NORTHWESTERN UNIVERSITY

Timing Optimization Algorithms for Sequential Circuits

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Chuan Lin

EVANSTON, ILLINOIS

June 2006

ABSTRACT

Timing Optimization Algorithms for Sequential Circuits

Chuan Lin

With the advent of deep sub-micron era (DSM), "System-on-chip (SOC)" has become a mainstream of IC industry. Semiconductor devices based on a smaller feature size offer the promise of faster and more highly integrated designs, but also provide a number of new challenges.

In SOCs, a large amount of communication time is spent on global multi-clock-period interconnects, which present themselves as the main performance limiting factor. How to handle global interconnects for performance optimization becomes an urgent issue. Another challenge is the increasing coupling effect (also known as *crosstalk*) between neighboring interconnects. Besides introducing noises on quiet interconnects, crosstalk could potentially change the interconnect delays and cause timing violations in the circuit. In this dissertation, we investigate and propose solutions to a few problems involving global interconnects and crosstalk.

To handle global interconnects, we propose techniques at different stages of the design flow. At physical layout stage, we propose to pipeline global interconnects by relocating flip-flops (an operation also known as *retiming*). Three efficient algorithms are designed to find

an optimal retiming with the minimal clock period. We then solve the problem of retiming under both setup and hold constraints more efficiently than the best-known algorithm in the literature. We also consider clock skew scheduling for prescribed skew domains and give an optimal polynomial-time algorithm to minimize the clock period with possible delay padding. At clustering stage, we propose an iterative algorithm that finds an optimal clustering with the minimal maximum-cycle-ratio. At register transfer level (RTL), we use delay relaxation to do interconnect planning.

For crosstalk, we propose a circular time representation under which coupling detection is easier and more efficient than state-of-the-art approaches. Using the circular time representation, clock schedule verification with crosstalk is more efficient. We show that the trade-off between a level-sensitive latch and an edge-triggered flop can be leveraged in a sequential circuit design with crosstalk, so that the clock period is minimized by selecting a configuration of mixed latches and flops. We design an effective and efficient algorithm to solve this problem.

To Jie

Acknowledgments

I cannot express enough gratitude to my advisor Professor Hai Zhou for bringing me into the exciting world of VLSI design automation and his tremendous help and encouragement over the years. It is from his personal example as well as verbal instruction that I learned the art of algorithm design, the importance of perseverance in research, the skills of formal mathematical writing, and the insight of problem solving. After four years of practice, I am starting to believe in what he once told me, "A researching life is a beautiful life."

I would also like to thank the members of my dissertation committee, Professors Robert P. Dick, Yehea Ismail, Ming-Yang Kao, Seda O. Memik, and Jorge Nocedal, for their interest and suggestions on my work.

Special thanks are due to Dr. Aiguo Xie for his guidance and patience during my first industry experience. He also helped a lot in Chapter 8.

I also take this opportunity to thank a few teachers, Ms. Guomei Zheng and Ms. Xia Chen from Songtao Elementary School, Mr. Yangui Weng from Longyan's First High School, and Ms. Yanni Liu from Tsinghua University. Their belief in me has been the constant drive of my twenty years of education.

Many thanks go to the members of the NuCAD Research Laboratory, specifically Ruiming Chen, Debasish Das, Nikos Liveris, Debjit Sinha, and Jia Wang, for their discussions and feedback.

I thank all my friends in Evanston for the fun time we had together, Bo Chen, Peng Duan, Gang Hua, Zhe Liu, Min Ni, Yujun Xie, and Bo Yang.

I am highly indebted to my parents and my sister Ying. This dissertation is the result of as much effort (if not more) on their parts than mine.

Last, but not the least, I thank Jie, my wife, for her love and support, without which this dissertation could not have been written.

CHUAN LIN

Northwestern University

June 2006

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Over the years, the rapid advancement of semiconductor technology has been driven by the constant downscaling in the feature size (i.e., the minimum transistor size) of VLSI circuits. The feature size decreased from about $0.35\mu m$ in 1996 to $0.09\mu m$ ($90nm$) today. It is predicted that the feature size will continue to shrink to about $45nm$ in Year 2010 [**86**]. Scaling down of VLSI geometries into deep sub-micron (DSM) dimensions has significant impact on both design methodology and IC design automation.

On one hand, semiconductor devices based on a smaller feature size offer the promise of faster and more highly integrated designs. Decrease in feature size results in reduction in transistor switching delays, which leads to faster signal transition times and higher clock frequencies. In addition, the transistor density on a chip grows quadratically with the rate of decrease in the feature size. There is also a trend of continuous increase in chip sizes. As a combined result, the number of transistors on a single chip has increased from less than 500,000 in 1985 to 190 million today and will reach 800 million in Year 2010 [**86**]. With the huge number of transistors available on a chip, "System-on-chip (SOC)" has become a mainstream of IC industry.

On the other hand, a number of new challenges emerge as technology advances to the DSM era. In the DSM era, the gate-level netlists still implement the functions. However, the results from physical design, such as interconnect delay and coupling effect, are largely determined by the layout of the designs. In a layout, long global interconnects are generally

required to maintain the communication among the huge number of transistors. The delays of the global interconnects are difficult to predict at early design stages. In addition, when interconnects are narrowed to reduce area, their heights are usually scaled at a much lower rate in order to keep the resistance low. As a result, the aspect ratio grows and the coupling capacitance becomes the dominant part of the total capacitance on an interconnect. This, together with the increasing number of interconnects and interconnect layers, makes coupling effect more unpredictable before layout.

Facing these challenges, conventional design methodology and tools break down, as is evidenced by the large number of iterations experienced by designers. One major cause of the large number of iterations has been the failure to meet timing. Since both interconnect delay and coupling effect contribute to timing significantly, it is an urgent necessity to design effective and efficient timing algorithms that address interconnect delay and coupling effect in the whole design flow.

1.1. Interconnect delay

Interconnect delays have dominated gate delays since the late 90's. From then on, when technology trends are investigated, an important distinction is made between two kinds of interconnects. *Local* interconnects connect gates or cells within each block. They become shorter with technology scaling. *Global* interconnects connect blocks together and usually span a significant portion of a chip. They do not shrink when devices become smaller, and even tend to increase with increasing chip sizes.

Unlike local interconnects, global interconnects have significant delays. This is due to the combined effect of decreasing driver resistance to drive the large load and large interconnect resistance due to the long length. Even with interconnect optimization technique such as

buffer insertion, the delay of a global interconnect may still be longer than one clock period, and multiple clock cycles are generally required to communicate such a global signal.

In SOCs, the conflicts between communication and computation will become prominent even on a chip. A big fraction of system time will shift from computation to communication. In sequential circuits, a large amount of communication time is spent on global multi-clock-period interconnects, which present themselves as the main performance limiting factor. In this dissertation, we consider how to handle global interconnects for performance optimization at difference stages of the design flow.

1.2. Coupling effect

Coupling effect (also known as *crosstalk*) refers to the phenomenon of noise induced on an interconnect when one of its neighboring interconnects has a signal switching. It is additive, and proportional not only to the rate of change in the signal waveform but also to the coupling capacitance between the neighboring interconnects. The closer the interconnects, the higher they are from the substrate voltage plane, and the longer they parallel each other, the larger the coupling capacitance they have.

Crosstalk increases delay when signals on coupling interconnects switch simultaneously in opposite directions. However, when signals switch simultaneously in the same direction, crosstalk decreases delay. The delay uncertainty introduced by crosstalk could potentially cause timing violations in the circuit. Therefore, it is necessary to control crosstalk.

Various approaches have been proposed to control crosstalk. Interconnect optimization techniques such as gate sizing and buffer insertion focus on slowing down the waveform change rate by adjusting drive strength and load capacitance. On the other hand, since crosstalk is heavily dependent on the coupling capacitance between interconnects, which in

turn are decided by the physical positions of interconnects, crosstalk-driven routing is another option. Different from these techniques that focus on reducing the magnitude of crosstalk, we consider in this dissertation how to eliminate the crosstalk-induced delay uncertainty by avoiding simultaneous switchings.

1.3. Dissertation overview

We give an overview of the dissertation as follows.

In Chapter 2, we propose to pipeline global interconnects by relocating flip-flops (an operation also known as *retiming*). Behaviorally, it means that both computation and communication are rescheduled for parallelism. The problem of retiming over a netlist of macro-blocks, where the internal structures may not be changed and flip-flops may not be inserted on some wire segments, is formulated as the wire retiming problem. We propose timing models for macro-blocks. Based on these models, a set of integer difference inequalities is used to quantified a feasible clock period, thus gives an algorithm that finds the minimal clock period by binary search.

In Chapter 3, we target the same wire retiming problem. We show that the constraints can be formulated as a fixpoint computation, which enables an iterative algorithm to check the feasibility of a given clock period more efficiently. In conjunction with binary search, it also improves the efficiency of finding the minimal clock period.

In Chapter 4, the same wire retiming problem is addressed from a completely different point of view. Contrary to the previous two algorithms that use binary search to check the feasibility of a range of candidate clock periods, we propose an even more efficient algorithm that directly checks the optimality of the current feasible clock period, and can thus either push down the clock period or certify the optimality. Since the algorithm does not involve

binary search, it is essentially incremental and has the potential of being combined with other optimization techniques.

In Chapter 5, we consider retiming under both setup and hold constraints. Given an edge-triggered sequential circuit $G = (V, E)$, a target clock period, a setup time and a hold time, we give an algorithm that finds a min-period retiming satisfying both setup and hold constraints in $O(|V|^2|E|)$ time. This is a much better result than the best-known algorithm in the literature with $O(|V|^3|E| \lg |V|)$ time complexity.

In Chapter 6, we deal with clock skew scheduling. Clock skews are the differences in clock arrival times among different flip-flops. It was observed in [**35**] that retiming and clock skew scheduling are discrete and continuous optimizations with the same effect. Given a finite set of prescribed skew domains, we propose a polynomial-time algorithm that finds an optimal domain assignment for each flip-flop such that the clock period is minimized with possible delay padding. We then consider how to insert extra paddings such that both setup and hold constraints are satisfied under the minimal clock period. We show that the existence of such a padding solution is guaranteed, and present an approach to find a padding by network flow technique.

In Chapter 7, we consider the effect of clustering on global interconnects. This is motivated by the need of integrating interconnect planning in high level synthesis to reduce the complexity of physical layout stage and to leverage that burden at early stages of the design flow. Clustering, being a stage between logic synthesis and physical design, helps to provide the first order information about global interconnect delays. We identify *processing rate*, the product of frequency and throughput, as the objective of clustering. We show that the maximum processing rate is upper bounded by the reciprocal of the maximum cycle ratio of

the clustered circuit, and propose an iterative algorithm to construct a clustering solution with the minimal maximum-cycle-ratio.

In Chapter 8, we perform interconnect planning at register transfer level (RTL). As we know that although retiming helps to relieve the criticality of global interconnects at physical design stage, it cannot change the total number of flip-flops along a topological cycle. However, it can be changed at RTL by a technique known as *delay relaxation*. Our contribution is twofold. Firstly, we propose a general formulation for design closure driven delay relaxation problem. We also relate the problem to retiming and propose a convex retiming formulation. Secondly, we show that the general formulation can be transformed into a convex cost integer dual network flow problem and solved in polynomial time by the approach in [**2**].

In Chapter 9, we explore the trade-off between a level-sensitive latch and an edge-triggered flop to control the crosstalk effect on interconnect delay. On one hand, latches are extensively used in high-performance sequential circuit designs to achieve high frequencies because of their good performance and time borrowing feature. On the other hand, however, the amount of timing uncertainty due to crosstalk accumulated through latches could be larger than the benefit gained by time borrowing. We show that the trade-off between a latch and a flop can be leveraged in a sequential circuit design with crosstalk, so that the clock period is minimized by selecting a configuration of mixed latches and flops. A circular time representation is proposed to make coupling detection easier and more efficient.

CHAPTER 2

Retiming for Wire Pipelining in System-On-Chip

With a great market drive for high performance and integration, operating frequencies and chip sizes of SOCs are dramatically increasing. Industry data showed that the frequencies of high-performance ICs approximately doubled every process generation and the die size also increased by about 25% per generation. With such short clock periods, the communication among different blocks on a SOC circuit of ever increasing complexity is becoming a bottleneck: even with interconnect optimization techniques such as buffer insertion, the delay from one block to another may be longer than one clock period, and multiple clock cycles are generally required to communicate such a global signal.

This trend has motivated recent research within Intel [**15**] and IBM [**42**] on how to insert flip-flops on a given net if the communication between the pins requires multiple clock cycles. However, inserting flip-flops within a circuit will change its functionality, and inserting arbitrary number of them on a net without considering global consistency will destroy the correctness of a circuit.

Retiming [**53**] is a traditional sequential optimization technique that moves flip-flops within a circuit while keeping its functionality. In traditional settings, retiming was used mainly on block level netlists [**90, 34, 30, 50, 77**]. Although some research incorporated wire delays in retiming [**50, 97, 21**], they did not consider the situation where multiple flip-flops may be on a global interconnect. With increasing communication delays as mentioned above,

this chapter explores the alternative utility of retiming—that is, besides its computational function, a flip-flop can be used to fulfill communication buffering requirements.

Since dominant wire delays can only happen on global wires, we solve the problem at the chip level, that is, the design we deal with is a netlist of macro-blocks. The wires within a block are relatively much shorter thus do not need multiple clock periods for propagation. In SOC design, many of these macro-blocks are IP (Intellectual Property) cores. Some of these blocks may be combinational circuits, and others sequential. In our problem formulation, we will use timing macro-models to model the timing behavior of the blocks. Because of the existence of pre-designed blocks such as IP cores or regular-structured blocks such as memories, (combinational) buffers or flip-flops may not be inserted everywhere [**111**]. We will incorporate buffering position restrictions in our solution.

The rest of this chapter is organized as follows. Section 2.1 introduces timing macro-models for both combinational and sequential blocks, based on which we present the problem formulation. Section 2.2 defines the notations and constraints used in this section, and presents theoretical results on the problem. Our algorithm is elaborated in Section 2.3. Section 2.4 presents the experimental results, followed by a discussion in Section 2.5.

2.1. Problem formulation

We consider wire pipelining via retiming on a SOC design under the assumption that a floorplan of the blocks and a global routing of the global wires are given. This problem may come from different design methodologies and different design stages. For example, it may come from an interconnect planning stage where the floorplan and global routing are done for estimation [**67, 66**], or it may come from a physical design stage where the floorplan and global routing are given. The wire pipelining problem in these two situations is the same

except that we may allow flip-flop insertions in soft blocks in the interconnect planning but only allow such insertions in pre-allocated buffer regions in physical design.

As an illustration, a SOC design with a floorplan and a global routing is depicted in Figure 2.1. Here, we have five blocks with the floorplan and the global routing of global wires. Each wire has an arrow to indicate the signal direction, and a weight to specify how many flip-flops are on the wire. Those with weight 0 have the weight omitted. For example, the wire from u to n has 1 flip-flop, while the wire from n to v has 0 and that from n to w has 2. Some segments of a wire may not accommodate any buffer or flip-flop because they run over macro-blocks that do not allow transistors to be added.

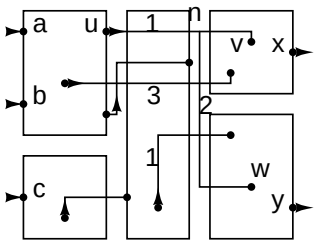

Figure 2.1. Global interconnections on a SOC design.

In order to take the delays within each block into consideration, timing models are used for specifying the timing behavior of each block. Due to the increasing popularity of SOC design and IP-core based design, recently there are increasing research activities on timing models for macro-blocks [**25, 72, 36**]. If a block is a pure combinational circuit such as an ALU or a multiplier, then a minimum and a maximum delay from each input pin to every output pin can be used to characterize the timing behavior of the block. This is a traditional approach. Recent researches are mainly focused on sequential blocks [**25, 72, 36**]. Generally speaking, if there are combinational paths from an input pin to an output pin, a minimum and maximum delay pair will be used to characterize the delay on each path. If an input pin

has a path to a flip-flop in the block, the arrival time of the input pin must be constrained to satisfy the set-up condition. Finally, if an output pin has paths from flip-flops in the block, then the arrival time of the pin is given by the path delays.

Traditional retiming is applied to logic level netlists that are composed of simple gates. In our application, the netlist is composed of macro-blocks. Since a combinational block can be viewed as a complex gate, moving a flip-flop over it is simply justified. Now we will show that retiming can be generalized to sequential blocks.

Lemma 2.1.1. *In a SOC design composed of macro-blocks, a flip-flop can be moved from every input to every output of a block or vice versa without changing the function of the design.*

Proof. The function of the design only depends on the synchronization of the data flows. Moving a flip-flop from every input to every output of a sequential block or vice versa, only changes the specific clock cycle during which the correct results are generated. In other words, the data flows are always synchronized under such flip-flop manipulation. Therefore, the function of the design is always kept. □

However, in order for a retiming procedure to move flip-flops over sequential blocks, the timing model for a sequential block must be a graph that connects its inputs to outputs. The traditional approach of treating sequential inputs as primary outputs and sequential outputs as primary inputs will cut off the connections through the block hence make it impossible to move flip-flops over the sequential block.

We will now consider how flip-flops can be moved over timing models of macro-blocks. When the block is a combinational circuit, as shown in Figure 2.2(a), we can use edges between inputs and outputs to represent the path delays between them, as shown in Figure

2.2(b). If we place these edges in traditional retiming formulation, flip-flops can be moved over them. However, there are two caveats. First, in traditional retiming, flip-flops may be placed on any edge. In order to avoid flip-flops being placed on the edges we introduced in timing models, we require that the retiming tags of the input and the output connected by such an edge be the same. This means that the number of flip-flops moved over the input is the same as the output. Therefore, no flip-flop will be left in between. Second, depending on the structure of the circuit within the block, an input may not have a path to every output. For example, in Figure 2.2(a), inputs a and b do not connect to output y, and input c does not connect to output x. If the macro-block is substituted by the edges in its timing model, then, the number of flip-flops moved over x may be different from that over y. On the other hand, if a block is treated as a super-gate, it is required that the retiming tags of the inputs and the outputs are all the same, and this may give sub-optimal solutions. Since the delay edges represent the path topology of the circuit, our timing model for macro-blocks gives us more flexibility.

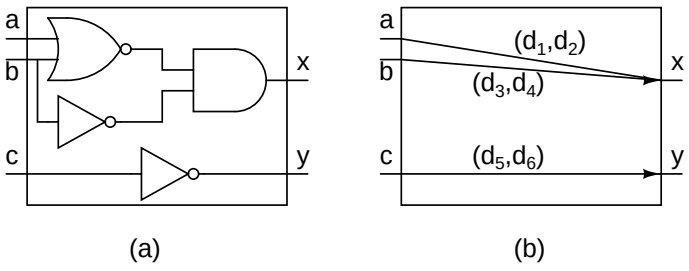

Figure 2.2. A combinational block and its timing model.

The case for sequential blocks is more tricky. To understand how to build a timing model for a sequential block that can be used for retiming, we use the sequential block shown in Figure 2.3(a) as an example. To simplify the presentation, we assume that the delays come only from gates, shown in the figure as d_1, d_2, and d_3. Delays from wires can be similarly

included. Since there is a combinational path of delay $d_1 + d_2$ from a to x, we introduce an edge (a, x) with delay $d_1 + d_2$ in the model. The input a has a combinational path of delay d_1 to the flip-flop f_1. This implies that the arrival time at a should not be larger than $T - d_1$, where T is the clock period. To enforce this set-up condition, we introduce a *virtual flip-flop* in the timing model, as shown in Figure 2.3(b), and add an edge with delay d_1 from a to this virtual flip-flop. Similarly, since the input b has a combinational path to the flip-flop f_2, an edge with delay d_3 is added from b to another virtual flip-flop. The concept of virtual flip-flops is also used to specify the arrival times at the block outputs that are dependent on interior flip-flops. For example, the output x has a combinational path of delay $d_1 + d_2$ from the output of the flip-flop f_1, thus an edge with delay $d_1 + d_2$ is added from a virtual flip-flop to x. Similar model is used for the output y. In Figure 2.3(b), all the virtual flip-flops are combined into one flip-flop. It is done to facilitate retiming on the block, and also to make the model simpler. Later, the virtual flip-flop is modeled as an edge with delay 0 and weight 1.

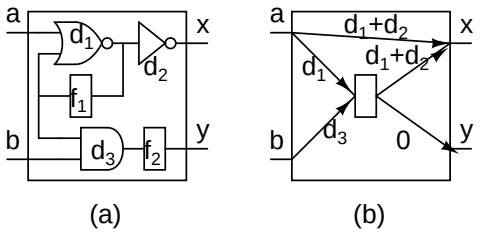

Figure 2.3. A sequential block and its timing model.

We also need to consider the modeling of a net. Since it is assumed that the (global) routing of nets is given, each net is represented as a Steiner tree. Based on routing positions of the wires, some wire segments of the tree may not accommodate buffers or flip-flops. For example, Figure 2.4(a) shows the route of a net with one source and three sinks. The shaded

regions represent the areas where no buffer or flip-flop can be inserted. The timing model used for this net is shown in Figure 2.4(b). Besides the sources and sinks of the net, vertices are created at the points where wires are getting into or out of buffer forbidden areas, and at the Steiner points not within buffer forbidden areas. Similar to combinational paths within a block, a delay edge will be used to represent the wire delay from an entering point to an exiting point through a buffer forbidden area. For example, the edge (u, v) in Figure 2.4(b) represents the wire delay from point u to point v. Applying the timing model, each net becomes a set of edges. Some of the edges, such as (x, y), (u, v), and (u, w) in Figure 2.4(b), do not allow buffers or flip-flops inserted. But others allow such insertions.

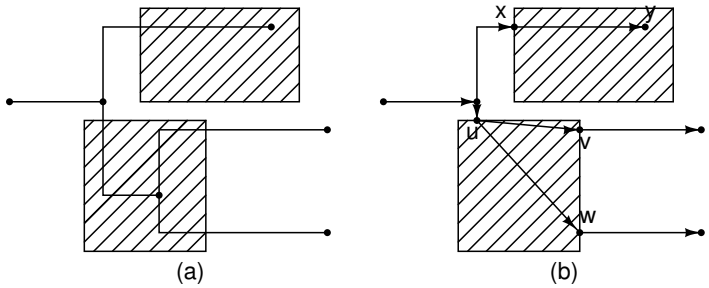

Figure 2.4. The route of a net and its timing model.

Since the timing model of a macro-block (whether it is combinational or sequential) is composed of a set of edges on which no flip-flop can be placed, and the timing model of a net is composed of a set of edges some of which accommodate buffers and flip-flops but others forbid them, after applying these models, our problem is represented as a directed graph with two types of edges: one allows buffer and flip-flop insertions but the other forbids them. According to [**76**], a buffer-allowable wire can be optimally buffered such that its delay becomes linear in terms of its length. Therefore, we assume the delay on a buffer-allowable edge to be linear. A buffer-forbidden edge may represent a combinational path within a block or a wire over a buffer-forbidden region. In the former case, its delay is given by the

pin-to-pin path delay; in the latter case, its delay can be computed as the Elmore delay of the wire under the assumption that it is buffered at the region boundaries.

In summary, in a graph model of the problem, a vertex is used to represent a source or sink of a net, the input or output of a virtual flip-flop, a point where a wire gets into or out of a buffer forbidden area, or a Steiner point outside buffer forbidden areas. A directed edge is used to represent a fan-out relation within a block or a wire connection outside buffer forbidden areas. On an edge representing a fan-out relation, the delay is either a non-negative constant, and no flip-flop change is allowed on it. On the other hand, flip-flops can be inserted on an edge representing a wire, and the delay of a segment is positive and proportional to its length. This graph model is very general, since it can also be used to represent flip-flop blockages over wires. In that case, wire segments over blockages can be represented as edges without flip-flop insertion. Based on this unified model, the problem we want to solve can be formulated as follows.

Problem 2.1.1 (Minimum Period Wire Retiming)**.**
Given a directed graph $G = (V, E)$ with two types of edges, buffer-forbidden edges E_1 and buffer-allowable edges E_2 ($E = E_1 \cup E_2$), where each edge $e \in E$ has a delay $d(e)$ and a weight $w(e)$ (representing number of flip-flops on it), find a retiming—i.e. a relocation of flip-flops in the graph—such that: 1. there is no flip-flop change on any edge $e \in E_1$; 2. the delay between two flip-flops on an edge $e \in E_2$ is linear in terms of their distance; 3. the clock period (i.e. the maximum delay between any two consecutive flip-flops, treating primary inputs (PIs) and primary outputs (POs) as flip-flops) is minimized.

Problem 2.1.1 wants to find a retiming solution to minimize the clock period. A reasonable step to solve this problem is to consider whether we can find a retiming to satisfy a given clock period. Such a problem is defined as follows.

Problem 2.1.2 (Fixed Period Wire Retiming)**.**
Given a clock period T and a directed graph $G = (V, E)$ with two types of edges, buffer-forbidden edges E_1 and buffer-allowable edges E_2 ($E = E_1 \cup E_2$), where each edge $e \in E$ has a delay $d(e)$ and a weight $w(e)$ (representing number of flip-flops on it), find a retiming—i.e. a relocation of flip-flops in the graph—such that: 1. there is no flip-flop change on any edge $e \in E_1$; 2. the delay between two flip-flops on an edge $e \in E_2$ is linear in terms of their distance; 3. the maximum delay between any two consecutive flip-flops (including PIs and POs) is no larger than T.

If we can solve this problem, Problem 2.1.1 can be solved by using a binary search. Furthermore, this problem is applicable if a designer only wants to target a fixed clock period.

2.2. Theoretical results

2.2.1. Notations and constraints

Before discussing the solutions to the two problems, we will first select some essential notations to help us to clearly state the requirements for a solution.

From the formulations of the problems, we already have a delay $d(u, v)$ and a weight $w(u, v)$, for each edge $(u, v) \in E$. We will follow the convention of Leiserson and Saxe [**53**] to use an integer variable $r(u)$ to represent the number of flip-flops moved from the outgoing edges of a vertex u to its incoming edges. In [**53**], these variables were sufficient to specify

a retiming solution since there was no wire delay and a solution only needed to tell whether a flip-flop was on a wire. In fact, it is not the absolute values of these variables but their differences that are important, since the number of flip-flops on a given edge (u, v) after retiming is given by

$$w(u, v) + r(v) - r(u).$$

However, in our problem, a retiming solution must include the positions of flip-flops on each wire. Because retiming can change the number of flip-flops in the system, it is not even known how many flip-flops there will be after retiming. Fortunately, we can overcome the problem by only specifying the arrival time of every vertex with respect to a clock period. For each vertex $u \in V$, we use $t(u)$ to represent its arrival time with respect to the nearest flip-flop on its incoming paths. For example in Figure 2.5, $t(u)$ is the maximum of the delays of the bold paths incident to u. Given $t(u)$, the positions of flip-flops directly fanning into u can be found, and the positions of other flip-flops can also be computed.

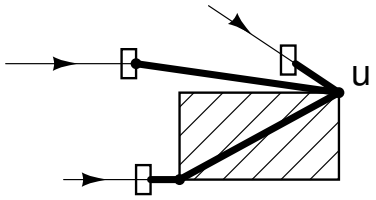

Figure 2.5. Illustration of $t(u)$.

Using these notations, the requirements for a retiming solution can be stated as follows. First, to ensure that no flip-flop is inserted on forbidden edges, we need

$$r(u) = r(v), \quad \forall (u, v) \in E_1. \tag{2.1}$$

Then, to make sure the availability of flip-flops, it must be true that

$$w(u,v) + r(v) - r(u) \geq 0, \quad \forall (u,v) \in E_2. \tag{2.2}$$

The following inequality will guarantee that the arrival times are all achievable.

$$t(v) \geq t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T, \quad \forall (u,v) \in E. \tag{2.3}$$

Finally, the set-up conditions at the flip-flop inputs are equivalently stated in the following inequalities.

$$0 \leq t(u) \leq T, \quad \forall u \in V. \tag{2.4}$$

For any path $p \in G$, we use $d(p)$ to denote the delay along p, which is the sum of the delays of p's constituent edges. Similarly, $w(p)$ denotes the number of flip-flops on p before retiming, which is the sum of the weights of p's constituent edges. When a path actually forms a cycle c, $d(c)$ $\big(w(c)\big)$ includes the delay (weight) of each edge in the cycle only once. Since retiming will not change the number of flip-flops in a cycle, $w(c)$ is independent of retiming. To avoid disturbing issues regarding zero weight cycles, we assume in this section that $w(c) > 0$ for all $c \in G$.

Note that the arrival times are computed based on the retimed graph. Edge (u,v) is called *a critical edge* if $t(v) = t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T$. Likewise, *critical paths* and *critical cycles* can be similarly defined. A solution (r,t,T) that satisfies (2.1)-(2.4) is called *a feasible solution*; the T is called *a feasible clock period*.

A vertex M is introduced into G, along with directed forbidden edges from each PO to it with zero delays and weights, and from it to each PI with delay zero but weight one. These forbidden edges ensure that if flip-flops are moved outside the circuit through PIs (POs),

they will be moved back into the circuit through POs (PIs). As a result, if $r(v)$, $\forall v \in V$, is a valid retiming, then $r(v)+K$, $\forall v \in V$, is also a valid retiming, for any $K \in Z$. On the other hand, due to the weight of one assigned on the forbidden edges to PIs, all PI$\rightsquigarrow$PO paths are transformed into cycles with positive weights, and thus will not contradict the assumption on positive cycle weights. Moreover, given that the arrival time $t(M)$ of vertex M can also be quantified by (2.3)-(2.4), we shall not differentiate M and the forbidden edges incident to it from other vertices and edges in the sequel.

We shall also clarify that the introduction of the vertex M is not necessarily required in our algorithm. In cases where flip-flops cannot be moved through PIs/POs due to the restrictions on the initial state [**32, 34, 95**], M will not be introduced. Our proposed algorithm is guaranteed to work as long as an optimal retiming can be reached from a given flip-flop configuration (may or may not be modified from the original circuit) by moving flip-flops in only one direction toward either the POs or the PIs.

2.2.2. Lower and upper bounds of clock period

From conditions (2.1)-(2.4), it is easy to get some lower bounds and upper bounds of the feasible clock periods. First, it is easy to see that any feasible clock period T must satisfy

$$T \geq T_1 \stackrel{\triangle}{=} \max_{path\ p \in E_1,\ w(p)=0} d(p).$$

By applying (2.3) on any cycle, we have the following lemma.

Lemma 2.2.1. *A feasible clock period T must satisfy*

$$T \ \geq \ T_2 \stackrel{\triangle}{=} \max_{c \in cycle} \frac{d(c)}{w(c)}, \tag{2.5}$$

Proof. Since T is feasible, in any cycle c, the delay between any two consecutive flip-flops must be no greater than T in order to satisfy the set-up conditions. Thus, $w(c)T \geq d(c)$ for any cycle c, i.e., $T \geq \max_{c \in \text{cycle}} \frac{d(c)}{w(c)}$. □

It has been shown in the literature [**77, 69, 21**] that when the forbidden edges are introduced only by gates, the above lower bounds are tight. In fact, we can generalize the result to the following lemma.

Lemma 2.2.2. *If each connected component in the subgraph $G_1 = (V, E_1)$ is a complete bipartite graph, the optimal clock period can be upper bounded by $T_1 + T_2$.*

Proof. If we ignore the flip-flop restrictions on all forbidden edges, we can construct a feasible solution with clock period T_2 because all the edges can accommodate flip-flops now. Together with Lemma 2.2.1, T_2 is then the optimal solution. However in reality, finding the optimal solution is much harder due to the flip-flop restrictions. But there is an easy way to construct a feasible solution by first solving the problem without considering flip-flop restrictions and then moving all the flip-flops on each forbidden edge out. If this can be fulfilled, the clock period of such feasible retiming solution will be at most T_1 larger than T_2, which is $T_1 + T_2$. What remains is to prove that such local adjustment of moving flip-flops out can always be fulfilled in a complete bipartite graph.

For the seek of contradiction, we assume that there exists a forbidden edge on which a flip-flop can not be moved out either through the input nor the output of the edge. The situation is shown in Figure 2.6, in which all the edges are forbidden edges.

Without loss of generality, (I_1, O_2) and (I_2, O_1) are two edges with no flip-flop assigned, which is actually the reason why the flip-flop on (I_1, O_1) can not be moved out. We now have $r(O_1) - r(I_1) = 1, r(O_2) = r(I_1)$ and $r(O_1) = r(I_2)$. These three equations lead to

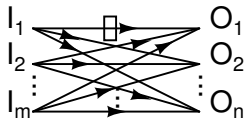

Figure 2.6. A flip-flop in a complete bipartite graph.

$r(I_2) - r(O_2) = 1$. However, it is impossible for a forbidden edge (I_2, O_2) to have $r(I_2) > r(O_2)$, which concludes our proof. □

Since a gate (with only one output) only gives a directed tree with forbidden edges, Lemma 2.2.2 subsumes previous results [**77, 69, 21**]. However, our result also shows that it can be extended to circuits with complex blocks such as multipliers and adders where each output depends on all inputs. Fortunately, the forbidden edges introduced by a net (as shown in Figure 2.4) are always a directed forest, thus will not give any trouble.

When the topology of forbidden edges does not satisfy the condition in Lemma 2.2.2, the optimal clock period may not be upper bounded by $T_1 + T_2$. As an example, consider a circuit shown in Figure 2.7(a), where the forbidden edges within the macro-block do not form a set of complete bipartite graphs. Suppose the delay and weight of each edge are given as the (delay, weight) label shown in the figure, we have $T_1 = 1$ and $T_2 = 34$. However, $T_1 + T_2 = 35$ is no longer an upper bound of the optimal clock period, since a period of 34 requires a retiming as shown in Figure 2.7(b), from which a feasible solution cannot be derived by moving only the flip-flops within the block.

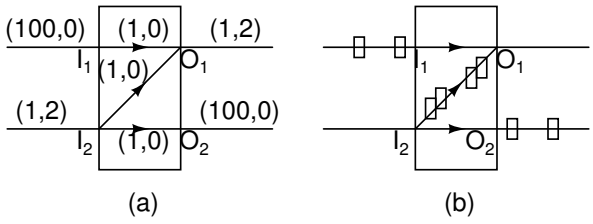

Figure 2.7. A non-complete bipartite block.

Cong and Yuan [**22**], in a study of multilevel placement and retiming, had the same problem with multiple-output clusters. To find an upper bound, they treated each cluster as a vertex which is equivalent to connecting all outputs of a cluster to a super output with edges of zero delay. Used on the circuit shown in Figure 2.7(a), their approach will add edges from O_1 and O_2 to a super output and thus the upper bound will be 202. As we can see from this example, even though their approach gives a correct upper bound, the bound is loose. Without any retiming, Figure 2.7(a) gives a much better upper bound of 101.

Our approach to find a tighter upper bound is as follows. First, we find an optimal retiming solution without considering forbidden edges. This can be done based on the computation of T_2. Then a local adjustment to move flip-flops out of forbidden edges is done to get a feasible solution. The objective in this step is to keep the increase of the clock period as small as possible. From any set of forbidden edges that form a complete bipartite graph, any flip-flop can be moved out with at most an increase of T_1 to the period. To move a flip-flop out of a forbidden edge in a non-complete bipartite graph, other flip-flops may be moved over the block, thus the increase of the period could be larger. However, the local adjustment will keep the number of flip-flops moved out of a non-forbidden edge as small as possible. In the example in Figure 2.7(a), an optimal retiming without considering forbidden edges is shown in Figure 2.7(b) and has a period of 34. The local adjustment will move two flip-flops out from O_1 and two from I_2. Therefore, our upper bound will be $3 \times 34 = 102$.

2.2.3. Fixed period wire retiming

From now on, we will consider how to check whether a clock period T that satisfies the above lower bounds can be realized by retiming. We will use an approach similar to Leiserson and Saxe [**53**] to solve this problem in polynomial time.

For any path p in the graph, we define the sequential delay $\mathrm{sd}(p)$ as follows

$$\mathrm{sd}(p) = d(p) - w(p)T$$

where T is the clock period. And we define the sequential delay $\mathrm{sd}(u, v)$ of any two vertices u, v to be

$$\mathrm{sd}(u, v) = \max_{p \in u \rightsquigarrow v} \mathrm{sd}(p).$$

Then, consider the following set of inequalities

$$r(v) - r(u) \geq \lceil \mathrm{sd}(u, v)/T \rceil - 1 \quad \forall u \neq v \in V \tag{2.6}$$

Using the definition of the sequential delay, for any edge $(u, v) \in E_2$, (2.6) means that

$$w(u, v) + r(v) - r(u) \geq \lceil d(p)/T \rceil - 1 \geq 0.$$

Thus, (2.6) implies (2.2). However, the importance of (2.6) is more than that. Under the lower bound condition of (2.5), formula (2.1) and (2.6) give a solution to the fixed period wire retiming problem.

Theorem 2.2.1. *The fixed period wire retiming problem is feasible if and only if (2.1), (2.5) and (2.6) have a solution.*

Proof. Suppose T is a feasible clock period, then considering any two vertices u, v, any path p between them would have $w(p) + r(v) - r(u)$ flip-flops. To make sure that the delay between any two consecutive flip-flops is not larger than T, we must have

$$d(p) \leq \big(w(p) + r(v) - r(u) + 1\big)T$$

which is $r(v)-r(u) \geq \big(d(p)-w(p)T\big)/T-1$, for any path p, or $r(v)-r(u) \geq \lceil \mathrm{sd}(u,v)/T \rceil - 1$.

The other direction is more difficult, since besides the number of flip-flops on each edge we also need to find the positions for those flip-flops such that the delay between any two consecutive ones is upper bounded by the clock period. In other words, we need to prove that there exists a solution for (2.3)-(2.4) determined by the solution of (2.1), (2.5) and (2.6).

Since (2.5) is true, once (2.1) and (2.6) have a solution, we can always compute a solution for all $t(u)$ satisfying (2.3) and $t(u) \geq 0$ by applying Bellman-Ford's algorithm [**23**], because it will not report a positive cycle under (2.5). We will have $t(u) = 0$ for every PI u. In the meanwhile, for any $v \in V$ whose $t(v) > 0$, there must exist at least one u such that $(u,v) \in E$ and $t(v) - t(u) = \mathrm{sd}(u,v) - \big(r(v) - r(u)\big)T$. We now prove that such solution also satisfies the requirement $t(u) \leq T$ for all $u \in V$.

For the seek of a contradiction, we assume there exists such a vertex v that $t(v) > T$. Starting from v we trace back along those critical edges (u,v) such that $t(v) - t(u) = \mathrm{sd}(u,v) - \big(r(v) - r(u)\big)T$ until we reach a vertex u whose $t(u) = 0$. In the worst case, u is a PI. Now for $t(u)$ and $t(v)$, we have

$$t(v) - t(u) = t(v) = \mathrm{sd}(u,v) - \big(r(v) - r(u)\big)T.$$

From (2.6), we know that

$$\mathrm{sd}(u,v) - \big(r(v) - r(u)\big)T \leq \mathrm{sd}(u,v) - \lceil \mathrm{sd}(u,v)/T \rceil \, T + T \leq T.$$

Hence $t(v) \leq T$, which contradicts our assumption. □

2.3. Algorithm

2.3.1. Detailed description

In this section, we give an algorithm (Figure 2.8) to solve both the fixed and minimum period wire retiming problems. The operation of our algorithm can be broken down to three main steps.

Algorithm Retiming for Wire Pipelining
Input: A graph representation $G = (V, E)$.
Output: A retiming with minimal clock period.

Find the upper bound T_u and lower bound T_l;
Create rG;
do
  $T = \frac{T_u + T_l}{2}$;
  Find $\mathtt{sd}(u, v) = \max_{p \in u \rightsquigarrow v} \mathtt{sd}(p)$ by Johnson's;
  For each $u \overset{p}{\rightsquigarrow} v$ do
    Update (u, v) in rG with weight $\left\lceil \frac{\mathtt{sd}(u,v)}{T} \right\rceil - 1$;
  Apply Bellman-Ford's to rG;
  If T is feasible then
    Record $r(u)$ for all $u \in V$;
    $T_u = T$;
  Else
    $T_l = T$;
while $T_u - T_l > \epsilon$;
Use T and $r(u)$ to compute $t(u), \forall u \in V$.

Figure 2.8. Pseudocode of the retiming algorithm.

In the first step, we apply an efficient algorithm to find the lower bound of the feasible clock periods. The formula of T_2 actually reflects a cycle property of the graph also known as *maximum cycle ratio* (MCR). Let $d(c)$, $w(c)$ denote two parameters associated with each

cycle c in G. The maximum cycle ratio ρ^* is then defined as

$$\rho^* = \max_c \frac{d(c)}{w(c)}, w(c) > 0.$$

In our application, $d(c)$ is the cycle delay and $w(c)$ is the number of flip-flops in the cycle. Then the value of ρ^* is exactly what we want for T_2. To compute MCR, a lot of algorithms have been designed and presented such as Burns's [**6**], Lawler's [**51**], Howard's [**16, 28**], etc. In terms of complexity, Burns's takes $O(|V||E|)$, Lawler's takes $O\big(|V||E|\lg(|V|W)\big)$, where W is the maximum edge delay, and Howard's takes $O(N|E|)$, where N is the product of the out-degrees of all the vertices in G. In [**28**], some popular algorithms that were widely used in CAD community were systematically compared and their comprehensive experimental results revealed that Howard's algorithm was by far the fastest algorithm though the only known bound of its running time is exponential. In our implementation, we adopt an improved version of Howard's algorithm [**16, 28**]. After T_2 is obtained, we compute a tight upper bound of the feasible clock periods in the way as described in section 2.2.2.

In the second step, a binary search is used to find the optimal clock period. Given a particular T, we need to apply Johnson's all-pair shortest path algorithm [**23**] first to compute all $\mathrm{sd}(u,v) = \max_{p \in u \rightsquigarrow v} \mathrm{sd}(p)$. Based on the results, we create a new graph rG incorporating all the vertices in V and edges (u,v) with weight $\lceil \mathrm{sd}(u,v)/T \rceil - 1$ if there exists a path from u to v in G, for all $u \neq v \in V$. Then we apply Bellman-Ford's algorithm to check if there exists a positive cycle in rG. When it terminates, we can decide how to adjust the two bounds accordingly. Note that once rG was created, its structure was kept throughout the rest of the algorithm and its edge weights were recomputed and updated every time T was changed.

In the third and last step, we use the optimal clock period and corresponding $r(u)$ computed in step two to calculate $t(u)$ for all $u \in V$. Due to Theorem 2.2.1, a feasible solution for $t(u)$ is guaranteed.

2.3.2. Pruning and optimization

Further examining (2.6), we found that some inequalities are actually redundant. For example, if $(u, v) \in E_2$, $(v, y) \in E_1$ and $w(v, y) = 0$, according to (2.6),

$$r(y) - r(u) \geq \lceil \mathrm{sd}(u, y)/T \rceil - 1 \geq \lceil \big(\mathrm{sd}(u, v) + d(v, y)\big)/T \rceil - 1 \geq \lceil \mathrm{sd}(u, v)/T \rceil - 1.$$

Since $r(v) = r(y)$, the inequality above actually implies $r(v) - r(u) \geq \lceil \mathrm{sd}(u, v)/T \rceil - 1$ which then becomes redundant. Generally speaking, we call an inequality in (2.6) a redundant inequality if it can be implied by other inequalities in (2.6) that have not yet been proved to be redundant.

To distinguish from the graph G, we call rG *retiming* graph and its edges *retiming* edges. We now present two pruning techniques which could help to reduce the redundancy of (2.6).

First, if there exist three distinct vertices u, v, y such that

$$\lceil \mathrm{sd}(u, y)/T \rceil - 1 + \lceil \mathrm{sd}(y, v)/T \rceil - 1 = \lceil \mathrm{sd}(u, v)/T \rceil - 1,$$

and either of $r(y) - r(u) \geq \lceil \mathrm{sd}(u, y)/T \rceil - 1$ and $r(v) - r(y) \geq \lceil \mathrm{sd}(y, v)/T \rceil - 1$ has not yet been proved to be redundant, then the inequality $r(v) - r(u) \geq \lceil \mathrm{sd}(u, v)/T \rceil - 1$ is redundant by definition and can be safely deleted.

Second, for the forbidden edges with no virtual flip-flop on them, all retiming edges from the ending points are redundant since their effects are implied by the retiming edges from the starting points. For the same reason, all retiming edges to the starting points of those forbidden edges are redundant too. By deleting the redundant inequalities, we can reduce the number of retiming edges dramatically. Although we did not improve the asymptotic complexity, we did get great benefit in reducing the running time in reality.

Using the two techniques above, the size of rG may be much smaller but we still have to run Bellman-Ford's algorithm during each test of the binary search. Since the outcome of Bellman-Ford's algorithm gives not only feasibility answer but all corresponding $r(u)$ values which are more than what we need during most of the tests. Instead, we can take advantage of the high efficiency of Howard's algorithm to calculate the maximum cycle ratio of rG. If during the calculation, it discovers a cycle with positive weight, current test is immediately finished. Hence, a second scheme to substitute the Bellman-Ford's algorithm is to use Howard's algorithm during the process of optimal period searching. After that, only one pass of Bellman-Ford's algorithm is needed to obtain those corresponding $r(u)$ values.

2.3.3. Computational complexity

In [**27**], the running time of Howard's algorithm is bounded by $O\big(|V|^2|E|(\omega_{max}-\omega_{min})/\epsilon\big)$ and $O(|V||E|\alpha)$, where α is the number of simple cycles in G, ω_{max} and ω_{min} are the maximum and minimum edge delays and ϵ is the precision. While in practice, using Howard's to get the lower bound takes much less time than doing the binary search.

Johnson's all-pair shortest path algorithm uses the Bellman-Ford's algorithm and Dijkstra's algorithm [**23**] as subroutines. Since we want to find the maximum sequential delay between any (u, v) pair if v is reachable through some path from u, we actually run Johnson's

algorithm to solve a longest path problem. Its running time is $O(|V|^2 \lg |V| + |V||E|)$ if the priority queue in Dijkstra's algorithm is implemented by a Fibonacci heap [**23**].

As expected, the dominant part of running time is consumed in applying Bellman-Ford's algorithm to rG. In the worst case, rG is a complete graph and consists of $|V|^2$ number of edges. Solving this will take $O(|V|^3)$ time. Therefore, the total running time inside the binary search loop is $O(|V|^3)$ and the time complexity for the whole algorithm is $O(|V|^3 \lg \frac{T_u - T_l}{\epsilon})$.

2.4. Experimental results

We implemented the algorithm in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. We performed retiming on the ISCAS-89 benchmark suite. In absence of delay information for ISCAS-89 circuits, we randomly assign delay values between 1.0 and 2.0 units to gates (we treat them as macro-blocks) and 0.2 to 5.0 to wires. In terms of the chip level we are focusing on, the delay range is intentionally chosen in order for the wire delay to be commensurate or even many times larger than the block delay.

In practical implementation, we found that the benefit we got using the first pruning technique presented in section 2.3.2 did not actually pay off the time penalty. The $\theta(|V|^3)$ complexity for the first technique is tight hence becomes the dominant part of the total running time. Therefore, we only apply the second pruning technique and compare the running time of the two schemes mentioned in section 2.3.2. Experimental results show that the performances of the two schemes are in the same level. Thus, we only report the running time of the second scheme in Table 2.1, where column "$|V|$" and "$|E|$" are the number of vertices and edges respectively. The lower bound T_2 of each circuit is also reported as a comparison with the optimal clock period we computed. Column "#Iter" lists the number

of binary search iterations. Column "t(sec)" lists the running time in seconds. Column "T^*" lists the minimal clock period found by the algorithm.

Table 2.1. Experimental Results

Circuit	$\|V\|$	$\|E\|$	T_2	w/o non-CB blocks			w/ non-CB blocks			
				#Iter	t(sec)	T^*	#Part	#Iter	t(sec)	T^*
myex	21	24	13.0	7	0.00	18.7	2	10	0.01	24.0
s386	519	700	51.0	5	1.97	51.1	50	10	3.67	55.0
s400	511	665	32.2	6	1.64	32.2	50	10	3.38	50.6
s444	557	725	35.0	5	2.23	35.2	40	10	4.31	63.2
s838	1299	1206	76.0	5	8.79	76.0	130	11	33.42	84.0
s953	1183	1515	60.6	5	9.76	60.6	110	10	17.56	69.5
s1238	1581	2100	100.2	5	7.88	100.3	150	11	28.45	100.3
s1488	2054	2780	70.1	5	35.17	70.6	200	11	98.88	73.3
s1494	2054	2792	76.8	5	34.13	76.9	160	11	62.86	80.0
s5378	7205	8603	111.0	5	684.60	111.2	500	13	1344.74	115.3

To further test the cases with non-complete bipartite ("non-CB" in Table 2.1) blocks, we apply hMETIS [**48**] to partition a circuit into groups. All edges inside a group are then treated as forbidden edges. For simplicity, we did not further apply our timing model to the partitions when generating the results. The number of partitions of a circuit, which is denoted as "#Part" in Table 2.1, plays an important role in determining the percentage of non-complete bipartite blocks. To better reflect the influence of non-complete bipartite blocks on the optimal clock period, we intensively choose these numbers in order for the resultant difference to be significant. Note that, although we did not change the topology of the circuit after partitioning, we have forced the type of the edges within a group to E_1. The consequent configuration of edges has little to do with that before partitioning. We report them in one table only to share the basic circuit information, such as $|V|$, $|E|$ and T_2 for clear comparisons.

