.

[21] T.-C. Hu. *Combinatorial Algorithms*. Dover Publication, second edition, April 2002.

[22] S. T. Huang, T. M. Parng, and J. M. Shyu. A polynomial-time heuristic approach to approximate a solution to the false path problem. In *Proc. of the Design Automation Conf.*, pages 118–122, 1993.

[23] M. Hutton, D. Karchmer, B. Archell, and J. Govig. Efficient static timing analysis and applications using edge masks. In *Proc. Int. Symp. on Field-Programmable Gate Arrays*, pages 174–183, 2005.

[24] N. P. Jouppi. Timing analysis for nmos vlsi. In *Proc. of the Design Automation Conf.*, pages 411–418, 1983.

[25] R. Kamikawai, M. Yamada, T. Chiba, K. Furumaya, and Y. Tsuchiya. A critical path delay check system. In *Proc. of the Design Automation Conf.*, pages 118–123, 1981.

[26] R. H. Katz and G. Borriello. *Contemporary Logic Design*. Pearson Prentice Hall, second edition, 2005.

[27] K. Keutzer, S. Malik, and A. Saldhana. Is redundancy necessary to reduce delay. *IEEE Trans. Computer-Aided Design*, 10:427–435, April 1991.

[28] Y. Kukimoto and R. K. Brayton. Hierarchical functional timing analysis. In *Proc. of the Design Automation Conf.*, pages 580–585, 1998.

[29] Y. Kukimoto, W. Gosti, R. K. Brayton, and A. Saldanha. Approximate timing analysis of combinational circuits under xbd0 model. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 176–181, 1997.

[30] W. K. C. Lam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Exact minimum cycle time for finite state machines. In *Proc. of the Design Automation Conf.*, pages 100–105, 1994.

[31] P. C. McGeer and R. K. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *Proc. of the Design Automation Conf.*, pages 561–567, 1989.

[32] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design*. Kluwer Academic Publishers, May 1991.

[33] C.W. Moon, H. Kriplani, and K. P. Belkhale. Timing model extraction of hierarchical blocks by graph reduction. In *Proc. of the Design Automation Conf.*, pages 152–157, 2002.

[34] H. Muller. On edge perfectness and classes of bipartite graphs. *Discrete Math.*, 149:159–187, 1996.

[35] M. Nourani and C. Papachristou. False path exclusion in delay analysis of rtl-based datapath-controller designs. In *Proc. of the European Design Automation Conf.*, pages 336–341, 1996.

[36] J. Orlin. Containment in graph theory: Covering graphs with cliques. *Indag. Math.*, 39:211–218, 1977.

[37] P.Ashar, S.Dey, , and S.Malik. Exploiting multi-cycle false paths in performance optimization. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 510–517, 1992.

[38] D. J. Pilling and H. B. Sun. Computer aided prediction of delays in lsi logic systems.

[39] A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Circuit structure relations to redundancy and delay: the kms algorithm revisited. In *Proc. of the Design Automation Conf.*, pages 245–248, 1992.

[40] A. Saldanha, H. Harkness, P. C. McGeer, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance oprunization using exact sensitization. In *Proc. of the Design Automation Conf.*, pages 425–429, 1994.

[41] T. Sasaki, A. Yamada, T. Aoyama, K Hasegawa, S. Kato, and S. Sato. Hierarchical design verification for large digital systems. In *Proc. of the Design Automation Conf.*, pages 105–112, 1981.

[42] C. Visweswariah and A. R. Conn. Formulation of static circuit optimization with reduced size, degeneracy and redundancy by timing graph manipulation. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 244–251, 1999.

[43] M. A. Wold. Design verification and performance analysis. In *Proc. of the Design Automation Conf.*, pages 264–270, 1978.

[44] H. Yalcin and J. P. Hayes. Hierarchical timing analysis using conditional delays. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 371–377, 1995.

[45] H. Yalcin, J. P. Hayes, and K. A. Sakallah. An approximate timing analysis method for datapath circuits. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 114–118, 1996.

[46] H. Yalcin, M. Mortazavi, R. Palermo, C. Bamji, and K. Sakallah. Functional timing analysis for ip characterization. In *Proc. of the Design Automation Conf.*, pages 731–736, 1999.

[47] H. Yalcin, M. Mortazavi, R. Palermo, C. Bamji, and K. Sakallah. Fast and accurate timing characterization using functional information. *IEEE Trans. Computer-Aided Design*, 20:315–330, 2001.

[48] S. Zhou, B. Yao, H. Chen, Y. Zhu, C.-K. Cheng, M. Hutton, and et al. Improving the efficiency of static timing analysis with false paths. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 527–531, 2005.