Ph.D. DISSERTATION

Methodology for Solving Timing Closure Problem by Utilizing Adjustable Delay Clock Buffers

가변 지연 시간 클락 버퍼 활용을 통한 타이밍 일치 문제 해결 방법론

BY

JUYEON KIM

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Methodology for Solving Timing Closure Problem by Utilizing Adjustable Delay Clock Buffers

가변 지연 시간 클락 버퍼 활용을 통한 타이밍 일치 문제 해결 방법론

BY

JUYEON KIM

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Methodology for Solving Timing Closure Problem by Utilizing Adjustable Delay Clock Buffers

가변 지연 시간 클락 버퍼 활용을 통한 타이밍 일치 문제 해결 방법론

지도교수 김 태 환

이 논문을 공학박사 학위논문으로 제출함

2017년 11월

서울대학교 대학원

전기 컴퓨터 공학부

김 주 연

김주연의 공학박사 학위 논문을 인준함

2017년 12월

Abstract

As clock timing is closely related to the performance of synchronous systems, many synthesis techniques were suggested to optimize clock distribution networks. Especially, meeting clock skew constraints is one of the most important objectives that should be achieved for successful operation of the design. Meanwhile, multiple power mode designs made the clock timing problem harder to tackle due to the dynamic delay change caused by varying supply voltages. Inserting adjustable delay buffers (ADBs) on the clock tree and controlling its delay can be a solution to the problem. However, because ADBs require non-negligible area and control overhead, it should be carefully inserted to minimize the number of ADBs. This work provides solutions to the ADB minimization problem under the environment of multiple power modes in which the clock path delay varies as power mode changes. Precisely, (1) an O($n \log n$) time algorithm that optimally solves the problem under clock skew bounds and (2) a graph based algorithm which supports useful skew scheduling are proposed, along with (3) their practical extensions, such as supporting discrete delay values and reducing more ADBs by integrating buffer sizing scheme. The experimental results showed that proposed ADB allocation algorithms under constant clock skew bound and useful skew constraints allocated 13.5% and 23.3% less number of ADBs on average, respectively, compared to the best known ADB allocation algorithm under the same constraints.

keywords: ADB, clock network design, multiple power mode, clock skew

student number: 2013-20776

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Introduction

Clock is a periodic signal which triggers the state transition of synchronous systems. It is one of the most important signals of synchronous digital systems, as all the operation of synchronous components such as flip-flops and latches rely on it. To prevent the timing failure of the design, the clock signal should arrive at its sinks at the accurate time. However, the increasing demands on integrated circuit (IC) chip performance are reducing the timing margin and making the clock design problem more challenging.

Clock signal is distributed to the whole components in the chip through clock distribution networks (CDNs), such as clock trees, clock meshes and clock spines. Clock tree is a tree-structured CDN, whose root corresponds to clock source and has clock sinks as its leaf nodes. Because it requires less resource on implementation and timing analysis is relatively simple, the clock tree structure is widely used so that synthesizing and optimizing clock trees has been one of the most important issues in VLSI design. The research on clock tree synthesis and optimization can be roughly classified into two directions: generating clock tree structures or topologies (e.g., [4–9]) and optimizing them for better timing quality (e.g., [10–16]). One common assumption of these prior works is that the developed clock network should be applied to single power mode designs, in which every design module operates in a single or global supply voltage.

However, as minimizing power consumption is one of the important design concerns, the design paradigm is shifting to multiple power mode designs, in which a design module can operate in different supply voltages on the different power modes. Multiple power mode design is very helpful for saving power consumption, but meeting the time constraints for every power mode by controlling the arrival times of clock signal becomes much more difficult because clock buffer delays vary depending on the power modes.

Meanwhile, inserting **Adjustable Delay Buffers** (ADBs) enables dynamic delay tuning by which the clock timing problem caused by the voltage change can be dealt with. The delay of an ADB is adjusted by its control inputs, thus the clock arrival times at each clock sink can be tuned during the operation. The idea of using ADBs in multiple power modes is to replace some of normal clock buffers with ADBs and control their delays by dynamically changing control inputs, so that the clock skew constraints on each power mode can be met [17].

The main drawback of inserting ADBs on clock tree is that it requires non-negligible area overhead. Thus, minimizing the number of ADBs to be inserted is the most essential problem to be solved for resolving timing violations of multiple power mode designs at a reasonable cost. This work provides a complete solution to the ADB minimization problem, which corrects timing violations in multiple power mode design with the minimum number of ADBs allocated.