2.5. Discussion

As stated in Section 2.1, the wire retiming problem is formulated at an abstract level that it may be used at different design stages such as interconnect planning and physical design. When flip-flops are relocated in the retiming process, there is an issue of how they will be accommodated in the given placement. A simple solution may allow the floorplan to be modified after the wire retiming stage if there is no enough space for flip-flops. Interconnect planning renders more flexibility to this approach, but we should in general avoid the iterations between floorplanning and retiming. A better approach will estimate and allocate buffer regions during floorplanning [**19**], and then treat only the buffer regions as buffer-enable areas. Furthermore, if we assume that the buffers (not the flip-flops) on the long wires are already placed, then when a flip-flop is moved to a wire, we can replace the closest buffer with the flip-flop. This will alleviate the impact of the relocated flip-flops on the area.

A few sentences may deem to be necessary for the linear delay model of the buffer-enable edges and for our above suggestion to substitute the closest buffer by the flip-flop. By assuming a continuous number of buffers and buffer sizes, [**76**] showed that the delay of a wire could be made linear in terms of its length. However, even when the buffer number is an integer and the size is fixed, the delay of a wire can still be bounded by a linear function of its length. The difference between the model and the "real" delay is at most the delay of one buffer. Since only very long global interconnects need to be wire pipelined, the difference of at most one buffer delay is negligible.

CHAPTER 3

Wire Retiming as Fixpoint Computation

The previous chapter presents an algorithm that solves the fixed period wire retiming problem in polynomial time. However, the time complexity of the algorithm is high, making it inhibitive for large circuits. To overcome this, we present another algorithm in this chapter that uses the iterative method to check the feasibility of a given period with great efficiency.

The rest of this chapter is organized as follows. Section 3.1 introduces a few more notations and rewrites the constraints in a more compact form. Section 3.2 establishes the equivalence between the conditions of a retiming solution and a fixpoint computation. Section 3.3 presents the algorithm. Experimental results are given in Section 3.4. Conclusions are given in Section 3.5.

3.1. Notations and constraints

Besides the notations we already defined in Section 2.2.1, we define $\mathrm{fd}(u)$ and $\mathrm{bd}(u)$ as

$$\mathrm{fd}(u) = \max_{\overset{p}{\rightsquigarrow} u, \forall (x,y) \in p, (x,y) \in E_1, d(x,y) > 0} d(p), \ \forall u \in V,$$

$$\mathrm{bd}(u) = \max_{u \overset{p}{\rightsquigarrow}, \forall (x,y) \in p, (x,y) \in E_1, d(x,y) > 0} d(p), \ \forall u \in V,$$

In other words, $\mathrm{fd}(u)$ is the delay from u's farthest preceding flip-flop to u through forbidden edges, and $\mathrm{bd}(u)$ is the delay from u to its farthest succeeding flip-flop through forbidden edges. Their physical meanings are illustrated in Figure 3.1, where $\mathrm{fd}(u)$ and $\mathrm{bd}(v)$ are the

maximum of the delays of the bold paths incident to u and v, respectively. Intuitively, they specify a range $[\mathrm{fd}(u), T - \mathrm{bd}(u)]$ for each vertex $u \in V$ such that the circuit can operate under T only if the arrival time $t(u)$ is within this range. In case there are no forbidden edges terminating at (originating from) u, we set $\mathrm{fd}(u) = 0$ ($\mathrm{bd}(u) = 0$).

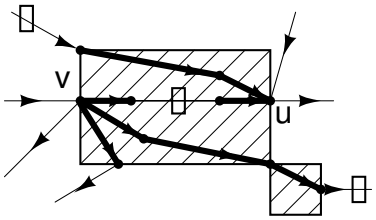

Figure 3.1. Illustration of fd and bd.

The requirements (2.1)-(2.4) for a retiming solution can be equivalently stated as follows.

$$0 \le \mathrm{fd}(u) \le t(u) \le T - \mathrm{bd}(u) \le T, \quad \forall u \in V, \tag{3.1}$$

$$r(u) = r(v), \quad \forall (u, v) \in E_1, \tag{3.2}$$

$$t(v) \ge t(u) + d(u, v) - \big(w(u, v) + r(v) - r(u)\big)T, \quad \forall (u, v) \in E. \tag{3.3}$$

Formula (3.3) actually implies the legality of a retiming, i.e., non-negative edge weights on $(u, v) \in E_2$ after retiming:

$$\begin{aligned}
& (3.3) \\
\overset{T>0}{\Longrightarrow}\ & w(u, v) + r(v) - r(u) \ge \big(t(u) + d(u, v) - t(v)\big)/T \\
\overset{(3.1)}{\Longrightarrow}\ & w(u, v) + r(v) - r(u) > \big(\mathrm{fd}(u) + 0 - T\big)/T \ge -1 \\
\Longrightarrow\ & w(u, v) + r(v) - r(u) \ge 0 \qquad .
\end{aligned}$$

Furthermore, we can establish the following key theorem.

Theorem 3.1.1. *(3.1)-(3.3) have a solution if and only if the following equation (3.4) has a solution for all $v \in V$,*

$$\big(r(v), t(v)\big) = \begin{cases} \big(r_1(v), t_1(v)\big) & \text{if } t_1(v) \le T - bd(v) \\ \big(r_1(v)+1, fd(v)\big) & \text{otherwise} \end{cases} \tag{3.4}$$

where

$$\begin{aligned} r_1(v) &= \max\Big(\max_{\forall(u,v) or (v,u)\in E_1, r(v)<r(u)} r(u), \\ &\qquad \max_{\forall(u,v)\in E_2} \left\lceil \frac{t(u)+d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1\Big), \end{aligned} \tag{3.5}$$

$$\begin{aligned} t_1(v) &= \max\Big(fd(v), \\ &\qquad \max_{\forall(u,v)\in E} t(u) + d(u,v) - \big(w(u,v) + r_1(v) - r(u)\big)T\Big). \end{aligned} \tag{3.6}$$

Proof. ($\leftarrow$): We want to show that if (3.4) has a solution, the solution also satisfies (3.1)-(3.3).

First of all, $t(v)$ equals either $t_1(v)$ or $\mathrm{fd}(v)$ by (3.4), $\forall v \in V$. The latter one establishes (3.1). The former one happens only when $t_1(v) \le T - \mathrm{bd}(v)$, which, together with $t_1(v) \ge \mathrm{fd}(v)$ by (3.6), also establishes (3.1). Therefore the solution satisfies (3.1). In fact, (3.2) is also satisfied, otherwise there exists an edge $(x, y) \in E_1$ such that $r(x) \neq r(y)$. For the case of $r(x) > r(y)$, we have $r_1(y) \ge r(x)$ by (3.5), thus $r_1(y) > r(y)$, which contradicts the fact that $r(y) \ge r_1(y)$ by (3.4). A contradiction can be similarly derived for the case of $r(x) < r(y)$. What remains is to show that the solution also satisfies (3.3).

To this aim, we first rewrite (3.6) as

$$t_1(v) = \max\big(\mathrm{fd}(v), \max_{\forall(u,v)\in E} s(u,v)\big),$$

where $s(u,v) = t(u) + d(u,v) - \big(w(u,v) + r_1(v) - r(u)\big)T$. For $(u,v) \in E_1$, given the facts that $w(u,v) \geq 0$, $r_1(v) \geq r(u)$ by (3.5), $d(u,v) \leq \mathrm{bd}(u)$ by the definition of $\mathrm{bd}(u)$, and $t(u) + \mathrm{bd}(u) \leq T$ by (3.1), we have

$$\begin{aligned} s(u,v) &\leq t(u) + d(u,v) - \big(0 + r_1(v) - r(u)\big)T \\ &\leq t(u) + d(u,v) \leq t(u) + \mathrm{bd}(u) \leq T \end{aligned}$$

For $(u,v) \in E_2$, by (3.5),

$$\begin{aligned} & r_1(v) \geq \left\lceil \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1 \\ \Longrightarrow\ & r_1(v) \geq \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) - 1 \\ \overset{T>0}{\Longrightarrow}\ & t(u) + d(u,v) - \big(w(u,v) + r_1(v) - r(u)\big)T \leq T \\ \Longrightarrow\ & s(u,v) \leq T \end{aligned}$$

Therefore, $s(u,v) \leq T$, $\forall(u,v) \in E$, thus

$$t_1(v) \leq T,\ \forall v \in V. \tag{3.7}$$

Then we assume that there exists an edge $(x,y) \in E$ on which (3.3) is violated, that is,

$$t(y) < t(x) + d(x,y) - (w(x,y) + r(y) - r(x))T$$

Since it satisfies (3.4), $\big(r(y), t(y)\big)$ is either $\big(r_1(y), t_1(y)\big)$ or $\big(r_1(y)+1, \mathrm{fd}(y)\big)$, that is, one of the following inequalities is true.

$$\begin{cases} t_1(y) < t(x) + d(x,y) - \big(w(x,y) + r_1(y) - r(x)\big)T \\ \mathrm{fd}(y) < t(x) + d(x,y) - \big(w(x,y) + r_1(y) + 1 - r(x)\big)T \end{cases}$$

By (3.7), $t_1(y) \leq T$, which implies that the second one is true only if the first is true. However, the first inequality contradicts (3.6). Therefore, such (x, y) does not exist and (3.3) is satisfied.

($\rightarrow$): We assume that (3.1)-(3.3) are solvable. Since (3.1) and (3.3) are inequalities, it is not the absolute values of the variables but their differences that make sense. Among all possible solutions, we are interested in the ones that satisfy the timing validity, that is, for every PI u, $t(u) = r(u) = 0$, and for all vertex $v \in V$ other than the PIs,

$$t(v) = \max_{(u,v)\in E} t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T. \tag{3.8}$$

Intuitively, timing validity rules out the solutions that have no physical meanings.

In addition, if v has no incident forbidden edges, it may have two equivalent satisfying assignments: $\big(T, r(v)\big)$ and $\big(0, r(v)+1\big)$. Physically, it means that if a flip-flop is placed right after v, it can also be moved ahead to be immediately before v. Without loss of generality, we choose the solution that assigns $\big(T, r(v)\big)$ to v. In other words, $0 < t(v) \leq T$ for such v. We will show in the following that the solution we quantified above is also a solution to (3.4). More specifically, $r(v) = r_1(v)$ and $t(v) = t_1(v) \leq T - \mathrm{bd}(v)$, $\forall v \in V$.

To show $r(v) = r_1(v)$, $\forall v \in V$, we first rewrite (3.5) as

$$r_1(v) = \max\big(r_2(v), r_3(v)\big),$$

where $r_2(v)$ and $r_3(v)$ are defined as

$$\begin{aligned} r_2(v) &= \max_{\forall(u,v) or (v,u)\in E_1, r(v)\le r(u)} r(u), \\ r_3(v) &= \max_{\forall(u,v)\in E_2} \left\lceil \frac{t(u)+d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1. \end{aligned}$$

Since (3.1) and (3.3) are satisfied, for all $(u,v) \in E$,

$$\begin{aligned} & \qquad\qquad (3.3) \\ & \overset{T>0}{\Longrightarrow} \; r(v) \ge \frac{t(u)+d(u,v)-t(v)}{T} - w(u,v) + r(u) \\ & \overset{t(v)\le T}{\Longrightarrow} \; r(v) \ge \frac{t(u)+d(u,v)}{T} - w(u,v) + r(u) - 1 \\ & \Longrightarrow r(v) \ge \left\lceil \frac{t(u)+d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1 \end{aligned}$$

That is,

$$r(v) \ge r_3(v), \; \forall v \in V. \tag{3.9}$$

We then divide the vertices into two categories V_1 and V_2 depending on whether they have incident forbidden edges or not, respectively. For vertex $v \in V_1$, we know $r_2(v) = r(v)$ otherwise (3.2) is violated. Therefore, $r_2(v) = r(v) \ge r_3(v)$ by (3.9), which leads to $r(v) = r_1(v)$ since $r_1(v) = \max\big(r_2(v), r_3(v)\big)$. On the other hand, if $v \in V_2$, then $r_2(v) = 0$, we have $r_1(v) = r_3(v)$. What remains is to show that $r(v) = r_3(v)$. Suppose otherwise $r(v) \ne r_3(v)$,

then $r(v) > r_3(v)$ by (3.9). By the definition of $r_3(v)$, we know that for all $(u, v) \in E_2$,

$$\begin{aligned} & r(v) > \left\lceil \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1 \\ \Longrightarrow \;& r(v) \geq \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) \\ \overset{T>0}{\Longrightarrow} \;& t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T \leq 0 \end{aligned}$$

Thus $t(v) = \max_{\forall (u,v) \in E} t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T \leq 0$, which contradicts the fact that $0 < t(v) \leq T$ for such v. Therefore, $r_1(v) = r(v)$, $\forall v \in V (= V_1 \cup V_2)$.

By replacing $r(v)$ with $r_1(v)$ in the timing validity (3.8), (3.6) can be written as $t_1(v) = \max\big(\mathrm{fd}(v), t(v)\big)$. Given that $\mathrm{fd}(v) \leq t(v) \leq T - \mathrm{bd}(v)$ by (3.1), we have $t_1(v) = t(v) \leq T - \mathrm{bd}(v)$, which concludes our proof. □

3.2. Wire retiming under a given period as fixpoint computation

In this section, we will formulate the wire retiming problem under a given clock period T as a fixpoint computation.

According to Theorem 3.1.1, (3.1)-(3.3) are equivalent to (3.4). For the brevity of presentation, we use a variable x_v to denote the assignment on vertex $v \in V$, that is, $x_v = \big(r(v), t(v)\big)$. Then (3.4) can be viewed as the following equation for each vertex v :

$$x_v = f_v(x_{v_1}, x_{v_2}, \ldots, x_{v_k}), \tag{3.10}$$

where $v_1, v_2, \ldots, v_k$ are vertices in V that have edges incident to v. A *partial order* ($\leq$) can be defined between two retiming values x_u and x'_u as follows.

$$x_u \leq x'_u \overset{\text{def}}{=} (r(u) < r'(u)) \vee (r(u) = r'(u) \wedge t(u) \leq t'(u))$$

In fact, a lexicographic order is defined. We use X to represent the vector $(x_1, x_2, \ldots, x_{|V|})$, that is, the retiming information for the whole circuit. The partial order on each vertex can be extended vertex-wisely to get a partial order on the vectors: two vectors $X = (x_1, x_2, \ldots, x_{|V|})$ and $Y = (y_1, y_2, \ldots, y_{|V|})$ satisfy $X \leq Y$ if and only if $x_i \leq y_i$ for all $1 \leq i \leq |V|$. Put all f_v, $v \in V$ together, we get a transformation for the whole system which can be written as

$$X = F(X) \tag{3.11}$$

There is a natural initialization to set $t(u) = r(u) = 0$ for each PI u and $t(v) = r(v) = -\infty$ for each vertex $v \in V$ other than the PIs. If we regard it as the bottom element($\perp$), and another extreme assignment with $t(v) = r(v) = \infty, \forall v \in V$ as the top element($\top$), according to the lattice theory [**29**], the solution space P becomes a *complete partially ordered set(CPO)*. Figure 3.2 shows an illustration of the solution space. For all $X \in P$, we have $\perp \leq X \leq \top$.

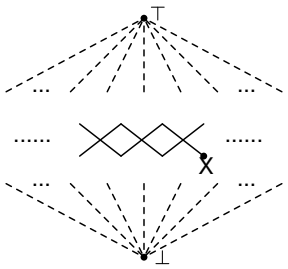

Figure 3.2. Illustration of solution space.

We know from Theorem 3.1.1 that the given period T is feasible if and only if (3.11) has a solution. To find a solution to (3.11), iterative method can be used. It starts with $X_0 = \perp$ as the initial vector, then iteratively computes new vectors from previous ones $X_1 = F(X_0), X_2 = F(X_1), \ldots$ until we find a X_m such that $X_m = X_{m-1}$. Then X_m is a solution to (3.11), also called a *fixpoint* of F. We will show in the following section that

(3.11) has a solution if and only if F is finitely convergent, or there exists a finite m such that $F^{m-1}(X_0) = F^m(X_0)$.

3.2.1. Finite convergence

We first show that F is a *monotonic* (or *order-preserving*) transformation.

Lemma 3.2.1. *For any X and X' satisfying $t(u) \leq T$ and $t'(u) \leq T$ for all $u \in V$, if $X \leq X'$ then $F(X) \leq F(X')$.*

Proof. We first prove that the member transformations $f_1, f_2, \ldots, f_n$ are monotonic.

Since $X \leq X'$, then $x_u \leq x'_u$ for all $u \in V$, that is, either $r(u) < r'(u)$ or $r(u) = r'(u) \wedge t(u) \leq t'(u)$. Together with $t(u) \leq T$ and $t'(u) \leq T$, we have

$$t(u)/T + r(u) \leq t'(u)/T + r'(u). \tag{3.12}$$

Therefore $r_1(v) \leq r'_1(v)$, $\forall v \in V$, by (3.5).

For $r_1(v) = r'_1(v)$, we have $t_1(v) \leq t'_1(v)$ by (3.6) and (3.12). Consider $f_v(X)$ and $f_v(X')$ by (3.4), there are three cases. If $t_1(v) \leq t'_1(v) \leq T - \mathrm{bd}(v)$, then $f_v(X) = \big(r_1(v), t_1(v)\big) \leq \big(r'_1(v), t'_1(v)\big) = f_v(X')$. It can be similarly shown that $f_v(X) \leq f_v(X')$ for the other two cases of $t_1(v) \leq T - \mathrm{bd}(v) \leq t'_1(v)$ and $T - \mathrm{bd}(v) \leq t_1(v) \leq t'_1(v)$.

For $r_1(v) < r'_1(v)$, we have two cases. If $t_1(v) \leq T - \mathrm{bd}(v)$, then $f_v(X) = \big(r_1(v), t_1(v)\big)$, thus $f_v(X) \leq \big(r'_1(v), t'_1(v)\big) \leq f_v(X')$. If $t_1(v) > T - \mathrm{bd}(v)$, then $f_v(X) = \big(r_1(v) + 1, \mathrm{fd}(v)\big)$. Since $r_1(v) < r'_1(v)$, we have $r_1(v) + 1 \leq r'_1(v)$. Together with $t'_1(v) \geq \mathrm{fd}(v)$ from (3.6), we have $f_v(X) \leq \big(r'_1(v), t'_1(v)\big) \leq f_v(X')$.

Therefore, $f_v(X) \leq f_v(X')$, for all $v \in V$. By the definition of partial order on vectors, $F(X) \leq F(X')$. □

The next result is a corollary of the above lemma.

Corollary 3.2.1.1. *For any vector X and X' satisfying $t(u) \leq T$ and $t'(u) \leq T$ for all $u \in V$, if $X \leq X'$ then $F^m(X) \leq F^m(X')$, $m \in Z^+$.*

The next lemma shows that if F is not finitely convergent but T is feasible, then r will keep increasing.

Lemma 3.2.2. *Let X denote any finitely reachable vector by the iterative method and $X_{|V|}$ denote the vector such that $X_{|V|} = F^{|V|}(X)$. If F is not finitely convergent but T is feasible, then there must exist a vertex $v \in V$ such that $r_{|V|}(v) > r(v)$.*

Proof. Suppose otherwise $r_{|V|}(v) = r(v)$ for all $v \in V$. Then (3.2) is always kept otherwise $r(v)$ will be increased for some v by (3.4) and (3.5). We divide the vertices into two categories V_1 and V_2 depending on whether they have incident forbidden edges or not, respectively. Consider one of the $|V|$ iterations.

For vertex $v \in V_1$, since (3.2) is satisfied, we have

$$r(v) = \max_{\forall (u,v) or (v,u) \in E_1, r(v) \leq r(u)} r(u).$$

Thus $r_1(v) \geq r(v)$ by (3.5). On the other hand, $r(v) \geq r_1(v)$ by (3.4). Therefore $r(v) = r_1(v)$.

For vertex $v \in V_2$ and any $(u, v) \in E$,

$$\begin{aligned}
&\overset{(3.5)}{\Longrightarrow} r_1(v) \geq \left\lceil \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) \right\rceil - 1 \\
&\Longrightarrow r_1(v) \geq \frac{t(u) + d(u,v)}{T} - w(u,v) + r(u) - 1 \\
&\overset{T>0}{\Longrightarrow} t(u) + d(u,v) - \big(w(u,v) + r_1(v) - r(u)\big)T \leq T
\end{aligned}$$

Together with (3.6), we have $t_1(v) \leq T = T - \mathrm{fd}(v)$, which also leads to $r(v) = r_1(v)$ by (3.4).

Therefore, $\big(r(v), t(v)\big) = \big(r_1(v), t_1(v)\big)$ is always chosen for all $v \in V$ during the $|V|$ iterations between X and $X_{|V|}$. In other words, F is reduced to only updating t by (3.6), which is exactly what Bellman-Ford's algorithm [**23**] does to solve a set of inequalities. Since F is not finitely convergent, after $|V|$ iterations, there must exist a positive cycle c in G such that $d(c) - w(c)T > 0$ (otherwise, F converges and becomes finitely convergent since X is finitely reachable), that is, $T < d(c)/w(c)$. By Lemma 2.2.1, T is infeasible, which contradicts that the given T is feasible.

Therefore, there exists a vertex $v \in V$ such that $r_{|V|}(v) \neq r(v)$. Since F cannot reduce r, we have $r_{|V|}(v) > r(v)$. □

Based on Corollary 3.2.1.1 and Lemma 3.2.2, we can establish the finite convergence in the following theorem.

Theorem 3.2.1. *F is finitely convergent if and only if T is feasible.*

Proof. If F is finitely convergent, we must have found a vector $\hat{X}$ such that $\hat{X} = F(\hat{X})$. By Theorem 3.1.1, $\hat{X}$ satisfies (3.1)-(3.3), thus T is feasible.

For the other direction, since T is feasible, we know that there exists a solution $\hat{X} = \Big(\big(\hat{r}(1), \hat{t}(1)\big), \ldots, \big(\hat{r}(|V|), \hat{t}(|V|)\big)\Big)$ such that $F(\hat{X}) = \hat{X}$. Starting from X_0, at most $|E|$ iterations are needed for each vertex $v \in V$ to have a finite $\big(r_{|E|}(v), t_{|E|}(v)\big)$. After that, if F is not finitely convergent, then, by Lemma 3.2.2, $r(v)$ for some $v \in V$ will be increased every $|V|$ iterations. Let

$$k = |E| + |V| \sum_{i=1}^{|V|} \big(\hat{r}(i) - r_{|E|}(i)\big)$$

and X_k be the vector such that $X_k = F^k(X_0)$. Lemma 3.2.2 implies that there exists a vertex $v \in V$ whose $r_k(v)$ exceeds $\hat{r}(v)$. However, since $X_0 \leq \hat{X}$, we have $F^k(X_0) \leq F^k(\hat{X}) = \hat{X}$ by Corollary 3.2.1.1, i.e., $r_k(u) \leq \hat{r}(u)$, $\forall u \in V$, which is a contradiction. Therefore, F is finitely convergent. □

In fact, according to the lattice theory [**29**], if F, defined on a complete partially ordered set, has a fixpoint, then F has a least fixpoint X_{lf}, defined as

$$\big(X_{lf} = F(X_{lf})\big) \wedge \big(\forall X : \bot \leq X \wedge X = F(X) : X_{lf} \leq X\big).$$

The next theorem shows that the iterative method will reach the least fixpoint.

Theorem 3.2.2. *The fixpoint found by iterative method from the bottom element ($\bot$) is the least fixpoint.*

Proof. Let X_{lf} and X_m be the least fixpoint and the fixpoint found by iterative method, respectively. Since X_{lf} is the least, it must be true that $X_{lf} \leq X_m$. On the other hand, since $X_0 = \bot \leq X_{lf}$, Corollary 3.2.1.1 implies $X_m = F^m(X_0) \leq F^m(X_{lf}) = X_{lf}$. Therefore, we have $X_m = X_{lf}$. □

3.2.2. Chaotic iteration

In this section we will establish that no matter what evaluation order is used, the iterative method will always converge to the same fixpoint, that is, the least fixpoint. This is called the scheme of *chaotic iteration* [**24**]. Here, a transformation F is composed of a set of partial transformations on each vertex. In each step, one or more partial transformations are applied to update retiming information on one or more vertices. All retiming information on other

vertices is kept unchanged. We will use F_S to represent such a partial transformation done in one step, where S represents the set of vertices whose retiming information is updated. We have an immediate result.

Lemma 3.2.3. *Given a subset of vertices $S \subseteq V$, if $X \leq F(X)$, then $X \leq F_S(X) \leq F(X)$.*

The following lemma states that no matter what evaluation order is used, the generated sequence is monotonic in the same direction.

Lemma 3.2.4. *Let $X'_0(=X_0), X'_1, \ldots, X'_{m'}, X'_{m'+1}(=X'_{m'})$ be the update sequence generated by a sequence of arbitrary partial transformations $F_{S_0}, F_{S_1}, \ldots, F_{S_{m'}}$. Then $X'_i \leq F_{S_i}(X'_i) \leq F(X'_i)$, for all $0 \leq i \leq m'$.*

Proof. We prove it by applying induction on i. Consider the base for $i = 0$, it is obvious that $X'_0 \leq F(X'_0)$ since $X'_0 = \perp$. With "X" replaced by "X'_0" in Lemma 3.2.3, we have $X'_0 \leq F_{S_0}(X'_0) = X'_1 \leq F(X'_0)$.

Suppose that the statement is true for $0 \leq i \leq k$, that is, $X'_k \leq F_{S_k}(X'_k) = X'_{k+1} \leq F(X'_k)$. Since $X'_k \leq X'_{k+1}$, we have $F(X'_k) \leq F(X'_{k+1})$ by Lemma 3.2.1. Together with the inductive hypothesis $\left(X'_{k+1} \leq F(X'_k)\right)$, we get $X'_{k+1} \leq F(X'_{k+1})$, which, by Lemma 3.2.3, leads to $X'_{k+1} \leq F_{S_{k+1}}(X'_{k+1}) \leq F(X'_{k+1})$. Thus the statement is also true for $i = k + 1$. The lemma is true. □

Furthermore, if the evaluation order is *fair*, that is, a partial transformation will always be applied if its inputs and outputs are not consistent, then the chaotic iteration will always reach the same fixpoint as F.

Theorem 3.2.3. *Any fair evaluation order will reach the same fixpoint as F, the least fixpoint.*

Proof. Let $X'_0(=X_0), X'_1, \ldots, X'_{m'}$ be the update sequence generated by a sequence of arbitrary partial transformations $F_{S_0}, F_{S_1}, \ldots, F_{S_{m'-1}}$ and $X_0, X_1, \ldots, X_m$ be the one generated by F. X_m and $X'_{m'}$ are fixpoints while any vector before them in the sequences is not. Our goal is to show that $X_m = X'_{m'}$.

First of all, note that if $X'_i \leq X_i$, then $F(X'_i) \leq F(X_i)$ by Lemma 3.2.1, and $F_{S_i}(X'_i) \leq F(X'_i)$ by Lemma 3.2.4, thus $F_{S_i}(X'_i) \leq F(X_i)$. Applying this relationship to both sequences with the fact that $X'_0 = X_0$, we have $X'_i \leq X_i$ for $0 \leq i \leq \min(m', m)$. If $m' < m$, then $X'_{m'} \leq X_{m'} \leq X_m$. Given that X_m is the least fixpoint by Theorem 3.2.2, we have $X_m \leq X'_{m'}$, thus $X'_{m'} = X_{m'} = X_m$, which contradicts the assumption that $X_{m'}$ is not a fixpoint. Therefore $m \leq m'$, which confirms the intuition that the number of iterations required by partial transformations is always larger than that by F. We then extend the sequence generated by F to the length of m' by assigning $X_i = X_m$, for all $m+1 \leq i \leq m'$, as illustrated in Figure 3.3.

$$
\begin{array}{ccccccccccc}
X'_0 & \leq & X'_1 & \leq & \cdots & \leq & X'_m & \leq & \cdots & \leq & X'_{m'} \\
\| & & \wedge & & & & \wedge & & & & \wedge \\
X_0 & \leq & X_1 & \leq & \cdots & \leq & X_m & = & \cdots & = & X_{m'}
\end{array}
$$

Figure 3.3. Update sequences by F any fair partial transformation.

It is obviously that $X'_i \leq X_i$, $m \leq i \leq m'$. Hence, $X'_{m'} \leq X_{m'} = X_m$. Given that X_m is the least fixpoint, we have $X'_{m'} = X_m$. □

3.3. Algorithm

3.3.1. Detailed description

(3.4) provides a clear scheme to do the updates. However, each update of a single variable requires a series of checkings on all the adjacent vertices, which could be inefficient. Since the fixpoint is independent on evaluation orders, an alternative is to propagate an update on a vertex to its adjacent vertices, instead of determining the update from them. In this way, updates are triggered only when they are needed, and unnecessary checkings are saved.

To propagate the update of x_u out, we decompose (3.4) into four different cases. We list the corresponding operations for each case in Figure 3.4-3.7 respectively. They are further illustrated in Figure 3.8.

Case 1. $(u, v) \in E_1$, $d(u, v) = 0$, $w(u, v) = 1$

Input: x_u; **Output**: $f_v(x_u)$.
`If` $\big(r(v) < r(u)\big)$ `then`
  $r(v) \leftarrow r(u)$; $t(v) \leftarrow \texttt{fd}(v) = 0$;

Figure 3.4. Update operations for Case 1.

Case 2. $(u, v) \in E_1$, $d(u, v) > 0$, $w(u, v) = 0$

Input: x_u; **Output**: $f_v(x_u)$.
`If` $\big(r(v) < r(u)\big)$ `then`
  $r(v) \leftarrow r(u)$;
  `If (3.3) is violated, then`
    $t(v) \leftarrow \max\big(t(u) + d(u, v), \texttt{fd}(v)\big)$;

Figure 3.5. Update operations for Case 2.

Case 3. $(u, v) \in E_2$

Input: x_u; **Output**: $f_v(x_u)$.
`If (3.3) is violated then`
$r(v) \leftarrow \left\lceil \frac{t(u)+d(u,v)}{T} + r(u) - w(u,v) \right\rceil - 1$;
$t(v) \leftarrow \max\Big(\texttt{fd}(v),$
$t(u) + d(u,v) - \big(w(u,v) + r(v) - r(u)\big)T\Big)$;
`If` $t\big((v) > T - \texttt{bd}(v)\big)$ `then`
$r(v) \leftarrow r(v) + 1$; $t(v) \leftarrow \texttt{fd}(v)$;

Figure 3.6. Update operations for Case 3.

Input: x_u; **Output**: $f_v(x_u)$.
`If` $\big(r(v) < r(u)\big)$ `then`
$r(v) \leftarrow r(u)$; $t(v) \leftarrow \texttt{fd}(v)$;

Figure 3.7. Update operations for Case 4.

Case 4. $(v,u) \in E_1$

It can be verified that the operations will satisfy (3.2) and (3.3) on (u,v). Take Case 2 as an example (arguments for other cases are similar), (3.2) is satisfied after $r(v)$ is assigned with the value of $r(u)$. Since $w(u,v) = 0$ for $(u,v) \in E_1$, (3.3) is satisfied after the $t(v)$ assignment. In addition, if $t(u)$ satisfies (3.1), then $t(v)$ will also satisfy (3.1), otherwise $T - \text{bd}(u) + d(u,v) \geq t(u) + d(u,v) = t(v) > T - \text{bd}(v)$, or $\text{bd}(u) < \text{bd}(v) + d(u,v)$, which contradicts the definition of bd.

With these explicit operations, any fair evaluation order will lead to the least fixpoint if F is finitely convergent. However, if T is infeasible, the update procedure will never stop. Therefore, we need to come up with some criteria to efficiently check the infeasibility and terminate the procedure. The following theorem provides one.

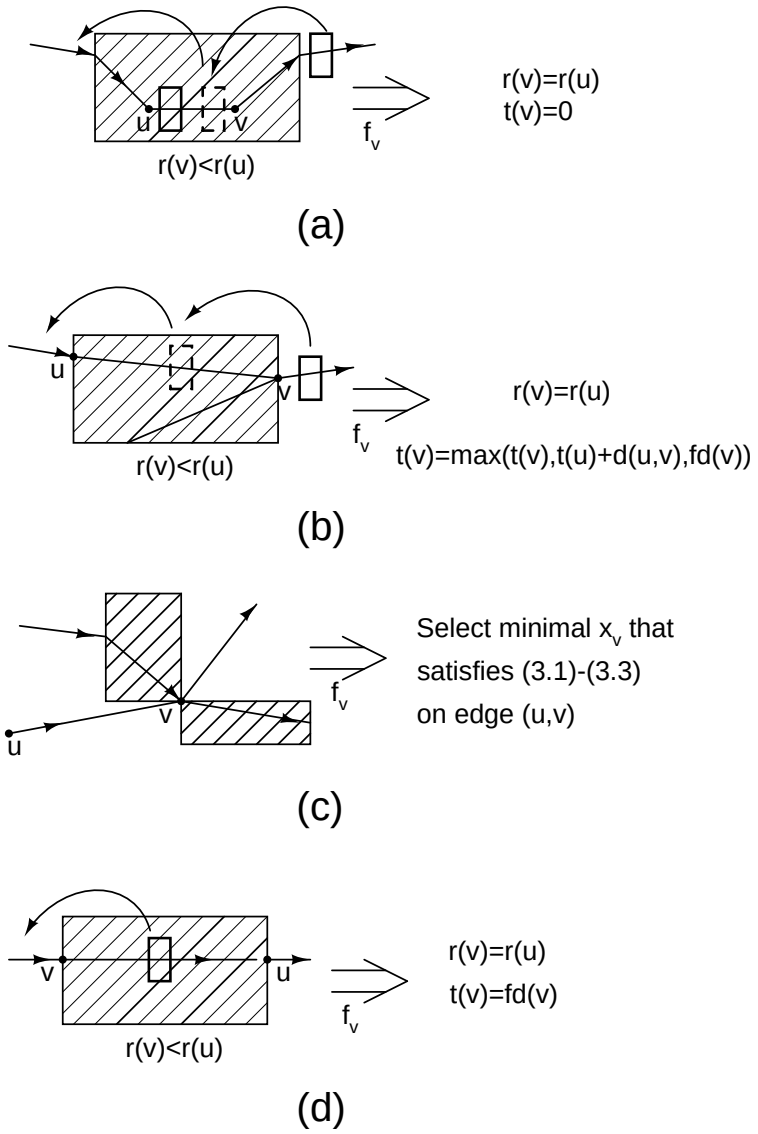

Figure 3.8. Four cases to propagate the update of x_u out.

Theorem 3.3.1. *A given T is infeasible if and only if $(\exists v \in V : r(v) > N_{\text{ff}})$ is true during the iteration, where $N_{\text{ff}} = \sum_{(u,v)\in E} w(u,v)$ is the total number of flip-flops in the original circuit.*

Proof. ($\rightarrow$): Suppose T is infeasible, then F has no fixpoint and will never converge. Similar to Lemma 3.2.2, we can show that r will be increased every $|V|$ iterations. In other words, $r(v)$ will exceed N_{ff} for some $v \in V$ after a finite number iterations.

($\leftarrow$): Suppose otherwise ($\exists v \in V : r(v) > N_{\text{ff}}$) and T is feasible. Let $\hat{X}$ be the least fixpoint of F. When it is reached, we have $\hat{r}(v) \geq r(v) > N_{\text{ff}}$ by Lemma 3.2.1. Let V' be the set of vertices that can reach v in G regardless of edge directions, including v. Let $u \in V'$ be the vertex whose $\hat{r}(u) = \min_{i \in V'} \hat{r}(i)$. There are two cases depending on whether a direct path exists between u and v.

If no, we can find another vertex $x \in V'$ distinct from u and v such that all acyclic paths between v and x are in one direction, either from v to x or from x to v. Suppose they are all from v to x (similar arguments apply to the other case), let p denote one such a path. We have $\hat{w}(p) = w(p) - \hat{r}(v) + \hat{r}(x)$. Since all paths between v and x are from v to x, the flip-flops that were moved into p through x are different from those that were originally in p. Thus, $w(p) + \hat{r}(x) \leq N_{\text{ff}}$. Given that $\hat{r}(v) > N_{\text{ff}}$, it leads to $\hat{w}(p_1) < 0$, which is impossible for a legal retiming with non-negative edge weights guaranteed by (3.3).

Therefore a direct path exists between u and v. This implies $\hat{r}(u) > 0$, otherwise more than N_{ff} number of flip-flops have to be moved into or out of the path between u and v, which is impossible by retiming. However we can construct another fixpoint by decreasing all $\hat{r}(i)$, $i \in V'$, by $\hat{r}(u)$, which contradicts Theorem 3.2.3 that the fixpoint we finally reach is the least fixpoint. Therefore, T is infeasible. $\square$

In Figure 3.9, we present our algorithm for feasibility checking under a given fixed clock period T. It uses an auxiliary queue Q to facilitate the update procedure. Initially, we have in Q all PIs. Vertices in Q are extracted one at a time to trigger possible updates on

adjacent vertices. A vertex is inserted into Q whenever its retiming information is updated. The algorithm terminates when Q is empty or it finds T infeasible. In conjunction with binary search, it solves the minimum period wire retiming problem.

```
Algorithm  Fixed period wire retiming
Input:  A graph representation G = (V, E), T
Output:  A fixpoint or infeasibility report.

Initialization, add PI into Q;
While (Q ≠ ∅) do
   u ←extract(Q);
   For each (u, v) or (v, u) ∈ E do
      If (x_v ≠ f_v(x_u)) then
         x_v ← f_v(x_u);
         Q ← v if v ∉ Q;
      If (r(v) > N_ff) then
         Return infeasibility report;
Return the fixpoint;
```

Figure 3.9. Pseudocode of algorithm for feasibility checking.

3.3.2. Speed-up techniques

3.3.2.1. Infeasibility checking criteria. Although Theorem 3.3.1 provides a criterion for infeasibility checking, it may not be efficient for all cases. For example, if G involves a critical cycle c (a cycle that determines the optimal clock period), N_{ff} is a large number and the given T is slightly smaller than the optimal clock period, then the algorithm will spend a long time to update the critical cycle before the infeasibility is discovered by Theorem 3.3.1. In this section, we provide three additional criteria to accelerate the infeasibility checking.

The first one makes use of the history of the updates. The idea is similar to Shenoy and Rudell [**90**] to maintain a predecessor vertex pointer denoted by $\pi : V \rightarrow V \cup \{\text{null}\}$ for each

vertex. Whenever x_v is updated through edge (u, v) or (v, u), we set $\pi(v) = u$, and reset $\pi(x)$ to "null" for those x with $\pi(x) = v$. Thus, (3.2) and (3.3) are always satisfied on edge $\big(\pi(v), v\big)$ or $\big(v, \pi(v)\big)$, $\forall v \in V$. This gives us another criterion.

Theorem 3.3.2. *If there exists a vertex u whose consecutive updates x_u and x'_u satisfy $t'(u) = t(u)$, and before resetting $\pi(v)$ to "null" for those v with $\pi(v) = u$, we find a cycle by following π from u, then T is infeasible.*

Proof. According to Lemma 3.2.4, $x_u \leq x'_u$. Given that $t'(u) = t(u)$, we have $r'(u) > r(u)$, otherwise $x'_u = x_u$ is not an update.

If the cycle involves only one vertex u, the only edge e in the cycle is not in E_1 since none of Case 1 (Figure 3.4), Case 2 (Figure 3.5) and Case 4 (Figure 3.7) will increase $r(u)$. For Case 3 (Figure 3.6), $r(u)$ is increased only when (3.3) is violated on e, that is, $t(u) < t(u) + d(e) - w(e)T$, or $d(e) > w(e)T$. By Lemma 2.2.1, T is infeasible.

For the cycle with multiple vertices, there exists a vertex v other than u such that $\pi(v) = u$ before it is reset to "null". Figure 3.10 gives an illustration, where the dashed edges represent the valid π pointers after $\pi(v)$ is reset, and the solid edges represent the cycle on which updates were carried out.

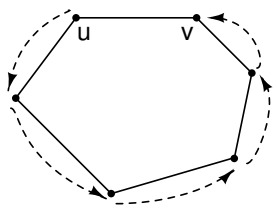

Figure 3.10. Valid π assignments after the x'_u update.

We now propagate the update on u to v. If it is Case 1 or Case 4, we will have $r'(v) = r'(u) > r(u) = r(v)$ and $t'(v) = t(v) = \mathrm{fd}(v)$. For Case 2 and Case 3, we have $r'(v) = r'(u) >$

$r(u) = r(v)$, and $t'(v) = t(v)$ because of $t'(u) = t(u)$. Therefore, $r'(v) - r(v) = r'(u) - r(u)$ and $t'(v) = t(v)$.

The above analysis can be applied to other edges along the cycle. As a consequence, the next update on u can be computed as: $r''(u) > r'(u)$, $t''(u) = t'(u)$. Since the evaluation order will not affect the final result, we can stick to this cycle for updates. Thus, $r(u)$ will keep increasing and eventually exceed N_{ff}. Therefore, T is infeasible. □

The second is inspired by the proof of Lemma 3.2.2.

Theorem 3.3.3. *let X'_k represent the retiming vector for the whole system when the k_{th} vertex is extracted from Q in Figure 3.9. For any X'_k and its $|V|^2_{th}$ succeeding representation $X'_{k+|V|^2}$, if $r'_k(u) = r'_{k+|V|^2}(u)$, for all $u \in V$, then T is infeasible.*

Proof. To characterize the update sequence executed in Figure 3.9, we insert dummies into Q. The dummies are only used as separators and no operation is actually executed when they are extracted from Q. We insert the first dummy right before we enter the while-loop. The second dummy is inserted when the first one is extracted from Q, and so on. Let $\tilde{X}_i$ and $\tilde{X}_{i+1}$ denote the retiming vector for the whole system when the i_{th} and $(i+1)_{th}$ dummies are exacted from Q, respectively. Then, the updates executed between them in the algorithm actually simulate one pass of F, that is, $\tilde{X}_{i+1} = F(\tilde{X}_i)$. Given that a vertex will not be inserted into Q when it is already in Q, the number of vertices between any two consecutive dummies is at most $|V|$. Therefore, $F^{|V|}(X'_k) \leq X'_{k+|V|^2}$.

Since $r'_k(u) = r'_{k+|V|^2}(u)$, for all $u \in V$, updates on t can only happen in Case 3. As a result, F is reduced to updating t as Bellman-Ford's algorithm. The only reason why it did not converge within $|V|$ passes is the discovery of a positive cycle c in G through which $d(c) - w(c)T > 0$. By Lemma 2.2.1, T is infeasible. □

The third one checks the r values at primary outputs.

Theorem 3.3.4. *If $r(v) > 0$ for some primary output v, then T is infeasible.*

Proof. Suppose T is feasible, then F is finitely convergent by Theorem 3.2.1. Let $\hat{X}$ denote the least fixpoint of F. When it is reached, we have $\hat{r}(v) = h \geq r(v) > 0$ by Lemma 3.2.1. It means that h number of flip-flops need to be moved into the circuit through PO v. This can only happen when these flip-flops are moved outside the circuit through PIs. As a result, $\hat{r}(u) = h$, for all PI/PO u. However, we can construct another fixpoint by decreasing all $\hat{r}(i)$, $i \in V$ by h, which contradicts the fact that $\hat{X}$ is the least fixpoint. Therefore, T is infeasible. □

3.3.2.2. Evaluation order setting. Although the evaluation order will not affect the final result, it is a key factor in determining the actual computational time because different orders lead to different update sequences.

Since every x_u has a dependency set of vertices, an efficient way to do the update is to update x_u when possibly most of its dependent variables are stable. Therefore, a good strategy to do the update is to first come up with a dependency graph[1] $G_d = G \cup \{(v, u)\}, \forall (u, v) \in E_1$. Then in G_d, we identify each of the strongly connected components, also called a cluster, and do the updates on each cluster until it converges, in a topological order of the clusters. By doing so, the evaluation order of each cluster is determined. But the vertices inside a cluster have no order and the convergent process can be any chaotic iteration.

[1]The reason why we include $\{(v, u)\}$, $\forall (u, v) \in E_1$ is because an update can follow the inverse direction of a forbidden edge.

3.3.3. Computational complexity

For the sake of simplicity, we put aside the evaluation order setting. The complexity can be broke down into two categories: processing the updates and criteria checking. According to Theorem 3.3.3, r will increase before the next $|V|^2_{th}$ vertex is extracted from Q, otherwise the program will terminate immediately. On the other hand, $r(v)$ is upper bounded by N_{ff} for all $v \in V$ by Theorem 3.3.1. Therefore, in the worst case, $O(|V|^3 N_{\text{ff}})$ number of vertices will be extracted from Q. Let $n(v)$ denote the number of incoming and outgoing edges incident to $v \in V$, and N denote the maximum of them, i.e.

$$N = \max_{v \in V} \ n(v).$$

Then processing the updates triggered by extracted vertices takes $O(N|V|^3 N_{\text{ff}})$ time. Checking the criterion in Theorem 3.3.2 takes $O(N|V|^4 N_{\text{ff}})$ in all because every vertex extracted from Q could induce N number of π cycle checkings, each of which takes at most $O(|V|)$ time. It dominates the complexity of checking Theorem 3.3.1, 3.3.3 and 3.3.4. Summing up these two categories, the program will terminate in $O(N|V|^4 N_{\text{ff}})$ time with either the fixpoint or an infeasibility answer.

Remark 3.3.1. *Caution should be taken on this bound. First of all, the design methodology behind the algorithm is to make it as efficient as possible. In other words, we try to improve the practical efficiency of the algorithm as much as we can. This explains why we incorporate the criterion Theorem 6 into the infeasibility checking process, though it has the dominant worst case bound* $O(N|V|^4 N_{\text{ff}})$. *If we drop it, the worst case bound of the algorithm can be reduced to* $O(N|V|^3 N_{\text{ff}})$. *On the other hand, we believe that the worst case bound we obtained is just an upper bound of the actual running time. Although we analyze the worst*

case bounds for both processing updates and criteria checking, the overall complexity that combines them is hard to analyze since they interact with each other. For example, if the circuit involves a critical cycle and the given period is slightly smaller than the optimal clock period, then the algorithm will reach the worst case bound assuming that Theorem 6 is not considered. However, with Theorem 6, a π-cycle will be found in $O(V^2)$ time, which will terminate the algorithm. So far, we cannot find a concrete example that reaches the worst case bound when all the proposed checking criteria are considered. A tighter bound may exist but its mathematical analysis is so complex that we cannot deduce it so far. The efficiency of our algorithm is confirmed by the experiments.

3.4. Experimental results

We implemented the algorithm with the proposed speed-up techniques in Theorem 3.3.1-3.3.4. In order to give a comparison with the algorithm in Chapter 2, we use the same test files. The computed optimal clock periods match the results in Table 2.1.

In Table 3.1, we highlight the running time differences between the algorithm in Chapter 2 and the current one, denoted as t_{pre} and t_{new} respectively. They both enjoy a precision of 0.1. The results clearly show that the fixpoint computation approach is much more efficient. It confirms that the bound on computational complexity in Section 3.3.3 is loose.

3.5. Conclusions

In this chapter, we formulate the constraints of the wire retiming problem as a fixpoint computation. An iterative algorithm is proposed with polynomial worst case runtime for feasibility checking. Although the asymptotic complexity is not improved compared with our

Table 3.1. Running Time Comparison

Circuit	`w/o non-CB blocks`		`w/ non-CB blocks`	
	t_{pre}(sec)	t_{new}(sec)	t_{pre}(sec)	t_{new}(sec)
s386	1.97	0.01	3.67	0.01
s400	1.64	0.01	3.38	0.01
s444	2.23	0.03	4.31	0.01
s838	8.79	0.03	33.42	0.02
s953	9.76	0.04	17.56	0.07
s1488	35.17	0.08	98.88	0.05
s1494	34.13	0.08	62.86	0.09
s5378	684.6	0.24	1344.74	0.29
s13207	-	1.07	-	206.52
s35932	-	18.63	-	6.19
s38584	-	7.44	-	21992.67

previous work [**110**], the practical efficiency of the new algorithm is revealed and corroborated by the experimental results.

CHAPTER 4

Optimal Wire Retiming Without Binary Search

We have seen in Chapter 3 that the fixed period wire retiming problem can be efficiently solved by a fixpoint computation. With a binary search, the algorithm for the minimum period wire retiming problem can also be improved. However, it still remains as a fully polynomial-time approximation scheme (FPTAS), that is, the overall complexity is dependent on a given binary search precision. In this chapter, a polynomial-time algorithm is proposed to solve the minimum period wire retiming problem without binary search. To the best of our knowledge, it is the first work that shows the minimum period wire retiming problem can be solved incrementally in polynomial time.

The rest of this chapter is organized as follows. Section 4.1 reviews the constraints. Section 4.2 is an overview of the algorithm. Section 4.3 elaborates the algorithm. Section 4.4 presents the experimental results, followed by a brief discussion in Section 4.5.

4.1. Notations and constraints

To ease the representation, we use $w_r(u, v)$ to denote the number of flip-flops on $(u, v) \in E$ after retiming, i.e.,

$$w_r(u, v) = w(u, v) + r(v) - r(u).$$

We also use $w_r(p)$ to denote the number of flip-flops on path $p = u \rightsquigarrow v$ after retiming: $w_r(p) = w(p) + r(v) - r(u)$. As mentioned before, since retiming will not change the number of flip-flops in a a cycle c, $w(c)$ is independent of retiming.

For the purpose of easy reference, we review the constraints below.

$$r(u) = r(v), \quad \forall (u,v) \in E_1 \tag{4.1}$$

$$w_r(u,v) \geq 0, \quad \forall (u,v) \in E_2 \tag{4.2}$$

$$t(v) = \max\big(0,\ \max_{\forall (u,v)\in E} t(u) + d(u,v) - w_r(u,v)T\big), \quad \forall v \in V \tag{4.3}$$

$$t(v) \leq T, \quad \forall v \in V \tag{4.4}$$

Our task is to find a solution (r, t, T) satisfying (4.1)-(4.4) with the minimal T.

4.2. Overview

The optimal wire retiming problem asks for a feasible solution with the minimal T. To this aim, our algorithm starts with a feasible solution and iteratively reduces T to the optimal while keeping (4.1)-(4.4) satisfied.

First of all, we notice that (4.1)-(4.2) are trivially satisfied with $r(v) = 0$, $\forall v \in V$. Based on this, we can easily find a feasible solution that also satisfies (4.3)-(4.4) if T is chosen large enough.

Since T is only involved in (4.3)-(4.4), one intuitive way to reduce it is to tighten (4.3)-(4.4) under the same r, for the purpose of which we use Burns's algorithm [**6**]. We shall always try this to reduce T. On the other hand, there must be a lower bound of T that is determined by the particular r. Once T is reduced to the lower bound by tightening (4.3)-(4.4), we will compare it with an imagined optimal solution and adjust r to approach the optimal solution. These two ways of reducing T are iteratively applied until we reach the optimal T.

4.3. Algorithm

4.3.1. Initialization

We want an initialization that returns a feasible solution. It is apparent that the original flip-flop configuration without any retiming must satisfy (4.1)-(4.2). A natural thought is to initialize it to satisfy the timing constraints (4.3)-(4.4) as well, which is easy to achieve when T is large.

To satisfy (4.3)-(4.4), we first compute t as if all flip-flops on each edge $(u, v) \in E$ are lined up immediately before v so that (4.4) is satisfied with $T = \max_{v \in V} t(v)$ and (4.3) is satisfied on edges with weight zero. But (4.3) might be violated on edges with positive weights. Consider the example in Figure 4.1(a), where $d(x, y) = 1$, $d(y, z) = 3$ and $w(y, z) = 2$. When the two flip-flops are lined up immediately before z, the computed $t(x)$, $t(y)$ and $t(z)$ are 0, 1 and 0 respectively. However, (4.3) is violated on (y, z) under $T = \max\big(t(x), t(y), t(z)\big) = 1$ since $t(z) = 0 < t(y) + d(y, z) - w(y, z)T = 2$.

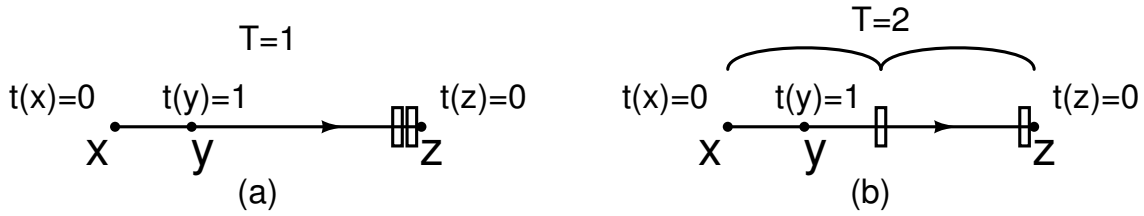

Figure 4.1. Flip-flop local distribution.

To fix the violation of (4.3), we propose to locally distribute the flip-flops on each edge and increase T accordingly. The idea is illustrated in Figure 4.1(b), where one flip-flop is moved to the middle of the path from x to z and (4.3) is satisfied by setting $T = 2$. In general, if (4.3) is violated some edge (u, v) with $w(u, v) > 0$, i.e., $t(v) < t(u) + d(u, v) - w(u, v)T$, we will increase T to $\frac{t(u)+d(u,v)-t(v)}{w(u,v)}$ under which (4.3) is merely satisfied on (u, v). Since T is

always increased while t remains unchanged, (4.4) will be kept. When completed, we have a feasible clock period T and the corresponding t values that satisfy (4.3)-(4.4).

In addition, we keep a label $f : V \rightarrow V$ such that if $t(v) > 0$, then $f(v)$ is the starting vertex of a critical path terminating at v with $t\big(f(v)\big) = 0$, $\forall v \in V$. The initialization procedure is summarized in Figure 4.2.

```
INIT(G)
  t(v), f(v) ← 0, v, ∀v ∈ V ; T ← 0; Q ← V ;
  While (Q ≠ ∅) do
    u ←dequeue(Q);
    For each (u, v) ∈ E with w(u, v) = 0 do
        If (t(v) < t(u) + d(u, v)) then
            t(v) ← t(u) + d(u, v);
            f(v) ← f(u);
            If (T < t(v)) then
                T ← t(v);
            Q ← Q ∪ {v} if v ∉ Q;
  For each (u, v) ∈ E with w(u, v) > 0 do
    If (T < (t(u)+d(u,v)−t(v))/(w(u,v))) then
        T ← (t(u)+d(u,v)−t(v))/(w(u,v))
```

Figure 4.2. Pseudocode of initialization.

4.3.2. Base algorithm

Starting with the initialization, there are two possible ways by which T can be reduced. Since T is only involved in (4.3)-(4.4), one intuitive way to reduce it, referred to as "t-ADJUST", is to tighten (4.3)-(4.4) under a given r, i.e., without changing the number of flip-flops on each edge. The other that allows such changes is referred to as "r-ADJUST". If the current r happens to be an optimal retiming, then we can reach the optimal T by "t-ADJUST" only.

Otherwise "r-ADJUST" is needed to adjust r toward an optimal retiming and the following lemma provides a direction of the adjustment.

Lemma 4.3.1. *Provided that (4.1)-(4.3) are satisfied, if $t(v) \geq T$ for some $v \in V$, then any retiming satisfying (4.1)-(4.4) but with a smaller clock period must have more than $w_r(p)$ number of flip-flops on p, where p is any critical path from $f(v)$ to v.*

Proof. Consider any critical path p from $f(v)$ to v. Since $t(v) \geq T > 0$, we have $t\big(f(v)\big) = 0$ by the definition of $f(v)$. In addition, since p is critical, it means that the delay between any two consecutive flip-flops on p is T, which implies that the path delay $d(p)$ is equal to $w_r(p)T + t(v)$.

For the sake of contradiction, we assume that there exists another retiming $(\bar{r}, \bar{t}, \bar{T})$ satisfying (4.1)-(4.4) with $\bar{T} < T$ but $w_{\bar{r}}(p) \leq w_r(p)$, i.e., $\bar{r}(v) - \bar{r}\big(f(v)\big) \leq r(v) - r\big(f(v)\big)$. Since $(\bar{r}, \bar{t}, \bar{T})$ satisfies (4.3), it must be true that

$$\bar{t}(v) \geq \bar{t}\big(f(v)\big) + d(p) - w_{\bar{r}}(p)\bar{T} \geq d(p) - w_{\bar{r}}(p)\bar{T}.$$

Together with the facts that $d(p) = w_r(p)T + t(v)$, $t(v) \geq T$, $\bar{T} < T$, and $w_{\bar{r}}(p) \leq w_r(p)$, the above inequality can be reduced to

$$\begin{aligned}
\bar{t}(v) &\geq w_r(p)T + t(v) - w_{\bar{r}}(p)\bar{T} \\
&\geq \big(w_r(p) + 1\big)T - w_{\bar{r}}(p)\bar{T} \\
&> \big(w_r(p) + 1 - w_{\bar{r}}(p)\big)T \\
&\geq T
\end{aligned}$$

Given that $(\bar{r}, \bar{t}, \bar{T})$ satisfies (4.4), we have $\bar{T} \geq \bar{t}(v) > T$, which contradicts $\bar{T} < T$. Therefore, $w_{\bar{r}}(p) > w_r(p)$ must be true. □

Since the initialization returns a feasible solution satisfying (4.1)-(4.4), we can check if $\big(\exists v \in V : t(v) = T\big)$ is true. If it is true, then, by Lemma 4.3.1, we know that T is the optimal under the current r. In order to reach a possible better solution, 'r-ADJUST" needs to be carried out. Otherwise $\big(\forall v \in V : t(v) < T\big)$ is true and "t-ADJUST" is employed to tighten (4.3)-(4.4). We now describe "t-ADJUST" and "r-ADJUST" in Section 4.3.3 and 4.3.4 respectively.

4.3.3. Clock period reduction with r unchanged

Since r is kept unchanged, (4.1)-(4.2) will not be violated. Our objective is to find a smaller T such that (4.3)-(4.4) are satisfied under the same r.

First of all, we identify the set of critical edges E_c, that is, $(u, v) \in E$ with $t(u) + d(u, v) - w_r(u, v)T = t(v)$. If E_c contains a critical cycle, then the current T coincides with the cycle ratio of the critical cycle, which, by Lemma 2.2.1, implies that the current T is the solution to the optimal wire retiming problem.

Otherwise, $G_c = (V, E_c)$ forms a directed acyclic graph. Suppose there exists a better solution $(r, \bar{t}, \bar{T})$ under the same r satisfying (4.1)-(4.4) with $\bar{T} < T$. Let θ denote the difference, i.e., $\theta = T - \bar{T}$. Consider critical edge (u, v) with $t(u) + d(u, v) - w_r(u, v)T = t(v)$. If $\bar{t}(u) = t(u)$, then $\bar{t}(v)$ is at least $w_r(u, v) \cdot \theta$ larger than $t(v)$; otherwise, the difference between $\bar{t}(u)$ and $t(u)$ also needs to be counted into $\bar{t}(v)$. To characterize $\bar{t}$, we define

$\Delta : V \rightarrow Z^+$ as follows:

$$\Delta(v) = \begin{cases} 0, & v \text{ is a root in } G_c \\ \max\big(\Delta(v), \Delta(u) + w_r(u,v)\big), & (u,v) \in E_c \end{cases}$$

More specifically, $\Delta(v)$ is the maximum number of flip-flops on critical paths from roots in G_c to v, $\forall v \in V$. Thus, for small θ, the following is true:

$$\bar{t}(v) = t(v) + \theta \cdot \Delta(v), \quad \forall v \in V.$$

The above relation collapses only when θ exceeds a threshold such that some of (4.3)-(4.4) is violated. The threshold is characterized below.

Note that (4.3) is violated only if the following inequality is true on some edge $(u,v) \in E$:

$$\bar{t}(v) < \bar{t}(u) + d(u,v) - w_r(u,v)\bar{T}.$$

Given that $\bar{t}(v) = t(v) + \theta \cdot \Delta(v)$, $\bar{t}(u) = t(u) + \theta \cdot \Delta(u)$ and $\bar{T} = T - \theta$, the above inequality can be rewritten as:

$$t(v) - \big(t(u) + d(u,v) - w_r(u,v)T\big) < \theta\big(\Delta(u) + w_r(u,v) - \Delta(v)\big).$$

Since (r,t,T) satisfies (4.3), the left hand side of the above inequality is no less than 0. It follows that $\Delta(u) + w_r(u,v) - \Delta(v) > 0$ and $\theta > \frac{t(v) - \big(t(u) + d(u,v) - w_r(u,v)T\big)}{\Delta(u) + w_r(u,v) - \Delta(v)}$. Therefore, as long as $\theta \leq \frac{t(v) - \big(t(u) + d(u,v) - w_r(u,v)T\big)}{\Delta(u) + w_r(u,v) - \Delta(v)}$ on edges with $\Delta(u) + w_r(u,v) - \Delta(v) > 0$, (4.3) is guaranteed to hold. We then have an upper bound for θ. This process of characterization for keeping (4.3) is exactly the same as for Burns's algorithm [**6**].

To maintain (4.4), we need to make sure that for all $v \in V$,

$$\bar{t}(v) \leq \bar{T} \Rightarrow t(v) + \theta \cdot \Delta(v) \leq T - \theta \Rightarrow \theta \leq \big(T - t(v)\big)/\big(\Delta(v) + 1\big)$$

It gives another upper bound. The threshold is then the smaller of the two bounds. Given that the first bound is evaluated on non-critical edges and that $\big(\forall v \in V : t(v) < T\big)$, the value of the threshold is always positive. By reducing T by the threshold and adjusting t accordingly, we obtain a better solution satisfying (4.1)-(4.4).

The above process can be iteratively applied as long as no critical cycle is formed and $\big(\forall v \in V : t(v) < T\big)$ is true. The pseudocode is given in Figure 4.3.

Due to the similarity with Burns's algorithm [**6**], procedure "t-ADJUST" has a provable complexity of $O(|V|^2|E|)$ before either the presence of a critical cycle or an evidence of $\big(\exists v \in V : t(v) = T\big)$, the former of which certifies the optimality of T while the latter necessitates "r-ADJUST".

4.3.4. Clock period reduction by changing r

Suppose that the current T is not the optimal. Let (r^*, t^*, T^*) denote an optimal solution. Given that $t(v) = T$ for some $v \in V$, we know by Lemma 4.3.1 that the optimal solution requires more flip-flops on the critical path from $f(v)$ to v, that is,

$$r^*(v) - r^*\big(f(v)\big) > r(v) - r\big(f(v)\big) \tag{4.5}$$

We can add more flip-flops on the critical path by either increasing $r(v)$ or decreasing $r\big(f(v)\big)$. No matter which one is chosen, the amount of change should only be 1 since we do

t–ADJUST(G, r, t, T)
Do
▷*Identify critical edges in* E
$E_c \leftarrow \{(u,v) \in E |\ t(v) = t(u) + d(u,v) - w_r(u,v)T\}$;
If E_c contains no cycle then
Topological sort $G_c = (V, E_c)$;
▷*Compute max #FF on critical paths from roots*
For $v \in V$ in topological order of G_c do
If v is a root in G_c then
$\Delta(v) \leftarrow 0$;
Else for each $(u,v) \in E_c$ do
$\Delta(v) \leftarrow \max\{\Delta(v),\ \Delta(u) + w_r(u,v)\}$;
▷*Compute the threshold* θ *for* T *reduction*
$\theta \leftarrow \infty$
For each $(u,v) \in E$ do
If $\big(\Delta(u) + w_r(u,v) > \Delta(v)\big)$ then
$\theta \leftarrow \min\{\theta,\ \frac{t(v) - \left(t(u) + d(u,v) - w_r(u,v)T\right)}{\Delta(u) + w_r(u,v) - \Delta(v)}\}$;
For each $v \in V$ do
$\theta \leftarrow \min\{\theta,\ \frac{T - t(v)}{\Delta(v)+1}\}$;
▷*Reduce* T *by* θ *and update* t *accordingly*
$T \leftarrow T - \theta$;
For each $v \in V$ do
$t(v) \leftarrow t(v) + \theta \cdot \Delta(v)$;
While $\Big(\neg(\mathit{critical\ \ cycle}) \wedge \big(\forall v \in V : t(v) < T\big)\Big)$;

Figure 4.3. Pseudocode of adjusting t with r unchanged.

not want to over-adjust r. Without loss of generality, we choose to increase $r(v)$ by 1 in our implementation, that is, $r'(v) = r(v) + 1$.

However, the increase of $r(v)$ might violate some of (4.1)-(4.3). For example, Figure 4.4(a) shows the flip-flop configuration around v before $r(v)$ is increased[1]. Note that the flip-flop on edge (v, x_5) actually stands right after v. We separate v and the flip-flop for the

[1]Cases of (v, x_2) and (v, x_3) may occur when $d(v, x_2) = d(v, x_3) = 0$, e.g., the edges incident to M, if introduced.

purpose of better visibility. After $r(v)$ is increased, (4.1) is violated on edges (x_1, v) and (v, x_2), (4.2) is violated on edge (v, x_3), and (4.3) is violated on edge (v, x_5).

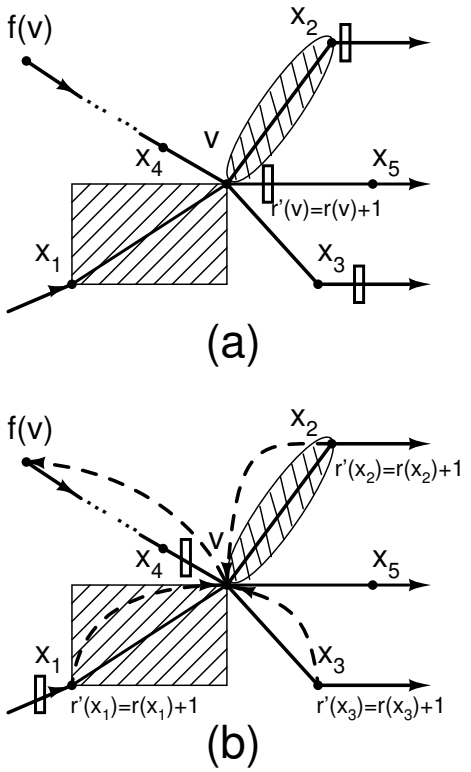

Figure 4.4. Restore (4.1)-(4.2) and assign m-labeling.

Our idea to restore (4.1)-(4.2) is illustrated in Figure 4.4(b). We first examine each edge incident to v. If (4.1) is violated on $(v, x) \in E_1$ or $(x, v) \in E_1$, it is actually $r'(v) = r(x) + 1$ due to the previous increase of $r(v)$. We then increase $r(x)$ by 1 to restore (4.1), i.e., $r'(x) = r'(v)$ with $r'(x) = r(x) + 1$. For the example in Figure 4.4(b), both $r(x_1)$ and $r(x_2)$ will be increased. If (4.2) is violated on $(v, x) \in E_2$, it is actually $w(v, x) + r(x) - r'(v) = -1$ due to the previous increase of $r(v)$. We then increase $r(x)$ by 1 to restore (4.2), i.e., $w(v, x) + r'(x) - r'(v) = 0$ with $r'(x) = r(x) + 1$. For the example in Figure 4.4(b), $r(x_3)$ will be increased. We do not need to consider the $(x, v) \in E_2$ case because the previous increase

of $r(v)$ ensures $w(x,v) + r'(v) - r(x) \geq 1$, which will not violate (4.2). Vertex x_4 in Figure 4.4(b) is this case. Likewise, we need to check the impact of the increase of $r(x)$, if any, on the edges incident to x, and so on. This process of checking will not stop until (4.1)-(4.2) are restored.

In our implementation, we employ a first-in-first-out (FIFO) queue rQ for the bookkeeping of the vertices whose r values are increased during the above process of restoring (4.1)-(4.2). For the example in Figure 4.4(b), v, x_1, x_2 and x_3 will be queued in rQ. We claim that no vertex will be queued more than once in rQ. Otherwise, let y be the first vertex that is queued twice ($r'(y) = r(y) + 2$) in order to restore (4.1)-(4.2) on some edge incident to y. If it is a violation of (4.1) on $(x,y) \in E_1$ or $(y,x) \in E_1$, then the violation will be fixed after the increase of $r(y)$, that is, $r'(x) = r'(y) = r(y) + 2$. Given that (r,t,T) satisfies (4.1), we have $r(x) = r(y)$, hence $r'(x) = r(x) + 2$, which contradicts the assumption that y is the first vertex whose r value is increased by 2 during the process. A contradiction can be similarly derived for the case of a violation of (4.2). Therefore the claim is true and we have $r'(u) \leq r(u) + 1$, $\forall u \in V$, after (4.1)-(4.2) are restored.

A key observation on Figure 4.4 is that if $r\big(f(v)\big)$ is increased before the next increase of $r(v)$, then the increase of $r(v)$ is necessitated because the increase of $r\big(f(v)\big)$ cancels the previous increase of $r(v)$ and makes (4.5) true again. The relation between $r\big(f(v)\big)$ and $r(v)$ is similar to that between $t\big(f(v)\big)$ and $t(v)$ where any increase of $t\big(f(v)\big)$ will be propagated to $t(v)$ unless the path between them ceases to be critical. The same relation exists between $r(v)$ and $r(x_1)$, $r(v)$ and $r(x_2)$, $r(v)$ and $r(x_3)$, and $r(v)$ and $r(x_5)$.

Therefore, we introduce another label $m : V \rightarrow V \cup \{\emptyset\}$, where $\emptyset$ is the default assignment, and define it as follows. For all $u \in V$, if $r(u)$ is increased due to (4.5), then $m(u)$ is set to $f(u)$; if $r(u)$ is increased due to the violation of (4.1)-(4.2) on some edge between u

and x, then $m(u)$ is set to x. For the example in Figure 4.4(b), we will have $m(v) = f(v)$ and $m(x_1) = m(x_2) = m(x_3) = v$, represented by dashed pointers.

Based on the definition of the m-labeling, we can formulate the relation between $r\big(m(u)\big)$ and $r(u)$ in the following lemma.

Lemma 4.3.2. *It is true before we reach an optimal retiming r^* that*

$$\Big(\forall u \in V, m(u) \in V : r^*\big(m(u)\big) - r\big(m(u)\big) \leq r^*(u) - r(u)\Big).$$

Proof. By the definition of the m-labeling, $m(u) \in V$ only if $r(u)$ has ever been increased since the initialization, $\forall u \in V$. For a particular vertex u, we will show that the inequality is kept after the first increase of $r(u)$ and continues to hold before an optimal r^* is reached.

Consider the first time that $r(u)$ is increased. Vertex u is queued in rQ either because (4.5) is satisfied on u or for the purpose of restoring (4.1)-(4.2). For the former case, $m(u)$ will be set to $f(u)$, and (4.5) becomes $r^*(u) - r^*\big(m(u)\big) > r(u) - r\big(m(u)\big)$, or $r^*\big(m(u)\big) - r\big(m(u)\big) < r^*(u) - r(u)$. After $r(u)$ is increased by 1, we have $r^*\big(m(u)\big) - r\big(m(u)\big) \leq r^*(u) - r(u)$. For the latter case, the increase of $r(u)$ is due to the violation of (4.1)-(4.2) on the edge between u and $m(u)$. If it is a violation of (4.1), then $r(u) = r\big(m(u)\big)$ after (4.1) is restored. It follows that $r^*\big(m(u)\big) - r\big(m(u)\big) = r^*(u) - r(u)$ since $r^*(u) = r^*\big(m(u)\big)$. If it is a violation of (4.2), then $w\big(m(u), u\big) + r(u) - r\big(m(u)\big) = 0$ after (4.2) is restored. Since $w\big(m(u), u\big) + r^*(u) - r^*\big(m(u)\big) \geq 0$, it follows that $r^*\big(m(u)\big) - r\big(m(u)\big) \leq r^*(u) - r(u)$.

In any case, the inequality is true after the first increase of $r(u)$. Even if $r\big(m(u)\big)$ may be increased thereafter, the inequality will remain true until the next increase of $r(u)$, when $m(u)$ will be assigned to a new value that may or may not differ from the original assignment.

By the same case study as above, we can show that the inequality will continue to hold. By induction, the lemma is true. □

In addition, if there exists a sequence of vertices x_i, $i = 0, 1, \ldots, k-1$, such that $x_i = m(x_{i+1})$ and $x_k = x_0$, we refer to it as an m-cycle. For the example in Figure 4.4(b), if $m\big(f(v)\big)$ is already equal to x_1, setting $m(v) = f(v)$ and $m(x_1) = v$ will cause an m-cycle among x_1, v and $f(v)$.

Since $m(u) = \emptyset$ only if $r(u)$ remains at 0 since the initialization, $\forall u \in V$. This fact enables us to conclude the following lemma.

Lemma 4.3.3. $\big(\forall u \in V : r(u) > 0\big)$ *implies an m-cycle.*

Proof. Since $r(u) > 0$, we know that $m(u) \in V$ by the definition of the m-labeling, $\forall u \in V$. If the m-labeling forms no cycle but paths, then the ending vertex u of one of the paths has to have $m(u) = \emptyset$, which contradicts $m(u) \in V$. Therefore, the m-labeling must form an m-cycle. □

The next result is a corollary to Lemma 4.3.3 which shows that an m-cycle can also appear when $r(v)$ exceeds an upper bound for some vertex $v \in V$.

Corollary 4.3.3.1. $\big(\exists v \in V : r(v) > N_{\text{ff}}\big)$ *implies an m-cycle, where N_{ff} is the total number of flip-flops in the original circuit with $N_{\text{ff}} = \sum_{(u,v)\in E} w(u, v)$.*

Proof. Suppose (4.1)-(4.2) are satisfied (if not, we shall wait until the process of restoring (4.1)-(4.2) is completed and $r(v) > N_{\text{ff}}$ is also true at that time). For the sake of contradiction, we assume that there is no m-cycle. We will show that this assumption contradicts Lemma 4.3.3 by conducting a case study. Let V' be the set of vertices that can reach v in G regardless of edge directions, including v. Suppose $u \in V'$ is such a vertex

whose $r(u) = \min_{i \in V'} r(i)$. By a proof similar to that for Theorem 3.3.1, we can show that there exists a direct path between u and v. This implies $r(u) > 0$, otherwise more than N_{ff} number of flip-flops have to be moved into or out of the path between u and v, which is impossible. In addition, V' cannot be V, otherwise it contradicts Lemma 4.3.3. However, since the vertices in $V - V'$ have no connection with those in V' at all, the vertices in V' and the edges connecting them actually constitute an independent sub-circuit $G' \subset G$. The flip-flop configuration of a sub-circuit is also independent on those of others. Thus, Lemma 4.3.3 can be applied to G' and $G - G'$ separately. Since we assume no m-cycle, Lemma 4.3.3 implies that $\big(\exists \gamma \in V' : r(\gamma) = 0\big)$ is true, which contradicts the fact that u is a vertex in V' with the minimal r value. Therefore, we must have an m-cycle. □

The reason why an m-cycle is important is because its appearance certifies the optimality of the current T.

Theorem 4.3.1. *If an m-cycle appears, then the current T is optimal.*

Proof. Suppose the last m assignment is $m(x_0) = x_k$, by setting which the m-labeling forms a cycle, that is, a sequence of vertices x_i, $i = 0, 1, \ldots, k-1$, such that $x_i = m(x_{i+1})$ and $x_k = x_0$.

For the sake of contradiction, we assume that the current T is not optimal. Let r denote the retiming before the current call of "r-ADJUST" and r' denote the retiming when the m-cycle appears. Recall that "r-ADJUST" is necessitated only when T is reduced to the optimal under r by "t-ADJUST". Since T is not optimal, it follows that r is not an optimal retiming. Then, by Lemma 4.3.2 and $m(x_0) = x_{k-1}$, we have $r^*(x_{k-1}) - r'(x_{k-1}) \leq r^*(x_0) - r'(x_0)$, which implies that $r^*(x_{k-1}) - r'(x_{k-1}) < r^*(x_0) - r(x_0)$ before the increase of $r'(x_0) = r(x_0) + 1$. On the other hand, Lemma 4.3.2 guarantees that $r^*(x_0) - r(x_0) \leq r^*(x_1) - r'(x_1)$ and

$r^*\big(m(x_i)\big) - r'\big(m(x_i)\big) \leq r^*(x_i) - r'(x_i)$, $2 \leq i \leq k-1$. It follows that $r^*(x_0) - r(x_0) \leq r^*(x_{k-1}) - r'(x_{k-1})$, which is a contradiction. Therefore, the current T is optimal. □

Once (4.1)-(4.2) are restored, restoring (4.3) is straightforward. First of all, we reset $t(v) = 0$, $\forall v \in V$. After that, for edges $(y, z) \in E$ with $w_r(y, z) = 0$, we update $t(z)$ with $\max\big(t(z), t(y) + d(y, z)\big)$. For others with $w_r(y, z) > 0$, if $t(y) < T$, then $t(z)$ is updated with $\max\big(t(z), t(y) + d(y, z) - w_r(y, z)T\big)$ as if all flip-flops are locally evenly distributed with delay T in between; otherwise, the first flip-flop on (y, z) will be positioned right after y and $t(z)$ is updated with $\max\big(t(z), d(y, z) - (w_r(y, z) - 1)T\big)$. The change of $t(z)$, if any, will propagate to the edges incident to z, and so on. For the example in Figure 4.4(b), if $t(x_4) < T$, then the flip-flop on (x_4, v) can be moved toward v until it hits v or the delay from x_4 to the flip-flop reaches $T - t(x_4)$; otherwise, the flip-flop is placed right after x_4.

In our implementation, another FIFO queue tQ is employed to facilitate this process. When completed, it produces t values satisfying (4.3).

However, if the resulting t satisfies $\big(\exists u \in V : t(u) \geq T\big)$, then, by Lemma 4.3.1, we have

$$r^*(u) - r^*\big(f(u)\big) > r(u) - r\big(f(u)\big).$$

All the above operations (increasing $r(u)$ and then restoring (4.1)-(4.3)) need to be carried out again on that particular u. On the other hand, if $\big(\forall v \in V : t(v) < T\big)$, (4.4) is restored and we obtain a better retiming. We then switch to "t-ADJUST".

The pseudocode for adjusting r is given in Figure 4.5. It turns out that Zhou's algorithm [**109**] for traditional retiming without binary search is a special case of Figure 4.5 with $d(u, v) = 0$, $\forall (u, v) \in E_2$.

```
r–ADJUST(G, r, t, T, Q)
 While ((Q ≠ ∅) ∧ ¬(m-cycle)) do
   v ←dequeue(Q);
   If (t(v) ≥ T) then
     r(v) ← r(v) + 1; m(u) ← f(v);
     rQ ← {v};
     ▷Operations to restore (4.1)-(4.2)
     While (rQ ≠ ∅) do
       u ←dequeue(rQ);
       m(u) ← f(v);
       For each e = (x, u) ∈ E or e = (u, x) ∈ E do
       If (((e ∈ E1) ∧ (r(x) ≠ r(u))) || (w_r(e) < 0)) then
         r(x) ← r(x) + 1; m(x) ← u;
         rQ ← rQ ∪ {x} if x ∉ rQ;
     ▷Operations to restore (4.3)
     t(v), f(v) ← 0, v, ∀v ∈ V; tQ ← V;
     While (tQ ≠ ∅) do
       y ←dequeue(tQ);
       For each (y, z) ∈ E do
         If (w_r(y, z) = 0) then
           If (t(z) < t(y) + d(y, z)) then
             t(z) ← t(y) + d(y, z);
             f(z) ← f(y);
             tQ ← tQ ∪ {z} if z ∉ tQ;
         Elsif (t(y) < T) then
           If (t(z) < t(y) + d(y, z) − w_r(y, z)T) then
             t(z) ← t(y) + d(y, z) − w_r(y, z)T;
             f(z) ← f(y);
             tQ ← tQ ∪ {z} if z ∉ tQ;
         Elsif (t(z) < d(y, z) − (w_r(y, z) − 1)T) then
           t(z) ← d(y, z) − (w_r(y, z) − 1)T;
           f(z) ← f(y);
           tQ ← tQ ∪ {z} if z ∉ tQ;
         ▷Checking (4.4) for further adjustments
         If (t(z) ≥ T) then
           Q ← Q ∪ {z} if z ∉ Q;
```

Figure 4.5. Pseudocode of adjusting r.

Having provided the pseudocodes of both "t-ADJUST" and "r-ADJUST" and the criterion of switching between them, we then present our algorithm in Figure 4.6. The structure of the algorithm is clear; it alternates between "t-ADJUST", which reduces T to the minimum under r, and "r-ADJUST", which adjusts r to approach an optimal retiming, and terminates when either a critical cycle or an m-cycle is found.

This is an elegant result that confirms our intuition. We know that if all edges can accommodate flip-flops, that is, $E_1 = \emptyset$, then (4.1) is dropped and we can compute a valid retiming by simply separating every two consecutive flip-flops by a delay value equal to the maximum cycle ratio ρ^*. The corresponding arrival times will satisfy (4.3)-(4.4) under $T = \rho^*$. In other words, the optimal period is equal to ρ^*. When the cycle with ratio ρ^* becomes critical under T, we have $T = \rho^*$ and T is the optimal. However, in the presence of forbidden edges, separating every two consecutive flip-flops with delay ρ^* may result in some flip-flop being placed on a forbidden edge, which is prohibited. Consequently, the optimal period T^* will be larger than ρ^*. When T is reduced to T^*, adding more flip-flops on critical paths will not help to further reduce T. In fact, the requirement for more flip-flops on one critical path will be propagated through forbidden edges to other critical paths, which is exactly captured by the m-labeling. When an m-cycle is formed, the total number of flip-flops in the m-cycle is a constant, independent of retiming. In other words, the requirement of more flip-flops in an m-cycle cannot be fulfilled by retiming. Therefore, the current T cannot be further reduced; it is the optimal.

4.3.5. Correctness proof and computational complexity

We prove the correctness of the algorithm by showing that it returns an optimal solution when terminates.

```
Algorithm  Incremental Optimal Wire Retiming
Input:  A graph representation G = (V, E).
Output:  A retiming with minimal clock period.

   INIT(G);  T* ← T;  r*(u) ← 0,  ∀u ∈ V;
   While (¬(critical cycle) ∧ ¬(m-cycle)) do
     Q ← {v},  ∀v ∈ V with t(v) = T;
     If (Q = ∅) then
        t–ADJUST(G, r, t, T);
        Update T* and r* with T and r;
     Else
        r–ADJUST(G, r, t, T, Q);
   Return T* and r*;
```

Figure 4.6. Pseudocode of retiming algorithm.

Theorem 4.3.2. *The algorithm in Figure 4.6 returns a retiming with the minimal clock period when it terminates.*

Proof. The algorithm terminates only if a critical cycle is found in procedure "t-ADJUST" or an m-cycle is found in "r-ADJUST". By Lemma 2.2.1 and Theorem 4.3.1, T is optimal, and the algorithm will return it along with a corresponding optimal retiming. □

We finally present the computational complexity of the algorithm to show that it terminates in polynomial time.

Theorem 4.3.3. *The algorithm in Figure 4.6 terminates in $O(|V|^3|E| \cdot N_{\mathrm{ff}})$ time, where N_{ff} is the total number of flip-flops in the original circuit.*

Proof. First of all, due to the similarity with Burns's algorithm [**6**], procedure "t-ADJUST" has a provable complexity of $O(|V|^2|E|)$ before either the presence of a critical cycle or an evidence of $\big(\exists v \in V : t(v) = T\big)$. The latter actually triggers procedure "r-ADJUST". Since no vertex is queued more than once in rQ, restoring (4.1)-(4.2) takes

$O(|V||E|)$ time. Restoring (4.3) also takes $O(|V||E|)$ time. Therefore, the complexity between two consecutive r increases is bounded by $O(|V|^2|E|)$.

On the other hand, by Corollary 4.3.3.1, the total number of r increases before the emergence of an m-cycle is bounded by $O(|V| \cdot N_{\text{ff}})$. Given that the initialization takes $O(|V||E|)$ time, we can conclude that the algorithm terminates in $O(|V|^3|E| \cdot N_{\text{ff}})$ time with either a critical cycle or an m-cycle. □

Theorem 4.3.2 and Theorem 4.3.3 together also confirm that checking critical cycle and m-cycle is sufficient in order to determine the optimality of T.

Remark 4.3.1. *The significance of Theorem 4.3.3 is not the actual formula of the bound, but showing that the optimal wire retiming problem can be indeed solved in polynomial time without using binary search. Furthermore, caution should be used on this bound. Firstly, a program usually has large running time variations on different problem instances. The worst case running time may only happen on a few rare instances, and thus it may not be a good indication of the efficiency on most other instances. For example, the worst case assumes that the algorithm terminates when all $r(v)$'s, $\forall v \in V$, are increased to N_{ff}, which is hardly the case in real applications. Secondly, even if the worst case does occur on most problem instances, a bound may be loose due to the difficulty of carrying out an accurate analysis. For example, reaching the minimal period under a given r typically takes $O(|E|)$ time on real circuits. These assumptions lead to an ideal worst case which is very unlikely to be attainable in reality. In other words, the bound in Theorem 4.3.3 is very loose. Since only necessary operations are conducted in the algorithm, it should be efficient on most instances. This is confirmed by our experimental results in Section 4.4.*

4.3.6. Speed-up technique (better initialization)

Let T_0 and T_0^* denote the clock period returned by the initialization and the minimal clock period under $r(v) = 0, \forall v \in V$, respectively. Intuitively, the closer T_0 is to T_0^*, the smaller the number of iterations the algorithm has to take to reach T_0^*. Recall that in the initialization procedure we proposed in Figure 4.2, flip-flops are not allowed to be distributed until the arrival times of all vertices are obtained. This may result in too conservative a value of T_0. For instance, a circuit with only one edge (u, v) and $w(u, v) = 1$ will be initialized with $T_0 = d(u, v)$. However, if we distribute flip-flops simultaneously during the computation of t, for this case we will place the flip-flop in the middle of (u, v), then the circuit ends up being initialized with $T_0 = d(u, v)/2$, which is exactly the optimal.

More specifically, the idea to compute t and distribute flip-flops simultaneously is as follows. For each edge $(u, v) \in E$, its flip-flops are first distributed with delay T in between, where T is the current maximum of t. Then, we check if the resulting $t(v) = \max\big(t(v), t(u) + d(u, v) - w_r(u, v)T\big)$ exceeds T. If it does exceed T, then both T and $t(v)$ are updated to $\frac{t(u)+d(u,v)}{w_r(u,v)+1}$, that is, flip-flops on (u, v) are evenly distributed.

When completed, it gives a much better T_0. However, (4.3) might not be satisfied because the flip-flops that are distributed earlier may have less than T_0 in between. As a remedy, the operations used in Figure 4.5 to restore (4.3) are employed. The resulting t will not violate (4.4). By doing so, we obtain a better T_0 within the same complexity as for Figure 4.2.

4.4. Experimental results

We implemented the algorithm and the speed-up technique. In order to give a comparison with the algorithms in Chapter 2 and 3, we use the same test files and report the results

Table 4.1. Minimal Clock Period

Circuit	$\|V\|$	$\|E\|$	N_{ff}	w/o non-CB blocks		w/ non-CB blocks		
				#Step	T^*	#Part	#Step	T^*
s386	519	700	6	37	51.1	50	1	55.0
s400	511	665	21	125	32.2	50	1	50.6
s444	557	725	21	312	35.2	40	1	63.2
s838	1299	1206	32	2	76.0	130	1	84.0
s953	1183	1515	29	39	60.6	110	2	69.5
s1488	2054	2780	6	108	70.6	200	1	73.3
s1494	2054	2792	6	63	76.9	160	1	79.9
s5378	7205	8603	179	50	111.2	500	1	115.3
s13207	19816	22999	669	270	239.5	1000	1	292.8
s35932	46097	58266	1728	125	148.3	2000	1	163.2
s38584	53473	66964	1452	341	204.0	2000	1	264.0

in a similar format in Table 4.1. They match the results in both Chapter 2 and 3[2]. Since we approach the optimal clock period by gradual reduction in the algorithm, we also report he total number of occurrences that T is successfully reduced during "t-ADJUST" in the column of "#Step".

In Table 4.2, we highlight the differences of running time among the algorithms in Chapter 2 and 3 (both with precision 0.1), the base algorithm in Figure 4.6, and the further improved one with the proposed speed-up technique, denoted as t_{bs1}, t_{bs2}, t_{base} and t_{new}, respectively. For a particular algorithm, we can compute the ratio of its running time to t_{new} for each test case[3] and obtain the arithmetic (geometric) mean of all the ratios in row "arith" ("geo"). The results clearly show that our new algorithm with speed-up technique achieves multiple-order improvement over the algorithms in Chapter 2 and 3, and the speed-up technique

[2]Although theoretically the algorithm in Figure 4.6 will reach the exact solution without being provided a precision, we have to consider the impact of floating point error introduced by practical finite precision arithmetic, due to the divisions involved in the computation of θ in Figure 4.3. In our experiments, we set the error to be 0.001, which is consistent with the implementation in Chapter 2 and 3.

[3]If the numerator is "-" or both the numerator and the denominator are "0.00", the case is ignored. If only the denominator is "0.00", it is treated as "0.01" for estimation.

contributes almost 2X speed-up by average. It also confirms that the bound in Theorem 4.3.3 is loose.

Table 4.2. Running Time Comparison (Seconds)

Circuit	w/o non-CB blocks				w/ non-CB blocks			
	t_{bs1}	t_{bs2}	t_{new}	t_{base}	t_{bs1}	t_{bs2}	t_{new}	t_{base}
s386	1.97	0.01	0.00	0.00	3.67	0.01	0.00	0.00
s400	1.64	0.01	0.01	0.01	3.38	0.01	0.00	0.00
s444	2.23	0.03	0.05	0.05	4.31	0.01	0.00	0.00
s838	8.79	0.03	0.00	0.00	33.42	0.02	0.00	0.00
s953	9.76	0.04	0.01	0.01	17.56	0.07	0.00	0.00
s1488	35.17	0.08	0.07	0.07	98.88	0.05	0.00	0.00
s1494	34.13	0.08	0.14	0.14	62.86	0.09	0.00	0.00
s5378	684.6	0.24	0.18	0.18	1344.74	0.29	0.00	0.00
s13207	-	1.07	3.94	4.07	-	206.52	0.02	0.06
s35932	-	18.63	4.81	5.02	-	6.19	0.17	0.21
s38584	-	7.44	17.18	117.76	-	21992.67	0.16	0.48
arith	851.3X	1.6X	1	1.7X	19610.3X	13442.9X	1	2.4X
geo	393.6X	1.1X	1	1.3X	2675.1X	23.0X	1	2.2X

Remark 4.4.1. *A few sentences are worth mentioning for a better understanding of the data presented in Table 4.2. First of all, though faster by average,* t_{new} *is a little slower than* t_{bs2} *for some cases "w/o non-CB blocks". This can be explained by looking into the nature of binary search. There are a couple of factors that will affect the runtime of a binary search based algorithm. First, the smaller the precision is chosen, the more the runtime will be. Second, the runtime is highly dependent on the binary search range. As we know that checking an infeasible period usually takes more time than checking a feasible one. Consequently, if the upper bound is chosen to be slightly larger than the optimal, then during most of the binary search iterations, the algorithm will be checking infeasible periods, which corresponds to the worst case runtime scenario. On the other hand, the best case scenario corresponds to the situation when the lower bound is slightly smaller than the optimal, then most binary*

search iterations will be on feasible periods. Our experiments on test cases "w/o non-CB blocks" happened to be the best scenario. That explains why t_{bs2} *is more efficient in some cases. Another phenomenon that needs to be explained is that the running times of the new algorithm actually decrease (to almost 0) when non-CB blocks are included, which counters the trend revealed by both* t_{bs1} *and* t_{bs2}*. The increasing trend of* t_{bs1} *and* t_{bs2} *can be explained by the higher binary search upper bounds caused by non-CB blocks, and the fact that for a significant number of iterations the binary search based algorithms are now checking infeasible periods. For the proposed new algorithm, however, since it always works on feasible periods, the running time per iteration is much less. If the initialization happens to generate a feasible solution that is close to the optimal, then the number of iterations to reach the optimal will be very few, as it is the case in our experiments shown in Table 4.1. Besides, the design methodology behind the new algorithm is to make it as efficient as possible. Three FIFO queues employed in the implementation are for this purpose.*

Interestingly, Chu *et al.* [**14**] also considered the retiming problem with interconnect delays. However, they formulated their problem at the block level, that is, only gates exist in the circuits. Therefore, they handled one issue (interconnect delay) in our work but not the other (retiming over blocks and buffer-forbidden regions). Furthermore, the approximation approach of first ignoring gate boundaries and then moving flip-flops out of gates does not work for non-complete bipartite blocks, that is, their near-optimal algorithm is not applicable to our second set of experiments (with non-CB blocks). In this sense, they solved a different problem, even though it looks similar to our problem.

For better comparison with their results within the first set of experiments (without non-CB blocks), we obtained their source code of the near-optimal algorithm and test files used

in [**14**], and ran our new algorithm on their test files. We report the running time differences in Table 4.3, where t_{chu} and t_{new} represent the running time for their near-optimal algorithm and our new algorithm with speed-up technique, respectively. Likewise, row "arith" ("geo") denotes the arithmetic (geometric) mean of running time improvement. Column "#Step" denotes the total number of occurrences that T is successfully reduced during the execution of our new algorithm. The near-optimal algorithm assumed 1% error bound with average 9.6 number of binary search iterations over all benchmark suite.

Note that, although all test files come from the same benchmark suite, they have different delay assignment. In their test files, wire delays were obtained by first implementing the circuits in a 0.25 um process, laying out the circuits by Silicon Ensemble, and then extracting the parasitics from the layout. Besides, since they formulated the problem at the block level, each gate was treated as a vertex. Thus, for a given benchmark circuit, the generated problem size in their formulation is approximately three times smaller than ours, where each input/output of a gate is treated as a vertex instead. For example, "s5378" in Table 4.1 and "s5378′" in Table 4.3 come from the same circuit, but the number of vertices and edges for "s5378" and "s5378′" is 7205 and 8603, and 2781 and 4261 respectively. Even with problem sizes that are three times larger, Table 4.3 reveals that our new algorithm is generally more efficient than their near-optimal algorithm. For those cases where $t_{\text{new}} > t_{\text{chu}}$, the long t_{new} is mainly due to the large "#Step", e.g., 3424 for "s6669′" and 5292 for "s38584′". In addition, since their near-optimal algorithm uses binary search, comments similar to Remark 4.4.1 can be used to explain the results in Table 4.3.

Table 4.3. Running Time Comparison with the work in [**14**] (Seconds)

Circuit	$\|V\|$	$\|E\|$	t_{chu}	t_{new}	`#Step`
s1488′	655	1405	0.20	0.01	8
s1494′	649	1411	0.22	0.01	19
s3271′	1574	2707	0.73	0.84	901
s3330′	1791	2890	0.40	0.05	42
s3384′	1687	2782	0.52	0.40	385
s4863′	2344	4093	2.42	0.03	11
s5378′	2781	4261	0.88	0.00	1
s6669′	3082	5399	1.16	10.50	3424
s9234′	5599	8005	3.13	1.47	180
s13207′	7953	11302	6.80	0.02	1
s15850′	9774	13794	19.18	27.84	1551
s35932′	16067	28590	48.40	0.08	1
s38417′	22181	32135	64.63	210.34	5292
s38584′	19255	33010	361.68	0.12	1
arith			298.8X	1	
geo			11.7X	1	

4.5. Discussion

A polynomial-time algorithm for optimal wire retiming is presented in this chapter. Contrary to all previous algorithms which used binary search to check the feasibility of a range of clock periods, the new algorithm directly checks the optimality of the current feasible clock period, and can thus either push down the period or certify the optimality.

The underlying idea looks into the nature of the binary search approach. At each step, the binary search gives the answer to the question of *whether the current clock period is feasible*. The optimality of a feasible clock period can only be established indirectly, that is, through the infeasibility of the next smaller clock period. However, in our algorithm, the question being answered at each step is *whether a feasible clock period smaller than the current one exists*. Since it gives the existence answer, the optimality is established directly once we reach such a step that gives the answer: "No".

Besides the difference of program methodology, our algorithm has many other advantages over the binary search approach. First of all, it is polynomial time bounded. No precision is required. Second, the implementation is simpler. No upper and lower bounds are needed. It is even automatically determined by the algorithm itself how far a necessary step can proceed. Third, the algorithm is very efficient in practice, which is confirmed by the experimental results. Last, but not the least, without using binary search, our algorithm is essentially incremental and has the potential of being combined with other optimization techniques, such as gate sizing, budgeting, etc., thus can be used in incremental design methodologies [**17**].

CHAPTER 5

An Efficient Retiming Algorithm Under Setup and Hold Constraints

Since its first creation twenty years ago by Leiserson and Saxe [**53**], retiming has firmly established its reputation as one of the most effective techniques for sequential circuit optimization.

Besides its steady improvements on performance [**90, 50, 34, 97, 109**], retiming has been extended to timing and area optimization of level-clocked circuits [**65, 45, 69**], skew scheduling [**84**], floorplan and placement [**101, 22**], circuit partitioning and clustering [**77, 20**], logic synthesis [**87**], power optimization [**88**], and testability [**70, 31**]. Recent progresses on semiconductor technology saw an increase in the number of global wires whose delays are longer than one clock period [**15, 43**], and retiming is again a promising technique that could be leveraged [**110, 14, 58, 103, 57**]. Our solutions to the wire retiming problem are presented in Chapter 2-4.

Among these researches, many were focus on minimizing clock period [**65, 97, 45, 14, 110, 58, 57, 109**]. However, the polynomial-time retiming algorithms proposed in these papers can only be used to satisfy setup constraint.

An algorithm was presented in [**92**] for retiming single-phase level-clocked circuits under both setup and hold constraints, but its worst-case running time was exponential. On the other hand, [**93**] proposed to handle hold constraint by inserting extra delays on short paths, but it may fail when the delay insertion is required to be discrete. The existence of

a polynomial-time algorithm for minimum period retiming of edge-triggered circuits under both setup and hold constraints has remained as an open problem until [**78**].

Given an edge-triggered sequential circuit $G = (V, E)$, a target clock period, a setup time S, and a hold time H, the algorithm in [**78**] computed a valid retiming satisfying both setup and hold constraints, or reported that there is no such a retiming. However, its worst-case running time was high. In conjunction with binary search, it took $O(|V|^3|E| \lg |V|)$ time to determine the minimum clock period.

In this chapter we present an algorithm that finds a minimum period retiming satisfying both setup and hold constraints in $O(|V|^2|E|)$ time. Interestingly, it was conjectured in [**78**] that an $O(|V|^2|E|)$-time algorithm may be designed for finding a valid retiming under a target clock period. Our algorithm not only confirms the conjecture, but also shows that finding a minimum period retiming can also be solved in $O(|V|^2|E|)$ time—a much better result.

The rest of this chapter is organized as follows. Section 5.1 presents the problem formulation. Our algorithm is presented in Section 5.2, followed by an illustration example in Section 5.3. Experimental results are given in Section 5.4. Conclusions are drawn at the end.

5.1. Problem formulation

We model an edge-triggered circuit as a directed graph $G = (V, E, d, D, w)$. Each vertex $v \in V$ represents a gate and each edge $e \in E$ represents a signal passing from one gate to another—with gate minimum delays $d : V \to \mathcal{R}^+$, gate maximum delays $D : V \to \mathcal{R}^+$ and register numbers $w : E \to \mathcal{N}$. Except for the minimum delays d, this model is identical to the one used by Leiserson and Saxe [**53**]. Without loss of generality, we assume that G is

strongly connected, which is consistent with [**78**]. All registers in the circuit are assumed to have equal setup times S and equal hold times H. To ease the presentation, register delays and clock skews are assumed to be zero. Non-zero values can be easily incorporated in the same way as in [**78**].

The problem we want to solve can be formally described as follows.

Problem 5.1.1. *Given a circuit $G = (V, E, d, D, w)$, find a relocation of registers such that both setup and hold constraints are satisfied under the minimum clock period.*

To guarantee that the new registers are actually a relocation of the old ones, a label $r : V \to \mathcal{Z}$ is used to represent how many registers are moved from the outgoing edges to the incoming edges of each vertex. It is consistent with Chapter 2-4. Based on this, the number of registers on an edge (u, v) after retiming can be computed as

$$w_r(u, v) \stackrel{\triangle}{=} w(u, v) + r(v) - r(u).$$

We use $w(p)$ to denote the number of registers on a simple path $p = u \rightsquigarrow v$ before retiming. Then the number of registers on p after retiming can be represented as

$$w_r(p) \stackrel{\triangle}{=} w(p) + r(v) - r(u).$$

A partial order $(\leq)$ can be defined between two labels r and r' as follows.

$$r \leq r' \stackrel{\triangle}{=} \big(\forall v \in V : r(v) \leq r'(v)\big).$$

Furthermore, to avoid explicitly enumerating paths, we introduce two labels $t : V \to \mathcal{R}^+$ and $T : V \to \mathcal{R}^+$ to represent the earliest and the latest output arrival times of a gate

respectively. In other words, $t(v)$ and $T(v)$ are the minimum and the maximum delays to the output of gate v from the nearest preceding registers. Using these notations, the validity of a retiming (r, t, T) is defined by the following conditions.

$$P0(r) \stackrel{\triangle}{=} \big(\forall (u,v) \in E : w_r(u,v) \geq 0\big)$$

$$P1(r) \stackrel{\triangle}{=} \big(\forall (u,v) \in E : w_r(u,v) \leq 1\big)$$

$$P2(r,t,T) \stackrel{\triangle}{=} \big(\forall (u,v) \in E : w_r(u,v) = 0 \;\Rightarrow$$
$$T(v) - T(u) \geq D(v) \;\wedge\; t(v) - t(u) \leq d(v)\big)$$

$$P3(r,t,T) \stackrel{\triangle}{=} \big(\forall (u,v) \in E : w_r(u,v) = 1 \Rightarrow$$
$$T(v) \geq D(v) \;\wedge\; t(v) \leq d(v)\big)$$

$$P4(r,t) \stackrel{\triangle}{=} \big(\forall (u,v) \in E : w_r(u,v) = 1 \;\Rightarrow\; t(u) \geq H\big)$$

The condition $P0$ states that a valid retiming should have non-negative number of registers on any edge. The condition $P1$ comes from the fact that an edge cannot accommodate more than one register without causing hold violations among the registers. The conditions $P2$ and $P3$ define T and t as the latest and the earliest arrival times. More specifically, the latest (earliest) arrival time of a gate is at least (at most) the summation of the gate delay and the latest (earliest) arrival time of its fanins. The condition $P4$ is the hold constraint. These conditions altogether characterize a valid retiming that permits a feasible clock period of $S + T_{max}$, where S is the setup time and

$$T_{max} \stackrel{\triangle}{=} \max_{v \in V} T(v).$$

We use P to denote the conjunction of $P0$-$P4$, i.e.,

$$P(r,t,T) \stackrel{\triangle}{=} P0(r) \wedge P1(r) \wedge P2(r,t,T) \wedge P3(r,t,T) \wedge P4(r,t)$$

The optimality of a min-period retiming (r,t,T) is given by the following condition.

$$P5(r,t,T) \stackrel{\triangle}{=} \left(\forall r',t',T' : P(r',t',T') \;\Rightarrow\; T_{max} \leq T'_{max}\right)$$

Therefore, (r,t,T) is a min-period retiming if and only if $P(r,t,T) \wedge P5(r,t,T)$, i.e., among all valid retimings—those satisfying P—, the current (r,t,T) has the minimum T_{max}.

5.2. Algorithms

We first present an algorithm in Section 5.2.1 for finding a retiming that satisfies the hold constraint. We start with the original circuit, i.e., $r(v) = 0$, $\forall v \in V$. While some of $P0$-$P4$ is violated, we will compare the current r with an imagined valid retiming and increase r to approach it.

Section 5.2.2 explains how T_{max} can be reduced by increasing r to approach an imagined optimal retiming. Since it only increases r, which is consistent with the r adjustments in the algorithm for finding a valid retiming, we combine them to yield a min-period retiming algorithm.

5.2.1. Retiming for hold constraint

Let $(\bar{r},\bar{t},\bar{T})$ be a valid retiming, i.e., $P(\bar{r},\bar{t},\bar{T})$. It should be noted that $(\bar{r}+c,\bar{t},\bar{T})$, where $c \in \mathcal{Z}$ is an arbitrary constant, is also a valid retiming. Therefore in the remainder of this

chapter, we assume that a valid retiming should satisfy

$$\big(\forall v \in V : \bar{r}(v) \geq 0\big) \wedge \big(\exists v \in V : \bar{r}(v) = 0\big).$$

In addition, since $(\bar{r}, \bar{t}, \bar{T})$ satisfies $P1$, there is an upper bound for $w_r(p)$ on any simple path p. This is stated in the following lemma.

Lemma 5.2.1. *For any valid retiming $(\bar{r}, \bar{t}, \bar{T})$, we have $w_{\bar{r}}(p) \leq |V|$ for every simple path or simple cycle p.*

The next result is a corollary of the above lemma.

Corollary 5.2.1.1. *For any valid retiming $(\bar{r}, \bar{t}, \bar{T})$, we have $\big(\forall v \in V : \bar{r}(v) \leq |V|\big)$.*

Proof. For the sake of contradiction, we assume that $\bar{r}(v) > |V|$ for some $v \in V$. Let u be the vertex such that $\bar{r}(u) = 0$. Since G is strongly connected, consider a simple path p from u to v, we have $w_{\bar{r}}(p) = w(p) + \bar{r}(v) - \bar{r}(u) = w(p) + \bar{r}(v)$. By Lemma 5.2.1, $w_{\bar{r}}(p) \leq |V|$, thus $\bar{r}(v) \leq |V| - w(p) \leq |V|$. □

Based on the above corollary, we can define a necessary condition for a valid retiming as

$$B(r) \stackrel{\triangle}{=} \big(\forall v \in V : 0 \leq r(v) \leq |V|\big) \wedge \big(\exists v \in V : r(v) = 0\big)$$

Thus, $B(\bar{r})$ is true for any valid retiming $\bar{r}$.

To reach $(\bar{r}, \bar{t}, \bar{T})$, we start with the original circuit, i.e., $r(v) = 0$, $\forall v \in V$, which trivially satisfies $P0$. If $P1$ is not satisfied under the current r, we have $w_r(u, v) > 1 \geq w_{\bar{r}}(u, v)$ for some $(u, v) \in E$, i.e.,

$$r(v) - r(u) > \bar{r}(v) - \bar{r}(u) \tag{5.1}$$

We need to remove at least $w_r(u,v)-1$ registers out of (u,v) by increasing $r(u)$. However, the increase of $r(u)$ may violate $P0$ on other edges (u,x) going out of u if $w_r(u,x)=0$ before the increase. In other words, we have $w_r(u,x)<0\leq w_{\bar{r}}(u,x)$ after the increase of $r(u)$, i.e.,

$$\bar{r}(x)-\bar{r}(u)>r(x)-r(u) \tag{5.2}$$

We then increase $r(x)$ by the same amount to restore $P0$ on (u,x). The above process is iterated until both $P0$ and $P1$ are satisfied on all edges.

After that, we find t to satisfy $P2$ and $P3$ by shortest path computation. In addition, we keep a label $f:V\rightarrow V$ such that a shortest combinational path to v starts from $f(v)$. We do not consider T since we only focus on the hold constraint in this section.

What remains is to check t against $P4$. If it is violated, it means that $t(u)<H$ for some $(u,v)\in E$ with $w_r(u,v)=1$. Let $z=f(u)$. By the definition of f, a shortest combinational path p to u starts from z. In addition, there is at least one edge $(y,z)\in E$ such that $w_r(y,z)=1$, otherwise p cannot be the shortest.

To fix the hold violation at u, we can either remove the register on (u,v), or increase $t(u)$ by moving the register on (y,z) further ahead. In general, the number of registers on the path $q=\{(y,z)\}\cup p\cup\{(u,v)\}$ should be less than 2, i.e., $w(q)+r(v)-r(y)=2>w(q)+\bar{r}(v)-\bar{r}(y)$, or

$$r(v)-r(y)>\bar{r}(v)-\bar{r}(y) \tag{5.3}$$

We will move r closer to $\bar{r}$ by increasing $r(y)$ by 1.

The pseudocode for finding a valid retiming is given in Figure 5.1.

The next lemma establishes an invariant during the execution of "ValidHold".

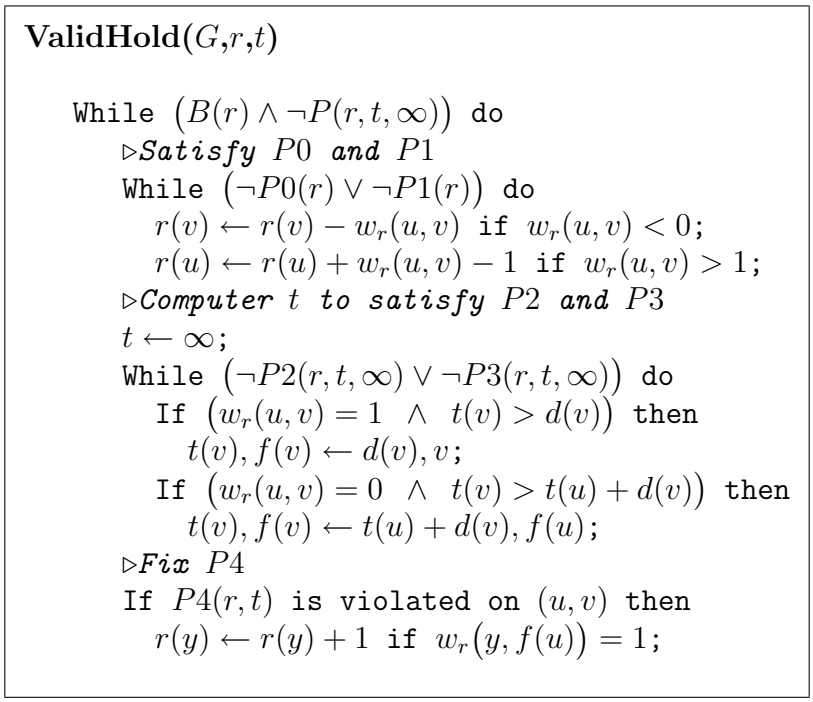

Figure 5.1. Pseudocode of finding a valid retiming.

Lemma 5.2.2. *If $r \le \bar{r}$, where $\bar{r}$ is a valid retiming, then $r \le \bar{r}$ after the execution of "ValidHold" in Figure 5.1.*

Proof. During "ValidHold", r is increased only when we attempt to satisfy $P0$ or $P1$, or to fix $P4$.

The first one happens when $\big(\exists (u,v) \in E : w_r(u,v) < 0\big)$. Let $r'(v) = r(v) - w_r(u,v) = r(v) - \big(w(u,v) + r(v) - r(u)\big) = r(u) - w(u,v)$. It follows that $r'(v) \le \bar{r}(u) - w(u,v)$ since $r(u) \le \bar{r}(u)$ due to $r \le \bar{r}$. Given that $w_{\bar{r}}(u,v) = w(u,v) + \bar{r}(v) - \bar{r}(u) \ge 0$, we have $\bar{r}(u) - w(u,v) \le \bar{r}(v)$. Therefore, $r'(v) \le \bar{r}(v)$, i.e., $r \le \bar{r}$ is kept after the increase of $r(v)$.

The second one happens when $\big(\exists (u,v) \in E : w_r(u,v) > 1\big)$. Let $r'(u) = r(u) + w_r(u,v) - 1 = r(u) + w(u,v) + r(v) - r(u) - 1 = w(u,v) + r(v) - 1$. It follows that $r'(u) \le w(u,v) + \bar{r}(v) - 1$ since $r(v) \le \bar{r}(v)$ due to $r \le \bar{r}$. Given that $w_{\bar{r}}(u,v) = w(u,v) + \bar{r}(v) - \bar{r}(u) \le 1$, we have

$w(u,v) + \bar{r}(v) - 1 \leq \bar{r}(u)$. Therefore, $r'(u) \leq \bar{r}(u)$, i.e., $r \leq \bar{r}$ is kept after the increase of $r(u)$.

When the last one happens, (5.3) is true. Given that $r \leq \bar{r}$, we have $r(y) < \bar{r}(y)$, otherwise $r(y) = \bar{r}(y)$, thus $r(v) > \bar{r}(v)$, which contradicts $r \leq \bar{r}$. Therefore, $r \leq \bar{r}$ holds after $r(y)$ is increased by 1.

Since none of the above three violates $r \leq \bar{r}$, it is kept after the execution of "ValidHold".

□

Based on the above lemma, we can establishes the correctness of "ValidHold".

Theorem 5.2.1. *Given that $r(v) \geq 0$, $\forall v \in V$, the procedure "ValidHold" terminates in $O(|V|^2|E|)$ time. It finds a valid retiming $\bar{r}$ with $r \leq \bar{r}$, if such an $\bar{r}$ exists.*

Proof. The procedure terminates only if $\neg B(r)$ or $P(r,t,\infty)$. For either case, the total number of r increases will not be larger than $|V|^2$ since we start with $r(v) \geq 0, \forall v \in V$. Given that satisfying $P2$ and $P3$ can be done in $O(|E|)$ time [**53**], the amount of time between two consecutive r increases is $O(|E|)$. Therefore, "ValidHold" will terminate in $O(|V|^2|E|)$ time.

If there exists a valid retiming $\bar{r}$ with $r \leq \bar{r}$, then Lemma 5.2.2 guarantees that $r \leq \bar{r}$ upon termination, which implies $B(r)$. Therefore, the procedure is terminated due to $P(r,t,\infty)$, i.e., it finds a valid retiming. □

As a result, we can apply "ValidHold" to the original circuit to obtain a valid retiming, if it exists.

5.2.2. Min-period retiming (for setup constraint)

Once we obtain a valid retiming r, we can find T to satisfy $P2$ and $P3$ by longest path computation. Similar to f, we introduce a label $F : V \rightarrow V$ such that a longest combinational path to v starts from $F(v)$.

Suppose that the valid retiming (r, t, T) we obtained via "ValidHold" from the original circuit is not optimal, i.e., $\neg P5(r, t, T)$. Let (r^*, t^*, T^*) be an optimal retiming. Lemma 5.2.2 guarantees that $r \leq r^*$. In this section we extend the idea in [**109**] to realize r^* by adjusting r.

Since $T^*_{max} < T_{max}$, there exists a vertex $v \in V$ such that $T(v) = T_{max} > T^*_{max} \geq T^*(v)$. Let $p = F(v) \rightsquigarrow v$ be a longest combinational path to v. The fact that $T^*(v) < T(v)$ implies that there must be at least one register on p in (r^*, t^*, T^*), i.e., $w_{r^*}(p) > 0 = w_r(p)$, or equivalently,

$$r^*(v) - r^*\big(F(v)\big) > r(v) - r\big(F(v)\big) \tag{5.4}$$

We can add more registers on p by increasing $r(v)$. The amount of increase should only be 1 since we do not want to over-adjust r. After that, we apply "ValidHold" to restore $P0$-$P4$. It is interesting to notice that increasing r to approach r^* is consistent with the r adjustments in "ValidHold", thus they can be combined smoothly.

The pseudocode for finding an optimal retiming is presented in Figure 5.2.

The following theorem establishes the correctness of our algorithm.

Theorem 5.2.2. *The algorithm in Figure 5.2 will terminate in $O(|V|^2|E|)$ time with an optimal retiming satisfying setup and hold constraints under the minimum clock period, or with a report that there is no valid retiming at all.*

```
Algorithm  Min-Period Retiming
Input:  A circuit G = (V, E, d, D, w),
        setup time S, hold time H.
Output:  An optimal retiming r^opt satisfying
        setup and hold constraints under
        minimum clock period φ^opt.

   r ← 0;
   ValidHold(G,r,t);
   Return ''No valid retiming'' if ¬B(r);
   φ^opt ← ∞;
   While (B(r)) do
      ▷Compute T to satisfy P2 and P3
      T ← 0;
      While (¬P2(r,t,T) ∨ ¬P3(r,t,T)) do
        If (w_r(u,v) = 1 ∧ T(v) < D(v)) then
          T(v), F(v) ← D(v), v;
        If (w_r(u,v) = 0 ∧ T(v) < T(u) + D(v)) then
          T(v), F(v) ← T(u) + D(v), F(u);
      ▷Update φ^opt and r^opt
      φ^opt, r^opt ← T_max, r if T_max < φ^opt;
      ▷Adjust r to reduce φ^opt
      r(v) ← r(v) + 1 for some v with T(v) ≥ φ^opt;
      ValidHold(G,r,t);
   ▷Take into account setup time
   φ^opt ← φ^opt + S;
   Return r^opt and φ^opt;
```

Figure 5.2. Pseudocode of retiming algorithm.

Proof. The algorithm terminates only if $\neg B(r)$. Since we start with $r(v) = 0$, $\forall v \in V$, the total number of r increases is no larger than $|V|^2$. Given that satisfying $P2$ and $P3$ can be done in $O(|E|)$ time, which is also the time between two consecutive r increases, the algorithm terminates in $O(|V|^2|E|)$ time.

The algorithm first attempts to find a valid retiming from the original circuit. If it fails, then, by Theorem 5.2.1, we know that there is no valid retiming at all.

Otherwise, we have $r \leq r^*$ before we enter the outer while loop. We use r^{opt} to record the valid retiming with the smallest clock period ϕ^{opt} we found so far. If ϕ^{opt} is not the minimum, then (5.4) is true. It implies that $r(v) < r^*(v)$, otherwise $r(v) = r^*(v)$, thus $r^*(u) < r(u)$, which contradicts $r \leq r^*$. Therefore, $r \leq r^*$ is kept after $r(v)$ is increased by 1. Together with Lemma 5.2.2, $r \leq r^*$ is kept before we reach r^*. The fact that $\neg B(r)$ upon termination implies that we have already reached r^* and went beyond it. Consequently, the value recorded in ϕ^{opt} is the minimum clock period. □

5.2.3. Additional termination criterion

From Section 5.2.2, we know that if a valid retiming is not optimal, then (5.4) is true and we will increase $r(v)$ by 1. A key observation is that if $r\big(F(v)\big)$ is increased before the next increase of $r(v)$, then another increase of $r(v)$ is necessitated because the increase of $r\big(F(v)\big)$ cancels the previous increase of $r(v)$ and makes (5.4) true again. The relation between $r\big(F(v)\big)$ and $r(v)$ is similar to that between $T\big(F(v)\big)$ and $T(v)$ where any increase of $T\big(F(v)\big)$ will be propagated to $T(v)$ unless the path between them ceases to be the longest.

In fact, the same relation exists between $r(v)$ and $r(u)$ when (5.1) is true, and between $r(u)$ and $r(x)$ when (5.2) is true, and between $r(v)$ and $r(y)$ when (5.3) is true.

Therefore, we introduce another label $m : V \rightarrow V \cup \{\emptyset\}$, where $\emptyset$ is the default assignment, and define it as follows. If $r(u)$ is increased due to (5.1), then $m(u)$ is set to v; if $r(x)$ is increased due to (5.2), then $m(x)$ is set to u; if $r(y)$ is increased due to (5.3), then $m(y)$ is set to v; if $r(v)$ is increased due to (5.4), then $m(v)$ is set to $F(v)$. Notice that the definition of m-labeling is very similar to that in Section 4.3.4 of Chapter 4. Similar results follow.

First of all, we can formulate the relation between $r\big(m(i)\big)$ and $r(i)$ for all $i \in V$ in the following lemma.

Lemma 5.2.3. *It is true before we reach an optimal retiming r^* that*

$$\Big(\forall i \in V, m(i) \in V : r^*\big(m(i)\big) - r\big(m(i)\big) \leq r^*(i) - r(i)\Big).$$

Proof. By the definition of the m-labeling, $m(i) \in V$ only if $r(i)$ has ever been increased from 0, $\forall i \in V$. For a particular vertex i, we will show that the inequality is kept after the first increase of $r(i)$ and continues to hold before an optimal r^* is reached.

Consider the first time that $r(i)$ is increased. Suppose it is due to (5.1) on edge (i, j). Then $m(i)$ will be set to j, and (5.1) becomes $r\big(m(i)\big) - r(i) > r^*\big(m(i)\big) - r^*(i)$, or $r^*\big(m(i)\big) - r\big(m(i)\big) < r^*(i) - r(i)$. After $r(i)$ is increased, we have $r^*\big(m(i)\big) - r\big(m(i)\big) \leq r^*(i) - r(i)$. Similar arguments apply to other cases due to the occurrences of (5.2)-(5.4).

Even if $r\big(m(i)\big)$ may be increased thereafter, the inequality will remain true until the next increase of $r(i)$, when $m(i)$ will be assigned to a new vertex which may or may not differ from the previous assignment. By the same case study as above, we can show that the inequality will continue to hold. By induction, the lemma is true. □

In addition, if there exists a sequence of vertices x_i, $i = 0, 1, \ldots, k-1$, such that $x_i = m(x_{i+1})$ and $x_k = x_0$, we refer to it as an m-cycle. The next theorem shows that the appearance of an m-cycle is an evidence that we have reached the optimal clock period.

Theorem 5.2.3. *If an m-cycle appears, then we have reached the optimal clock period.*

Proof. Suppose the last m assignment is $m(x_0) = x_{k-1}$, by setting which the m-labeling forms a cycle, i.e., a sequence of vertices x_i, $i = 0, 1, \ldots, k-1$, such that $x_i = m(x_{i+1})$ and $x_k = x_0$.

For the sake of contradiction, we assume that an optimal r^* is not reached yet. Therefore, by Lemma 5.2.3 and $m(x_0) = x_{k-1}$, we have $r^*(x_{k-1}) - r(x_{k-1}) \leq r^*(x_0) - r'(x_0)$ after $r(x_0)$ is

increased to $r'(x_0)$ due to any of (5.1)-(5.4). It implies that $r^*(x_{k-1})-r(x_{k-1}) < r^*(x_0)-r(x_0)$ before the increase. On the other hand, Lemma 5.2.3 guarantees that $r^*\big(m(x_i)\big)-r\big(m(x_i)\big) \leq r^*(x_i) - r(x_i)$, $1 \leq i \leq k-1$. It follows that $r^*(x_0) - r(x_0) \leq r^*(x_{k-1}) - r(x_{k-1})$, which is a contradiction. Therefore, an optimal retiming is already reached. □

Based on Theorem 5.2.3, the algorithm will terminate if an m-cycle is found. Since checking m-cycle is carried out after every r increase and the total number of r increases is no larger than $|V|^2$, the complexity of checking m-cycle is at most $O(|V|^3)$. Therefore, adding m-cycle checking in the algorithm will not affect the worst case complexity $O(|V|^2|E|)$ in Theorem 5.2.2.

5.3. Example

In this section we use an example in Figure 5.3 to show how the proposed algorithm finds the minimum period and builds m-labeling to certify the optimality.

Figure 5.3(a) shows the original sequential circuit (taken from [**78**]). It has five combinational logic blocks connected in a ring and two edge-triggered registers. The pair of integers in each block gives the maximum and the minimum propagation delays of the data through that block. For example, whenever data propagate through block A, they always require at least 1 time unit and never more than 10 time units. For simplicity, each register is assumed to have zero setup time and a hold time H of 4. The steps of the algorithm are shown by the sequence of figures (b)-(f) in Figure 5.3.

Starting with the original circuit, i.e., $r(A) = r(B) = r(C) = r(D) = r(E) = 0$, "ValidHold" is called. Since no edge has more than one register, $P0$ and $P1$ are satisfied. Then it calculates the earliest output arrival time of each block, which is shown beside each block. Since none of the earliest arrival times is smaller than H, the current r is a valid

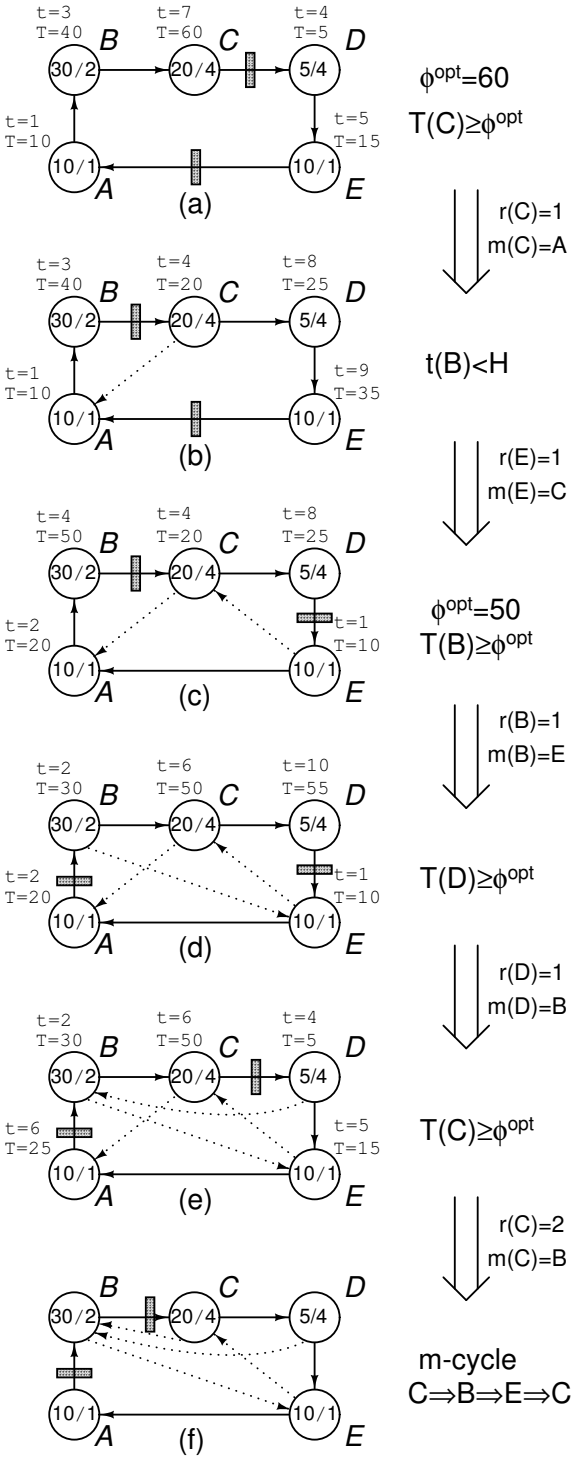

Figure 5.3. Applying the proposed algorithm on an example.

retiming. The algorithm proceeds to compute the latest output arrival time of each block and assign the maximum of them, 60, to ϕ^{opt}.

In order to reduce ϕ^{opt}, any block whose output arrival time is at least ϕ^{opt} will have its r value increased. In our case we have $T(C) = \phi^{\text{opt}}$, thus $r(C)$ is increased by 1, which means that a register is moved from the output of C to its input. This increment is accompanied by setting $m(C) = F(C) = A$ (because a longest combinational path to C starts from A), which is represented by a dotted edge in Figure 5.3(b). After that, "ValidHold" is called again. In this case, a hold violation at B is identified since $t(B) = 3 < H = 4$.

To fix the violation, the algorithm increases $r(E)$ by 1. In other words, the register on edge (E, A) is moved ahead to increase $t(B)$. As an accompaniment, $m(E)$ is set to C to indicate that future increases in $r(C)$ will be propagated to $r(E)$ to avoid hold violation on path from E to C. It turns out that the hold violation is fixed and we obtain another valid retiming whose $T_{max} = 50$, as shown in Figure 5.3(c). Therefore, ϕ^{opt} is updated to 50.

Next, the algorithm attempts to reduce ϕ^{opt} by successively increasing $r(B)$, $r(D)$ and $r(C)$ in Figure 5.3(d), 5.3(e), and 5.3(f) respectively until the appearance of an m-cycle. It is an evidence that there is no valid retiming with a period less than 50, hence 50 is the optimal period and it is realized in Figure 5.3(c).

It can be seen from the example that m-labeling can only form a forest before the optimal period is reached. In addition, the process of adjusting r to improve ϕ^{opt} and the process of building m to certify optimality are carried out simultaneously. As a result, the algorithm can keep reducing ϕ^{opt} until the optimality is certified in the last iteration.

5.4. Experimental results

We implemented the algorithm on a Sun Ultra 10 machine. Our test files were generated from ISCAS-89 benchmark suite using ASTRA [**84**]. Each gate was assigned a maximum delay between 2 and 100; the minimum delay was equal to the maximum delay. For simplicity, we assumed zero setup time. Hold time H was set to 2.

Reported in Table 5.1 are results for large circuits. Column "T_{max}" lists the maximum combinational delays of the original circuits. Note that T_{max} may not be a feasible clock period since the original circuit may have more than one register on an edge. The minimum periods by retiming when only setup constraint is considered are shown in column "ϕ_S". Taking both setup and hold constraints into account, our proposed algorithm computes the minimum periods in column "ϕ_{SH}", where we use "NO" to indicate that there is no valid retiming. Cases with $\phi_S \neq \phi_{SH}$ are highlighted. For runtime comparison, we obtained the source code of the algorithm in [**109**], which considers only setup constraint for minimum period retiming, and reported the runtime in column "[**109**]". The runtime of our algorithm is listed in column "ours". We then compute the ratio between them for each case[1] and obtain the arithmetic (geometric) mean of all the ratios in row "arith" ("geo").

Table 5.1 reveals two things. Firstly, more than half of the circuits have $\phi_{SH} \neq \phi_S$. In other words, the minimum period retimings given by the algorithm in [**109**] for these circuits have hold violations. The difference between ϕ_{SH} and ϕ_S could be significant even for small $H = 2$, e.g., $(\phi_{SH} - \phi_S)/\phi_S = 53.8\%$ for "s838.1". Two circuits "s13207.1" and "s38584.1" do not even have a valid retiming because of reconvergent paths. It happens when one path from u to v has extra registers and thus requires $r(u) > r(v)$ while another path from u to v has no register and requires $r(u) \leq r(v)$, which contradict each other. Secondly, the

[1] If the runtime of one algorithm is "0.00", the case is ignored.

Table 5.1. Experimental Results

name	#gates	T_{max}	period		t(sec)	
			ϕ_S	ϕ_{SH}	[**109**]	ours
s838.1	288	94	52	**80**	0.00	0.00
s1238	428	110	110	110	0.01	0.00
s1423	490	332	254	**280**	0.01	0.00
s1494	558	166	164	**166**	0.01	0.01
s5378	1004	92	92	92	0.01	0.01
s9234	2027	178	162	162	0.08	0.10
s9234.1	2027	178	162	162	0.09	0.11
s13207.1	2573	286	270	**NO**	0.08	0.01
s15850	3448	372	154	**210**	0.17	0.21
s15850.1	3448	372	290	290	0.17	0.20
s35932	12204	138	124	**138**	1.23	0.84
s38417	8709	220	112	**120**	0.32	0.75
s38584.1	11448	306	290	**NO**	0.37	0.03
arith					1	1.01X
geo					1	0.72X

proposed algorithm is efficient. Although it checks both setup and hold constraints, it is as efficient as the algorithm in [**109**] on average.

5.5. Conclusion

A new algorithm is presented for retiming edge-triggered circuits to achieve the minimum clock period under both setup and hold constraints. The worst-case running time of the algorithm is $O(|V|^2|E|)$, which is asymptotically more efficient than the best known result $O(|V|^3|E| \lg |V|)$ in [**78**]. Experimental results confirm the efficiency of the algorithm.

CHAPTER 6

Clock Skew Scheduling with Delay Padding for Prescribed Skew Domains

In a sequential circuit, due to the differences of interconnect delays in the clock distribution network, clock signals do not arrival at all flip-flops at the same time. The consequent differences in clock arrival times are also known as *clock skews*. Since the setup and hold constraints of a sequential circuit are complicated by clock skews, an approach that has been followed by [**46, 47, 104, 79, 80**] is to deliberately design the clock distribution network so as to ensure zero clock skew.

Clock skew scheduling [**35**], on the other hand, views clock skew as a manageable resource rather than a liability. It intentionally introduces skews to flip-flops to improve the circuit performance. The designated skews are then implemented by specific layout of the clock distribution network. However, in practice, a skew schedule with a large set of arbitrary values cannot be realized in a reliable manner. This is because the implementation of dedicated delays using additional buffers and interconnects is highly susceptible to intra-die variations of process parameters.

Instead of tuning clock skews of flip-flops, retiming [**53**] physically relocates flip-flops to balance the delays without changing the functionality of the circuit. It was observed in [**35**] that retiming and clock skew scheduling are discrete and continuous optimizations with the same effect. The equivalence between retiming and skew has been used in previous researches [**71, 64, 11, 84**]. Although retiming is a powerful sequential optimization

technique, its practical use is limited due to the impact on the verification methodology, i.e., equivalence checking and functional simulation. Furthermore, the use of retiming for maximum performance may cause a steep increase in the number of flip-flops [**34**], requiring a larger effort for clock distribution and resulting in higher power consumption.

Recently, multi-domain clock skew scheduling was proposed in [**81**]. Multiple clocking domains are routinely applied in designs to realize several clocking frequencies and also to address specific timing requirements. For example, a special clocking domain that delivers a phase-shifted clock signal to the flip-flops close to the chip inputs and outputs is regularly used to achieve timing closure for ports with extreme constraints on their arrival and required times. The motivation behind the multi-domain skew scheduling is based on the fact that large phase shifts between clocking domains can be implemented reliably by using dedicated, possibly expensive circuit components such as "structured clock buffers" [**9**], adjustments to the PLL circuitry, or simply by deriving the set of phase-shifted domains from a higher frequency clock using different tapping points of a shift flip-flop. In [**81**], Ravindran *et al.* showed that a clock skew schedule using a few domains combined with a small within-domain latency can reliably implement the full optimization potential of clock skew scheduling. They proposed an algorithm based on a branch-and-bound search to assign flip-flops to clock domains and used a modified Burns's algorithm [**6**] to compute the skews.

Although the algorithm in [**81**] computed for a user-given number of domains the optimal skew for each domain, the user had no control on the distribution of the domains. Nor did the algorithm consider delay padding [**93**], which is a technique that fixes hold violations by inserting extra delays on short paths without increasing the delay on any long path. In other words, the clock period obtained by the algorithm in [**81**] may be sub-optimal if delay padding is allowed, as demonstrated in [**49**].

In this chapter we formulate the clock skew scheduling problem on a user-given finite set of prescribed clock domains. For example, one can require the skews of flip-flops to be either zero or half the clock period. We propose a polynomial-time algorithm that finds an optimal domain assignment for each flip-flop such that the clock period is minimized with possible delay padding. The obtained skew schedule respects the user requirement. We then consider how to insert extra paddings such that both setup and hold constraints are satisfied under the minimal clock period. We show that the existence of such a padding solution is guaranteed, and present an approach to find a padding by network flow technique. Experimental results confirm the efficiency of our algorithm.

The rest of the chapter is organized as follows. Section 6.1 presents a motivation example and the problem formulation. Notations and constraints are explained in Section 6.2. Our algorithm is elaborated in Section 6.3. Experimental results are presented in Section 6.4, followed by conclusions in Section 6.5.

6.1. Motivation and problem formulation

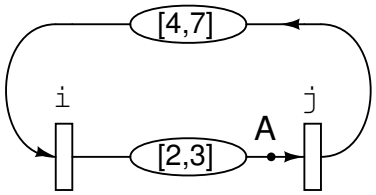

Figure 6.1. Effect of clock skew scheduling and delay padding on circuit performance.

We now use an example in Figure 6.1 to illustrate the effect of clock skew scheduling and delay padding on circuit performance. In this example we have two flip-flops i and j triggered at the falling edge of a clock. The ellipses between the flip-flops represent the combinational blocks with their minimum and maximum delays. Suppose that the setup

and hold times are all zero, and that the skew to each flip-flop can be either zero or half the period. In order for the circuit to operate under a given period T, the following conditions must be satisfied, where the first two are from the setup constraint and the other two are from the hold constraint.

$$skew(i) + 3 - T \leq skew(j)$$

$$skew(j) + 7 - T \leq skew(i)$$

$$skew(i) + 2 \geq skew(j)$$

$$skew(j) + 4 \geq skew(i)$$

Depending on the skew assignment, we have three cases. Firstly, $skew(i) = skew(j)$, which leads to $T \geq 7$. Secondly, $skew(i) = T/2$ and $skew(j) = 0$, we have $6 \leq T \leq 8$. For $skew(i) = 0$ and $skew(j) = T/2$, the setup constraint requires $T \geq 14$ while the hold constraint needs $T \leq 4$. In other words, there is no such a T satisfying both constraints. However, if we insert an extra delay of 5 at point A, then the minimum and maximum delays from i to j become 7 and 8 respectively, which results in a feasible period of $T = 14$.

The above example reveals two things. Firstly, assigning skews to flip-flops may help to reduce the period of a circuit. On the other hand, cautions should be taken when choosing the skews since the circuit may end up having no feasible period at all. Secondly, delay padding can be used to remedy a skew assignment so that it permits feasible periods after the insertion of extra delays. In some cases, delay padding is required to reach a smaller period. For the above example, if the minimum delay from j to i is not 4 but 2, then a delay of 1 needs to be inserted on the minimum delay path to obtain the optimal period 6. This motivates us to solve a problem formulated as follows.

Problem 6.1.1 (Optimal Skew Scheduling)**.**

Given a sequential circuit and a finite set of prescribed skew domains, find a domain assignment for each flip-flop such that the circuit satisfies both setup and hold constraints with possible delay padding under the minimal clock period.

For simplicity, we assume that flip-flops are triggered at the same clock edge of a single phase clock. However, the proposed approach can be extended to multiple clock phases.

6.2. Notations and constraints

Suppose that we are given N skew domains. Without loss of generality, we assume that the skew values of the N domains are $s_0T, s_1T, ..., s_{N-1}T$ with respect to the period T, where $s_0, s_1, ..., s_{N-1}$ are constants between 0 and 1 in the increasing order, i.e., $0 = s_0 < s_1 < ... < s_{N-1} < 1$. In particular, the domains are *evenly distributed* if $s_n = \frac{n}{N}T$, for all $0 \leq n \leq N-1$.

A directed graph $G = (V, E)$ is used to represent a sequential circuit, where V is the set of gates and flip-flops, and E is the set of interconnects. Each gate $v \in V$ has a maximum delay $D(v)$ and a minimum delay $d(v)$. Each interconnect $e \in E$ has a delay $w(e)$. Delay padding increases $w(e)$ by inserting extra delays on e. For any combinational path $p = u \rightsquigarrow v$, we use $D(p)$ to represent the maximum delay along p without padding, which is the sum of the constituent interconnect delays and maximum gate delays, except for $D(u)$. The minimum delay along p without padding is denoted by $d(p)$. With extra paddings on p, the maximum and minimum delays become $\Delta(p)$ and $\delta(p)$ respectively. Note that

$$\Delta(p) - \delta(p) = D(p) - d(p).$$

We also construct a timing graph $G_t = (V_t, E_t)$ of G as follows. Let $V_t \subset V$ be the set of flip-flops in the circuit. An edge (i, j) is introduced in E_t if there is a combinational path $p \in G$ from flip-flop i to flip-flop j. We define

$$D(i,j) \stackrel{\triangle}{=} \max_{p \in G: i \rightsquigarrow j} D(p), \qquad \Delta(i,j) \stackrel{\triangle}{=} \max_{p \in G: i \rightsquigarrow j} \Delta(p)$$
$$d(i,j) \stackrel{\triangle}{=} \min_{p \in G: i \rightsquigarrow j} d(p), \qquad \delta(i,j) \stackrel{\triangle}{=} \min_{p \in G: i \rightsquigarrow j} \delta(p)$$

In other words, $D(i, j)$ and $d(i, j)$ are the maximum and minimum combinational delays from i to j without padding, respectively. They become $\Delta(i, j)$ and $\delta(i, j)$ with padding. We say that p is a *long* path from i to j if $\Delta(p) = \Delta(i, j)$; p is a *short* path if $\delta(p) = \delta(i, j)$. For all $i \in V_t$, we use $X(i)$ and $H(i)$ to denote the setup and hold times at flip-flop i respectively. A label $l : V_t \rightarrow \{0, ..., N - 1\}$ is introduced to represent the index of the domain that a flip-flop is assigned to.

Using these notations, the setup and hold constraints under a given period T can be formulated as follows.

$$0 \leq l(i) \leq N - 1, \quad \forall i \in V_t \tag{6.1}$$

$$s_{l(i)}T + \Delta(i,j) - s_{l(j)}T \leq T - X(j), \quad \forall (i,j) \in E_t \tag{6.2}$$

$$s_{l(i)}T + \delta(i,j) - s_{l(j)}T \geq H(j), \quad \forall (i,j) \in E_t \tag{6.3}$$

A partial order ($\leq$) can be defined between two labels l and l' as follows.

$$l \leq l' \stackrel{\triangle}{=} l(i) \leq l'(i), \quad \forall i \in V_t$$

We say that T is a *feasible period* if and only if we can find an l and a delay padding such that (6.1)-(6.3) are satisfied under T. The optimal skew scheduling problem asks for the minimal feasible T, with possible padding insertion.

6.3. Algorithm

In order to find the minimum feasible T, we first compute a lower bound T_{lb} for it in Section 6.3.1. Then we observe that for any l satisfying (6.1), since $s_n < 1$, $\forall 0 \leq n \leq N-1$, we have $1 + s_{l(j)} - s_{l(i)} > 0$, $\forall (i,j) \in E_t$. Thus, the setup constraint (6.2) can be written as $T \geq \big(\Delta(i,j) + X(j)\big)/\big(1 + s_{l(j)} - s_{l(i)}\big)$. Together with $\Delta(i,j) \geq D(i,j)$, it characterizes a minimal period T_S under setup constraint only. We propose an algorithm in Section 6.3.2 to compute $T^* = \max(T_{lb}, T_S)$ and the corresponding l^*. We then show in Section 6.3.3 that there always exists a padding solution such that both setup and hold constraints are satisfied under T^* and l^*. Therefore, T^* and l^* are the solution to the optimal skew scheduling problem.

6.3.1. Lower bound for feasible period

Consider any combinational path p from $i \in V_t$ to $j \in V_t$. Since $\Delta(i,j) \geq \Delta(p)$ and $\delta(i,j) \leq \delta(p)$ by the definition of $\Delta(i,j)$ and $\delta(i,j)$, (6.2)-(6.3) imply that

$$s_{l(i)}T + \Delta(p) - s_{l(j)}T \leq T - X(j)$$

$$s_{l(i)}T + \delta(p) - s_{l(j)}T \geq H(j)$$

Subtracting the second one from the first yields $\Delta(p) - \delta(p) \leq T - X(j) - H(j)$. Since $\Delta(p) - \delta(p) = D(p) - d(p)$, we have a lower bound for T in the next lemma.

Lemma 6.3.1. *A feasible clock period T must satisfy*

$$T \;\geq\; T_{lb} \stackrel{\triangle}{=} \max_{(i,j)\in E_t, p\in E: i\rightsquigarrow j} \big(D(p) - d(p) + X(j) + H(j)\big)$$

To compute T_{lb}, let $\theta(v)$ be the difference between the maximum and the minimum delays at gate v, defined as

$$\theta(v) \stackrel{\triangle}{=} D(v) - d(v), \quad \forall v \in V - V_t \tag{6.4}$$

For flip-flop j, we define

$$\theta(j) \stackrel{\triangle}{=} X(j) + H(j), \quad \forall j \in V_t$$

Let $\Theta(v)$ be the length of the longest combinational path terminating at v with respect to θ, i.e.,

$$\Theta(v) \stackrel{\triangle}{=} \max_{combinational\ p\in E:\rightsquigarrow v} \theta(p), \qquad \forall v \in V \tag{6.5}$$

Then, finding T_{lb} is equivalent to computing the maximum $\Theta(j)$, $\forall j \in V_t$, which can be done by longest path computation in $O(|E| + |V| \log |V|)$ time [**23**].

6.3.2. Minimum period under setup constraint

We use T_S to denote the minimal period under which the setup constraint is satisfied without padding, i.e.,

$$0 \leq l(i) \leq N - 1, \quad \forall i \in V_t \tag{6.1}$$

$$s_{l(i)}T + D(i,j) - s_{l(j)}T \leq T - X(j), \quad \forall (i,j) \in E_t \tag{6.6}$$

Let $T^* = \max(T_{lb}, T_S)$. We use l^* to denote a domain assignment satisfying (6.1) and (6.6) under T^*. For example, the corresponding domain assignment under T_S is such an l^*.

To find an l^*, we start with $l(i) = 0$, $\forall i \in V$, and obtain $T = \max_{(i,j)\in E_t} \frac{D(i,j)+X(j)}{1+s_{l(j)}-s_{l(i)}}$ since $1+s_{l(j)}-s_{l(i)} > 0$, $\forall (i,j) \in E_t$. If $T \leq T_{lb}$, then we have $T^* = T_{lb}$, and thus the current l is an l^*. Otherwise, let $(x,y) \in E_t$ be the edge that determines T, i.e., $T = \frac{D(x,y)+X(y)}{1+s_{l(y)}-s_{l(x)}}$. Suppose $T > T^*$, it follows that $D(x,y) + X(y) > \big(1 + s_{l(y)} - s_{l(x)}\big)T^*$. On the other hand, T^* and l^* should satisfy the setup constraint on (x,y), i.e., $\big(1 + s_{l^*(y)} - s_{l^*(x)}\big)T^* \geq D(x,y) + X(y)$. As a result, we have

$$s_{l^*(y)} - s_{l^*(x)} > s_{l(y)} - s_{l(x)} \tag{6.7}$$

We can move l closer to l^* by increasing $l(y)$. The amount of increase should only be 1 since we do not want to over-adjust l. This process is iterated until the optimality of T is certified. We present the pseudocode in Figure 6.2.

MinPeriod(G_t, T^*, l^*)

$T^*, l \leftarrow \infty, 0$;
`While` $\Big(T^* > T_{lb} \ \wedge \ \big(\forall i \in V_t : l(i) < N\big)\Big)$ `do`
  `Extract` (x,y) `from` E_t `with max` $\frac{D(x,y)+X(y)}{1+s_{l(y)}-s_{l(x)}}$;
  $T \leftarrow \frac{D(x,y)+X(y)}{1+s_{l(y)}-s_{l(x)}}$;
  `If` $(T < T^*)$ `then`
    $T^*, l^* \leftarrow \max(T, T_{lb}), l$;
  $l(y) \leftarrow l(y) + 1$;

Figure 6.2. Pseudocode of minimum period computation.

The next lemma states an invariant that is kept throughout "MinPeriod".

Lemma 6.3.2. *$l \leq l^*$ is kept during the execution of "MinPeriod" in Figure 6.2 before we reach an l^*.*

Proof. First of all, $l \leq l^*$ before we enter the while loop since we initialize $l(i) = 0$, $\forall i \in V_t$. What remains is to show that $l \leq l^*$ is preserved after $l(y)$ is increased by 1 for some $y \in V_t$ until T^* is reached.

Assume that $T > T^*$, thus (6.7) is true. Since $l(x) \leq l^*(x)$ and $l(y) \leq l^*(y)$ due to $l \leq l^*$, we have $l(y) < l^*(y)$, otherwise $l(y) = l^*(y)$, which leads to $l(x) > l^*(x)$, which is a contradiction. Therefore, $l \leq l^*$ is kept after the increase of $l(y)$. The lemma is true. □

The correctness and complexity of "MinPeriod" is given in the following theorem.

Theorem 6.3.1. *The procedure "MinPeriod" will terminate in $O(N|V_t|B_t \log |E_t|)$ time, where B_t is the maximum incoming and outgoing degrees of the vertices in V_t. Upon termination, it gives $T^* = \max(T_{lb}, T_S)$, and an l^* satisfying (6.1) and (6.6) under T^*.*

Proof. The outer while loop cannot be executed more than $(N-1)|V_t|$ times since each traversal results in an increase in $l(y)$ for some $y \in V_t$. The complexity of extracting the edge (x, y) in E_t with the maximum $\frac{D(x,y)+X(y)}{1+s_{l(y)}-s_{l(x)}}$ is $O(\log |E_t|)$ if we choose Fibonacci heap [**23**]. After $l(y)$ is increased by 1, we need to adjust the values of $\frac{D(i,y)+X(y)}{1+s_{l(y)}-s_{l(i)}}$ for all incoming edges $(i, y) \in E_t$ to y, and the values of $\frac{D(y,j)+X(j)}{1+s_{l(j)}-s_{l(y)}}$ for all outgoing edges $(y, j) \in E_t$ from y, which takes $O(B_t \log |E_t|)$ time. Therefore, the overall complexity is $O(N|V_t|B_t \log |E_t|)$.

When it terminates, we have either $T^* = T_{lb}$ or $l(y) = N > l^*(y)$ for some $y \in V_t$. In the first case, we have $T_S \leq T_{lb}$, thus $T^* = \max(T_{lb}, T_S)$ is true. By Lemma 6.3.2, the second case implies that we have already reached an l^* and went beyond it. Therefore, the obtained

T^* is T_S. Since $T^* > T_{lb}$, $T^* = \max(T_{lb}, T_S)$ is also true. In both cases, the obtained l^* satisfies (6.1) and (6.6) under T^*. □

Since padding increases $D(i,j)$, we have $\Delta(i,j) \geq D(i,j)$, $\forall (i,j) \in E_t$. Therefore, T^* is a lower bound for any feasible period with padding.

6.3.3. Padding for hold constraint

Given T^* and l^* from "MinPeriod", we will check the hold constraint (6.3) without padding. If we have a hold violation at some $j \in V_t$, it means that there is a short path p from $i \in V_t$ to j such that $s_{l^*(i)}T^* + d(p) - s_{l^*(j)}T^* < H(j)$. Intuitively, if p has an interconnect that does not lie on any long path, then we can insert extra delays on it to fix the hold violation. The following lemma [**93**] provides the condition under which the existence of such an interconnect is guaranteed.

Lemma 6.3.3. *Let p be a short path to $j \in V_t$, where a hold violation occurs under T^* and l^*. There exists an edge e on p such that e does not lie on any long path if $D(p) - d(p) \leq T^* - X(j) - H(j)$.*

The next result is a corollary of the above lemma.

Corollary 6.3.3.1. *There exists a delay padding solution under T^* and l^* satisfying both setup and hold constraints.*

Proof. Since $T^* \geq T_{lb}$, the definition of T_{lb} in Lemma 6.3.1 implies that $D(p) - d(p) \leq T^* - X(j) - H(j)$ for all path p to j, $\forall j \in V_t$. Consequently, if a hold violation occurs at j under T^* and l^*, Lemma 6.3.3 ensures that we can successively identify interconnects for padding insertion without affecting any long path until the hold violation is fixed. □

Based on Corollary 6.3.3.1 and the fact that T^* is a lower bound for any feasible period with padding, we know that T^* and l^* are the solution to the optimal skew scheduling problem.

To find a padding solution, we can treat flip-flop outputs as primary inputs (PIs) and flip-flop inputs as primary outputs (POs), and find a padding for each individual combinational component. To ease the presentation, we will focus on padding for a combinational component $G_c = (V_c, E_c) \subseteq G$ under T^* and l^*.

For each gate $v \in G_c$, we use $A(v)$ and $a(v)$ to denote the latest and the earliest arrival times at the output of v, which are the longest and the shortest combinational delays from PIs to v, respectively. Let $p(u, v)$ be the padding on $(u, v) \in G_c$. The problem (denoted as MP) of finding a minimum delay padding under T^* and l^* can be formulated as follows [**93**].

$$MP: \quad \text{Minimize} \sum_{(u,v)\in G_c} p(u,v)$$

$$a(i) = s_{l^*(i)}T^*, \quad \forall i \in PI \tag{6.8}$$

$$a(v) \leq a(u) + d(v) + w(u,v) + p(u,v), \quad \forall (u,v) \in G_c \tag{6.9}$$

$$a(j) \geq H(j) + s_{l^*(j)}T^*, \quad \forall j \in PO \tag{6.10}$$

$$A(i) = s_{l^*(i)}T^*, \quad \forall i \in PI \tag{6.11}$$

$$A(v) \geq A(u) + D(v) + w(u,v) + p(u,v), \quad \forall (u,v) \in G_c \tag{6.12}$$

$$A(j) \leq T^* - X(j) + s_{l^*(j)}T^*, \quad \forall j \in PO \tag{6.13}$$

$$p(u,v) \geq 0, \quad \forall (u,v) \in G_c \tag{6.14}$$

The conditions (6.8)-(6.9) characterize the earliest arrival times at gate outputs. (6.10) is the hold constraint at POs. The conditions (6.11)-(6.12) characterize the latest arrival times. (6.13) is the setup constraint. Inequality (6.14) enforces nonnegative padding.

We say that p is a *feasible padding* if and only if there exist a and A such that (6.8)-(6.14) are satisfied under T^* and l^*. The *feasible region* of MP contains all the feasible paddings. Since both the objective and the constraints are linear, MP can be solved by any linear programming solver.

In the following, we will show how to find a "reasonably good" padding using network flow technique, which is more efficient than linear programming.

First of all, we observe that subtracting (6.9) from (6.12) yields

$$A(v) - a(v) \geq A(u) - a(u) + \theta(v), \qquad \forall (u,v) \in G_c$$

where $\theta(v) = D(v) - d(v)$ is defined in (6.4). Therefore, $A(v) - a(v) \geq \Theta(v)$ by the definition of $\Theta(v)$ in (6.5). Given a feasible padding, we can insert extra delays on each edge such that (6.12) becomes an equality. When (6.12) is an equality, we have

$$A(v) - a(v) = \max_{(u,v) \in G_c} A(u) - a(u) + \theta(v), \quad \forall v \in G_c$$

As a result, there exists a combinational path p to v such that $A(v) - a(v) = \theta(p) \leq \Theta(v)$. Together with $A(v) - a(v) \geq \Theta(v)$, we have $A(v) - a(v) = \Theta(v)$, $\forall v \in G_c$. Since $\Theta(v)$ can be obtained by longest path computation with respect to θ, we can replace $A(v)$ by $\Theta(v) + a(v)$ in the conditions (6.11)-(6.13) to simplify the problem.

Lemma 6.3.4. *The minimum padding problem MP with (6.12) being an equality is equivalent to the following problem (EMP):*

$$
\begin{aligned}
EMP: \qquad & \text{Minimize} \sum_{(u,v)\in G_c} p(u,v) \\
a(i) &= s_{l^*(i)}T^*, \quad \forall i \in PI, && (6.8)\\
a(v)+\Theta(v) &= a(u)+\Theta(u)+D(v)+w(u,v)+p(u,v), \quad \forall (u,v)\in G_c && (6.15)\\
a(j) &\geq H(j)+s_{l^*(j)}T^*, \quad \forall j \in PO && (6.10)\\
a(j)+\Theta(j) &\leq T^*-X(j)+s_{l^*(j)}T^*, \quad \forall j \in PO && (6.16)\\
p(u,v) &\geq 0, \quad \forall (u,v)\in G_c && (6.14)
\end{aligned}
$$

Proof. Since both MP and EMP have the same objective, what remains is to show that they have the same feasible region when (6.12) is an equality.

Suppose that p is a feasible padding to EMP. Since $\Theta(v) \geq \Theta(u)+\theta(v)$ by (6.5), we know that (6.15) implies (6.9). For $i \in PI$, since $\Theta(i)=0$ and (6.8), we have $A(i) = \Theta(i)+a(i) = s_{l^*(i)}T^*$, which is (6.11). Given that $A(v)=\Theta(v)+a(v)$, (6.15) is an equality form of (6.12). (6.16) is the same as (6.13) since $A(j)=\Theta(j)+a(j)$. Therefore, p is also a feasible padding to MP when (6.12) is an equality. Similarly, if p is feasible to MP with (6.12) being an equality, p is also feasible to EMP, which concludes our proof. □

Note that EMP is a dual of a min-cost flow problem [**3**]. Let $\bar{p}$ be an optimal solution to EMP. Since $\bar{p}$ satisfies the setup constraint, any p with $p(u,v) \leq \bar{p}(u,v)$, $\forall (u,v)\in G_c$, should also satisfy the setup constraint. Therefore, we use $\bar{p}$ as an upper bound for p and solve the minimum padding problem with the hold constraint only. This is formulated as

the following problem (BMP).

$$
\begin{aligned}
BMP: \quad & \text{Minimize} \sum_{(u,v)\in G_c} p(u,v) \\
a(i) &= s_{l^*(i)}T^*, \quad \forall i \in PI && (6.8)\\
a(v) &\leq a(u) + d(v) + w(u,v) + p(u,v), \quad \forall (u,v) \in G_c && (6.9)\\
a(j) &\geq H(j) + s_{l^*(j)}T^*, \quad \forall j \in PO && (6.10)\\
0 &\leq p(u,v) \leq \bar{p}(u,v), \quad \forall (u,v) \in G_c && (6.17)
\end{aligned}
$$

Note that BMP is a dual of a convex-cost flow problem [**2**]. From [**3**] and [**2**], both EMP and BMP can be solved in polynomial time bounded by $O\big(|V_c||E_c| \log(|V_c|^2/|E_c|) \log(|V_c|T^*)\big)$. The next theorem provides the condition under which an optimal solution to BMP is also an optimal solution to MP.

Theorem 6.3.2. *If MP has an optimal solution p^* such that $p^*(u,v) \leq \bar{p}(u,v)$ for all $(u,v) \in G_c$, then an optimal solution to BMP is one such p^*.*

Proof. Since $p^* \leq \bar{p}$, p^* is feasible to BMP. Given that the feasible region of MP contains the feasible region of BMP, p^* is an optimal solution to BMP. Therefore, any optimal solution to BMP has the same amount of padding as p^*, and hence is an optimal solution to MP. □

When the condition in Theorem 6.3.2 does not hold, solving BMP only gives a feasible padding. However, our experiments show that the feasible padding we obtained is close to the minimum padding.

Our algorithm for optimal skew scheduling is presented in Figure 6.3. It first applies "MinPeriod" to compute T^* and l^*. Then, it finds a padding solution for each combinational component under T^* and l^*. The overall complexity is $O\big(N|V_t|B_t \log |E_t| + |V||E| \log(|V|^2/|E|) \log(|V|T^*)\big)$ by Theorem 6.3.1 and the complexity for solving EMP and BMP.

Input : A circuit $G = (V, E)$ and N skew domains.
Output: Optimal period T^* under domain assignment l^* and delay padding p.

```
Construct timing graph G_t from G;
MinPeriod(G_t, T*, l*);
For each combinational component G_c ⊆ G do
   Find padding p by solving BMP for G_c;
Return T*, l* and p;
```

Figure 6.3. Pseudocode of optimal skew scheduling algorithm.

6.4. Experimental results

We implemented the algorithm in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. Our test files were generated from ISCAS-89 benchmark suite using ASTRA [**84**]. Each gate was assigned a maximum delay between 2 and 100; the minimum delay was equal to the maximum delay. For simplicity, interconnect delays were set to zero. The circuits used are summarized in Table 6.1.

Without loss of generality, we used four evenly distributed skew domains, i.e., $s_n = nT/4$, $0 \leq n \leq 3$. Setup and hold times of each flip-flop were set to 2. Thus, $T_{lb} = 4$. The results are reported in Table 6.2. Column "$|s_n|$", $0 \leq n \leq 3$, lists the number of flip-flops that are assigned to domain s_n in the obtained optimal skew schedule. The sum of the setup time

Table 6.1. Sequential circuits from ISCAS-89

Circuit	$\|V\|$	$\|E\|$	$\|V_t\|$	$\|E_t\|$
s838	618	959	172	3160
s1196	560	1053	31	94
s1423	896	1407	239	11320
s5378	3080	4561	301	3068
s9234	6198	8593	601	11035
s9234.1	6176	8588	579	10843
s13207.1	9337	12702	1386	9803
s15850	11449	15408	1677	63184
s15850.1	11348	15370	1576	52504
s35932	21880	34403	5815	32049
s38417	25058	35012	2879	59054
s38584	26651	40431	7398	101700

and the maximum combinational delay of the original circuit is listed in Column "T_{ub}". It is an upper bound for T^*. The computed minimal period is listed in column "T^*". The running time of "MinPeriod" for finding T^* is reported in column "t(sec)" in seconds. The improvement ratio $(T_{ub} - T^*)/T_{ub}$ for each circuit is listed in column "impr%". We obtain the arithmetic (geometric) mean of all the ratios in row "arith" ("geo"). Once T^* and l^* are obtained, we solve EMP and BMP to get a padding solution under T^* and l^*. We compare the solution with the minimum padding computed by MOSEK solver [**1**]. The amount of padding and running time are listed in column "padding" and "time(sec)" respectively.

We can see from Table 6.2 that, except for "s5378", all circuits have smaller periods after skew scheduling with possible padding. The improvement could be significant, e.g., 42.9% in "s15850". The average improvement is 16.9%. In addition, "MinPeriod" is efficient. It takes only 0.59 second for the largest circuit "s38584". Although the padding solution to BMP is about 1.5X the minimum padding, the actual area overhead will be reasonably small since the delay padding is amortized over the whole circuit. For running time comparison, solving

BMP by network flow technique is much more efficient than solving MP by MOSEK. The average speed-up is more than one order of magnitude.

To see how skew scheduling helps to improve a retiming solution, we used the algorithm in Chapter 5 to compute a minimum period retiming under the setup and hold constraints, and then applied our algorithm on the retimed circuit. The results are reported in Table 6.3. We use "s838′" to denote the optimal retiming of "s838", and so on for other optimal retimings. Column "[**78**]" lists the minimal period computed by the retiming algorithm in [**78**], where we use "NO" to indicate that there is no feasible retiming. For circuits without feasible retiming, we obtained their min-period retimings under setup constraint only. We highlight the cases where the periods are further improved by our skew scheduling algorithm. Comparing Table 6.3 with Table 6.2, we observe the following results.

Firstly, half of the circuits have their minimal periods further reduced after skew scheduling with possible padding insertion. The improvement is 7.3% on average and up to 27.8%. This is significant considering that our algorithm is applied after a minimum period retiming. Two circuits "s13207.1" and "s38584" do not even have a feasible retiming due to the discrete nature of retiming. It happens when there exist reconvergent paths where the retiming requirements from different paths contradict each other. However, by skew scheduling and delay padding, we are able to find the minimal periods for them.

Secondly, it appears that the optimal skew schedule for the retimed circuit uses less number of domains. The number of flip-flops that are assigned to domains other than s_0 is also reduced. In other words, applying skew scheduling after retiming improves the implementability of the obtained optimal skew schedule.

Thirdly, in all test cases, except for "s838", applying skew scheduling on the retimed circuit results in less amount of delay padding than the original circuit. In addition, the solution to BMP is closer to (about 1.1X) the minimum padding of the retimed circuit.

6.5. Conclusion

We present a polynomial time algorithm that finds an optimal skew schedule over a finite set of prescribed skew domains such that the period is minimized with possible delay padding. We show that the existence of a padding solution under the minimal period is guaranteed and propose an approach to find a padding solution by network flow technique. Experimental results validate the efficiency of our algorithm.

Table 6.2. Optimal skew schedule with delay padding

Circuit	$\|s_0\|$	$\|s_1\|$	$\|s_2\|$	$\|s_3\|$	T_{ub}	T^*	impr%	t(sec)	padding		time(sec)	
									MOSEK	ours	MOSEK	ours
s838	94	28	43	7	96.0	65.6	31.6%	0.01	56.8	56.8	0.20	0.01
s1196	30	1	0	0	102.0	100.0	2.0%	0.00	25.0	87.0	0.45	0.01
s1423	150	70	19	0	334.0	256.0	23.4%	0.07	966.0	1425.4	0.49	0.03
s5378	301	0	0	0	94.0	94.0	0.0%	0.01	0.0	0.0	2.65	0.09
s9234	597	4	0	0	180.0	164.0	8.9%	0.04	38.0	75.0	5.57	0.37
s9234.1	575	4	0	0	180.0	164.0	8.9%	0.05	38.0	75.0	5.72	0.43
s13207.1	1381	5	0	0	288.0	272.0	5.6%	0.02	546.0	562.0	9.25	0.88
s15850	1102	291	158	126	374.0	213.7	42.9%	0.29	12441.4	14305.6	11.34	1.28
s15850.1	1362	210	4	0	374.0	292.0	21.9%	0.28	1862.0	2829.0	13.63	1.49
s35932	4951	864	0	0	140.0	126.0	10.0%	0.16	15840.0	22896.0	13.21	2.83
s38417	1945	430	230	274	222.0	128.0	42.3%	0.28	6612.0	8224.0	38.55	3.44
s38584	6042	1355	1	0	308.0	290.0	5.8%	0.59	22932.5	23920.0	19.19	5.14
arith							16.9%		1	1.53X	15.6X	1
geo							18.4%		1	1.42X	12.5X	1

Table 6.3. Effect of retiming on skew scheduling

Circuit	$\|s_0\|$	$\|s_1\|$	$\|s_2\|$	$\|s_3\|$	[78]	T^*	impr%	t(sec)	padding		time(sec)	
									MOSEK	ours	MOSEK	ours
s838′	54	38	40	34	82.0	**59.2**	27.8%	0.01	147.6	158.4	0.22	0.01
s1196′	35	0	0	0	100.0	100.0	0.0%	0.00	0.0	0.0	0.46	0.02
s1423′	239	17	0	0	282.0	**256.0**	9.2%	0.05	514.0	999.7	0.46	0.03
s5378′	301	0	0	0	94.0	94.0	0.0%	0.01	0.0	0.0	2.63	0.09
s9234′	603	0	0	0	164.0	164.0	0.0%	0.05	0.0	0.0	6.13	0.36
s9234.1′	581	0	0	0	164.0	164.0	0.0%	0.05	0.0	0.0	6.71	0.43
s13207.1′	1126	5	0	0	NO	**272.0**	n/a	0.02	36.0	52.0	9.17	0.79
s15850′	928	398	201	4	212.0	**156.0**	26.4%	0.10	9479.0	10783.8	9.80	1.40
s15850.1′	1515	0	0	0	292.0	292.0	0.0%	0.17	0.0	0.0	10.77	1.20
s35932′	4321	864	0	0	140.0	**126.0**	10.0%	0.14	14752.0	21808.0	13.83	2.69
s38417′	4688	0	0	0	122.0	122.0	0.0%	0.03	0.0	0.0	27.97	3.95
s38584′	4159	1355	1	0	NO	**290.0**	n/a	0.66	18284.0	19265.7	18.28	4.32
arith							7.3%		1	1.19X	13.9X	1
geo							8.0%		1	1.16X	11.7X	1

CHAPTER 7

Clustering for Processing Rate Optimization

Circuit clustering (or partitioning) is often employed between logic synthesis and physical design to decompose a large circuit into parts. Each part will be implemented as a separate cluster that satisfies certain design constraints, such as the size of a cluster. Clustering helps to provide the first order information about interconnect delays as it classifies interconnects into two categories: intra-cluster ones are local interconnects due to their spatial proximity while inter-cluster ones may become global interconnects after floorplan/placement and routing (also known as circuit layout).

Due to aggressive technology scaling and increasing operating frequencies, interconnect delay has become the main performance limiting factor in large scale designs. Industry data shows that even with interconnect optimization techniques such as buffer insertion, the delay of a global interconnect may still be longer than one clock period, and multiple clock periods are generally required to communicate such a global signal. Since global interconnects are not visible at logic synthesis when the functionality of the implementation is the major concern, a design that is correct at the logic synthesis level may have timing closure problems after layout due to the emergence of multiple-clock-period interconnects.

This gap has motivated recent research to tackle the problem from different aspects of view. Some of them resort to retiming [**53**], which is a traditional sequential optimization technique that moves flip-flops within a circuit without destroying its functionality. It was used in [**110, 14, 103, 61, 57**] to pipeline global interconnects so as to reduce the clock

period. Although retiming helps to relieve the criticality of global interconnects, there is a lower bound of the clock periods that can be achieved because retiming cannot change the latency of either a (topological) cycle or an input-to-output path in the circuit. In case that the lower bound does not meet the frequency requirement, redesign and re-synthesis may have to be carried out.

One way to avoid redesign is to insert extra wire-pipelining units like flip-flops to pipeline long interconnects, as done within Intel [**15**] and IBM [**42**]. It can be shown that if the period lower bound is determined by an input-to-output path, pipelining can reduce the lower bound without affecting the functionality. However, if the period lower bound is given by a cycle, inserting extra flip-flops in it will change its functionality.

C-slow transformation [**53**] is a technique that slows down the input issue rate[1] of the circuit to accommodate higher frequencies. It was thus used in [**75**] to retain the functionality when extra flip-flops were inserted in cycles. The slowdown C is dictated by the slowest cycle in the circuit where the ratio between the extra flip-flops and the original flip-flops is the maximum. Extra flip-flops are inserted in other cycles to match the slowdown. As a result, throughput is sacrificed (becomes $1/C$) to meet the frequency requirement.

Instead of slowing down the throughput uniformly over the whole circuit, Latency Insensitive Design (LID) [**8, 7**], on the other hand, employs a protocol that slows down the throughput of a part of the circuit only when it is needed. As a result, LID can guarantee minimal throughput reduction while satisfying the frequency requirement.

We show in Section 7.1 that the aforementioned three approaches (retiming, pipelining with C-slow, and pipelining with LID) can be unified under the same objective function

[1]The issue rate is defined as the number of clock periods between successive input changes. An issue rate of 1 indicates that the inputs can change every clock period.

of maximizing the *processing rate*, defined as the product of frequency and throughput, as illustrated in Figure 7.1. In addition, the processing rate of a sequential system is upper bounded by the reciprocal of the maximum cycle ratio of the system, which is only dependent on the clustering. Therefore, we propose an optimal algorithm that finds a clustering with the minimal maximum-cycle-ratio.

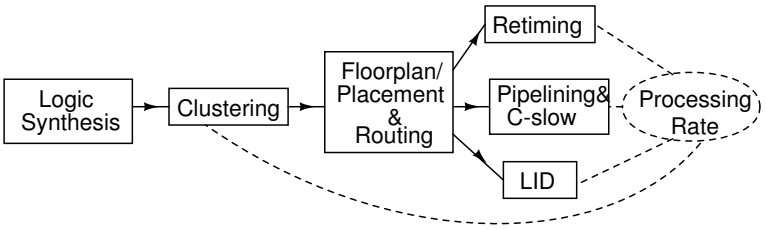

Figure 7.1. A logical and physical design flow.

The rest of this chapter is organized as follows. Section 7.1 presents the problem formulation. Two previous works are reviewed in Section 7.2. Section 7.3 defines the notations and constraints used in this chapter. Following an overview in Section 7.4, our algorithm is elaborated in Section 7.5 and 7.6. Section 7.7 presents the speed-up techniques. Section 7.8 reviews the techniques in [**77**] for cluster and replication reduction. We present some experimental results in Section 7.9. Conclusions are given in Section 7.10.

7.1. Problem formulation

We consider clustering subject to a size limit for clusters. More specifically, each gate has a specified size, as well as each interconnect. We require that the size of each cluster, defined as the sum of the sizes of the gates and the interconnects in the cluster, should be no larger than a given constant A. Replication of gates is allowed, i.e., a gate may be assigned to more than one cluster in the layout. When a gate is replicated, its incident interconnects

are also replicated so that the clustered circuit is logically equivalent to the original circuit. Figure 7.2 (taken from [**77**]) shows an example of gate replication in a clustering.

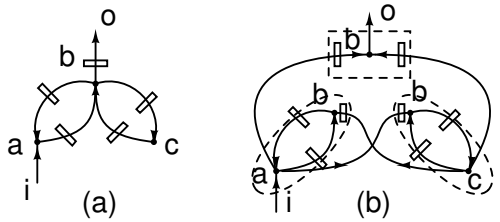

Figure 7.2. (a) An example circuit; (b) A clustering with 3 replicas of gate b.

Given a particular clustering c, we treat the replicas of gates and the original ones distinctly and denote them all as V_c. We use E_c to denote the set of interconnects among V_c. The clustered circuit is represented as $G_c = (V_c, E_c)$. In order for the circuit to operate at a specified clock period λ, additional wire-pipelining flip-flops are inserted. For all cycle o_c in G_c, let $d(o_c)$ denote the cycle delay, $w(o_c)$ and $w_\lambda(o_c)$ denote the number of flip-flops in o_c before and after additional pipelining flip-flops are inserted, respectively. Assuming $w(o_c) > 0$, the cycle ratio of o_c is defined as $\phi(o_c) = d(o_c)/w(o_c)$. Note that $\phi(o_c)$ is defined using $w(o_c)$, not $w_\lambda(o_c)$. The *maximum cycle ratio* over all the cycles in G_c is denoted as $\phi_c = \max_{o_c \in G_c} \phi(o_c)$.

We define *processing rate* as follows.

Definition 7.1.1. *For a sequential system, processing rate is defined as the length of processed input sequence per unit time. In particular, it is the product of frequency and throughput in a synchronous system.*

The larger the processing rate, the better the sequential system. Given the above definition, the approach of retiming actually maximizes the processing rate by minimizing the

period while keeping the throughput. It is interesting to notice that the approach of pipelining with C-slow transformation also maximizes the processing rate for a specified period by computing the least slowdown of the issue rate, which is transformed into throughput reduction. As an alternative, Latency Insensitive Design (LID) helps the clustered circuit reach the maximum throughput for a specified period. Therefore, all the three approaches can be unified under the same objective function of maximizing the processing rate.

It was shown in [**10**] that the maximum throughput ρ_λ of an LID for a specified period λ can be computed as

$$\rho_\lambda = \min_{\text{cycle } o_c \in G_c} \frac{w(o_c)}{w_\lambda(o_c)}.$$

On the other hand, the fact that the circuit can operate at the specified period λ after the insertion of additional flip-flops implies that $w_\lambda(o_c)\lambda \geq d(o_c)$, i.e., $\frac{1}{w_\lambda(o_c)} \leq \frac{1}{d(o_c)/\lambda}$, $\forall$cycle $o_c \in G_c$. Substitute this into the formula of ρ_λ to get

$$\rho_\lambda \leq \min_{o_c \in G_c} \frac{w(o_c)}{d(o_c)/\lambda} = \min_{o_c \in G_c} \frac{\lambda}{\phi(o_c)} = \frac{\lambda}{\phi_c}.$$

It follows that the maximum processing rate of an LID is upper bounded by $\frac{1}{\phi_c}$ since

$$\textit{max processing rate} = \frac{1}{\lambda} \cdot \rho_\lambda \leq \frac{1}{\lambda} \cdot \frac{\lambda}{\phi_c} = \frac{1}{\phi_c}.$$

It is also an upper bound of the maximum processing rate obtained by the approach of retiming, as shown in [**110**]. In other words, all the three approaches share the same upper bound of their common objective.

To maximize the processing rate, one can either maximize the upper bound or try to achieve the upper bound. They are equally important. However, since achieving the upper bound requires further knowledge on physical design, such as buffer and flip-flop allowable

regions [**19, 110**], while the upper bound itself is only dependent on the maximum cycle ratio of the clustered circuit, we will consider how to optimally cluster the circuit such that the upper bound is maximized, or equivalently, the maximum cycle ratio is minimized.

In order to compute the maximum cycle ratio, we need to know how to compute the delay of a cycle during clustering. Although local interconnect delays can be obtained using some delay models at synthesis, the delays of global interconnects are not available until layout. Therefore during clustering, we assume that each global interconnect induces an extra constant delay D, as proposed in [**77**]. More specifically, if an interconnect (u, v) with delay $d(u, v)$ is assigned to be inter-cluster, then its delay becomes $d(u, v) + D$.

The single-value inter-cluster delay model is the best approximation to distinguish potential global interconnects from local ones. In floorplanning, critical global interconnects are made short by placing the relevant modules closer. On the other hand, the values of cluster size A and inter-cluster delay D can be chosen deliberately to make this model practical. The intra-cluster interconnects will be very long if large A is selected. On the other hand, A shall not be too small otherwise there will be too many clusters and the inter-cluster delays will be similar to the intra-cluster delays after floorplanning. By carefully choosing D, the single-value model can fulfill the need of integrating inter-cluster delay information in clustering.

Since we want to minimize the maximum cycle ratio, the path delays from primary inputs (PIs) to primary outputs (POs) can be ignored since they can be mitigated by pipelining. This motivates us to formulate the problem in a strongly connected graph.

Problem 7.1.1 (Optimal Clustering Problem)**.**

Given a directed, strongly connected graph $G = (V, E)$, where each vertex $v \in V$ has a delay

$d(v)$ and a specified size, and each edge $(u, v) \in E$ has a delay $d(u, v)$, a specified size and a weight $w(u, v)$ (representing the number of flip-flops on it), find a clustering of vertices with possible vertex replication such that: 1. the size of each cluster is no larger than a given constant A; 2. each global interconnect induces an extra constant delay D; 3. the maximum cycle ratio of the clustered circuit is minimized.

We assume that all delays are integral[2] and thus all cycle ratios are rational. In addition, we assume that each gate has unit size and the size of each interconnect is zero. Our proposed algorithm can be easily extended to handle various size scenarios.

7.2. Previous work

Pan *et al.* [**77**] proposed to optimally cluster a sequential circuit such that the lower bound of the period of the clustered circuit was minimized with retiming. However, the period lower bound may not come from a cycle ratio. In addition, their algorithm needs to start from PIs, thus cannot be used to solve our problem in a strongly connected graph. Applying their algorithm with arbitrary PI selection may lead to a sub-optimal solution. In this sense, they solved a different problem, even though it looks similar to ours.

Their problem was solved by binary search, using a test for feasibility as a subroutine. For each target period, they used a procedure called *labeling computation* to check the feasibility. The procedure starts with label assignment 0 for PIs and $-\infty$ for the other vertices, and repeatedly increases the label values until they all converge or the label value of some PO exceeds the target period, for which the target period is considered infeasible. For each vertex, the amount of increase in its label is computed using another binary search that

[2]This assumption is not really restrictive in practice because computer works with rational numbers which we can convert to integers by multiplying by a suitably large number.

basically selects the minimum from a candidate set. Because of the nested binary searches, their algorithm is relatively slow. In addition, the algorithm requires $O(|V|^2)$ space to store a pre-computed all-pair longest-path matrix, which is impractical for large designs. Cong *et al.* [**20**] improved the algorithm by tightening the candidate set to speed up the labeling computation, and by reducing the space complexity to linear dependency. But the improved algorithm still needs the nested binary searches.

Besides the difference in problem formulation, our algorithm differs from theirs in two algorithmic aspects. Firstly, our algorithm focuses on cycles, thus can work on any general graph. Secondly, no binary search is employed in our algorithm. As a result, our algorithm is efficient and essentially incremental. Like [**20**], our algorithm does not need pre-computed information on paths either.

Except for these differences, [**77**] revealed some important results on clustering, which we summarize here to simplify our notations.

- Each cluster has only one output, which is called the *root* of the cluster. If there is a cluster with more than one output, we can replicate the cluster enough times so that each copy of the cluster has only one output.
- For each vertex in V, there is exactly one cluster rooted at it and no cluster rooted at its replicas. Its arrival time (defined in Section 7.3) is no larger than the arrival times of its replicas.
- If $u \in V$ is an input of the cluster rooted at $v \in V$, then the cluster rooted at v must not contain a replica of u.

7.3. Notations and constraints

First of all, we define notations with respect to G and G_c (the clustered circuit of a particular clustering c) respectively.

We use p to denote a path in G. Let $w(p)$ be the number of flip-flops on p, which is the sum of the weights of p's constituent edges. Let $d(p)$ be the delay along p, which is the sum of the delays of p's constituent edges and vertices, except for $d(u)$. When a path actually forms a cycle o, $w(o)$ includes the weight of each edge in the cycle only once. Similarly, $d(o)$ includes the delay of each edge and vertex in the cycle only once. We assume in this chapter that $w(o) > 0$ for all cycle $o \in G$.

Each of the above notations is appended with a subscript c when it is referred to with respect to G_c. More specifically, a path in G_c is denoted as p_c with $w(p_c)$ flip-flops and $d(p_c)$ delays. Note that the delay of an inter-cluster edge $(u, v) \in G_c$ is $d(u, v) + D$. A cycle in G_c is denoted as o_c with $w(o_c)$ flip-flops and $d(o_c)$ delays. We have $w(o_c) > 0$ for all cycle $o_c \in G_c$. We use $\phi(o_c)$ to denote the cycle ratio of o_c, and ϕ_c to denote the maximum cycle ratio of G_c.

Since we only need to consider clusters rooted at the vertices in V, exactly one for each vertex, we use c_v to refer to the set of vertices that are included in the cluster rooted at $v \in V$. Let $i_v \subset V$ be the set of inputs of c_v. In the remainder of this chapter, when we say $u \in c_v$ $(u \neq v)$, we mean that the cluster rooted at $v \in V$ contains a replica of $u \in V$. For example, Figure 7.3(a) shows a circuit before clustering. There are five vertices (a-e) and seven edges. Figure 7.3(b) illustrates a clustering of the circuit with size limit $A = 3$, where dashed circles represent clusters. For each cluster, the vertex whose index is outside

the cluster indicates the root. For example, c_a contains replicas of vertices c and e with the input set $i_a = \{d\}$.

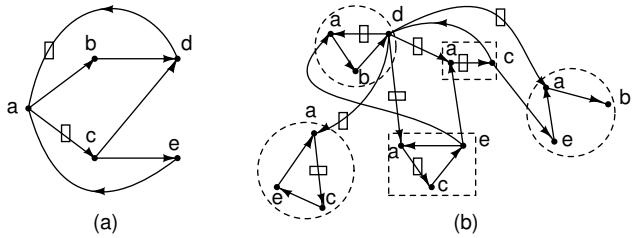

Figure 7.3. An example of clustering representation.

We use a label $t : V \rightarrow \Re$ to denote the arrival time of the vertex. To ease the presentation, we will extend the domain of t to V_c to represent the arrival times of the replicas of the vertices. Based on this, a clustering that satisfies the cluster size requirement and has a maximum cycle ratio no larger than a given rational value ϕ can be characterized as follows.

$$t(v) \geq 0, \qquad \forall v \in V \tag{7.1}$$

$$t(v) \geq t(u) + d(u,v) + d(v) - w(u,v)\phi, \tag{7.2}$$

$$\forall \text{ intra-cluster } (u,v) \in E_c$$

$$t(v) \geq t(u) + d(u,v) + D + d(v) - w(u,v)\phi, \tag{7.3}$$

$$\forall \text{ inter-cluster } (u,v) \in E_c$$

$$|c_v| \leq A, \qquad \forall v \in V \tag{7.4}$$

where (7.1)-(7.3) guarantee that the arrival times are all achievable, and (7.4) is the cluster size requirement.

Following the convention, $(u,v) \in E_c$ is a *critical edge* under ϕ iff it is intra-cluster with $t(v) = t(u) + d(u,v) + d(v) - w(u,v)\phi$, or it is inter-cluster with $t(v) = t(u) + d(u,v) +$

$D + d(v) - w(u,v)\phi$. A *critical path* under ϕ refers to a path whose constituent edges are all critical under ϕ. Vertex $u \in G$ is a *critical input* of c_v under ϕ iff $u \in i_v$ and v can be reached by u through a critical path $p = u \rightarrow x \rightsquigarrow v$ under ϕ where the sub-path $x \rightsquigarrow v$ is in c_v. When a critical path actually forms a cycle, it is then called a *critical cycle*. Cycle o_c is critical under ϕ iff $d(o_c) = w(o_c)\phi$.

A *legal clustering* must satisfy (7.4). When the arrival times of a legal clustering satisfy (7.1)-(7.3) under ϕ, it is called a *feasible clustering* under ϕ. When a critical cycle is present in a feasible clustering under ϕ, it is called a *critical clustering* under ϕ. A given ϕ is *feasible* iff there exists a feasible clustering under ϕ. We must note that for a legal clustering, its maximum cycle ratio is feasible. In fact, any value larger than the maximum cycle ratio of a legal clustering is also feasible.

Given a feasible clustering c under ϕ, consider an edge $(u,v) \in E$, $\forall v \in V$. It is either in E_c with $u \in i_v$, or there is an edge (u',v) such that u' is a replica of u. In either case, the following inequality is true because of (7.2)-(7.3) and the fact that $t(u) \leq t(u')$ from [**77**].

$$t(v) \geq t(u) + d(u,v) + d(v) - w(u,v)\phi, \forall (u,v) \in E \tag{7.5}$$

The following lemma provides a lower bound for ϕ.

Lemma 7.3.1. *A feasible ϕ is no smaller than the maximum cycle ratio of G, denoted as ϕ_{lb}.*

Proof. Since ϕ is feasible, then, by definition, there exists a clustering c satisfying (7.1)-(7.4) under ϕ. Therefore, (7.5) is true, and we have $d(o) \leq w(o)\phi$, for all cycle $o \in G$, i.e., $\phi \geq \phi_{\text{lb}}$. □

Define

$$\Delta(u, v, \phi) \stackrel{\triangle}{=} \max_{p \in u \rightsquigarrow v \ in \ G} \big(d(p) - w(p)\phi\big), \forall u, v \in V$$

Lemma 7.3.1 ensures that $\Delta(u, v, \phi)$ is well-defined on feasible ϕ's and can be obtained by longest path computation in $O(|E| + |V| \log |V|)$ time [**23**].

7.4. Overview

The optimal clustering problem asks for a legal clustering with the minimal maximum-cycle-ratio. Since $A > 0$, the clustering with each vertex being a cluster is certainly legal. Starting from it, we will iteratively improve the clustering by reducing its maximum cycle ratio until the optimality is certified.

First of all, the maximum cycle ratio of a legal clustering is feasible and can be efficiently computed using Howard's algorithm [**16, 28**]. Given a feasible ϕ, we show that, unless ϕ is already the optimal solution, a particular legal clustering can be constructed whose maximum cycle ratio is smaller than ϕ. The smaller ϕ can be obtained by applying Howard's algorithm on the constructed clustering. Therefore, we alternate between applying Howard's algorithm and constructing a better clustering until the minimal ϕ is reached.

7.5. Clustering under a given $\phi > \phi_{\text{lb}}$

Given $\phi > \phi_{\text{lb}}$, if ϕ is feasible, we show in this section how to construct a feasible clustering under ϕ, i.e., a clustering satisfying (7.1)-(7.5) under ϕ, whose maximum cycle ratio is no larger than ϕ.

7.5.1. Choosing (7.1) and (7.5) as invariant

We choose to first satisfy (7.1) and (7.5) because they are independent on clustering, and iteratively update $t(v)$ and c_v to satisfy (7.2)-(7.4) while keeping (7.1) and (7.5).

Let T denote the arrival time vector, i.e.,

$$\mathrm{T} = \big(t(1), t(2), \ldots, t(|V|)\big).$$

A partial order ($\leq$) can be defined between two arrival time vectors T and T$'$ as follows.

$$\mathrm{T} \leq \mathrm{T}' \stackrel{\triangle}{=} t(v) \leq t'(v),\ \forall v \in V.$$

According to the lattice theory [**29**], if we treat assignment $t(v) = 0$, $\forall v \in V$ as the bottom element ($\perp$) and assignment $t(v) = \infty$, $\forall v \in V$ as the top element ($\top$), then the arrival time vector space $\Re^{|V|}$ becomes a *complete partially ordered set*, that is, for all $\mathrm{T} \in \Re^{|V|}$, $\perp \leq \mathrm{T} \leq \top$.

To satisfy (7.1), we set $t(v) = 0$, $\forall v \in V$. Then we apply a modified Bellman-Ford's algorithm, defined as $MBF(\mathrm{T}, \phi)$, on E to satisfy (7.5) under ϕ. The modified Bellman-Ford's algorithm is the same as Bellman-Ford's algorithm [**23**] except that it takes two inputs: a given arrival time vector T and a value of ϕ. The value of ϕ is used to specify (7.5) so that we can perform relaxation starting from the given arrival time vector T. The relaxation is guaranteed to converge when $\phi \geq \phi_{\mathrm{lb}}$.

Given $\perp$ under ϕ, the resulting arrival time vector of MBF is denoted as

$$\mathrm{T}_0 = MBF(\perp, \phi).$$

In fact, T_0 is the *least* vector satisfying (7.1) and (7.5) under ϕ, as stated in the following lemma.

Lemma 7.5.1. $\mathrm{T}_0 \leq \mathrm{T}$*, for all* T *satisfying (7.1) and (7.5) under* ϕ*.*

Proof. Suppose we have a T satisfying (7.1) and (7.5) under ϕ with $t(v) < t_0(v)$ for some $v \in V$. It follows that $t_0(v) > 0$ since $t(v) \geq 0$ by (7.1). The modified Bellman-Ford's algorithm guarantees that there exists a path $p = u \rightsquigarrow v$ in G such that $t_0(u) = 0$ and $t_0(v) = t_0(u) + d(p) - w(p)\phi$. Since T satisfies (7.5) under ϕ, we have $t(v) \geq t(u) + d(p) - w(p)\phi = t(u) + t_0(v) \geq t_0(v)$, which contradicts $t(v) < t_0(v)$. Therefore, such a T does not exist and the lemma is true. □

7.5.2. Transformation $\mathcal{L}_v$ to satisfy (7.2)-(7.4) under ϕ

In order to satisfy (7.2)-(7.4) while keeping (7.1) and (7.5) under ϕ, we define transformation $\mathcal{L}_v : (\Re^{|V|}, \Re) \rightarrow \Re$, $\forall v \in V$, as follows.

For all $v \in V$, we will construct a new cluster c'_v, as opposed to the current c_v. The procedure starts with $c'_v = \{v\}$ and grows c'_v progressively by including one critical input at a time. Let $t'(v)$ denote the arrival time of v in c'_v, as opposed to $t(v)$ in c_v. Note that when a vertex is put in c'_v, its preceding vertices that are outside c'_v become inputs of c'_v. Therefore, $t'(v)$ varies every time c'_v grows. The procedure will stop only when either $|c'_v| = A$ or $t'(v) \leq t(v)$. When it stops, we compare $t'(v)$ with $t(v)$. If $t'(v) < t(v)$, we keep $t(v)$ and c_v unchanged; otherwise we update $t(v)$ and c_v with $t'(v)$ and c'_v respectively. The resulting arrival time of v is defined as $\mathcal{L}_v(\mathrm{T}, \phi)$. The next lemma helps to identify the critical input to be included at each time.

Lemma 7.5.2. *Given that (7.1) and (7.5) are satisfied under ϕ, we have $t'(v) \geq t(x) + D + \Delta(x, v, \phi)$, for all $x \notin c'_v$. In particular, if u is a critical input of c'_v, then $t'(v) = t(u) + D + \Delta(u, v, \phi)$.*

Proof. For all $x \notin c'_v$, let p be the path from x to v in G such that $\Delta(x, v, \phi) = d(p) - w(p)\phi$. Since $x \notin c'_v$, there exists a vertex $y \in i'_v$ on p which divides p into two sub-paths: p_1 from x to y, and p_2 from y to v. In addition, p_2 has the form of $y \rightarrow z \rightsquigarrow v$, where $z \rightsquigarrow v$ is in c'_v. By (7.3), we have $t'(v) \geq t(y)+D+d(p_2)-w(p_2)\phi$. In addition, $t(y) \geq t(x)+d(p_1)-w(p_1)\phi$ by (7.5). Therefore, $t'(v) \geq t(x) + D + d(p) - w(p)\phi = t(x) + D + \Delta(x, v, \phi)$.

If c'_v has a critical input u, then, by definition, there exists a path $p = u \rightsquigarrow v$ in G such that $t(u)+D+d(p)-w(p)\phi = t'(v)$. Since $u \notin c'_v$, we have $t'(v) \geq t(u)+D+\Delta(u, v, \phi)$, thus $d(p) - w(p)\phi \geq \Delta(u, v, \phi)$. On the other hand, $d(p) - w(p)\phi \leq \Delta(u, v, \phi)$ since $\Delta(u, v, \phi)$ is the largest among all paths from u to v in G. Therefore, $d(p) - w(p)\phi = \Delta(u, v, \phi)$, which concludes our proof. □

Lemma 7.5.2 implies that when a vertex is put in c'_v, its preceding vertices that are already in c'_v can be ignored for the computation of $t'(v)$.

7.5.3. Implementation of $\mathcal{L}_v$

Our implementation of $\mathcal{L}_v$ is similar to the label computation in [**20**]. To characterize critical inputs, we introduce another label $\delta(u)$, $\forall u \in V$. Before the construction of c'_v, we assign $\delta(u)$ with $-\infty$ for all $u \neq v$ in V while $\delta(v)$ with 0. At each time, the vertex $u \in i'_v$ with the largest $t(u) + \delta(u)$ is identified. If $t(u) + D + \delta(u) \leq t(v)$, the construction is completed. Otherwise, we put it in c'_v and update $\delta(x)$ with $\max\big(\delta(x),\ \delta(u)+d(u)+d(x, u)-w(x, u)\phi\big)$,

for all $(x, u) \in E$. This procedure will iterate until either $|c'_v| = A$ or the last vertex u identified has $t(u) + D + \delta(u) \leq t(v)$.

To validate the above procedure, we need to show that it is equivalent to $\mathcal{L}_v(\mathrm{T}, \phi)$, or equivalently, to show that it can always identify the critical input of c'_v. This is fulfilled by the next lemma and corollary.

Lemma 7.5.3. *Given that (7.1) and (7.5) are satisfied under ϕ, we have $\delta(u) = \Delta(u, v, \phi)$, for all $u \in c'_v$.*

Proof. We prove it by induction on the size of c'_v. At the beginning, $c'_v = \{v\}$ and $\delta(v) = 0 = \Delta(v, v, \phi)$. Suppose that the lemma is true for $|c'_v| \leq k < A$, we need to show that the lemma is also true for $|c'_v| = k + 1$. Therefore we start with $|c'_v| = k$ and let $u \in i'_v$ be the vertex with the largest $t(u) + D + \delta(u) > t(v)$.

We use p to denote the path where $d(p) - w(p)\phi = \Delta(u, v, \phi)$. Since $u \notin c'_v$, there exists a vertex $x \in i'_v$ on p such that the sub-path from x to v has the form of $x \to y \rightsquigarrow v$, where $y \rightsquigarrow v$ is in c'_v. Let p_1 and p_2 be the sub-path from u to x and from y to v, respectively. Since $d(p) - w(p)\phi = \Delta(u, v, \phi)$, we have $d(p_1) - w(p_1)\phi = \Delta(u, x, \phi)$ and $d(p_2) - w(p_2)\phi = \Delta(y, v, \phi)$.

Given that $u \in i'_v$ is the vertex with the largest $t(u) + \delta(u)$, we know that $t(u) + \delta(u) \geq t(x) + \delta(x)$. (7.5) ensures that $t(x) \geq t(u) + d(p_1) - w(p_1)\phi$. On the other hand, we have $\delta(x) \geq \delta(y) + d(x, y) - w(x, y)\phi$ since we updated $\delta(x)$ to be no less than $\delta(y) + d(x, y) - w(x, y)\phi$

when y was put in c'_v. Further, $\delta(y) = \Delta(y, v, \phi)$ by the inductive hypothesis. Consequently,

$$\begin{aligned}
& t(u) + \delta(u) \\
\geq\ & t(x) + \delta(x) \\
\geq\ & \big(t(u) + d(p_1) - w(p_1)\phi\big) + \delta(x) \\
\geq\ & \big(t(u) + d(p_1) - w(p_1)\phi\big) + \big(\delta(y) + d(x, y) - w(x, y)\phi\big) \\
=\ & \big(t(u) + d(p_1) - w(p_1)\phi\big) + \big(\Delta(y, v, \phi) + d(x, y) - w(x, y)\phi\big) \\
=\ & \big(t(u) + d(p_1) - w(p_1)\phi\big) + \big((d(p_2) - w(p_2)\phi) + d(x, y) - w(x, y)\phi\big) \\
=\ & t(u) + d(p) - w(p)\phi \\
=\ & t(u) + \Delta(u, v, \phi),
\end{aligned}$$

or $\delta(u) \geq \Delta(u, v, \phi)$. However, $\delta(u) \leq \Delta(u, v, \phi)$ since $\Delta(u, v, \phi)$ is the largest among all paths from u to v. Therefore, $\delta(u) = \Delta(u, v, \phi)$. □

Corollary 7.5.3.1. *Given that (7.1) and (7.5) are satisfied under ϕ, the vertex $u \in i'_v$ with the largest $t(u) + D + \delta(u) \geq t(v)$ is the critical input of c'_v.*

Proof. Suppose u is not a critical input. Let p denote the path where $d(p) - w(p)\phi = \delta(u)$. Since p is not critical, we have $t'(v) > t(u) + D + d(p) - w(p)\phi = t(u) + D + \delta(u) \geq t(v)$. It implies that c'_v has a critical input x, otherwise the vertices that have critical paths to v are all inside c'_v and we have $t'(v) = t_0(v) \leq t(v)$, which is a contradiction. Let p' be a critical path $x \to y \rightsquigarrow v$ from x to v whose sub-path $p'_1 : y \rightsquigarrow v$ is in c'_v and $t'(v) = t(x) + D + d(p') - w(p')\phi$. In addition, $t'(v) = t(x) + D + \Delta(x, v, \phi)$ by Lemma 7.5.2. Thus, $\Delta(x, v, \phi) = d(p') - w(p')\phi$, which implies that $\Delta(y, v, \phi) = d(p'_1) - w(p'_1)\phi$.

On the other hand, since $y \in c'_v$, we have $\delta(y) = \Delta(y, v, \phi)$ by Lemma 7.5.3. Hence $\delta(x) \geq \delta(y) + d(x, y) - w(x, y)\phi = d(p') - w(p')\phi = \Delta(x, v, \phi)$. Since $\Delta(x, v, \phi)$ is the largest among all paths from x to v, we have $\delta(x) = \Delta(x, v, \phi)$. It follows that $t(x) + D + \delta(x) = t'(v) > t(u) + D + \delta(u)$, which contradicts that u is the input with the largest $t(u) + \delta(u)$. Therefore, the lemma is true. □

The pseudocode for computing $\mathcal{L}_v(\mathrm{T}, \phi)$ is given in Figure 7.4. It employs a heap Q for bookkeeping the vertices $u \in i'_v$ whose $t(u) + D + \delta(u) > t(v)$. At each iteration, it puts in c'_v the input $u \in Q$ with the largest $t(u) + D + \delta(u)$ and updates $\delta(x)$ for each fanin of u that becomes an input in i'_v. In our implementation, we choose Fibonacci heap [**23**] for Q.

Input: $G = (V, E)$, A, D, T, ϕ, $v \in V$.
Output: $t'(v)$, c'_v.

$\delta(u) \leftarrow -\infty$, $\forall u \in V$; $\delta(v) \leftarrow 0$;
$Q \leftarrow \{v\}$; $c'_v \leftarrow \emptyset$;
`While` $\big((Q \neq \emptyset) \wedge (|c_v| < A)\big)$ `do`
  $u \leftarrow$ `extract from` Q `with max` $t(u) + \delta(u)$;
  $t'(v) \leftarrow t(u) + D + \delta(u)$;
  $c'_v \leftarrow c'_v \cup \{u\}$;
  `For` $e = (x, u) \in E$ `with` $x \notin c'_v$ `do`
    `If` $\big(\delta(x) < \delta(u) + d(u) + d(x, u) - w(x, u)\phi\big)$ `then`
      $\delta(x) \leftarrow \delta(u) + d(u) + d(x, u) - w(x, u)\phi$;
    `If` $\Big((x \notin Q) \wedge \big(t(x) + D + \delta(x) > t(v)\big)\Big)$ `then`
      $Q \leftarrow Q \cup \{x\}$;
`If` $(Q \neq \emptyset)$ `then`
  $t'(v) \leftarrow$ `max` $t(u) + D + \delta(u)$ `in` Q;
`Else` $t'(v) \leftarrow t(v)$;
`Return` $t'(v)$ `and` c'_v;

Figure 7.4. Pseudocode of $\mathcal{L}_v(\mathrm{T}, \phi)$.

The complexity of $\mathcal{L}_v(\mathrm{T}, \phi)$ in Figure 7.4 is given in the following lemma.

Lemma 7.5.4. *The procedure in Figure 7.4 terminates in $O(A|E|\log|V|)$ time.*

Proof. At each iteration, the complexity of extracting the vertex u with the maximum $t(u)+\delta(u)$ is $O(\log|V|)$ by Fibonacci heap [**23**]. For each $(x,u)\in E$, updating $\delta(x)$ takes $O(\log|V|)$ time. On the other hand, since a vertex cannot be put in Q more than once, the total number of edges processed by the inner for-loop is $O(|E|)$. Therefore, the complexity the procedure is $O(A\log|V|+|E|\log|V|)$, or $O(|E|\log|V|)$. □

7.5.4. Clustering under ϕ as a fixpoint computation

We define $\mathcal{L}(\mathrm{T},\phi)$ as the arrival time vector when all the $\mathcal{L}_v(\mathrm{T},\phi)$'s, $\forall v\in V$, are applied once, followed by the modified Bellman-Ford's algorithm to ensure (7.5), expressed as

$$\mathcal{L}(\mathrm{T},\phi)\stackrel{\triangle}{=}MBF\Big(\big(\mathcal{L}_1(\mathrm{T},\phi),\mathcal{L}_2(\mathrm{T},\phi),...,\mathcal{L}_{|V|}(\mathrm{T},\phi)\big),\phi\Big).$$

The following lemma shows that $\mathcal{L}$ is an order-preserving transformation.

Lemma 7.5.5. *For any* T *and* $\bar{\mathrm{T}}$ *satisfying (7.1) and (7.5) under* ϕ*, if* $\mathrm{T}\le\bar{\mathrm{T}}$*, then* $\mathcal{L}(\mathrm{T},\phi)\le\mathcal{L}(\bar{\mathrm{T}},\phi)$.

Proof. We first show that $\mathcal{L}_v(\mathrm{T},\phi)\le\mathcal{L}_v(\bar{\mathrm{T}},\phi)$, $\forall v\in V$.

For the sake of contradiction, we assume that $\mathcal{L}_v(\mathrm{T},\phi)>\mathcal{L}_v(\bar{\mathrm{T}},\phi)$ for some $v\in V$. The procedure of $\mathcal{L}_v$ guarantees that $\mathcal{L}_v(\mathrm{T},\phi)=\max\big(t'(v),t(v)\big)$ and $\mathcal{L}_v(\bar{\mathrm{T}},\phi)=\max\big(\bar{t}'(v),\bar{t}(v)\big)$. Since $t(v)\le\bar{t}(v)$ by $\mathrm{T}\le\bar{\mathrm{T}}$, we have $t'(v)>t(v)$, otherwise $\mathcal{L}_v(\mathrm{T},\phi)=t(v)\le\bar{t}(v)\le\mathcal{L}_v(\bar{\mathrm{T}},\phi)$, which contradicts the assumption that $\mathcal{L}_v(\mathrm{T},\phi)>\mathcal{L}_v(\bar{\mathrm{T}},\phi)$. In addition, since T satisfies (7.1) and (7.5) under ϕ, Lemma 7.5.1 ensures that $t(v)\ge t_0(v)$, where $t_0(v)$ is the arrival time of v in vector $\mathrm{T}_0=MBF(\perp,\phi)$. Thus, $t'(v)>t_0(v)$, which implies that cluster

c'_v has a critical input u, otherwise the vertices that have critical paths to v are all inside c'_v and we have $t'(v) = t_0(v)$, which is a contradiction. The existence of a critical input u implies that $|c'_v| = A$, otherwise u should have been put in c'_v since $t'(v) > t(v)$. Let p be the path where $\Delta(u, v, \phi) = d(p) - w(p)\phi$. Figure 7.5(a) shows an example of c'_v.

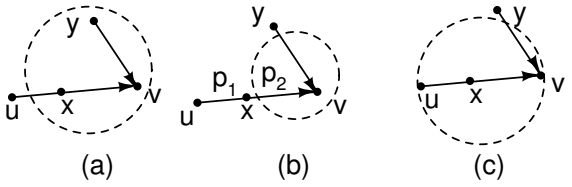

Figure 7.5. (a) Cluster c'_v; (b) and (c) two cases of cluster $\bar{c}'_v$.

Now consider cluster $\bar{c}'_v$, there are two cases. Firstly, $u \notin \bar{c}'_v$. Thus there exists a vertex $x \in p$ such that $x \in \bar{i}'_v$, as illustrated in Figure 7.5(b). x divides p into two sub-paths: p_1 from u to x, and p_2 from x to v. We know that $\Delta(x, v, \phi) = d(p_2) - w(p_2)\phi$ otherwise p cannot subsume p_2. Together with $t(x) \geq t(u) + d(p_1) - w(p_1)\phi$ by (7.5), we have $t(x) + \Delta(x, v, \phi) \geq t(u) + \Delta(u, v, \phi)$. On the other hand, since $x \in \bar{i}'_v$, we have $\bar{t}'(v) \geq \bar{t}(x) + D + \Delta(x, v, \phi)$ by Lemma 7.5.2. Given that $t(x) \leq \bar{t}(x)$ (since $\mathrm{T} \leq \bar{\mathrm{T}}$), we have $\bar{t}'(v) \geq t(u) + D + \Delta(u, v, \phi) = t'(v)$.

Secondly, $u \in \bar{c}'_v$. Given that $|c'_v| = A$ and u is not in c'_v, we know that there exists a vertex $y \in c'_v$ such that $y \in \bar{i}'_v$, as shown in Figure 7.5(c). By the same argument, we can show that $\bar{t}'(v) \geq \bar{t}(y) + D + \Delta(x, v, \phi) \geq t(y) + D + \Delta(x, v, \phi) \geq t(u) + D + \Delta(u, v, \phi) = t'(v)$.

In either case, we have $\max\big(t(v), t'(v)\big) \leq \max\big(\bar{t}(v),\ \bar{t}'(v)\big)$, i.e., $\mathcal{L}_v(\mathrm{T}, \phi) \leq \mathcal{L}_v(\bar{\mathrm{T}}, \phi)$, which is a contradiction. Therefore, the assumption is wrong and $\mathcal{L}_v(\mathrm{T}, \phi) \leq \mathcal{L}_v(\bar{\mathrm{T}}, \phi)$ is true, $\forall v \in V$. It is easy to verify that $\mathcal{L}(\mathrm{T}, \phi) \leq \mathcal{L}(\bar{\mathrm{T}}, \phi)$ after applying the modified Bellman-Ford's algorithm. $\square$

We say that T is a *fixpoint of* $\mathcal{L}$ under ϕ if and only if $\mathrm{T} = \mathcal{L}(\mathrm{T}, \phi)$. The following theorem bridges the existence of a fixpoint and the feasibility of ϕ.

Theorem 7.5.1. *ϕ is feasible if and only if $\mathcal{L}$ has a fixpoint under ϕ.*

Proof. ($\rightarrow$): If ϕ is feasible, then, by definition, there exists a legal clustering c whose arrival time vector T satisfies (7.1)-(7.3) and (7.5) under ϕ. We claim that $\mathcal{L}_v(\mathrm{T}, \phi) \leq t(v)$, $\forall v \in V$. Otherwise, $t'(v) > t(v) \geq t_0(v)$ for some $v \in V$, and c'_v has a critical input u, as shown in Figure 7.5(a). We can conduct a similar case study as Figure 7.5(b) and 7.5(c) to show that $t(v) \geq t'(v)$, which is a contradiction. On the other hand, $t(v) \leq \mathcal{L}_v(\mathrm{T}, \phi)$ by the procedure of $\mathcal{L}_v(\mathrm{T}, \phi)$. Therefore, $\mathcal{L}_v(\mathrm{T}, \phi) = t(v)$, $\forall v \in V$. Given that T satisfies (7.5), applying the modified Bellman-Ford's algorithm gives $\mathrm{T} = \mathcal{L}(\mathrm{T}, \phi)$, i.e., T is a fixpoint of $\mathcal{L}$ under ϕ.

($\leftarrow$): If $\mathcal{L}$ has a fixpoint under ϕ, then, by the definition of $\mathcal{L}$, the constructed clustering is legal and the arrival time vector satisfies (7.1)-(7.3) and (7.5) under ϕ. Therefore, ϕ is feasible. □

In fact, according to the lattice theory [**29**], if $\mathcal{L}$, defined on a complete partially ordered set, has a fixpoint under ϕ, then it has a *least fixpoint* T^ϕ, defined as

$$\big(\mathrm{T}^\phi = \mathcal{L}(\mathrm{T}^\phi, \phi)\big) \wedge \big(\forall \mathrm{T} : \mathrm{T} = \mathcal{L}(\mathrm{T}, \phi) : \mathrm{T}^\phi \leq \mathrm{T}\big).$$

We use c^ϕ to denote the clustering constructed by $\mathcal{L}(\mathrm{T}^\phi, \phi)$. In fact, if $t^\phi(v) > t_0(v)$, then there is a critical path from $u \in V$ to v with $t^\phi(u) = t_0(u)$. This is made precise in the following lemma.

Lemma 7.5.6. *If $t^{\phi}(v) > t_0(v)$, then there exists a sequence of vertices $x_i \in V$, $i = 0, 1, ..., k-1$ such that $t_0(x_0) = t^{\phi}(x_0)$, $x_k = v$, and x_i is a critical input of cluster $c^{\phi}_{x_{i+1}}$.*

Proof. Since $t^{\phi}(v) > t_0(v)$, we know that cluster c^{ϕ}_v has critical inputs, otherwise the vertices that have critical paths to v are all inside c^{ϕ}_v and we have $t_0(v) = t^{\phi}(v)$, which is a contradiction.

Suppose otherwise that such a sequence does not exist, namely, all the critical paths terminating at v are actually critical cycles, where each constituent vertex $u \in V$ has $t^{\phi}(u) > t_0(u) \geq 0$. Choose the one that contains all of them, denoted as O. For the example in Figure 7.6, we will choose O to be $a \rightsquigarrow e \rightsquigarrow d \rightsquigarrow c \rightsquigarrow g \rightsquigarrow f \rightsquigarrow d \rightsquigarrow c \rightsquigarrow b \rightsquigarrow a$. Now consider any

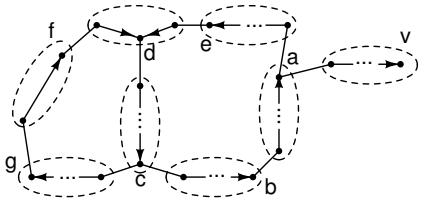

Figure 7.6. Vertices that have critical paths to v.

incoming edge of O (from a vertex outside of O to a vertex in O), it must be non-critical, otherwise we can trace back from this edge and find another critical cycle that is not in O, which is a contradiction. Since the arrival times of the vertices in O are all greater than zero, we can decrease them simultaneously while keeping the arrival times of other vertices unchanged until some incoming edge of O becomes critical or the arrival time of some vertex in O is reduced to zero. For either case, we obtain a fixpoint less than T^{ϕ}, which is a contradiction. Therefore, the lemma is true. □

To reach a fixpoint, iterative method can be used on $\mathcal{L}$. It starts with T_0 as the initial vector, iteratively computes new vectors from previous ones $\mathrm{T}_1 = \mathcal{L}(\mathrm{T}_0, \phi), \mathrm{T}_2 = \mathcal{L}(\mathrm{T}_1, \phi), \ldots$

until it finds a T_n such that $\mathrm{T}_n = \mathrm{T}_{n-1}$. The following lemma states that applying iterative method on $\mathcal{L}$ will converge to its least fixpoint in a finite number of iterations.

Lemma 7.5.7. *If ϕ is feasible, applying iterative method on $\mathcal{L}$ will converge to T^ϕ in a finite number of iterations.*

Proof. Since we start with $\mathrm{T} = \mathrm{T}_0 \leq \mathrm{T}^\phi$, Lemma 7.5.5 ensures that $\mathrm{T} \leq \mathrm{T}^\phi$ at each iteration. Therefore, if $\mathcal{L}$ converges, the fixpoint has to be the least fixpoint. What remains is to show that $\mathcal{L}$ is finitely convergent.

By Lemma 7.5.2 and 7.5.6, if $t^\phi(v) > t_0(v)$, then $t^\phi(v)$ can be written as

$$t^\phi(v) = t_0(x_0) + \sum_{0 \leq i \leq k-1} \big(D + \Delta(x_i, x_{i+1}, \phi)\big),$$

where $x_i \in V$ and $x_k = v$. Given that each vertex in V has exactly one cluster rooted at it, we know that $k \leq |V|$, thus $t^\phi(v) \leq U$, where $U = (|V| - 1)D + |V| \max_{u,v \in V} \Delta(u, v, \phi)$.

On the other hand, since ϕ is a rational number, it can be expressed as r/q, where r and q are integers and $q \neq 0$. If $t(v)$ is increased during the iteration, the amount of increase will be at least $1/q$. Therefore, if $\mathcal{L}$ does not converge after $|V|Uq$ iterations, then there exists a vertex $v \in V$ whose $t(v) > U \geq t^\phi(v)$, which contradicts $\mathrm{T} \leq \mathrm{T}^\phi$, which concludes our proof. □

The next result is a corollary of Lemma 7.5.5-7.5.7.

Lemma 7.5.8. $\big(\forall v \in V : t(v) > t_0(v)\big)$ *implies that ϕ is infeasible.*

Proof. Suppose otherwise that ϕ is feasible. Then, by Lemma 7.5.5 and 7.5.7, when T^ϕ is reached, we have $\big(\forall v \in V : t^\phi(v) > t_0(v)\big)$, which contradicts Lemma 7.5.6. Therefore, ϕ is infeasible. □

7.6. Optimal clustering algorithm

Given a legal clustering c, its maximum cycle ratio ϕ_c is feasible. If ϕ_c is not optimal, then we can find a feasible $\phi_{c'} < \phi_c$, which is specified in the following lemma.

Lemma 7.6.1. *Given that ϕ_c is the maximum cycle ratio of a legal clustering c, if ϕ_c is not optimal, then $\phi_c - 1/(|V|N_{\text{ff}})^2$ is also feasible, where N_{ff} is the total number of flip-flops in G.*

Proof. Let o_c denote the cycle with the maximum cycle ratio, that is, $\phi_c = d(o_c)/w(o_c)$. If ϕ_c is not optimal, it means that there exists another legal clustering c' whose maximum cycle ratio $\phi_{c'}$ is smaller than ϕ_c. Let $o_{c'}$ be the cycle with $\phi_{c'} = d(o_{c'})/w(o_{c'})$. The difference between $\phi_{c'}$ and ϕ_c can be written as

$$\phi_c - \phi_{c'} = \frac{d(o_c)}{w(o_c)} - \frac{d(o_{c'})}{w(o_{c'})} = \frac{d(o_c)w(o_{c'}) - d(o_{c'})w(o_c)}{w(o_c)w(o_{c'})} \geq \frac{1}{w(o_c)w(o_{c'})}$$

since all delays are integers. In addition, since each vertex in V has exactly one cluster rooted at it, both $w(o)$ and $w(o')$ can pass at most $|V|$ clusters. Thus, neither $w(o)$ nor $w(o')$ will be larger than $|V|N_{\text{ff}}$. Therefore, $\phi_c - \phi_{c'} \geq 1/(|V|N_{\text{ff}})^2$. In other words, $\phi_c - 1/(|V|N_{\text{ff}})^2$ is also feasible. □

It implies that we can certify the optimality of ϕ_c by checking the feasibility of $\phi_c - 1/(|V|N_{\text{ff}})^2$. The algorithm for finding the optimal ϕ is presented in Figure 7.7. It first computes a feasible ϕ by treating each vertex as a cluster, and computes a lower bound of ϕ_{lb} by Lemma 7.3.1. After that, it checks the feasibility of $\phi - 1/(|V|N_{\text{ff}})^2$ by iterative method on $\mathcal{L}$. If $\mathcal{L}$ converges, it means that we find a better clustering whose maximum cycle ratio is at most $\phi - 1/(|V|N_{\text{ff}})^2$ and can be computed by Howard's algorithm. The

evidence of $\big(\forall v \in V : t(v) > t_0(v)\big)$ or the fact that ϕ is reduced below $\phi_{\text{lb}} + 1/(|V|N_{\text{ff}})^2$ immediately certifies the optimality of the current feasible ϕ.

Algorithm `Optimal clustering`
Input: `A directed graph` $G = (V, E)$, A, D.
Output: `A clustering` c^{opt} `with` ϕ^{opt}.

$c_v^{\text{opt}} \leftarrow c_v \leftarrow \{v\}$, $\forall v \in V$;
$\phi^{\text{opt}} \leftarrow \phi \leftarrow$ `maximum cycle ratio of` G_c;
$\phi_{\text{lb}} \leftarrow$ `maximum cycle ratio of` G;
`While` $\big(\phi \geq \phi_{\text{lb}} + 1/(|V|N_{\text{ff}})^2\big)$ `do`
  $\phi \leftarrow \phi - 1/(|V|N_{\text{ff}})^2$;
  $\mathrm{T} \leftarrow \mathrm{T}_0 \leftarrow MBF(\perp, \phi)$;
  `While` $\Big(\big(\mathrm{T} \neq \mathcal{L}(\mathrm{T}, \phi)\big) \wedge \big(\exists v \in V : t(v) = t_0(v)\big)\Big)$ `do`
    $\mathrm{T} \leftarrow \mathcal{L}(\mathrm{T}, \phi)$;
  `If` $\big(\forall v \in V : t(v) > t_0(v)\big)$ `then`
    `break`;
  $\phi \leftarrow$ `maximum cycle ratio of` G_{c^ϕ};
  $c^{\text{opt}} \leftarrow c^\phi$; $\phi^{\text{opt}} \leftarrow \phi$;
`Return` c^{opt} `and` ϕ^{opt};

Figure 7.7. Pseudocode of optimal clustering algorithm.

We prove the correctness of the algorithm by showing that it returns the optimal ϕ when it terminates.

Theorem 7.6.1. *The algorithm in Figure 7.7 will return a clustering with the optimal ϕ when it terminates.*

Proof. When the algorithm terminates, we have either $\phi < \phi_{\text{lb}} + 1/(|V|N_{\text{ff}})^2$, or $\big(\forall v \in V : t(v) > t_0(v)\big)$ under $\phi - 1/(|V|N_{\text{ff}})^2$. For the first case, Lemma 7.6.1 ensures that ϕ is optimal, otherwise $\phi - 1/(|V|N_{\text{ff}})^2$ is feasible, which contradicts Lemma 7.3.1. For the second case, $\phi - 1/(|V|N_{\text{ff}})^2$ is infeasible by Lemma 7.5.8, which, by Lemma 7.6.1, implies

that ϕ is optimal. The optimal ϕ and the corresponding clustering are recorded in ϕ^{opt} and c^{opt}. □

We finally present the worst-case complexity of the algorithm in the next theorem.

Theorem 7.6.2. *The algorithm in Figure 7.7 stops in* $O\big(|V|^9|E|N_{\text{ff}}^5 D(D+\Delta_{\text{lb}})\log|V|\big)$ *time, where* $\Delta_{\text{lb}} = \max_{u,v\in V}\Delta(u,v,\phi_{\text{lb}})$ *and* N_{ff} *is the total number of flip-flops in* G.

Proof. First of all, ϕ is reduced during the execution of the outer while-loop in Figure 7.7. Since the amount of decrease in ϕ is at least $1/(|V|N_{\text{ff}})^2$ after each loop, the algorithm will terminate in $(\phi_{\text{ub}} - \phi_{\text{lb}})|V|^2N_{\text{ff}}^2$ loops, where ϕ_{ub} is an upper bound of ϕ. Since $\phi_{\text{ub}} \leq \phi_{\text{lb}} + |V|D$, the number of loops can be bounded by $|V|^3N_{\text{ff}}^2D$.

At each loop, it takes $O(|V||E|)$ to compute T_0 by the modified Bellman-Ford's algorithm. The complexity of maximum-cycle-ratio computation can be bounded by $O(|V_c|^2|E_c|)$ [**28**], or $O(A^2|V|^3|E|)$ since G_c consists of $|V|$ clusters and the size of each cluster is no larger than A.

We next analyze the complexity of checking the feasibility of ϕ. According to the proof of Lemma 7.5.7, if ϕ is feasible, iterative method will converge in $|V|Uq$ iterations, where $U = (|V|-1)D + |V|\max_{u,v\in V}\Delta(u,v,\phi)$ and q is an integer such that the product of ϕ and q is integral. Since $\phi \geq \phi_{\text{lb}}$, we have $\Delta(u,v,\phi) \leq \Delta(u,v,\phi_{\text{lb}})$, $\forall u,v \in V$. On the other hand, since ϕ is the maximum cycle ratio of a legal clustering minus $1/(|V|N_{\text{ff}})^2$, it is true that $q \leq |V|^3N_{\text{ff}}^3$. As a result, the number of iterations can be bounded by $O\big(|V|^5N_{\text{ff}}^3(D+\Delta_{\text{lb}})\big)$, where $\Delta_{\text{lb}} = \max_{u,v\in V}\Delta(u,v,\phi_{\text{lb}})$. The complexity of each iteration can be computed as $O(|V||E|\log|V|)$ by Lemma 7.5.4.

Therefore, the computational complexity of each loop is $O\big(|V|^6|E|N_{\text{ff}}^3(D+\Delta_{\text{lb}})\log|V|\big)$ and the theorem is true. □

Remark 7.6.1. *The significance of Theorem 7.6.2 is not the actual formula of the bound, but showing that the optimal clustering problem has a pseudo-polynomial time complexity. Furthermore, caution should be taken on this bound. The worst-case complexity is based on a series of assumptions that are very unlikely to be attainable in reality. For example, although we do feasible checking on a value that is slightly smaller than a given feasible ϕ, the improvement at each loop is not small. This is because the new clustering will have a different structure whose maximum cycle ratio is usually much smaller than the given ϕ. This is confirmed by our experiments in Section 7.9. Therefore, we believe that the worst case bound we obtained is just an upper bound of the actual running time. A tighter bound may exist but its mathematical analysis is so complex that we cannot deduce it so far. The efficiency of our algorithm is confirmed by the experiments.*

7.7. Speed-up techniques

7.7.1. Variations of $\mathcal{L}$

In Section 7.5, $\mathcal{L}(\mathrm{T}, \phi)$ is defined as applying all the $\mathcal{L}_v$'s, $\forall v \in V$, once followed by the modified Bellman-Ford's algorithm. In our implementation, all the $\mathcal{L}_v(\mathrm{T}, \phi)$'s are not computed at the same time. Intuitively, if previously computed $\mathcal{L}_v$'s can be taken into account in later computations of others, the convergence rate may be accelerated.

This motivates our study on a variation of $\mathcal{L}$, in which later computations of $\mathcal{L}_v$'s are based on previously computed ones, and each computation of $\mathcal{L}_v$ is followed by the modified Bellman-Ford's algorithm. Let $\mathcal{J}_v(\mathrm{T}, \phi)$ denote the vector after $t(v)$ is updated with $\mathcal{L}_v(\mathrm{T}, \phi)$, that is,

$$\mathcal{J}_v(\mathrm{T}, \phi) = \big(t(1), ..., t(v-1), \mathcal{L}_v(\mathrm{T}, \phi), t(v+1), ..., t(|V|)\big).$$

Define

$$\bar{\mathcal{L}}(\mathrm{T},\phi) \stackrel{\triangle}{=} MBF\Big(\mathcal{J}_{i_{|V|}}\big(...MBF\big(\mathcal{J}_{i_1}(\mathrm{T},\phi),\phi\big),...,\phi\big),\phi\Big),$$

where $i_1,...,i_{|V|} \in V$. It can be seen that different evaluation orders of V give different $\bar{\mathcal{L}}$'s. However, they all satisfy the following relation.

Lemma 7.7.1. *For any* $\mathrm{T} \leq \mathrm{T}^\phi$ *satisfying (7.1) and (7.5) under a feasible* ϕ *and any evaluation order of* V, $\mathcal{L}(\mathrm{T},\phi) \leq \bar{\mathcal{L}}(\mathrm{T},\phi) \leq \bar{\mathcal{L}}(\mathrm{T}^\phi,\phi) = \mathrm{T}^\phi$.

Proof. Let $\bar{\mathcal{L}}_v(\mathrm{T},\phi)$ denote the arrival time of $v \in V$ in $\bar{\mathcal{L}}(\mathrm{T},\phi)$. Since T satisfies 7.1) and (7.5) under ϕ, the definition of $\bar{\mathcal{L}}(\mathrm{T},\phi)$ implies that $\mathcal{L}_v(\mathrm{T},\phi) \leq \bar{\mathcal{L}}_v(\mathrm{T},\phi)$, $\forall v \in V$, independent of the evaluation order. It follows that $\mathcal{L}(\mathrm{T},\phi) \leq \bar{\mathcal{L}}(\mathrm{T},\phi)$. On the other hand, since $\mathcal{L}_v(\mathrm{T}^\phi,\phi) = t^\phi(v)$, we have $\mathcal{J}_v(\mathrm{T}^\phi,\phi) = \mathrm{T}^\phi$, hence $\bar{\mathcal{L}}(\mathrm{T}^\phi,\phi) = \mathrm{T}^\phi$. What remains to show is $\bar{\mathcal{L}}(\mathrm{T},\phi) \leq \mathrm{T}^\phi$. To this aim, we observe that $MBF\big(\mathcal{J}_v(\mathrm{T},\phi)\big) \leq \mathrm{T}^\phi$, $\forall v \in V$, provided that $\mathrm{T} \leq \mathrm{T}^\phi$. Based on this, we can show by induction that $\bar{\mathcal{L}}(\mathrm{T},\phi) \leq \mathrm{T}^\phi$. Therefore, the lemma is true. □

As a corollary, the next result ensures that we can apply iterative method on $\bar{\mathcal{L}}$ to reach T^ϕ.

Corollary 7.7.1.1. *If* ϕ *is feasible, applying iterative method on* $\bar{\mathcal{L}}$ *will converge to* T^ϕ *in a finite number of iterations, independent of the evaluation order of* V.

Proof. Since ϕ is feasible, $\mathcal{L}$ is finitely convergent by Lemma 7.5.7. Let K be the number of iterations such that $\mathcal{L}^K(\mathrm{T}_0,\phi) = \mathrm{T}^\phi$. The corollary can be proved if we can show that $\bar{\mathcal{L}}^K(\mathrm{T}_0,\phi) = \mathrm{T}^\phi$, or equivalently, $\mathcal{L}^K(\mathrm{T}_0,\phi) \leq \bar{\mathcal{L}}^K(\mathrm{T}_0,\phi) \leq \bar{\mathcal{L}}^K(\mathrm{T}^\phi,\phi)$.

The former part can be derived from Lemma 7.7.1 because

$$\mathcal{L}^K(\mathrm{T}_0, \phi) \leq \bar{\mathcal{L}}\big(\mathcal{L}^{K-1}(\mathrm{T}_0, \phi)\big) \leq \ldots \leq \bar{\mathcal{L}}^K(\mathrm{T}_0, \phi)$$

The latter part is also a consequence of Lemma 7.7.1 since $\mathrm{T}_0 \leq \mathrm{T}^\phi$. □

7.7.2. Reduced clustering representation

It was shown in [**28**] that Howard's algorithm was by far the fastest algorithm for maximum-cycle-ratio computation. Given a clustered circuit $G_c = (V_c, E_c)$ with edge delays and weights specified, Howard's algorithm finds the maximum cycle ratio in $O(N_c|E_c|)$ time, where N_c is the product of the out-degrees of all the vertices in V_c. Since vertex replication is allowed, N_c and $|E_c|$ could be $|V|N$ and $|V||E|$ respectively, where N is the product of the out-degrees of the vertices in V.

To reduce the complexity, we propose a reduced clustering representation for maximum-cycle-ratio computation. For each cluster, we use edges from its inputs to its output (root) to represent the paths between them such that the delay and weight of an edge correspond to the delay and weight of an acyclic input-to-output path. Figure 7.8 shows the reduced representation of the clustered circuit in Figure 7.3(b).

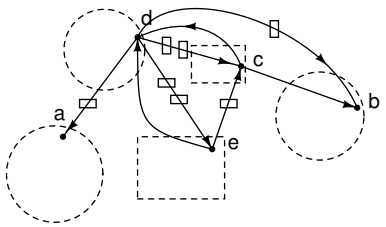

Figure 7.8. The reduced clustering representation of Figure 7.3(b).

Let ϕ_c^r denote the maximum cycle ratio of the reduced representation for clustering c. The following lemma formulates the relation among ϕ_c, ϕ_c^r and the lower bound ϕ_{lb} defined in Lemma 7.3.1.

Lemma 7.7.2. *For any clustering c, $\phi_c = \max(\phi_c^r, \phi_{\text{lb}})$.*

Proof. All the cycles in G_c can be classified into two groups according to whether they contain an inter-cluster edge or not. If a cycle contains only intra-cluster edges, its maximum cycle ratio is upper bounded by ϕ_{lb}. If a cycle contains inter-cluster edges, it is present in the reduced representation and thus is upper bounded by ϕ_c^r. □

One benefit of the reduced clustering representation is that we can now represent the clustered circuit without explicit vertex replication, that is, using V instead of V_c. Let $G_c^r = (V, E_c^r)$ denote the reduced representation for clustering c. We call an edge in G_c^r *redundant* if its removal will not affect the maximum cycle ratio of G_c^r. The following lemma provides a criterion to prune the redundant edges so that Howard's algorithm can find the maximum cycle ratio of G_c^r more efficiently.

Lemma 7.7.3. *Let c denote a feasible clustering under ϕ, E_c^r denote its reduced representation, e_1 and e_2 denote two edges from $u \in V$ to $v \in V$ in E_c^r, $d(e_1)$ and $d(e_2)$ denote their delays respectively, and $w(e_1)$ and $w(e_2)$ denote their weights respectively. If $w(e_1) \geq w(e_2)$ and $d(e_1) - w(e_1)\phi \geq d(e_2) - w(e_2)\phi$, then e_2 can be pruned.*

Proof. Since c is feasible under ϕ, we know that $\phi \geq \phi_c \geq \phi_c^r$. If e_2 is not involved in any cycle in G_c^r, then e_2 can be safely pruned as it will not affect the computation of the maximum cycle ratio. Otherwise, let o_2 be a cycle in G_c^r involving e_2, and o_1 be the cycle

in G_c^r such that $o_1 = \{e_1\} \cup o_2 - \{e_2\}$. What remains to show is that if o_2 is a critical cycle under ϕ_c^r, so is o_1.

By definition, if o_2 is a critical cycle under ϕ_c^r, then $d(o_2) - w(o_2)\phi_c^r = 0$. Since o_1 differs o_2 in e_1 only, we have

$$\begin{aligned}
& d(o_1) - w(o_1)\phi_c^r \\
= \;& d(o_2) - d(e_2) + d(e_1) - \big(w(o_2) - w(e_2) + w(e_1)\big)\phi_c^r \\
= \;& d(e_1) - w(e_1)\phi_c^r - \big(d(e_2) - w(e_2)\phi_c^r\big) \\
= \;& d(e_1) - w(e_1)\phi - \big(d(e_2) - w(e_2)\phi\big) + \big(w(e_1) - w(e_2)\big)(\phi - \phi_c^r) \\
\geq \;& 0,
\end{aligned}$$

provided that $d(e_1) - w(e_1)\phi \geq d(e_2) - w(e_2)\phi$ and $w(e_1) \geq w(e_2)$. On the other hand, $d(o_1) \leq w(o_1)\phi_c^r$ since ϕ_c^r is the maximum cycle ratio. Therefore, $d(o_1) = w(o_1)\phi_c^r$, i.e., o_1 is also a critical cycle under ϕ_c^r. This implies that we can safely remove e_2 and the resulting representation has the same maximum cycle ratio as G_c^r. □

The next result is a corollary of the above lemma that provides an upper bound for the number of non-redundant edges in E_c^r.

Corollary 7.7.3.1. *The number of non-redundant edges in E_c^r is $O(|V|^2 N_{\text{ff}})$, where N_{ff} is the total number of flip-flops in G.*

Proof. First of all, for all $e \in E_c^r$, we know that $w(e) \leq N_{\text{ff}}$. According to Lemma 7.7.3, there are at most N_{ff} non-redundant edges from $u \in V$ to $v \in V$. Therefore, the corollary is true. □

In practice, the number of flip-flops on an input-to-output path in a cluster is much smaller than N_{ff}, which enables the efficiency of the reduced clustering representation.

In our implementation, we employ another two parameters $\text{pd} : V \to \{R\}$ and $\text{pw} : V \to \{Z\}$ to record the path delays and weights from the inputs of a cluster to its output, respectively. More specifically, we set $\text{pw}(u) = \text{pd}(u) = \emptyset$, $\forall u \in V$, before $\mathcal{L}_v(\text{T}, \phi)$ is about to be carried out for some $v \in V$. After that, whenever a vertex u is put in c'_v, we compute the pd and pw values of its preceding vertices based on $\text{pd}(u)$ and $\text{pw}(u)$, followed by pruning.

7.8. Cluster and replication reduction

In this section, we briefly review the techniques that were used in [**73, 77**] to reduce the number of clusters and vertex replication.

In Section 7.2, we assume that each cluster has one output. If this assumption is relaxed, a post-processing step can be added to reduce the number of clusters. For example, if the arrival time of a vertex $v \in V$ in its own cluster is equal to the arrival time of a copy of v in another cluster, then the entire cluster at v can be removed, and replaced by the copy.

Replicated vertices can also be reduced as follows. If the arrival times of two copies of a vertex differ by an amount greater than or equal to the inter-cluster delay D, then the output of the copy with the smaller arrival time can replace the copy with the larger arrival time. In addition, there are slacks available for vertices on noncritical paths and their arrival times need not be the least fixpoint. This property can also be used to further remove replicated vertices. Once this reduction of replicated vertices is carried out, there may be clusters that are not completely filled. We can merge some of the clusters, provided that the area bound is not exceeded. Using these techniques, the area overhead can be reduced to 14%, as shown in [**20**].

It is worthy to point out that it is both unnecessary and memory-wise prohibitive to keep all clusters during clustering under a given ϕ. The cluster rooted at each vertex $v \in V$ is dynamically built during the execution of procedure $\mathcal{L}_v(\mathrm{T}, \phi)$ and released when the procedure finishes. The existence of such a cluster is implied by the updated arrival time of v. The whole clustered circuit needs to be built only when we want to compute its maximum cycle ratio, or when ϕ^{opt} is found. For the former case, reduced clustering representation helps to manage the storage requirement. For the latter case, a reasonable overhead can be obtained using the aforementioned techniques.

7.9. Experimental results

We implemented the algorithm in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. To compare with the algorithm in [**77**], we used the same test files, which were generated from the ISCAS-89 benchmark suite. For each test case, we introduced a flip-flop with directed edges from each PO to it and from it to each PI so that every PI-to-PO path became a cycle. As in [**77**], the size and delay of each gate was set to 1, intra-cluster delays were 0, and inter-cluster interconnects had delays $D = 2$. The circuits used are summarized in Table 7.1. We also list the maximum cycle ratio of the circuit before clustering in column ϕ_{lb}, which provides a lower bound of the solution by Lemma 7.3.1.

Although theoretically the algorithm in Figure 7.7 will reach the exact solution without being provided a precision, we have to consider the impact of floating point error introduced by practical finite precision arithmetic, due to the divisions involved in the maximum-cycle-ratio computation. In our experiments, we set the error to be 0.001. Since $1/(|V|N_{\mathrm{ff}})$ is generally smaller than 0.001, we set the precision of ϕ^{opt} to be 0.01.

Table 7.1. Sequential Circuits from ISCAS-89

Circuit	$\|V\|$	$\|E\|$	N_{ff}	ϕ_{lb}
s208	104	183	40	10.00
s349	161	285	35	14.00
s420	218	385	84	12.00
s635	286	478	97	66.00
s838	446	789	172	16.00
s1196	529	1024	31	24.00
s1423	657	1170	239	53.00
s1512	780	1286	185	22.50
s3330	1789	2890	435	14.00
s4863	2342	4093	105	30.00
s5378	2779	4262	301	21.00
s9234	5597	6932	333	38.00
s35932	16065	28590	5815	27.00
s38584	19253	33061	7372	48.00

For each circuit, we tested three size bounds: A is 5%, 10% and 20% of the number of gates. The optimal maximum-cycle-ratio for each circuit is shown in Table 7.2.

Table 7.2. Optimal Maximum-cycle-ratio

Circuit	ϕ^{opt}		
	$A = 5\%\|V\|$	$A = 10\%\|V\|$	$A = 20\%\|V\|$
s208	13.00	11.00	10.00
s349	18.00	16.00	14.67
s420	14.00	13.00	12.00
s635	75.00	70.00	68.00
s838	17.00	16.00	16.00
s1196	26.00	25.00	24.00
s1423	55.00	53.00	53.00
s1512	23.78	22.50	22.50
s3330	14.33	14.00	14.00
s4863	30.25	30.00	30.00
s5378	21.00	21.00	21.00
s9234	38.00	38.00	38.00
s35932	27.00	27.00	27.00
s38584	48.00	48.00	48.00

To illustrate the advantage of our incremental algorithm over binary search, we ran binary search to find the optimal maximum-cycle-ratio using the proposed iterative method as a subroutine for feasibility checking. The lower bound of the binary search was ϕ_{lb} and the upper bound was the maximum-cycle-ratio of the clustering where each vertex itself is a cluster. The binary search precision was also set to be 0.01. We report the results in Table 7.4, where column "#step" lists the number of search steps and column "time(s)" lists the running time in seconds. "BS" refers to the binary search based algorithm and "INC" refers to our incremental algorithm. Row "arith" ("geo") gives the arithmetic (geometric) mean of the running times. It can be seen that the incremental algorithm is more efficient.

To compare the running time in [**77**], where the optimal clock period is integral, we set the precision of ϕ^{opt} to be 1 and ran the algorithm again for $A = 5\%|V|, 10\%|V|, 20\%|V|$ respectively. The obtained ϕ^{opt} matches the result in [**77**] for all the scenarios of A. The only running time information given in [**77**] is the largest running time per step among the three scenarios, which we list in Table 7.3 under column "[**77**]". We then compute ours in column "ours". Note that the running time from [**77**] was based on an UltraSPARC 2 workstation.

We observe that, for most of the circuits, our algorithm finds the optimal solution in just a few steps, which is generally less than the number of iterations conducted in a binary search, which are not given in [**77**].

7.10. Conclusion

Processing rate, defined as the product of frequency and throughput, is identified as an important metric for sequential circuits. We show that the processing rate of a sequential circuit is upper bounded by the reciprocal of its maximum cycle ratio, which is only dependent on the clustering of the circuit. The problem of processing rate optimization is

Table 7.3. Running Time Comparison with [**77**] (Seconds)

Circuit	`time/step (s)`	
	[**77**]	ours
s208	0.05	0.00
s349	1.48	0.02
s420	0.25	0.00
s635	0.76	0.03
s838	1.06	0.01
s1196	0.65	0.02
s1423	5.20	0.04
s1512	9.75	0.06
s3330	8.52	0.17
s4863	658.37	1.04
s5378	60.62	0.31
s9234	n/a	1.20
s35932	n/a	11.46
s38584	n/a	11.07

formulated as seeking an optimal clustering with minimal maximum-cycle-ratio in a general graph. An iterative algorithm is proposed that finds the minimal maximum-cycle-ratio. Since our algorithm avoids binary search and is essentially incremental, it has the potential to be combined with other optimization techniques, such as gate sizing, budgeting, etc., thus can be used in incremental design methodologies [**17**]. In addition, since maximum cycle ratio is a fundamental metric, the proposed algorithm can be adapted to suit other traditional designs.

Table 7.4. Improvement over Binary Search in Running Time

Circuit	$A = 5\%\|V\|$				$A = 10\%\|V\|$				$A = 20\%\|V\|$			
	#step		time(s)		#step		time(s)		#step		time(s)	
	BS	INC	BS	INC	BS	INC	BS	INC	BS	INC	BS	INC
s208	12	4	0.98	0.20	12	4	0.14	0.09	12	3	0.01	0.04
s349	13	14	2.56	1.73	13	21	0.81	0.47	13	14	4.83	1.39
s420	13	3	1.95	0.33	13	3	0.94	0.34	13	2	0.01	0.00
s635	15	4	19.20	5.71	15	4	66.14	8.02	15	4	65.64	6.43
s838	13	3	2.18	0.69	13	2	0.05	0.01	13	2	0.05	0.01
s1196	14	3	8.05	2.86	14	3	21.19	3.40	14	2	0.08	0.02
s1423	15	3	6.68	2.77	15	2	0.43	0.07	15	2	0.47	0.06
s1512	14	15	76.08	166.36	14	8	1.37	0.79	14	8	0.32	0.68
s3330	13	11	16.15	11.34	13	10	0.79	1.82	13	10	0.79	1.82
s4863	14	8	1688.22	133.54	14	5	3.27	3.85	14	5	1.86	3.70
s5378	13	3	2.66	0.94	13	3	1.61	0.73	13	3	1.36	0.73
s9234	14	3	12.86	3.78	14	3	5.86	2.98	14	3	6.97	2.98
s35932	14	4	108.47	48.13	14	4	46.26	47.79	14	4	48.54	46.59
s38584	15	2	104.56	21.92	15	2	47.77	21.90	15	2	58.22	21.46
arith			3.70X	1			3.00X	1			3.09X	1
geo			2.87X	1			2.22X	1			1.78X	1

CHAPTER 8

Design Closure Driven Delay Relaxation Based on Convex Cost Network Flow

Design closure is said to occur when constraints from high levels are satisfied at the low level. With the advent of ultra deep sub micron era, achieving design closure is becoming harder and harder. Inaccuracy of system level predictions, unpredictability in circuit behavior, critical design objectives, high degree of sensitivity among various design objectives are among a few factors to name.

Delay relaxation is a technique at RTL (Register Transfer Level) that relaxes the timing constraints of functional resources by assigning extra clock latencies (or clock cycles), called budgets, to them without violating any of the data flow or scheduling constraints. The problem of delay relaxation with maximum sum of budgets is refereed to as *maximum budget delay relaxation* problem. Polynomial algorithms can be found in [**74**]. In [**37**], a polynomial optimal algorithm was proposed using network flow technique. The same technique was used in [**106**] for sequential budgeting. It was shown in [**98**] that an RTL design with maximum budget resulted in fewer design iterations and faster design closure.

However, a design with maximum budget usually has many critical edges (edges with zero latency slack) since the budget of none of the resources can be further increased. These critical edges will have tighter timing constraints in later stages of placement and routing. On the other hand, due to aggressive technology scaling and increasing operating frequencies,

the delay of a global interconnect can be longer than one clock period even with buffer insertion, which makes the timing constraints on critical edges even harder to satisfy.

In [**110, 18, 58, 75, 57**], wire pipelining was applied while considering placement and availability of pipeline registers. In [**110, 58, 57**], retiming was explored to distribute pipeline registers to fulfill communication buffering requirements on global interconnects. Although retiming helps to relieve the criticality of global interconnects, it cannot change the total number of registers along a topological cycle. Therefore, we need to integrate interconnect planning in high level synthesis to reduce the complexity of physical layout stage and leverage that burden on early stages of design flow.

Recently, researchers in [**94**] explored delay relaxation for interconnect budgeting at RTL. An open problem was posted whose objective is an RTL design with maximum budget such that the number of non-critical edges is also maximized. They formulated the problem as a mixed integer linear programming and proposed a heuristic algorithm to solve it. Compared with an arbitrary maximum budget solution, design closure was achieved 2.8 times faster on average using the solution given by their algorithm. In addition, they found that the trade-off between resource budgeting and interconnect budgeting can be further leveraged to improve circuit performance.

Our contribution in this chapter is twofold. Firstly, we propose a general formulation for design closure driven delay relaxation problem. The open problem in [**94**] is a special case of our formulation. We also relate the problem to retiming and propose a convex retiming formulation. Secondly, we show that the general formulation can be transformed into a convex cost integer dual network flow problem and solved in strongly polynomial time by

the approach in [**2**]. The transformed formulation is also amicable to incorporating the trade-off between resource budgeting and interconnect budgeting. Experimental results confirm the efficiency of the approach.

The rest of this chapter is organized as follows. Section 8.1 formulates the general problem where resource budgeting and interconnect budgeting are treated as two separate objectives. Section 8.2 merges the two objectives to obtain a dual network flow problem, facilitating a single-step solution. As another point of view, Section 8.3 presents a convex retiming formulation that relates the problem to retiming. Theoretical approach from [**2**] is reviewed in Section 8.4 and Section 8.5, followed by experimental results in Section 8.6. Conclusions are drawn in Section 8.7.

8.1. Problem formulation

We model a data flow graph (DFG) as a directed acyclic graph (DAG) $G = (V, E)$, where V is the set of inputs and outputs of functional resources and $E = E_1 \cup E_2$ is the union of two subsets: input-to-output relations E_1 and interconnects E_2 among resources. Figure 8.1 illustrates an example DFG and its corresponding DAG, where dashed edges are in E_1 and solid edges are in E_2.

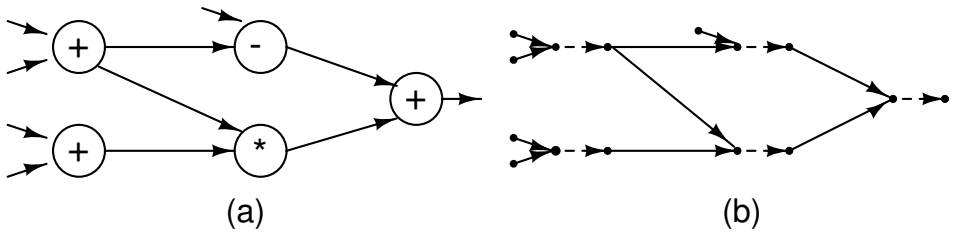

Figure 8.1. (a) A DFG and (b) its corresponding DAG.

For all $e \in E$, we use $d(e) \in \mathcal{Z}^+$ (positive integer set) to denote the original clock latencies (or clock cycles) on that edge. For example, if $e \in E_1$, $d(e)$ could be the minimum amount

of clock cycles determined by the functionality of that resource. We use $b(e) \in \mathcal{Z}^+$ to denote the budget we assign on edge $e \in E$ during delay relaxation. Each resource edge $e \in E_1$ is associated with a concave function δ_e such that $\delta_e\big(b(e)\big) \in \mathcal{Z}^+$ represents the benefit we gain by assigning $b(e)$ budget on edge e. More latencies for a resource means more possible slow-down in its logic components, which can be transformed to improvements in area, power dissipation, or other design quality metrics. Similarly, each interconnect edge $e \in E_2$ is associated with a concave function λ_e such that $\lambda_e\big(b(e)\big) \in \mathcal{Z}^+$ represents the budgeting gain on it. More latencies for an interconnect helps to further relieve its criticality. Intuitively, concave functions δ_e and λ_e help to distribute budgets evenly over all edges. Without loss of generality, we assume that all concave functions are linear between successive integers. We use $T \in \mathcal{Z}^+$ to denote the given latency upper bound at primary outputs (POs).

We introduce $t(v)$ to represent the arrival time at $v \in V$. More specifically, $t(v)$ is the maximum latency along any path from primary inputs (PIs) to v. The validity of a delay relaxation is defined by the following conditions.

$$P0(t) \stackrel{\triangle}{=} \big(\forall v \in V : 0 \leq t(v) \leq T\big)$$

$$P1(b) \stackrel{\triangle}{=} \big(\forall e \in E : 0 \leq b(e) \leq T - d(e)\big)$$

$$P2(t, b) \stackrel{\triangle}{=} \big(\forall (u, v) \in E : t(v) \geq t(u) + d(u, v) + b(u, v)\big)$$

The condition $P0$ follows from the requirement that the arrival times at each PO after delay relaxation should be no larger than the given latency constraint T. For the same reason, the condition $P1$ assigns a budget between 0 and $T - d(e)$ to each edge $e \in E$. The condition $P2$ computes the arrival time at each vertex. Based on these conditions, we can

characterize a valid delay relaxation b by $P(b)$, defined as

$$P(b) \stackrel{\triangle}{=} \big(\exists t : P0(t) \wedge P1(b) \wedge P2(t,b)\big)$$

We use $\Delta(b)$ and $\Lambda(b)$ to denote the total gain on E_1 and E_2 respectively, i.e.,

$$\Delta(b) \stackrel{\triangle}{=} \sum_{e \in E_1} \delta_e\big(b(e)\big), \qquad \Lambda(b) \stackrel{\triangle}{=} \sum_{e \in E_2} \lambda_e\big(b(e)\big).$$

The problem we want to solve can be formulated as follows.

Problem 8.1.1.

Given a directed acyclic graph $G = (V, E_1, E_2, d, \delta, \lambda, T)$, find a valid delay relaxation b with maximum $\Delta(b)$ such that $\Lambda(b)$ is also maximized.

This formulation is very general. δ_e and λ_e can be chosen independently according to the application's needs. For example, maximum budget delay relaxation is given by choosing $\delta_e\big(b(e)\big) = b(e)$, $\forall e \in E_1$. If we set

$$\lambda_e\big(b(e)\big) = \begin{cases} 0, & \text{if } b(e) = 0 \\ 1, & \text{if } b(e) > 0 \end{cases}$$

for all $e \in E_2$, we actually maximize the number of non-critical edges. Both δ_e and λ_e defined above are concave. Therefore, the problem of finding a maximum budget delay relaxation with minimal number of critical edges in [**94**] is a special case of Problem 8.1.1.

8.2. Transformation to a convex cost integer dual network flow problem

The formulation of Problem 8.1.1 seems to imply a two-step process, i.e., finding the delay relaxations b that maximize $\Delta(b)$ and then choosing among them the one that has the

maximum $\Lambda(b)$. Intuitively, if these two objectives can be appropriately combined into one equivalent alternative, then we may be able to solve it efficiently in one step.

First of all, we define λ_s as follows.

$$\lambda_s \stackrel{\triangle}{=} \sum_{e \in E_2} \big(\max_{0 \le i \le T - d(e)} \lambda_e(i) - \min_{0 \le i \le T - d(e)} \lambda_e(i) \big)$$

Consider the following problem.

Problem 8.2.1.

$$\begin{aligned} \text{Maximize} \quad & (\lambda_s + 1)\Delta(b) + \Lambda(b) \\ \text{subject to} \quad & P(b) \end{aligned}$$

It turns out that a solution to the above problem is the solution we want.

Theorem 8.2.1. *A solution to Problem 8.2.1 is also a solution to Problem 8.1.1.*

Proof. Let b' be a solution to Problem 8.1.1 and b^* be a solution to Problem 8.2.1. We first show that $\Delta(b^*) = \Delta(b')$. Suppose otherwise $\Delta(b^*) \neq \Delta(b')$, we have $\Delta(b^*) < \Delta(b')$ since b' maximizes $\Delta(b)$ for all valid b. On the other hand, since b^* maximizes $(\lambda_s + 1)\Delta(b) + \Lambda(b)$ and both b^* and b' are valid, we have

$$(\lambda_s + 1)\Delta(b^*) + \Lambda(b^*) \ge (\lambda_s + 1)\Delta(b') + \Lambda(b'),$$

that is, $(\lambda_s + 1)\big(\Delta(b') - \Delta(b^*)\big) \le \Lambda(b^*) - \Lambda(b')$. Given that $\Delta(b')$ and $\Delta(b^*)$ are in $\mathcal{Z}^+$, it follows that $\Delta(b') - \Delta(b^*) \ge 1$. Thus, $\Lambda(b^*) - \Lambda(b) \ge \lambda_s + 1$. However, the definition of λ_s implies that $\Lambda(b^*) - \Lambda(b) \le \lambda_s$, which is a contradiction. Therefore, $\Delta(b^*) = \Delta(b')$.

We then show that $\Lambda(b^*) = \Lambda(b')$. Otherwise $\Lambda(b^*) < \Lambda(b')$, thus $(\lambda_s+1)\Delta(b^*)+\Lambda(b^*) < (\lambda_s+1)\Delta(b')+\Lambda(b')$, which contradicts that b^* maximizes $(\lambda_s+1)\Delta(b)+\Lambda(b)$. Therefore, the theorem is true. □

If we denote $k = \lambda_s + 1$, $h = 1$, and define

$$\bar{F}_e\big(b(e)\big) \stackrel{\triangle}{=} \begin{cases} -k\delta\big(b(e)\big), & \text{if } e \in E_1 \\ -h\lambda\big(b(e)\big), & \text{if } e \in E_2 \end{cases}$$

then we can rewrite Problem 8.2.1 as

$$\begin{aligned} \text{Minimize} \quad & \textstyle\sum_{e\in E} \bar{F}_e\big(b(e)\big) \\ \text{subject to} \quad & P(b) \end{aligned}$$

Note that $\bar{F}_e$ is convex on each $e \in E$. This problem is also known as *convex cost integer dual network flow problem* [**2**]. It is worthy to point out that the trade-off between resource budgeting $\Delta(b)$ and interconnect budgeting $\Lambda(b)$ can be handled smoothly by changing Δ and Λ.

8.3. Convex retiming formulation

In this section, we relate Problem 8.2.1 to retiming [**53**]. We propose a convex retiming formulation for Problem 8.2.1 and show that it can be solved by relocating the latencies in the graph, which is similar to traditional retiming where registers are relocated for performance/area optimization.

First of all, note that if $t(v) > t(u) + d(u,v) + b(u,v)$ for some edge $(u,v) \in E_1$ in the solution, then $\delta_{(u,v)}\big(b(u,v)\big)$ has to be the maximum over the range $[0, T - d(u,v)]$

since $\delta_{(u,v)}$ is convex. We can modify $\delta_{(u,v)}$ by assigning $\delta_{(u,v)}(i) = \delta_{(u,v)}\big(b(u,v)\big)$ for all $b(u,v) \leq i \leq T - d(u,v)$. Similar modification applies to λ_e. By doing so, the modified problem has a solution with all edges being critical, from which a solution to the original problem can be easily generated. We henceforth assume that there exists a solution to Problem 8.2.1 with all edges being critical after the budget assignment.

Such a budget assignment b_0 can be obtained by the following process. First of all, we compute the arrival time at each vertex with respect to $b(e) = 0$, $\forall e \in E$, i.e., no budgeting at all. After that, we assign $b_0(u,v) = T - t(u)$ for each (u,v) with v being a PO, and assign $b_0(u,v) = t(v) - t(u)$ for other edges. We use $r : V \rightarrow \mathcal{Z}$ to represent the number of latencies that are moved from the outgoing edges to the incoming edges of each vertex. Denote

$$b_r(u,v) \stackrel{\triangle}{=} b_0(u,v) + r(v) - r(u).$$

Consider the following problem.

$$\begin{aligned} \text{Minimize} \quad & \textstyle\sum_{e \in E} \bar{F}_e\big(b_r(u,v)\big) \\ \text{subject to} \quad & b_r(u,v) \geq 0, \quad \forall (u,v) \in E \\ & r(v) = 0, \quad \forall v \in \{PI, PO\} \end{aligned}$$

We refer to it as *convex retiming* formulation. The next theorem shows that a solution to the convex retiming formulation can be used to generate a solution to Problem 8.2.1.

Theorem 8.3.1. *Let r^* be a solution to the convex retiming formulation. Then $b_{r^*}(u,v) = b_0(u,v) + r^*(v) - r^*(u)$, $\forall (u,v) \in E$, is a solution to Problem 8.2.1.*

Proof. We first show that b_{r^*} is valid, i.e., $P(b_{r^*})$. Let t^* be the arrival times with respect to b_{r^*}, thus $P2(t^*, b_{r^*})$ is true. With $r^*(v) = 0$, $\forall v \in \{PI, PO\}$, the number of latencies along any path from PI to PO cannot be changed by retiming, and thus are kept as T. Therefore, $P0(t^*)$ is true, which, together with $P2(t^*, b_{r^*})$, implies $b_{r^*}(e) \leq T - d(e)$, hence $P1(b_{r^*})$ is also satisfied. What remains is to show that b_{r^*} minimizes $\sum_{e \in E} \bar{F}_e\big(b(u,v)\big)$.

Suppose otherwise that b' is the solution to Problem 8.2.1 such that $\sum_{e \in E} \bar{F}_e\big(b'(e)\big) < \sum_{e \in E} \bar{F}_e\big(b_{r^*}(e)\big)$. Then, we can obtain a valid retiming r' from b' such that $r'(v) - r'(u) \geq b'(u,v) - b_0(u,v)$, $\forall (u,v) \in E$. This is possible by longest path computation on the acyclic graph G as follows. First of all, let $r'(i) = 0$, $\forall i \in \{PI\}$. Consider any $(u,v) \in E$. Let p_1 be the longest path to v, p_2 be the longest path to u, and $p_3 = p_2 \cup \{(u,v)\}$. We use $b(p)$ to denote the latency of path p. Then $r'(v) = b'(p_1) - b_0(p_1)$ and $r'(u) = b'(p_2) - b_0(p_2)$. Since b_0 has no non-critical edges, any path to v should have the same latency in b_0. This is also true for b'. Thus, $b'(p_1) - b_0(p_1) = b'(p_3) - b_0(p_3)$. In other words, $r'(v) = r'(u) + b'(u,v) - b_0(u,v)$, $\forall (u,v) \in E$. Consider any PO j and the longest path p to it from a PI i, we have $r'(j) = r'(i) + b'(p) - b_0(p)$. Since $b'(p) = b_0(p) = T$, it follows that $r'(j) = r'(i) = 0$. Therefore, r' satisfies both constraints of convex retiming. However,

$$\sum_{e \in E} \bar{F}_e\big(b_{r'}(u,v)\big) = \sum_{e \in E} \bar{F}_e\big(b'(e)\big) < \sum_{e \in E} \bar{F}_e\big(b_{r^*}(u,v)\big),$$

which contradicts that r^* minimizes $\sum_{e \in E} \bar{F}_e\big(b_r(u,v)\big)$. It concludes the proof. □

When both δ_e and λ_e are linear, the convex retiming formulation is reduced to minimum area retiming [**53**], which can be solved by a minimum cost flow algorithm. For general convex functions, Ahuja *et al.* [**2**] showed that Problem 8.2.1 can be transformed into a

convex primal network flow problem and solved in strongly polynomial time by cost-scaling approach. We review his approach in the following.

8.4. Transformation to a primal network flow problem

8.4.1. Constraint simplification

First of all, $\bar{F}_e$ can be modified to eliminate the bounds on $b(e)$ as follows.

$$F'_e\big(b(e)\big) \stackrel{\triangle}{=} \begin{cases} \bar{F}_e(0) - Mb(e), & \text{if } b(e) < 0 \\ \bar{F}_e\big(b(e)\big), & \text{if } 0 \le b(e) \le T - d(e) \\ \bar{F}_e\big(T - d(e)\big) + M\big(b(e) + d(e) - T\big), & \text{otherwise} \end{cases}$$

where M is a sufficiently large number such that F'_e is still a convex function and minimizing $\sum_{e\in E} F'_e\big(b(e)\big)$ subject to only $P0(t)$ and $P2(t,b)$ will not have a solution that violates $P1(b)$. For example, $M = \sum_{e\in E} \max_{0\le b(e)\le T-d(e)} \bar{F}_e\big(b(e)\big)$ will suffice.

Similarly, the bounds on $t(v)$ can also be eliminated by adding into the objective a convex cost function $B_v\big(t(v)\big)$ defined as follows.

$$B_v\big(t(v)\big) \stackrel{\triangle}{=} \begin{cases} -Mt(v), & \text{if } t(v) < 0 \\ 0, & \text{if } 0 \le t(v) \le T \\ M\big(t(v) - T\big), & \text{if } t(v) > T \end{cases}$$

Together with the discussion in Section 8.3 that there exists a solution for which $P2(t,b)$ is an equality constraint, Problem 8.2.1 can be transformed to

$$\begin{aligned} \text{Minimize} \quad & \sum_{e\in E} F'_e\big(b(e)\big) + \sum_{v\in V} B_v\big(t(v)\big) \\ \text{subject to} \quad & t(u) - t(v) = -d(u,v) - b(u,v), \quad \forall (u,v) \in E \end{aligned}$$

To further simplify it, let

$$w(e) \stackrel{\triangle}{=} -d(e) - b(e), \quad \forall e \in E$$

Define function $F_e\big(w(e)\big)$ such that $F_e\big(w(e)\big) = F'_e\big(b(e)\big)$, which is illustrated in Figure 8.2.

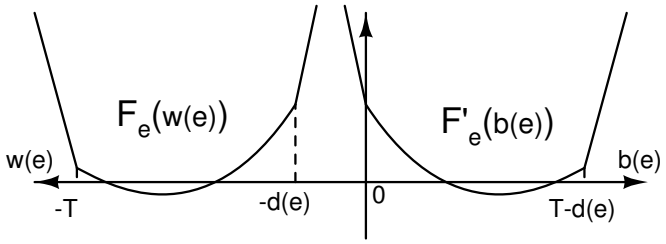

Figure 8.2. Illustration of $F'_e\big(b(e)\big)$ and $F_e\big(w(e)\big)$.

Note that $F_e\big(w(e)\big)$ is also convex. Substituting $F'_e\big(b(e)\big)$ by $F_e\big(w(e)\big)$, Problem 8.2.1 becomes

$$\begin{aligned} \text{Minimize} \quad & \textstyle\sum_{e\in E} F_e\big(w(e)\big) + \sum_{v\in V} B_v\big(t(v)\big) \\ \text{subject to} \quad & t(u) - t(v) = w(u,v), \quad \forall (u,v) \in E \end{aligned}$$

8.4.2. Problem transformation by Lagrangian relaxation

In this section we will apply Lagrangian relaxation [**68**] to transform Problem 8.2.1 to a primal network flow problem. Lagrangian relaxation is a general technique for solving constrained optimization problems. In Lagrangian relaxation, "troublesome" constraints are "relaxed" and incorporated into the objective after multiplying them by constants called Lagrangian multipliers, one multiplier for each constraint. For given multipliers, the relaxed problem is called *Lagrangian subproblem*. Finding the optimal multipliers under which

the Lagrangian subproblem attains the best objective value is called *Lagrangian multiplier problem*.

The Lagrangian subproblem of Problem 8.2.1 is as follows.

$$L(x) = \min_{w,t} \Big(\sum_{e \in E} F_e\big(w(e)\big) + \sum_{v \in V} B_v\big(t(v)\big) - \sum_{(u,v) \in E} x(u,v)\big(w(u,v) + t(v) - t(u)\big) \Big),$$

where $x(u,v)$ is the Lagrangian multiplier associated with the constraint $t(u) - t(v) = w(u,v)$. Note that

$$\sum_{(u,v) \in E} x(u,v)\big(t(v) - t(u)\big) = \sum_{v \in V} t(v)\big(\sum_{(i,v) \in E} x(i,v) - \sum_{v,j} x(v,j)\big)$$

Define

$$x(v,0) \stackrel{\triangle}{=} \sum_{(i,v) \in E} x(i,v) - \sum_{(v,j) \in E} x(v,j), \quad \forall v \in V$$

Substituting it into $L(x)$ yields

$$L(x) = \min_{w,t} \Big(\sum_{e \in E} \big(F_e\big(w(e)\big) - x(e)w(e)\big) + \sum_{v \in V} \big(B_v\big(t(v)\big) - x(v,0)t(v)\big) \Big)$$

To simplify $L(x)$, a vertex 0 is introduced into G as well as edges $(v,0)$, $\forall v \in V$. Let $G^0 = (V^0, E^0)$ be the resultant DAG. We define

$$w(v,0) \stackrel{\triangle}{=} t(v), \qquad F_{(v,0)}\big(w(v,0)\big) \stackrel{\triangle}{=} B_v\big(t(v)\big)$$

Then the Lagrangian subproblem becomes

$$L(x) = \min_{w} \sum_{e \in E^0} \big(F_e\big(w(e)\big) - x(e)w(e)\big)$$
$$\text{subject to} \quad \sum_{(i,v)\in E^0} x(i,v) = \sum_{(v,j)\in E^0} x(v,j), \quad \forall v \in V^0$$

It is important to notice that, for a given x, each term of $\sum_{e\in E^0} F_e\big(w(e)\big) - x(e)w(e)$ is a function of $w(e)$. Since $w(e)$ are independent among all edges $e \in E^0$, the objective is minimized only when each term of it is minimized. In other words, we can minimize each term separately. Define

$$H_e\big(x(e)\big) \triangleq \min_{w} \Big(F_e\big(w(e)\big) - x(e)w(e)\Big), \quad \forall e \in E^0$$

Then it becomes $L(x) = \sum_{e\in E^0} H_e\big(x(e)\big)$. The *Lagrangian multiplier problem* of Problem 8.2.1 can be formulated as follows.

$$L(x^*) = \max_{x} L(x) = \max_{x} \sum_{e \in E^0} H_e\big(x(e)\big)$$
$$\text{subject to} \quad \sum_{(i,v)\in E^0} x(i,v) = \sum_{(v,j)\in E^0} x(v,j), \quad \forall v \in V^0$$

This is a primal network flow problem. The constraint is also known as *flow conservation*, i.e., incoming flow equals outgoing flow. x is called a flow on G^0 if it satisfies flow conservation. For example, $x(e) = 0$ for all $e \in E^0$ is a flow. $H_e\big(x(e)\big)$ is the gain function on $e \in E^0$ with respect to the flow $x(e)$. The Lagrangian multiplier problem asks for an optimal flow such that the total gain over G^0 is maximized. The following theorem [**3**] establishes a connection between Problem 8.2.1 and its corresponding Lagrangian multiplier problem.

Theorem 8.4.1. *Let x^* be a solution to the Lagrangian multiplier problem of Problem 8.2.1. Then $L(x^*)$ equals the optimal objective value of Problem 8.2.1.*

8.5. Convex cost-scaling approach

In the following, we will characterize function H_e and elaborate the algorithm developed in [**2**] for computing x^*. In Section 8.5.3 we describe how to use x^* to construct a solution to Problem 8.2.1.

8.5.1. Function $H_e\big(x(e)\big)$

Consider the case when $x(e) > M$ for some $e \in E^0$, where M is the large number we introduced in Section 8.4.1 to eliminate the bounds on $b(e)$. By the definition of F_e, we have

$$\lim_{w(e)\to\infty} \Big(F_e\big(w(e)\big) - x(e)w(e)\Big) = \lim_{w(e)\to\infty} \big(Mw(e) - x(e)w(e)\big) = -\infty$$

Thus $H_e\big(x(e)\big) = -\infty$ when $x(e) > M$. Recall that the Lagrangian multiplier problem asks for an x^* that maximizes $\sum_{e\in E^0} H_e\big(x(e)\big)$. It implies that we can assume $x^*(e) \le M$. Similarly, it can be shown that $H_e\big(x(e)\big) = -\infty$ when $x(e) < -M$, which implies that $x^*(e) \ge -M$. Therefore we consider $-M \le x(e) \le M$, $\forall e \in E^0$, in the Lagrangian multiplier problem.

Let $\perp(e) = -T$ and $\top(e) = -d(e)$ for $e \in E$, and $\perp(e) = 0$ and $\top(e) = T$ for $e \in E^0 - E$. For each $e \in E_0$, we define

$$f_e(\theta) \stackrel{\triangle}{=} F_e(\theta+1) - F_e(\theta), \quad \perp(e) \le \theta \le \top(e) - 1 \tag{8.1}$$

Then $H_e\big(x(e)\big)$ for $-M \le x(e) \le M$ can be analytically expressed by a set of linear segments specified in the next lemma [**2**].

Lemma 8.5.1. *$H_e\big(x(e)\big)$ for $-M \leq x(e) \leq M$ is a piecewise linear concave function of $x(e)$ as follows, where $\perp(e) < i < \top(e)$,*

$$H_e\big(x(e)\big) = \begin{cases} F_e\big(\perp(e)\big) - \perp(e)x(e), & \text{if } -M \leq x(e) \leq f_e\big(\perp(e)\big) \\ \qquad\qquad \cdots\cdots & \\ F_e(i) - ix(e), & \text{if } f_e(i-1) \leq x(e) \leq f_e(i) \\ \qquad\qquad \cdots\cdots & \\ F_e\big(\top(e)\big) - \top(e)x(e), & \text{if } f_e\big(\top(e)-1\big) \leq x(e) \leq M \end{cases}$$

Figure 8.3 gives an illustration of $H_e\big(x(e)\big)$ for $F_e\big(w(e)\big) = -1/w(e)$, $\perp(e) = -5$ and $\top(e) = -1$.

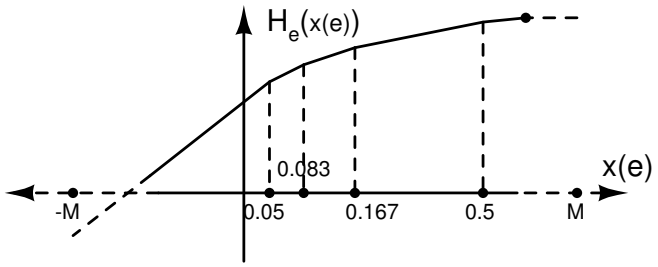

Figure 8.3. Illustration of $H_e\big(x(e)\big)$.

8.5.2. Algorithm

Define cost function $C_e\big(x(e)\big) = -H_e\big(x(e)\big)$. The Lagrangian multiplier problem can be alternatively restated as

$$\begin{aligned} \text{Minimize} \quad & \sum_{e \in E^0} C_e\big(x(e)\big) \\ \text{subject to} \quad & \sum_{(i,v) \in E^0} x(i,v) = \sum_{(v,j) \in E^0} x(v,j), \quad \forall v \in V^0 \\ & -M \leq x(e) \leq M, \quad \forall e \in E^0 \end{aligned}$$

Notice that if C_e is linear, i.e., $C_e\big(x(e)\big) = \rho x(e)$ where ρ is a constant, then the above problem is a minimum cost flow problem, which can be solved in strongly polynomial time [**3**]. For general convex C_e, since it is piecewise linear, a straightforward approach is to represent C_e as a set of segments, each of which is linear within a small range, and solve it using any minimum cost flow algorithm. However, since the number of segments for each e is $\top(e) - \bot(e) + 1$, minimum cost flow algorithms would not in general run in strongly polynomial time. Intuitively, if the minimum cost flow algorithm can be modified to be aware of the change of cost when the flow is changed, then we can apply it directly to C_e, which may help to solve the problem more efficiently.

In [**2**], a strongly polynomial time algorithm was developed for general convex functions based on cost-scaling algorithm [**39**].

The cost-scaling algorithm defines *pseudoflow* x such that x satisfies $-M \leq x(e) \leq M$ for all $e \in E$ but may violate the flow conservation. For any pseudoflow x, the *imbalance* of vertex v is defined as $o(v) = \sum_{(i,v) \in E^0} x(i,v) - \sum_{(v,j) \in E^0} x(v,j)$. If $o(v) > 0$, v is called an *excess* vertex. The algorithm proceeds by constructing and manipulating the *residual*

graph $\mathcal{G}(x)$ defined as follows with respect to a pseudoflow x. For each $(u, v) \in E^0$, $\mathcal{G}(x)$ may contain two edges: *forward* edge (u, v) and *backward* edge (v, u). Forward edge (u, v) has a cost $c(u, v)$ equal to the right slope of $C_{(u,v)}\big(x(u, v)\big)$ at $x(u, v)$, and a capacity of $cap(u, v) = M - x(u, v)$. Backward edge (v, u) has a cost $c(v, u)$ equal to the negative of the left slope of $C_{(u,v)}\big(x(u, v)\big)$ at $x(u, v)$, and a capacity of $cap(v, u) = x(u, v) - (-M)$. Note that if $C_{(u,v)}\big(x(u, v)\big) = \rho x(u, v)$, then $c(u, v) = -c(v, u) = \rho$. The residual graph consists only of edges with positive capacities. The algorithm also maintains a value $\pi(v)$ for each $v \in V^0$, which is refereed to as the *vertex potential*.

For a given residual graph $\mathcal{G}(x)$ and a set of vertex potentials, it defines the *reduced cost* of edge (u, v) as $c^\pi(u, v) = c(u, v) + \pi(v) - \pi(u)$. For a given value of ϵ, an edge (u, v) is called *admissible* if $-\epsilon \leq c^\pi(u, v) < 0$. A flow or pseudoflow x is called ϵ-*optimal* for some $\epsilon \geq 0$ if for some vertex potential π, the following ϵ-*optimality condition* is satisfied: $c^\pi(u, v) \geq -\epsilon$ for all $(u, v) \in \mathcal{G}(x)$. The cost-scaling algorithm treats ϵ as a parameter and iteratively obtains ϵ-*optimal* flows for successively smaller values of ϵ. Initially, ϵ is set to be the maximum edge cost (which is T in our problem), thus any flow is ϵ-*optimal*. The algorithm then performs cost-scaling phases by repeatedly applying an *improve-approximation* procedure that transforms an ϵ-*optimal* flow into an $\epsilon/2$-*optimal* flow. When $\epsilon < 1/|V^0|$, the algorithm terminates with an optimal flow x^*. The pseudocode of the algorithm is presented in Figure 8.4.

The basic operation in the *improve-approximation* procedure is to select an excess vertex u with $o(u) > 0$. When u has no admissible edges, its potential is increased by $\epsilon/2$; otherwise, it performs *pushes* on admissible edges emanating from it. If $C_e\big(x(e)\big)$ is linear, the amount of flow pushed on an admissible edge (u, v) will be the minimum of the excess and the edge capacity, i.e., $\min\big(o(u), cap(u, v)\big)$. However, when $C_e\big(x(e)\big)$ is piecewise linear, the cost

```
Algorithm convex cost-scaling
Input: A circuit G = (V^0, E^0) and C_e, ∀e ∈ E^0
Output: An optimal flow x*.

   π ← 0 and ε ← max_{e∈E^0} ⊤(e);
   Let x* be any flow;
   While (ε ≥ 1/|V^0|) do
     improve-approximation(ε, x*, π);
     ε ← ε/2;
   Return x*;

Procedure improve-approximation(ε, x, π)
   For each admissible edge (u, v) ∈ E^0 do
     Send q(u, v) flow on (u, v);
   Compute flow imbalance on each vertex;
   While there is an excess vertex u do
     If there is an admissible edge (u, v) then
       push min(o(u), q(u, v)) flow on (u, v);
     Else
       π(u) ← π(u) + ε/2;
```

Figure 8.4. Pseudocode of convex cost-scaling algorithm.

(slope) will change with the change of the flow. For this case, [**2**] showed that the amount of flow should be $\min\big(o(u), q(u,v)\big)$, where $q(u,v)$ is defined as follows. If (u,v) is a forward edge, then

$$q(u,v) = \begin{cases} M - x(u,v), & \text{if } \pi(u) - \pi(v) \geq \top(u,v) \\ f_{(u,v)}(\lfloor \pi(u) - \pi(v) \rfloor) - x(u,v), & \text{otherwise} \end{cases}$$

where $f_{(u,v)}$ is defined in (8.1) in Section 8.5.1. If (u,v) is a backward edge, then

$$q(u,v) = \begin{cases} M + x(v,u), & \text{if } \pi(v) - \pi(u) \leq \bot(v,u) \\ x(v,u) - f_{(v,u)}(\lfloor \pi(v) - \pi(u) \rfloor), & \text{otherwise} \end{cases}$$

Intuitively, we push the flow along the current linear segment of the piecewise linear function until we hit a joint point where the cost is about to change. The change of flow is followed by an update in edge cost (slope) based on Lemma 8.5.1. The procedure terminates when there is no more excess vertex. Note that the replacement of $cap(u, v)$ by $q(u, v)$ and the following cost update is the only difference between a convex cost-scaling algorithm and a linear cost-scaling algorithm.

The correctness and complexity of the algorithm is given in the next theorem [**2**].

Theorem 8.5.1. *The convex cost-scaling algorithm in Figure 8.4 solves the Lagrangian multiplier problem of Problem 8.2.1 in* $O(|V||E| \log(|V|^2/|E|) \log(|V|T))$ *time, where the complexity of the improve-approximation procedure is bounded by* $O(|V||E| \log(|V|^2/|E|))$ *and* $O(\log(|V|T))$ *bounds the number of iterations.*

The practical efficiency of the algorithm is confirmed by our experimental results in Section 8.6.

8.5.3. Solution transformation

The convex cost-scaling algorithm upon termination gives an optimal flow x^* and the corresponding potentials π. Both x^* and π may not be integer. However, since each cost function is piecewise linear, it follows that there always exists an integer optimal potential π^*. To determine it, we construct the residual graph $\mathcal{G}(x^*)$ with respect to x^* and solve a shortest path problem to determine the shortest path distance $d(v)$ from vertex 0 to every other vertex $v \in V$ in terms of costs. Since all edge costs in $\mathcal{G}(x^*)$ are integer, d is also

integer. Then $\pi^*(v) = -d(v)$, $\forall v \in V$, gives an integer potential for the Lagrangian multiplier problem. The solution (t^*, w^*, b^*) to Problem 8.2.1 is obtained by assigning $t^* = \pi^*$, $w^*(u,v) = t^*(u) - t^*(v)$, and $b^*(u,v) = -d(u,v) - w^*(u,v)$, $\forall (u,v) \in E$.

8.6. Experimental results

We used the linear cost-scaling algorithm by Goldberg in [**38**] and adapted it to convex cost case. To compare with the work in [**94**], we set δ_e and λ_e as in Section 8.1 so that we focus on the same problem as in [**94**], i.e., finding a maximum budget delay relaxation with minimal number of critical edges. We also used the same test files, which were extracted from MediaBench [**52**]. In addition, since the test files are relatively small, we included ISCAS-85 benchmark suite, where each gate was treated as a resource. Without loss of generality, the original clock latency was set to 1 for each resource and 0 for each interconnect. The latency constraint T was set to the maximum number of resources in a PI-to-PO data flow path. All tests were conducted in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. The results are reported in Table 8.1, where column "time(s)" lists the running time for each test file in seconds.

It can be seen that the algorithm is very efficient. All MediaBench test can be finished in 0.02 second (the running time of the heuristic algorithm in [**94**] is not given). The running times appear to scale well with problem size in ISCAS-85.

8.7. Conclusion

A general problem formulation is proposed for design closure driven delay relaxation. We show that it can be transformed to a convex cost integer dual network flow problem. It can also be viewed as a convex retiming problem. When cost functions are linear, it is

Table 8.1. Experimental Results

name	$\|V\|$	$\|E\|$	T	`time(s)`
inv1	400	397	9	0.01
inv2	430	426	16	0.01
inv3	702	706	15	0.02
jpg1	234	244	16	0.01
jpg2	530	527	13	0.01
mat1	218	221	10	0.00
mat2	192	200	14	0.00
rot1	160	153	6	0.00
c1355	1092	1402	23	0.06
c1908	1760	2300	39	0.12
c2670	2386	3043	31	0.19
c3540	3338	4302	46	0.53
c5315	4614	6185	48	0.91
c6288	4832	6704	123	1.57
c7552	7024	9348	42	1.57

reduced to minimum area retiming. The proposed formulation is amicable to incorporating the trade-off between resource budgeting and interconnect budgeting.

Using Lagrangian relaxation technique, the dual network flow problem is further transformed to a primal network flow problem, which can be solved in strongly polynomial time by the convex cost-scaling algorithm in [**2**].

CHAPTER 9

Trade-off between Latch and Flop for Min-Period Sequential Circuit Designs with Crosstalk

In a sequential circuit, clock signals are usually applied at memory elements to lock correct state values and to filter out unintended transitions. Generally speaking, memory elements can be categorized into two groups according to how they respond to clock signals: edge-triggered *flops* store the data when the clock switches; level-sensitive *latches* let the output have the input value during its active duration.

Latches dominate high-performance designs as they have smaller delays and occupy less area than flops. More importantly, since signals can pass through a latch anytime during its active duration, time borrowing between two consecutive latches is possible. As a result, circuits designed with latches can operate at higher frequencies than their edge-triggered counterparts [**33**].

For correct operation the data is required to be stable before the latching edge by an amount of time called the *setup time*—this is known as the *setup condition*. The data has to remain stable after the latching edge for an amount of time called the *hold time*—this is known as the *hold condition*. A *clocking violation* refers to a violation of the setup or the hold condition.

Clock schedule verification (also known as timing verification) for edge-triggered circuits is straightforward. Since signals cannot pass transparently through flops, the output times of the flops are used as the input times of the combinational components, and the setup and

hold conditions of the flops are checked against the combinational outputs. However, clock schedule verification involving latches is much harder. Since signals can pass through latches transparently during their active durations, the setup and hold conditions on paths between any two latches need to be considered. These conditions become even more complex in the presence of a multi-phase clock schedule.

On the other hand, with increasing clock frequencies and shrinking process geometries in deep sub-micron technology, both capacitive and inductive coupling (also known as crosstalk) become big concerns in designs. For present day processes, the coupling capacitances can be as high as the sum of the area capacitances and the fringing capacitances. Trends indicate that the role of crosstalk will be even more dominant in the future as feature sizes shrink [**85**]. Besides introducing noises on quiet wires, crosstalk may greatly change the wire delays, hence affect the timing analysis. If an aggressor and a victim switch simultaneously in the same direction, the victim will speed up, called *assistive coupling*. Likewise, if an aggressor and a victim switch simultaneously in opposite directions, the victim will slow down, called *opposing coupling*. Assuming that coupling capacitances and inductances dominate all other capacitances and inductances of wires, failure to take crosstalk effects into timing analysis may produce wrong results. Our work in this chapter equally applies to crosstalk induced by capacitive and inductive couplings. However, to simplify the presentation, we only talk about capacitive couplings in the rest of the chapter.

Since latches allow signals to pass transparently, crosstalk effects on switching windows may accumulate on a long path and cause a clocking violation in the end. These phenomena will be explained in more details in Section 9.1. It is interesting to observe that the switching window at the output of a flop is just a single point, or a switching window with width zero (for the ease of the presentation, we assume no clock skew uncertainty). A natural thought

is to select a set of latches such that when they are replaced with flops, the crosstalk effects contributed by their original output switching windows are greatly reduced and the previous clocking violations are removed.

Our contribution in this chapter is twofold. Firstly, we propose a circular time representation for coupling detection. We show that detecting coupling under the circular time representation is easier than state-of-the-art approaches used in [**44, 107**]. Furthermore, since complicated phase translations are avoided, clock schedule verification under the circular time representation is more efficient. Secondly, we formulate the trade-off between a latch and a flop in sequential circuits with crosstalk as a problem of seeking a configuration of mixed latches and flops to minimize the clock period. We present an effective and efficient heuristic algorithm to solve this problem. Experiments showed promising results.

The rest of this chapter is organized as follows. Section 9.1 presents the motivation and problem formulation. Section 9.2 describes the models we will use in this chapter. Section 9.3 recaps the previous work. The circular time representation is proposed in Section 9.4. Our algorithm is presented in Section 9.5, followed by experimental results in Section 9.6. Conclusions are given in Section 9.7.

9.1. Motivation and problem formulation

We now use an example in Figure 9.1(a) to show that the amount of timing uncertainty due to crosstalk accumulated through latches could be larger than the benefit gained by time borrowing. In this example, we have six latches driven by a clock ϕ with period 2 shown in Figure 9.1(b), which has a duty cycle ratio of 0.5, i.e., the active duration is half the period. The ellipses between the latches represent the combinational blocks with their minimum and maximum delays. Opposing coupling increases the delay by 0.5 while assistive

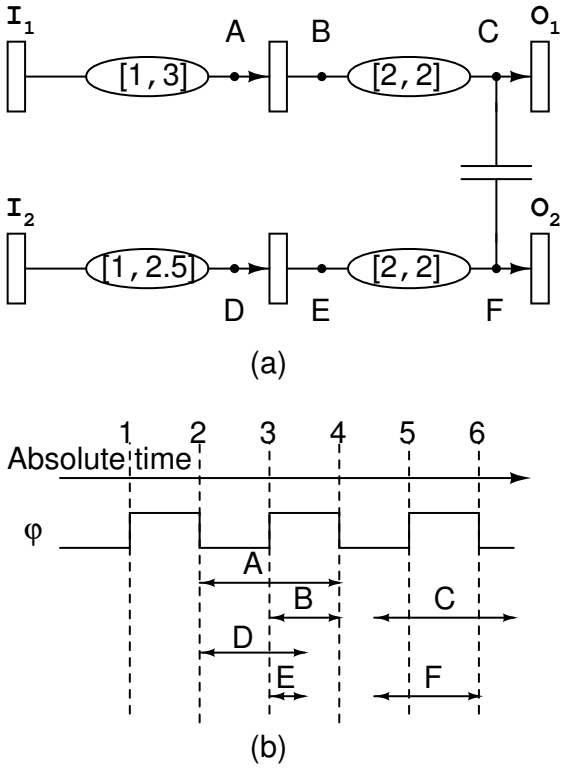

Figure 9.1. Crosstalk effects on system performance.

coupling decreases the delay by 0.5. Suppose that the setup and hold times are all zero and the inputs are available at I_1 and I_2 at time 1. Then the switching windows of A and D are $[2, 4]$ and $[2, 3.5]$ respectively, illustrated in Figure 9.1(b). According to the transparent nature of latches, the switching windows of B and E are $[3, 4]$ and $[3, 3.5]$ respectively. If the coupling capacitor between C and F does not exist, then the switching windows of C and F will be $[5, 6]$ and $[5, 5.5]$ respectively. Since no clocking violation occurs, 2 is a feasible clock period. However, due to the presence of the coupling capacitor and the fact that the switching windows of B and E overlap, opposing and assistive couplings are assumed in the blocks they feed. As a result, the switching windows of C and F become $[4.5, 6.5]$ and $[4.5, 6]$

respectively. We detect a setup violation at latch O_1. Therefore, this circuit has to work at a period no smaller than 2.2 (assuming duty cycle scales proportionally with period).

However, if we replace the latch between A and B by a flop that operates at the falling edge of ϕ, then the switching window of B is shrunken to $[4, 4]$. Since the new window does not overlap with that of E, coupling is not triggered. Therefore, the switching windows of C and F become $[6, 6]$ and $[5, 5.5]$ respectively. The previous setup violation at O_1 is now removed. The new circuit can still work under period 2.

This example shows that the trade-off between a latch and a flop can be leveraged to improve sequential circuit designs with crosstalk. We formulate it as a problem of finding a configuration of mixed latches and flops to minimize the clock period.

Problem 9.1.1. *(Optimal Latch-Flop Configuration)*
Given a sequential circuit and its clock schedule, find a configuration of mixed latches and flops for the memory elements such that the circuit satisfies both setup and hold conditions with crosstalk under the minimum clock period.

9.2. Models and notations

9.2.1. Circuit model

A directed graph $G = (V, E)$ is used to represent a sequential circuit, where $V = V_G \cup V_L$ is the union of two sets of vertices: the gates V_G and the memory elements V_L, and $E = E_I \cup E_C$ is the union of two sets of edges: the interconnects E_I and the coupling capacitances E_C. Interconnect delays are available since we are considering circuits after placement and routing. We assume that all primary inputs and primary outputs are latched.

We designate the latest (earliest) arrival time at a vertex v as A_v (a_v). The latest (earliest) departure time from a vertex is denoted by D_v (d_v). $[a_v, A_v]$ and $[d_v, D_v]$ are called the input and output switching windows of v respectively.

If v is a memory element, $p(v)$ denotes its given phase. Depending on the particular configuration, v could be a latch with phase $p(v)$, or a flop operating at the rising or falling edge of $p(v)$. For simplicity, we only talk about latches and the flops that operate at the falling edges. In the remainder of this chapter, we simply use flops to refer to those that operate at the falling edges.

9.2.2. Delay model

We choose to use the dynamically bounded delay model [**41**] to capture the delay variations due to crosstalk. More specifically, each vertex v has a combinational delay range $[\delta_v, \Delta_v]$. For each coupling capacitance between v and an aggressor u, if the switching window at v's input overlaps with that at u's input within a specified amount of time $\tau_{u,v}$, then opposing and assistive couplings are assumed at u and v. As a result, δ_v is decreased by $\delta_{u,v}$ and Δ_v is increased by $\Delta_{u,v}$. In this model, multiple aggressors are treated independently, i.e., their effects on victim are additive.

Although more accurate models are possible, such as [**102, 54, 5**], where superposition is used to calculate the total crosstalk effects without using coupling factors, the chosen model is a generalization of discrete coupling models, such as ones that assume a 0 X, 1 X, or 2 X effective coupling capacitance, e.g., [**83**].

9.2.3. Clock model

A *clock schedule* for a circuit is a set of periodic signals $\phi_1, ..., \phi_n$ with width w_i of the phase ϕ_i and a common period T. A three-phase clock schedule is shown in Figure 9.2. Selecting a period of length T as the *global time reference*, we can order the phases ϕ_i by its starting times and ending times (s_i, e_i) with respect to the reference. Note that it is possible to have $s_i > e_i$ based on the selection of the reference. We generally order the phases such that $e_i < e_j$ if $i < j$, and choose e_n as the global time reference. In particular, if $e_i = \frac{i}{n}T$, the clock schedule is *symmetric* [**45**]. The setup and hold times are denoted as X_i and H_i, respectively. We assume that both s_i and e_i scale proportionally with T, but X_i and H_i remain the same. A clock schedule is *valid* if and only if the circuit has no clocking violation under it. The common period of a valid clock schedule is called a *feasible* period. For simplicity, we assume zero clock skews, i.e., clock signals arrive at all the memory elements at the same time.

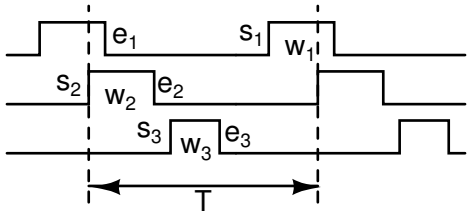

Figure 9.2. Three-phase clock schedule with period T.

9.3. Previous work

Several techniques have been proposed to evaluate crosstalk effects on combinational delays. Some are based on iterative techniques [**5, 83**]; some are based on the propagation of events [**13**]; others are based on more complex mathematical formulations [**41**]. Consideration of the functional correlation of the victim and the aggressors allows further accuracy in

analysis [**4, 12, 105**]. The worst case victim delay can be obtained by driver modeling using reduced order modeling and worst case alignment of the aggressors relative to the victim [**26, 40, 96**]. What these techniques have in common is that crosstalk effects are expressed by expanding the switching window at the victim to accommodate more possible switchings.

The problem of clock schedule verification without considering crosstalk has been elegantly solved by Szymanski and Shenoy [**100, 91**]. Without crosstalk, we only need to consider switching windows at memory elements. The input switching window of a memory element i can be computed by propagating the output switching windows of i's fanins through shortest and longest path delays to i. Timing conditions for latches are formulated as [**91**]

$$A_i = \max_{j \to i}(D_j + C_{j,i} - E_{p(j)p(i)}) \tag{9.1}$$

$$a_i = \min_{j \to i}(d_j + c_{j,i} - E_{p(j)p(i)}) \tag{9.2}$$

$$D_i = \max(A_i, T - w_{p(i)}) \tag{9.3}$$

$$d_i = s_{p(i)} = T - w_{p(i)} \tag{9.4}$$

$$A_i \leq T - X_i \tag{9.5}$$

$$a_i \geq H_i \tag{9.6}$$

where $C_{j,i}$ and $c_{j,i}$ represent the maximum and minimum combinational delays from latch j to latch i respectively, and $E_{p(j)p(i)}$ is a phase shift operator defined as [**82**]

$$E_{p(j)p(i)} = \begin{cases} e_{p(i)} - e_{p(j)}, & \text{if } p(j) < p(i) \\ T + e_{p(i)} - e_{p(j)}, & \text{otherwise} \end{cases}$$

Note that used here are local times referring to local time zones that end with the phase falling edges. Timing conditions involving flops are the same except for (9.3)-(9.4), where the output window of a flop is a single point $[T, T]$. We say that j is the *latest critical predecessor* of i if $A_i = D_j + C_{j,i} - E_{p(j)p(i)}$. If $a_i = d_j + c_{j,i} - E_{p(j)p(i)}$, then j is called the *earliest critical predecessor* of i.

We choose to characterize d_i by (9.4) instead of the more aggressive formulation with $d_i = \max(a_i, T - w_{p(i)})$ in [**82**]. This is because [**99**] showed that there are common situations, such as a latch driven by a qualified clock signal, in which the aggressive formulation is incorrect, and a similar problem arises in circuits which permit the clock to be stopped between adjacent latches to save power.

In the presence of crosstalk, however, switching windows at both memory elements and gates need to be considered because crosstalk may change the shortest and longest path delays between them.

Hassoun *et al.* [**44**] proposed to assign each gate v a phase, also denoted as $p(v)$, which can be derived by analyzing the phases of the memory elements in the combinational fanin and fanout of the gate. For example, $p(v)$ could be the phase of the clock driving the nearest memory element in v's combinational fanin. Based on the phases at gates, the switching windows at gates can be computed in a similar fashion as those at memory elements, expressed

as

$$a_v = \begin{cases} \min_{(u,v)\in E_I}(d_u - E_{p(u)p(v)}), & \text{if } p(u) \neq p(v) \\ \min_{(u,v)\in E_I} d_u, & \text{if } p(u) = p(v) \end{cases} \tag{9.7}$$

$$A_v = \begin{cases} \max_{(u,v)\in E_I}(D_u - E_{p(u)p(v)}), & \text{if } p(u) \neq p(v) \\ \max_{(u,v)\in E_I} D_u, & \text{if } p(u) = p(v) \end{cases} \tag{9.8}$$

$$d_v = a_v + \delta_v \tag{9.9}$$

$$D_v = A_v + \Delta_v \tag{9.10}$$

Their approach of verifying clock schedules with crosstalk works as follows. At the beginning, no crosstalk is assumed to take effect and the Szymanski and Shenoy algorithm is used to find a solution. Then the solution is used to compute the switching windows at gates by (9.7)-(9.10). The results are used to check for switching window overlaps and to modify the delays based on crosstalk effects. These three processes are repeated until there is no change on delays. Zhou [**107**] proposed an improved algorithm that essentially combined the three processes together and computed the accumulated switching windows during the iterations.

However, detecting overlap in both Hassoun *et al.*'s and Zhou's approaches was not easy. In [**44**], Hassoun *et al.* showed that when two vertices couple, the victim's input window can overlap with either one, two, or three of the three possible switching windows at the aggressor's input: the previous, the current, and the following windows. The input window of the aggressor must be translated to the victim's local time zone to perform a meaningful comparison. They showed that the previously defined phase shift operator $E_{p(j)p(i)}$ cannot be used for such a translation since it may lead to coupling detection misses. Consider the

example in Figure 9.3 (taken from [**44**]), where $T = 10$, $e_{p(u)} - e_{p(v)} = 1$, $E_{p(u)p(v)} = 9$, $\tau_{u,v} = 1.01$ and 50% duty cycle. If $E_{p(u)p(v)}$ is used to translate the aggressor's ranges: the previous $[5 - T, 7 - T]$, the current $[5, 7]$ and the following $[5 + T, 7 + T]$, then the ranges, respectively, become $[-14, -12]$, $[-4, -2]$, and $[6, 8]$. If the victim occurrence is $[13, 15]$, then comparing the translated aggressor ranges against the victim's will not indicate a coupling problem. However, coupling should have been detected because the fourth occurrence $[16, 18]$ is within $\tau_{u,v}$ of the victim occurrence.

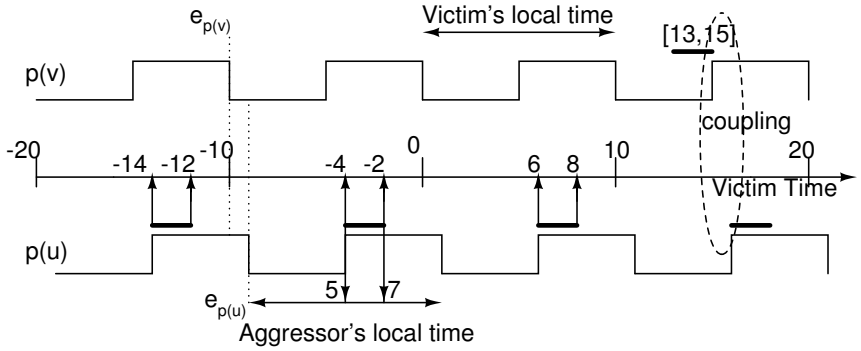

Figure 9.3. An example where coupling is missed by $E_{p(u)p(v)}$.

As a remedy, they proposed a new phase shift operator $E'_{p(j)p(i)}$, defined as

$$E'_{p(j)p(i)} = \begin{cases} e_{p(i)} - e_{p(j)} + T, & \text{if } e_{p(i)} - e_{p(j)} < -\frac{T}{2} \\ e_{p(i)} - e_{p(j)}, & \text{if } -\frac{T}{2} \leq e_{p(i)} - e_{p(j)} \leq +\frac{T}{2} \\ e_{p(i)} - e_{p(j)} - T, & \text{if } e_{p(i)} - e_{p(j)} > +\frac{T}{2} \end{cases}$$

Intuitively, if the difference between $e_{p(i)}$ and $e_{p(j)}$ is small, then the amount of translation should be equal to the difference. A simple example is $e_{p(i)} = e_{p(j)}$, where the aggressor and the victim have the same phase and aligned time zones. Thus, no phase translation is needed, instead of having $E_{p(j)p(i)} = T$ by the previous phase shift operator. T will be involved in phase translation only when $e_{p(i)}$ and $e_{p(j)}$ are far away from each other.

Consider the example in Figure 9.3 again with the new phase shift operator. In this case, $E'_{p(u)p(v)} = -1$. Subtracting this phase shift operator, the three aggressor ranges become $[-4, -2]$, $[6, 8]$, and $[16, 18]$ respectively. The previously missed coupling is now detected.

9.4. Circular time representation for coupling detection

Although coupling can be detected with $E'_{p(j)p(i)}$, the process of detection is complicated. Occurrences on three time zones need to be translated and checked. An important observation here is that if we translate the input windows of both the aggressor and the victim to the global time reference and take *modulo* T on them, coupling can be easily detected. For the example in Figure 9.3, assuming that the global time reference coincides with v's local time zone, we have $[13, 15] \mod T = [3, 5]$ for v, and $[-4, -2] \mod T = [6, 8]$ for u. Given that $\tau_{u,v} = 1.01$, coupling is successfully detected. The following theorem establishes the correctness of this approach.

Theorem 9.4.1. *Two vertices couple if and only if their switching windows overlap within a specified amount of time after being translated to the global time reference and taken modulo T.*

Proof. ($\rightarrow$): If two vertices u and v couple, then, by the definition of coupling (assistive or opposing), it is possible that they switch simultaneously, i.e., their switching windows overlap within a specified amount of time. The fact of overlap will be preserved after being translated to the global time reference. In addition, since all the clock phases share a common period T, all switching windows are periodic with T, so are the overlaps among them. Therefore, the overlap between u and v will remain after being taken modulo T.

($\leftarrow$): On the other hand, since a clock schedule is periodic with a common period T, the fact that two switching windows overlap after being taken modulo T implies a coupling between them, which is independent on phase translations. $\square$

In addition, if switching windows at all the vertices are always computed with respect to the global time reference, phase translations are not necessary at all.

A formal treatment of taking modulo T on switching windows can be illustrated in a circular time representation. Figure 9.4(a) shows the circular time representation of the three-phase clock schedule in Figure 9.2. The whole circle represents the global time reference we selected. The time reference at the top point is 0. It increases clockwise until it goes back to the top, where reference T coincides with 0. Each phase i is then represented as an arc with endpoints s_i and e_i. Under this representation, each switching window becomes an arc. When propagating through a gate, an arc will be shifted clockwise and expanded, as shown in Figure 9.4(b). The modulo T operation can be treated as a transformation from the original axis representation to the circular time representation to ensure that the values of the endpoints are within $[0, T)$, whenever an arc is shifted or expanded beyond the top point.

We must note that it is possible to have $A_v < a_v$ under the circular time representation. Therefore, $[a_v, A_v]$ is no longer interpreted as an interval in the original real axis, but a *record* that indicates the starting and ending points of an arc. In the remainder of this chapter, we will use "window" and "arc" interchangeably.

We then propose clock schedule verification with crosstalk under the circular time representation. It has two phases. In the first phase, we check if the clock schedule is valid (having no clocking violation) without crosstalk using the Szymanski and Shenoy algorithm. Given

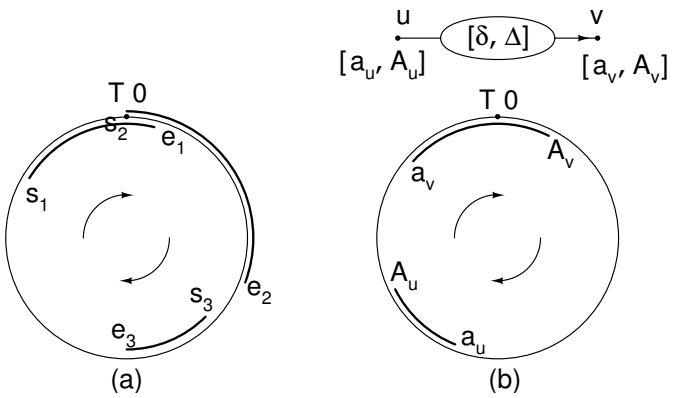

Figure 9.4. (a) The circular time representation of the clock schedule in Figure 9.2. (b) Switching window propagation through a combinational vertex under the circular time representation.

that we use (9.4) to characterize d_i instead of the more aggressive one in [**82**], it is guaranteed by [**108**] that the switching window without crosstalk is a subset of the switching window with crosstalk at each vertex. Therefore, a clock schedule is ensured to be invalid if it has clocking violations at this phase. If there is no clocking violation, (9.7)-(9.10) are carried out to obtain the switching windows at gates. We then translate the switching windows at memory elements and gates to the global time reference, and take modulo T on them, i.e., represent them as arcs.

In the second phase, the occurrences of overlaps are used to update delays and switching arcs to take crosstalk effects into consideration. Whenever a switching arc is changed, the amount of change is propagated to its fanouts and coupled wires. Propagation through a combinational vertex v can be expressed as

$$[d_v, D_v] = [(a_v + \delta_v) \text{mod } T,\ (A_v + \Delta_v) \text{mod } T]$$

Propagation through a latch i is formulated as

$$\begin{aligned} d_i &= s_{p(i)} \\ D_i &= \begin{cases} A_i & \text{if } (A_i - a_i)(A_i - s_{p(i)})(s_{p(i)} - a_i) > 0 \\ s_{p(i)} & \text{otherwise} \end{cases} \end{aligned}$$

Propagation through a flop is formulated as

$$d_i = D_i = e_{p(i)}$$

The change on one arc may trigger or cancel other succeeding couplings and result in more changes. The process of update is carried out over the whole circuit until convergence or a clocking violation is found.

For all $i \in V_L$, let

$$[R_i^X, R_i^H] \stackrel{\triangle}{=} [(e_{p(i)} - X_{p(i)}) \text{mod } T,\ (e_{p(i)} + H_{p(i)}) \text{mod } T]$$

as shown in Figure 9.5.

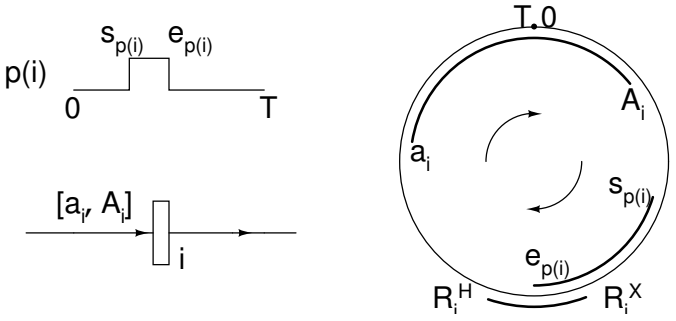

Figure 9.5. Illustration of R_i^X and R_i^H for memory element i.

The following theorem states the necessary and sufficient conditions for a valid clock schedule with crosstalk.

Theorem 9.4.2. *Given that a clock schedule is valid without crosstalk, it is valid with crosstalk if and only if the input switching arc* $[a_i, A_i]$ *of each memory element* $i \in V_L$ *does not intersect with* $[R_i^X, R_i^H]$ *anytime before convergence.*

Proof. Let $[\bar{a}_i, \bar{A}_i]$ denote the input switching arc at memory element i without crosstalk, $\forall i \in V_L$. Since the circuit is free of clocking violations under the clock schedule without crosstalk, $\bar{A}_i$ and $\bar{a}_i$ satisfy the setup and hold conditions (9.5)-(9.6), which implies that $[\bar{a}_i, \bar{A}_i]$ does not intersect with $[R_i^X, R_i^H]$.

Considering crosstalk, we have $[\bar{a}_i, \bar{A}_i] \subseteq [a_i, A_i]$ by [**108**]. If $[a_i, A_i]$ does not intersect with $[R_i^X, R_i^H]$ before convergence, $\forall i \in V_L$, then (9.5)-(9.6) are still kept with crosstalk. The clock schedule is still valid. On the other hand, if an intersection occurs at i before convergence, we must have either R_i^X or R_i^H (or both) on the arc $[a_i, A_i]$. The former stands for a hold violation while the latter is a setup violation. The clock schedule is then invalid with crosstalk. □

Based on this, a setup violation is caused when A_i is shifted clockwise to make the following inequality satisfied.

$$(A_i - a_i)(A_i - R_i^X)(R_i^X - a_i) \;>\; 0 \tag{9.11}$$

A hold violation is caused when a_i is shifted counterclockwise to make the following inequality satisfied.

$$(A_i - a_i)(A_i - R_i^H)(R_i^H - a_i) \;>\; 0 \tag{9.12}$$

The next theorem states that the convergence or a clocking violation will be found in polynomial time.

Theorem 9.4.3. *The complexity of verifying clock schedule with crosstalk under the circular time representation is* $O(|V_L|^3 + |E_C||E_I||V_L|)$.

Proof. Given a clock period, computing switching windows at memory elements by the Szymanski and Shenoy algorithm takes $O(|V_L|^3)$ time. If there is no clocking violation at this phase, i.e., (9.5)-(9.6) are satisfied, we then proceed to compute switching windows at gates. To do this, we will treat the outputs of memory elements as primary inputs and the inputs of memory elements as primary outputs, and compute the switching windows at gates in their topological order, which takes $O(|E_I|)$ time. Translating all the switching windows to the global time reference and representing them under the circular time representation take $O(|V|)$ time.

To analyze the complexity of the second phase, we separate it into passes. At the beginning of each pass, we detect the occurrences of overlaps and update delays, which takes $O(|E_C|)$ time. Then during the rest of the pass, we propagate the delay updates to other vertices without detecting overlap. The propagation through gates can be conducted in their topological order. It was shown in [**100**] that the propagation takes $O(|E_I||V_L|)$ time before convergence or a clocking violation is found. Given that the number of effective coupling capacitances increases every pass (otherwise we stop with a converged solution), the number of passes is upper bounded by $O(|E_C|)$. Therefore, the complexity of the second phase is $O(|E_C||E_I||V_L|)$.

Summing up the complexities of the two phases, we obtain the complexity of verifying clock schedule with crosstalk under the circular time representation as $O(|V_L|^3 + |E_C||E_I||V_L|)$. □

Compared with the methods in [**44, 107**], our approach has two advantages. Firstly, no phase translation is needed in the second phase of our approach. Secondly, the overlap detection is much easier under the circular time representation. As a result, the process of verification is accelerated, which is confirmed by the experiments in Section 9.6.

9.5. Algorithm for an optimal latch-flop configuration

9.5.1. Lower and upper bounds of a feasible clock period

In [**89**], a set of constraints equivalent to (9.1)-(9.6) was proposed to characterize a feasible clock period, expressed as

$$e_{p(i)} \geq e_{p(j)} - w_{p(j)} + C^q_{j,i} - K^q_{j,i}T + X_{p(i)} \tag{9.13}$$

$$e_{p(i)} \leq e_{p(j)} - w_{p(j)} + c^q_{j,i} + (1 - K^q_{j,i})T - H_{p(i)} \tag{9.14}$$

if j is a latch, and

$$e_{p(i)} \geq e_{p(j)} + C^q_{j,i} - K^q_{j,i}T + X_{p(i)} \tag{9.15}$$

$$e_{p(i)} \leq e_{p(j)} + c^q_{j,i} + (1 - K^q_{j,i})T - H_{p(i)} \tag{9.16}$$

if j is a flop. $C^q_{j,i}$ and $c^q_{j,i}$ are the maximum and minimum combinational delays along any path q between j and another memory element i without crosstalk. Path q is combinational in (9.14) and (9.16); q could be sequential in (9.13) and (9.15), i.e., there could be memory elements on q. $K^q_{j,i}$ counts the number of occurrences that two consecutive memory elements

x and y on q have $e_{p(x)} \geq e_{p(y)}$, recursively defined as

$$K^q_{j,k} = \begin{cases} 0 & \text{if } k = j \\ K^q_{j,b(k)} & \text{if } e_{p(b(k))} < e_{p(k)} \\ K^q_{j,b(k)} + 1 & \text{if } e_{p(k)} \leq e_{p(b(k))} \end{cases} \tag{9.17}$$

where $b(k)$ represents the preceding memory element of k on q.

We use T_l to denote the minimum period that satisfies (9.13) for all q. Since e_i and w_i scale proportionally with T, we represent them as $e_i = \rho^e_i T$ and $w_i = \rho^w_i T$, where $0 \leq \rho^e_i, \rho^w_i < 1$. Then (9.13) becomes

$$(K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j)T \geq C^q_{j,i} + X_{p(i)}$$

By the definition of $K^q_{j,i}$ in (9.17), if $e_j < e_i$, then $K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j > 0$ since $K^q_{j,i} \geq 0$; if $e_j = e_i$, then $K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j \geq 1$ since $K^q_{j,i} \geq 1$; if $e_j > e_i$, then $K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j > 0$ since $K^q_{j,i} \geq 1$ and $\rho^e_i - \rho^e_j > -1$. Therefore, we have $K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j > 0$ for all q. As a result, T_l can be written as

$$T_l = \max_{q: j \rightsquigarrow i} \frac{C^q_{j,i} + X_{p(i)}}{K^q_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j} \tag{9.18}$$

The next lemma states that T_l is a lower bound of a feasible clock period.

Lemma 9.5.1. *The minimum period that can be possibly achieved with crosstalk is lower bounded by T_l.*

Proof. Let q^+ be the path such that

$$(K^{q^+}_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j)T_l = C^{q^+}_{j,i} + X_{p(i)}$$

Since the delays may be increased due to opposing couplings, the maximum delay along q^+ with crosstalk becomes $C'^{q^+}_{j,i} \geq C^{q^+}_{j,i}$. Therefore,

$$(K^{q^+}_{j,i} + \rho^e_i - \rho^e_j)T_l \leq (K^{q^+}_{j,i} + \rho^e_i - \rho^e_j + \rho^w_j)T_l \leq C'^{q^+}_{j,i} + X_{p(i)}$$

It implies that any value below T_l will cause a setup violation on q^+ with crosstalk independent of whether j and i are latches or flops. Hence, T_l is a lower bound of the solution. □

To find an upper bound T_u of a feasible clock period, we use the minimum period that is feasible under the worst case coupling scenario (assuming all the couplings take effect). Both T_l and T_u can be computed by the algorithm in [**89**] in $O(|V_L|n^2 \log(|V_L|))$ time, where n is the number of clock phases.

Although an upper and a lower bound can be obtained, we cannot use binary search to find the optimal period. This is because the solution space may not be convex with crosstalk [**44**]. Therefore, we examine each candidate period from the lower bound incrementally. The first feasible period under which the clock schedule is valid with crosstalk is the optimal solution.

9.5.2. A heuristic

Given a clock period, we first scale the clock schedule proportionally and treat all the memory elements as latches to take advantage of time borrowing. Then we compute the switching windows with crosstalk under the circular time representation. If there is no clocking violation, the clock period is feasible; otherwise, we apply a heuristic to remove the violations by replacing a subset of latches by flops.

Suppose a clocking violation occurs at $i \in V_L$. In order to find the latches for replacement, we recursively define the *source graph* of a vertex $v \in V$, or $sG(v)$, as

$$sG(v) = \begin{cases} \emptyset, & \text{if } v \in V_L \text{ and } d_v = D_v \\ sG(u) \cup \{u, (u,v)\}, & \text{if } (u,v) \in E_I \text{ or they couple} \end{cases}$$

Then $sG(i)$ can be obtained by tracing back from i along the interconnect edges and the coupling capacitances, where crosstalk takes effect, until we reach a memory element j with $d_j = D_j$ or a previously traversed vertex on each branch. Figure 9.6 shows an example of $sG(i)$, where vertical rectangles represent latches and horizontal rectangles represent the input switching windows of the vertices below them with the gray parts indicating the crosstalk effects.

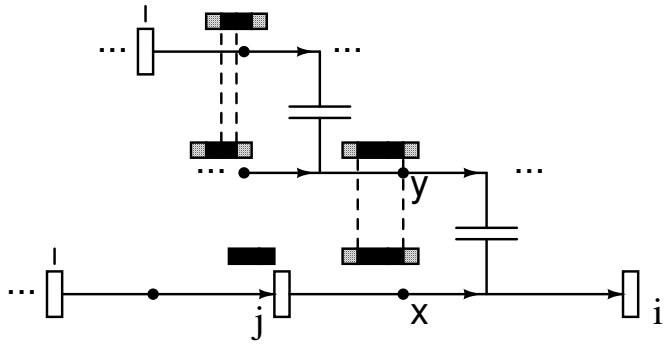

Figure 9.6. The source graph of latch i.

The following theorem states that the latches in $sG(i)$ are the candidates for replacement.

Theorem 9.5.1. *At least one latch in $sG(i)$ has to be replaced with a flop in order to remove the clocking violation at i.*

Proof. Suppose that the violation at i can be removed by only replacing some latches outside $sG(i)$. By the definition of $sG(i)$, there is no directed interconnect path from these

latches to the vertices in $sG(i)$. It implies that the effect of the replacement on $sG(i)$ is to introduce more couplings, which cause more switching window expansions at the vertices in $sG(i)$. However, the violation at i cannot be removed by merely expanding the switching windows of the vertices in $sG(i)$. Therefore, some latch in $sG(i)$ has to be chosen for replacement. □

If $sG(i)$ involves no coupling capacitance, the input switching window of i is determined by the minimum and maximum combinational delays from its critical predecessors. To remove a hold violation at i, a_i needs to be increased. This, by (9.2), implies that the earliest departure time at the earliest critical predecessor of i needs to be increased. Since the earliest departure time at the output of a latch is a constant by (9.4), the only way to increase it is to replace it by a flop. For a setup violation at i, we can trace back from i along the latest critical predecessors until we reach a memory element j in $sG(i)$ with $d_j = D_j$ such that A_i is determined by D_j through the maximum combinational delay from j to i. However, A_i cannot be decreased independent of whether j is a latch or a flop. In other words, the setup violation at i cannot be removed even without crosstalk effects. Therefore, the current clock period is infeasible.

Alternatively, if $sG(i)$ contains coupling capacitances, identifying the latches for replacement becomes very hard. Replacing a latch with a flop shrinks its output switching arc to a point. Unlike the situation without crosstalk, shrinking one switching arc may lead to decreases in maximum arrival times at the succeeding latches due to coupling misses. For the example in Figure 9.6, when latch j is replaced by a flop, if the resulting switching window at x does not overlap with that at y and the maximum arrival time at x is smaller than the original D_x, then the maximum arrival time at latch i is actually decreased due to the

replacement, which may help to fix the setup violation at i. In other words, the effect of a placement is dependent on the coupling configuration in the circuit, which in turn depends on the replacement. They are mutually dependent.

Therefore, we propose a heuristic that considers each latch in $sG(i)$ as a candidate. For each candidate, we estimate the effect of the replacement on the input switching window of i, assuming that the switching windows at other vertices will not change. Among the vertices whose individual replacements remove the violation, we choose the one whose latest arrival time is the closest to the falling edge of its clock phase. Intuitively, we want the increase of the latest arrival time of the latch being replaced to be as small as possible, so that the chance of introducing clocking violations at its succeeding memory elements is small. If the violation cannot be removed by replacing any latch in $sG(i)$, we choose the one that gives the least amount of violation. After replacing the chosen latch by a flop, we perform clock schedule verification with crosstalk on the new circuit. The above process is iteratively conducted until there is no clocking violation or the source graph has no candidate latch, for which we consider the current clock period infeasible. Since we replace a latch by a flop at each iteration and the number of latches in $sG(i)$ is no more than $|V_L|$, the number of iterations is upper bounded by $|V_L|$.

Since switching windows at many vertices may vary when a latch is replaced by a flop, computing the effect of the replacement using the original switching windows may produce a result that is different from the later one by actually carrying out a verification on the new circuit. However, since we only allow one latch to be replaced at a time, it ensures that our assumption yields good estimations.

In Figure 9.7, we give the pseudocode of the heuristic algorithm for finding the minimum clock period and the corresponding configuration of mixed latches and flops.

Algorithm
Input : A directed graph $G = (V, E)$ and
an n-phase clock schedule with period T.
Output: A configuration of mixed latches $V - V_r$
and flops V_r with the minimum period T^*.

```
Compute T_l and T_u;
T* ← T_l − ΔT;
Do
  Restore flops to latches;
  V_r ← ∅;
  T* ← T* + ΔT;
  Scale the clock schedule from T to T*;
  Do
    Compute switching windows w/o coupling under T*;
    If (9.5)-(9.6) are satisfied then
      Compute switching windows at gates by (9.7)-(9.10);
      Translate switching windows to global time;
      Take modulo T* on all the switching windows;
      Compute switching windows w/ coupling under T*;
      Check clocking violations by (9.11)-(9.12);
    If a clocking violation occurs on latch i then
      Choose latch j from sG(i) by heuristic;
      Replace j with a flop;
      V_r ← V_r ∪ {j};
  While (a new j is added into V_r);
While (clocking violations exist);
Return V_r and T*;
```

Figure 9.7. Pseudocode of the algorithm.

The following theorem gives the complexity of the algorithm.

Theorem 9.5.2. *The algorithm in Figure 9.7 terminates with a feasible clock period in* $O\big(|V_L|n^2\log(|V_L|) + |V_L|(|V_L|^3 + |E_C||E_I||V_L|)\frac{T_u - T_l}{\Delta T}\big)$ *time.*

Proof. It takes $O(|V_L|n^2 \log(|V_L|))$ to compute T_l and T_u by the algorithm in [**89**]. Computing the latch-to-latch minimum and maximum delays can be done in $O\big(|V_L|(|V_G|+|E_I|)\big)$ time if we use the algorithm in [**84**].

Given a clock period, it takes $O(|V_L|^3 + |E_C||E_I||V_L|)$ time to verify it with crosstalk by Theorem 9.4.3. If there is a clocking violation at latch i, then finding $sG(i)$ and identifying the latch for replacement take $O(E)$ and $O(V_L V)$ respectively. Therefore, the complexity for checking the feasibility of a given period by our heuristic is $O\big(|V_L|(|V_L|^3 + |E_C||E_I||V_L|)\big)$, where $O(|V_L|)$ bounds the number of iterations.

In conjunction with incremental search, the complexity of the algorithm can be computed as $O\big(|V_L|n^2 \log(|V_L|) + |V_L|(|V_L|^3 + |E_C||E_I||V_L|)\frac{T_u - T_l}{\Delta T}\big)$. In the worst case, the algorithm will return T_u, which is feasible. □

9.6. Experimental results

We implemented the algorithm in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. Our test files were generated from ISCAS-89 benchmark suite. First of all, each combinational vertex in the circuit was randomly assigned a maximum delay within 0.5 and 2.5 (assuming no coupling capacitors); the minimum delay was randomly initialized with a value that is at most 0.5 less than the maximum delay. We then randomly added coupling capacitors equal in number to 50% of the gates so that each gate has a capacitor incident to it by average. Each coupling capacitor was randomly assigned a delay between 0.0 and 2.0. After that, the delay of each coupling capacitor is added into the minimum and maximum delays of its two joint vertices. Finally, we converted flops to back-to-back ϕ_1/ϕ_2 latches and used ASTRA, a min-period retiming tool developed by Sapatnekar

and Deokar [**84**], to determine a symmetric, non-overlapping, two-phase retiming. The circuits used are summarized in Table 9.1.

Table 9.1. Sequential circuits from ISCAS-89

Circuit	$\|V\|$	$\|E\|$	IO's	latches	gates
s386	197	397	14	24	159
s420	339	519	19	102	218
s838	691	1067	35	210	446
s1196	713	1250	28	156	529
s1488	752	1553	27	72	653
s3271	2156	3470	40	544	1572
s3330	2459	3691	113	557	1789
s3384	2443	3714	69	689	1685
s4863	2919	4911	65	512	2342
s5378	3707	5475	84	844	2779
s6669	4248	6852	138	1030	3080
s9234	6662	9666	75	990	5597
s13207	9460	13558	214	1295	7951
s15850	11718	16685	227	1719	9772
s35932	21627	35958	355	5207	16065

To find the optimal clock period, we search the space starting with the lower bound specified by T_l in (9.18), incrementing this period by 1 until we find a feasible period. We cannot use binary search because the solution space may not be convex. This is confirmed by our experiments that feasible periods may interleave with infeasible ones. For simplicity, we assume $\tau = 0$ (strict overlap) and zero setup and hold times. The results are reported in Table 9.2. Column "T_l" lists the period lower bounds. Column "T_L^{opt}" lists the optimal periods for circuits with only latches. The optimal periods computed by our heuristic algorithm are listed in column "T_{L+F}^{opt}". Column "redu%" lists the percentage of period reduction with respect to the maximum possible reduction, i.e., $(T_L^{\text{opt}} - T_{L+F}^{\text{opt}})/(T_L^{\text{opt}} - T_l)$. Column "impr%" lists the percentage of period improvement with respect to the minimum period by using only latches, i.e., $(T_L^{\text{opt}} - T_{L+F}^{\text{opt}})/T_L^{\text{opt}}$. Since we choose the search step to be 1, the number of

iterations from "T_l" to "T_{L+F}^{opt}" for each circuit is equal to their difference. Column "#veri" lists the number of clock schedule verifications that our algorithm has carried out for each circuit before T_{L+F}^{opt} is found. The runtime is reported in column "t(sec)" in seconds.

Table 9.2. Computed clock period

Circuit	T_l	T_L^{opt}	T_{L+F}^{opt}	redu%	impr%	#veri	t(sec)
s386	33	38	33	100.0%	13.2%	9	0.00
s420	35	42	35	100.0%	16.7%	4	0.00
s838	49	62	49	100.0%	21.0%	38	0.04
s1196	74	89	89	0.0%	0.0%	769	0.68
s1488	72	76	75	25.0%	1.3%	36	0.04
s3271	94	116	97	86.4%	16.4%	59	0.30
s3330	143	196	143	100.0%	27.0%	8	0.02
s3384	102	120	120	100.0%	15.0%	14	0.09
s4863	155	170	170	0.0%	0.0%	97	0.71
s5378	162	220	162	100.0%	26.4%	2	0.02
s6669	158	209	164	88.2%	21.5%	86	1.49
s9234	254	344	254	100.0%	26.2%	41	1.52
s13207	543	573	553	66.7%	3.5%	80	4.04
s15850	446	654	654	0.0%	0.0%	4198	235.09
s35932	687	780	702	83.9%	10.0%	1263	198.04
avg				70.0%	13.2%		

The results in Table 9.2 reveal three things. Firstly, the values of T_l are the minimum periods we can possibly get for a symmetric two-phase clock schedule. However, crosstalk effects prevent the circuits from operating at T_l, which is illustrated by the fact that $T_L^{\text{opt}} > T_l$ for all the tested circuits. The overheads could be significant for some circuits. Secondly, by replacing a subset of latches into flops, our algorithm always successfully finds a feasible clock period T_{L+F}^{opt} with crosstalk. The effectiveness of our algorithm is shown by the circuits with $T_{L+F}^{\text{opt}} < T_L^{\text{opt}}$. In fact, half of the circuits are able to run at the indicated period lower bounds after the replacement, i.e., $T_{L+F}^{\text{opt}} = T_l$. For these circuits, our algorithm achieves 100% period reduction. On average, the percentage of period reduction is 70.0% and the

percentage of period improvement is 13.2%. Lastly, our algorithm is efficient. For the longest case "s15850" with over 11,000 vertices, the runtime 235.09 seconds are measured for 208 iterations and overall 4198 clock schedule verifications with crosstalk, i.e., 0.056 second per verification. Given that the approaches in [**44, 107**] take over 30 seconds (scaled based on the difference in CPU frequency between their machines and ours) to do a single clock schedule verification with crosstalk for circuits with around 4,000 vertices, the proposed approach under the circular time representation is much more efficient. The efficiency is due to the proposed efficient coupling detection.

For the circuits with $T_{L+F}^{\text{opt}} < T_L^{\text{opt}}$, we list in Table 9.3 the number of contributing capacitors that affect switching windows before and after the replacement in column "cap_{L}" and "$\text{cap}_{\text{L+F}}$", respectively. The number of latches chosen for replacement in each circuit under T_{L+F}^{opt} is listed in column "flop". The ratio between the number of flops and the number of vertices is listed in column "flop/$|V|$". We can see from Table 9.3 that "$\text{cap}_{\text{L+F}}$" is less than "cap_{L}" for all circuits. It means that the crosstalk effects are reduced after the replacement. In addition, the "flop/$|V|$" ratio is small. It implies that if a flop is implemented as two back-to-back latches, the area overhead due to the replacement is negligible.

To illustrate the crosstalk effects on clock period when the number of coupling capacitors varies, we use "s9234" as an example. The results are presented in Figure 9.8, where the x-axis is the percentage of the number of capacitors with respect to the number of gates, and the y-axis is the clock period. The results confirm that the overhead due to crosstalk effects, i.e., $(T_L^{\text{opt}} - T_l)/T_l$, grows severer as the number of capacitors increases. On the other hand, we find that the effectiveness of our algorithm is hardly affected by the number of capacitors. It achieves 100% period reduction, i.e., $T_{L+F}^{\text{opt}} = T_l$, for almost all different

Table 9.3. Change of contributing capacitors

Circuit	total cap	cap_L	cap_{L+F}	flop	flop/$\|V\|$
s386	79	54	42	8	4.06%
s420	109	32	1	3	0.88%
s838	223	136	18	37	5.35%
s1488	326	279	13	11	1.46%
s3271	786	503	6	48	2.23%
s3330	894	104	27	7	0.28%
s3384	842	610	595	13	0.53%
s5378	1389	908	190	1	0.03%
s6669	1540	1051	12	30	0.71%
s9234	2798	1714	231	40	0.60%
s13207	3975	3412	249	23	0.24%
s35932	8032	4296	2760	98	0.45%

capacitor numbers. As a result, the percentage of period improvement increases when more capacitors are present in the circuit.

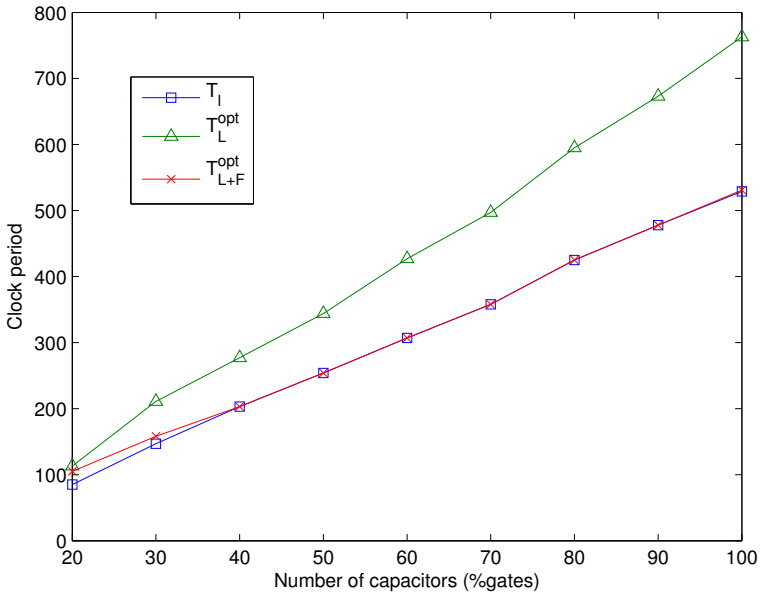

Figure 9.8. Crosstalk effects on clock period as the number of capacitors varies for "s9234".

We also notice that there are a few circuits in Table 9.2 that our algorithm cannot improve. It happens when replacing a latch by a flop in a circuit causes clocking violations at its succeeding memory elements. Recall that the replacement of a latch not only shrinks the width of the switching window but also increases the window to the falling edge of its clock phase. Intuitively, if we can adjust the falling edge of the clock phase after the replacement, the violations at the succeeding memory elements may be avoided. The technique for adjusting clock phases is also known as *clock skew scheduling* [**35**]. We will consider incorporating it into the proposed algorithm as our future work.

9.7. Conclusion

The trade-off between a latch and a flop in sequential circuit designs with crosstalk is formulated as seeking a configuration of mixed latches and flops to minimize the clock period. A circular time representation is proposed for coupling detection without additional phase translations, which are otherwise required in state-of-the-art approaches [**44, 107**]. We show that clock schedule verification under the circular time representation is easier. A heuristic algorithm is presented for finding the optimal configuration of latches and flops. Experimental results show that our algorithm is both effective and efficient.

The proposed framework and solution approach can equally apply to crosstalk induced by capacitive and inductive couplings.

References

[1] The mosek optimization tools version 3.2 user's manual and reference. [online] http://www.mosek.com.

[2] R. K. Ahuja, D. S. Hochbaum, and J. B. Orlin. Solving the convex cost integer dual network flow problem. *Management Science*, 49(7):950–964, July 2003.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Application*. Prentice Hall, 1993.

[4] R. Arunachalam, R. D. Blanton, and L. T. Pileggi. False coupling interactions in static timing analysis. In *Proc. of the Design Automation Conf.*, pages 726–731, 2001.

[5] R. Arunachalam, K. Rajagopal, and L. T. Pilleggi. Taco: Timing analysis with coupling. In *Proc. of the Design Automation Conf.*, pages 266–269, Los Angeles, CA, June 2000.

[6] S. M. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Institute of Technology, Computer Science Department, 1991.

[7] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. Intl. Conf. on Computer-Aided Design*, 1999.

[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In *Proc. of the Computer Aided Verification Conf.*, 1999.

[9] K. M. Carrig. Chip clocking effect on performace for ibms sa-27e asic. *IBM Micronews*, 6(3):12–16, 2000.

[10] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proc. of the Design Automation Conf.*, 2004.

[11] L.-F. Chao and E. H.-M. Sha. Retiming and clock skew for synchronous systems. In *Proc. Intl. Symposium on Circuits and Systems*, pages 1.283–1.286, 1994.

[12] P. Chen and K. Keutzer. Toward true crosstalk noise analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 132–137, 1999.

[13] P. Chen, D. A. Kirkpatrick, and K. Keutzer. Switching window computation for static timing analysis in presence of crosstalk noise. In *Proc. Intl. Conf. on Computer-Aided Design*, San Jose, CA, November 2000.

[14] C. Chu, E. F. Y. Young, D. K. Y. Tong, and S. Dechu. Retiming with interconnect and gate delay. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 221–226, 2003.

[15] P. Cocchini. Concurrent flip-flop and repeater insertion for high performance integrated circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 268–273, 2002.

[16] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat. Numerical computation of spectral elements in max-plus algebra. In *Proc. IFAC Conf. on Syst. Structure and Control*, Nantes, France, 1998.

[17] J. Cong, O. Coudert, and M. Sarrafzadeh. Incremental CAD. In *Proc. Intl. Conf. on Computer-Aided Design*, 2000.

[18] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang. Architectural synthesis integrated with global placement for multi-cycle communications. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 536–543, 2003.

[19] J. Cong, T. Kong, and D. Z. Pan. Buffer block planning for interconnect-driven floorplanning. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 358–363, 1999.

[20] J. Cong, H. Li, and C. Wu. Simultaneous circuit partitioning/clustering with retiming for performance optimization. In *Proc. of the Design Automation Conf.*, pages 460 – 465, 1999.

[21] J. Cong and S. K. Lim. Physical planning with retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 2–7, November 2000.

[22] J. Cong and X. Yuan. Multilevel global placement with retiming. In *Proc. of the Design Automation Conf.*, pages 208–213, Anaheim, CA, 2003.

[23] T. H. Cormen, C. E. Leiserson, and R. H. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[25] A. J. Daga, L. Mize, S. Sripada, C. Wolff, and Q. Wu. Automated timing model generation. In *Proc. of the Design Automation Conf.*, pages 146–151, 2002.

[26] F. Dartu and L. T. Pileggi. Calculating worst-case gate delays due to dominant capacitance coupling. In *Proc. of the Design Automation Conf.*, pages 46–51, Anaheim, CA, June 1997.

[27] A. Dasdan, S. Irani, and R. K. Gupta. An experimental study of minimum mean cycle algorithms. Technical Report 98-32, Univ. of California, Irvine, July 1998.

[28] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio. In *Proc. of the Design Automation Conf.*, pages 37–42, 1999.

[29] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[30] R. B. Deokar and S. S. Sapatnekar. A fresh look at retiming via clock skew optimization. In *Proc. of the Design Automation Conf.*, pages 310–315, 1995.

[31] S. Dey and S. Chakradhar. Retiming sequential circuits to enhance testability. In *Proceedings of the 12th IEEE VLSI Test Symposium*, pages 28–33, April 1994.

[32] S. Dey, M. Potkonjak, and S. G. Rotweiler. Performance optimization of sequential circuits by eliminating retiming bottlenecks. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 504–509, 1992.

[33] C. Ebeling and B. Lockyear. On the performance of level-clocked circuits. In *Advanced Research in VLSI*, pages 242–356, 1995.

[34] G. Even, I. Y. Spillinger, and L. Stok. Retiming revisited and reversed. *IEEE Transactions on Computer Aided Design*, 15(3):348–357, March 1996.

[35] J. P. Fishburn. Clock skew optimization. *IEEE Transactions on Computers*, 39:945–951, July 1990.

[36] M. Foltin, B. Foutz, and S. Tyler. Efficient stimulus independent timing abstraction model based on a new concept of circuit block transparency. In *Proc. of the Design Automation Conf.*, pages 158–163, 2002.

[37] S. Ghiasi, E. Bozorgzadeh, S. Choudhary, and M. Sarrafzadeh. Unified theory of timing budget management. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 653–659, 2004.

[38] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22:1–29, 1997.

[39] A. V. Goldberg and R. E. Tarjan. Solving minimum cost flow problem by successive approximation. In *"ACM Sympos. Theory Comput."*, pages 7–18, 1987.

[40] P. D. Gross, R. Arunachalam, K. Rajagopal, and L. T. Pileggi. Determination of worst-case aggressor alighment for delay calculation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 212–219, San Jose, CA, November 1998.

[41] S. Hassoun. Critical path analysis uing a dynamically bounded delay model. In *Proc. of the Design Automation Conf.*, pages 260–265, Los Angeles, CA, June 2000.

[42] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 247–253, 2002.

[43] S. Hassoun and C. J. Alpert. Optimal path routing in single- and multiple-clock domain systems. *IEEE Transactions on Computer Aided Design*, 22(11):1580– 1588, November 2003.

[44] S. Hassoun, C. Cromer, and E. C-Gamez. Static Timing analysis for level-clocked circuits in the presence of crosstalk. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 22(9):1270–1277, September 2003.

[45] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM*, 44(1):148–199, 1997.

[46] M. A. B. Jackson, A. Srinivasan, and E. S. Kuh. Clock routing for high-performance ic's. In *Proc. of the Design Automation Conf.*, 1990.

[47] A. Kahng, J. Cong, and G. Robins. High-performance clock routing based on recursive geometric matching. In *Proc. of the Design Automation Conf.*, 1991.

[48] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning:application in vlsi domain. In *Proc. of the Design Automation Conf.*, pages 526–529, 1997.

[49] Y. Kohira and A. Takahashi. Clock period minimization method of semi-synchronous circuits by delay insertion. *IEICE Trans. Fundamentals*, E88-A(4):892–898, April 2005.

[50] K. N. Lalgudi and M. C. Papaefthymiou. An efficient tool for retiming with realistic delay modeling. In *Proc. of the Design Automation Conf.*, San Francisco, CA, June 1995.

[51] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[52] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *"International Symposium on Microarchitecture"*, pages 330–335, 1997.

[53] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proc. of the Third Caltech Conf.*, pages 86–116, Rockville, MD, 1983. Computer Science Press.

[54] R. Levy, D. Blaauw, G. Braca, A. Dasgupta, A. Grinshpon, C. Oh, B. Orshav, S. Sirichotiyakul, and V. Zolotov. Clarinet: A noise analysis tool for deep submicron design. In *Proc. of the Design Automation Conf.*, pages 233–238, 2000.

[55] C. Lin and H. Zhou. Optimal wire retiming without binary search. *IEEE Transactions on Computer Aided Design*. Accepted for publication.

[56] C. Lin and H. Zhou. Retiming for wire pipelining in system-on-chip. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 215–220, 2003.

[57] C. Lin and H. Zhou. Optimal wire retiming without binary search. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 452–458, 2004.

[58] C. Lin and H. Zhou. Wire retiming for system-on-chip by fixpoint computation. In *Proc. DATE: Design Automation and Test in Europe*, pages 1092–1097, 2004.

[59] C. Lin and H. Zhou. Clustering for processing rate optimization. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 189–195, 2005.

[60] C. Lin and H. Zhou. Trade-off between latch and flop for min-period sequential circuit designs with crosstalk. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 329–334, 2005.

[61] C. Lin and H. Zhou. Wire retiming as fixpoint computation. *IEEE Transactions on Very Large Scale Integration Systems*, 13(12):1340–1348, December 2005.

[62] C. Lin and H. Zhou. An efficient retiming algorithm under setup and hold constraints. In *Proc. of the Design Automation Conf.*, 2006.

[63] C. Lin, H. Zhou, and A. Xie. Design closure driven delay relaxation based on convex cost network flow. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2006.

[64] B. Lockyear and C. Ebeling. Minimizing the effect of clock skew via circuit retiming. Technical Report UW-CSE-93-05-04, Dept. of Computer Science and Engineering, University of Washington, Seattle, 1993.

[65] B. Lockyear and C. Ebeling. The practical application of retiming to the design of high performance systems. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 288–295, 1993.

[66] R. Lu and C. K. Koh. Interconnect planning with local area constrained retiming. In *Proc. DATE: Design Automation and Test in Europe*, pages 442–447, 2003.

[67] R. Lu, G. Zhong, C. K. Koh, and K. Y. Chao. Flip-flop and repeater insertion for early interconnect planning. In *Proc. DATE: Design Automation and Test in Europe*, pages 690–695, 2002.

[68] D. G. Luenberger. *Linear and nonlinear programming*. Addison-Wesley, Reading, Massachusetts, 1984.

[69] N. Maheshwari and S. S. Sapatnekar. Optimizing large multi-phase level-clocked circuits. *IEEE Transactions on Computer Aided Design*, 18(9):1249–1264, September 1999.

[70] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):74–84, January 1991.

[71] H.-G. Martin. Retiming by combination of relocation and clock delay adjustment. In *Proc. European Conf. on Design Automation*, pages 384–389, 1993.

[72] 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.

[73] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli. On clustering for minimum delay/area. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 6–9, 1991.

[74] R. Nair, C. L. Berman, P. S. Hauge, and E. J. Yoffa. Generation of Performance Constraints for Layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 8(8):860–874, August 1989.

[75] V. Nookala and S. S. Sapatnekar. A method for correcting the functionality of a wire-pipelined circuit. In *Proc. of the Design Automation Conf.*, pages 570–575, 2004.

[76] R. H. J. M. Otten and R. Brayton. Planning for performance. In *Proc. of the Design Automation Conf.*, pages 122–127, 1998.

[77] P. Pan, A. K. Karandikar, and C. L. Liu. Optimal clock period clustering for sequential circuits with retiming. *IEEE Transactions on Computer Aided Design*, 17(6):489–498, June 1998.

[78] M. C. Papaefthymiou. Asymptotically efficient retiming under setup and hold constraints. In *Proc. Intl. Conf. on Computer-Aided Design*, 1998.

[79] S. Pullela, N. Menezes, and L. T. Pillage. Reliable nonzero clock skew trees using wire width optimization. In *Proc. of the Design Automation Conf.*, 1993.

[80] S. Pullela, N. Menezes, and L. T. Pillage. Skew and delay optimization for reliable buffered clock trees. In *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[81] K. Ravindran, A. Kuehlmann, and E. Sentovich. Multi-domain clock skew scheduling. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 801–808, 2003.

[82] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun. *check*T_c and *mint*$_c$: Timing verification and optimal clocking of synchronous digital circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 552–555, November 1990.

[83] S. S. Sapatnekar. A timing model incorporating the effect of crosstalk on delay and its application to optimal channel routing. *IEEE Transactions on Computer Aided Design*, 19(5):550–559, May 2000.

[84] S. S. Sapatnekar and R. B. Deokar. Utilizing the retiming-skew equivalence in a practical algorithm for retiming large circuits. *IEEE Transactions on Computer Aided Design*, 15(10):1237–1248, October 1996.

[85] Semiconductor Industry Association, http://public.itrs.net. *International Technology Roadmap for Semiconductors*, 2001.

[86] Semiconductor Industry Association, http://public.itrs.net. *International Technology Roadmap for Semiconductors*, 2005.

[87] E. M. Sentovich and R. K. Brayton. Preserving Don't Care Conditions During Retiming. In *Proc. Intl. Conf. on VLSI*, pages 461–470, August 1991.

[88] F. Sheikh, A. Kuehlmann, and K. Keutzer. Minimum-power retiming for dual-supply cmos circuits. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2002.

[89] N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Graph algorithms for clock schedule optimization. In *Proc. Intl. Conf. on Computer-Aided Design*, 1992.

[90] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, 1994.

[91] N. V. Shenoy. *Timing Issues in Sequential Circuits*. PhD thesis, UC Berkeley, 1993.

[92] N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming of circuits with single phase level-sensitive latches. In *Proc. Intl. Conf. on Computer Design*, 1991.

[93] N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Minimum padding to satisfy short path constraints. In *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[94] L. Singhal and E. Bozorgzadeh. Fast timing closure by interconnect criticality driven delay relaxation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 792–797, 2005.

[95] V. Singhal, S. Malik, and R. K. Brayton. The Case for Retiming with Explicit Reset Circuitry. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 618–625, November 1996.

[96] S. Sirichotiyakul, D. Blaauw, C. Oh, R. Levy, V. Zolotov, and J. Zuo. Driver modeling and alignment for worst-case delay noise. In *Proc. of the Design Automation Conf.*, pages 720–725, 2001.

[97] T. Soyata, E. G. Friedman, and J. H. Mulligan. Incorporating interconnect, register, and clock distribution delays into the retiming process. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 16(1):105–120, January 1997.

[98] A. Srivastava, S. O. Memik, B.-K. Choi, and M. Sarrafzadeh. Achieving design closure through delay relaxation parameter. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 54–57, 2003.

[99] T. G. Szymanski. Computing optimal clock schedules. In *Proc. of the Design Automation Conf.*, 1992.

[100] T. G. Szymanski and N. Shenoy. Verifying clock schedules. In *Proc. Intl. Conf. on Computer-Aided Design*, 1992.

[101] A. Tabbara, B. Tabbara, R. K. Brayton, and A. R. Newton. Integration of retiming with architectural floorplanning. *INTEGRATION, the VLSI journal*, 29:25–43, 2000.

[102] P. F. Tehrani, S. W. Chyou, and U. Ekambaram. Deep sub-micron static timing analysis in presence of crosstalk. In *International Symposium on Quality Electronic Design*, pages 505–512, 2000.

[103] D. K. Y. Tong and E. F. Y. Young. Performance-driven register insertion in placement. In *International Symposium on Physical Design*, pages 53–60, 2004.

[104] R.-S. Tsay. Exact zero skew. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 336–339, 1991.

[105] T. Xiao and M. Marek-Sadowska. Functional correlation analysis in crosstalk induced critical paths identification. In *Proc. of the Design Automation Conf.*, pages 653–656, 2001.

[106] C.-Y. Yeh and M. Marek-Sadowska. Minimum-area sequential budgeting for fpga. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 813–817, 2003.

[107] H. Zhou. Clock schedule verification with crosstalk. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2002.

[108] H. Zhou. Timing analysis with crosstalk is a fixpoint on a complete lattice. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 22(9):1261–1269, September 2003.

[109] H. Zhou. Deriving a new efficient algorithm for min-period retiming. In *Proc. Asian and South Pacific Design Automation Conference*, 2005.

[110] H. Zhou and C. Lin. Retiming for wire pipelining in system-on-chip. *IEEE Transactions on Computer Aided Design*, 23(9):1338–1345, September 2004.

[111] H. Zhou, D. F. Wong, I-M. Liu, and A. Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. Proc. of the Design Automation Conf., 1999.

Vita

Chuan Lin was born in Longyan, Fujian, China on May 8, 1980, the only son of Yixing Lin and Huiying Zheng. With the 2nd Order Award (top 50) in National Physics Olympiad, he was honored with the privilege of entering Tsinghua University in Beijing, China in 1998, waived of the national college entrance examination. He received the degree of Bachelor of Engineering in 2002 from Tsinghua University with the Outstanding Graduate Award. In September 2002 he joined the NuCAD Research Laboratory in Northwestern University with the prestigious Walter P. Murphy Fellowship for graduate studies.

He has published more than 10 technical papers in leading journals and conferences in the area of VLSI design automation. When he was pursuing his doctoral degree, he was employed as a teaching and research assistant in the Electrical Engineering and Computer Science department of Northwestern University. He also worked as a Summer Intern from June 2005 to September 2005 at Calypto Design Systems Inc, Santa Clara, California.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]The macros used in formatting this dissertation were written by Miguel A. Lerma, Mathematics Department of Northwestern University.