1.2 Contributions of This Dissertation

In this dissertation, the methods of ADB allocation and delay assignment are proposed to optimize the area utilization while resolving the clock skew variation problem in multiple power mode designs.

- In Chapter 4, an O($n \log n$) time algorithm that optimally solves the problem of minimizing the number of ADBs to be allocated under the given clock skew bound is proposed. It includes the extended version of the algorithm that supports the use of ADBs which have discrete delays, and replacing the ADBs with various sizes of buffers to reduce the area further.

- In Chapter 5, a graph based algorithm solving the problem with useful skew scheduling is proposed, along with the acceleration techniques to trade-off runtime and quality of the result, and supporting discrete or bounded delay values of ADBs.

The outcomes of this work will enable the use of ADBs with small overhead, and they can also be applied usefully to the diverse environments, such as non-uniform thermal effect, in which the clock skew varies dynamically during the operation.

Chapter 2

BACKGROUND

2.1 Multiple Power Mode Design

One of the most efficient way of reducing power is to lessen the supply voltage for the design when high performance is not required [18, 19]. Reducing the voltage levels depending on the tasks on operation, called dynamic voltage scaling, cannot only be applied in global chip level, but also for smaller units called voltage islands.

Multiple power mode design is a design strategy that exploits the advantage of dynamic voltage scaling in higher resolution. If some modules in a design have less tasks to do, applying low voltage to them will be more effective than applying high voltage for the whole circuit only for few modules that should have higher performance. In multiple power mode design, the circuit is divided into several voltage islands and different voltages are applied for each voltage island depending the operating mode. For instance, some tasks would require more computations in microprocessor unit, while the others would use digital signal processor more. Then, in each operating mode, the modules with heavy load use high voltage and the others use low one.

The drawback of this method is that the delay of clock buffers are also affected by the applied voltage. As two clock paths might cross different voltage domains, the relative clock arrival times on their sinks will vary depending on the operating modes.

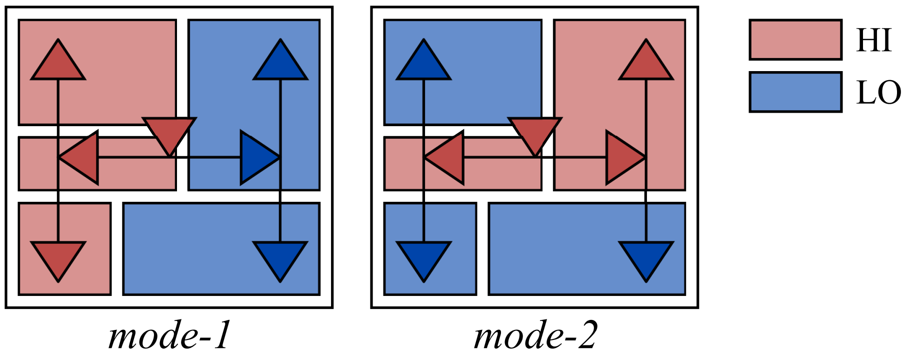


Figure 2.1: In multiple power mode design, applied voltage for each voltage island varies depending on the operating mode.

As a result, the circuit suffers from timing violations [20,21]. In Figure 2.1, the voltage level of each voltage island shifts from high to low, considering the operation speed required in the current mode. Then, the clock buffers operate under the voltage applied on voltage islands they locate, so that the clock arrival times on sinks cannot maintain their values as the power mode changes.

2.2 Setup Time and Hold Time Constraints

Time constraints describe the conditions which prevent the circuit from the timing failures. Especially, arrival time of clock events on synchronous components is one of the most important concerns to be considered for the timing closure. The input signal of a flip-flop in a synchronous circuit should be held steady before and after the clock event, to ensure the correct data being captured. The minimum duration of time that the input should be stable before and after the clock arrival is called setup time and hold time, respectively.

Thus, the timing failures in synchronous circuits can be classified into two situations, the valid input data not being able to arrive until the point capturing starts, and the data of the next cycle affecting the input before the capturing finishes. Figure 2.2

shows the timing waveforms at clock and data input pin of a flip-flop. The data input in Case 1 becomes stable at the latest setup time before the clock arrival, and retain stability until hold time after the clock. In this case, the flip-flop successfully stores the correct data. On the other hand, the input signals in Case 2 and Case 3 are not stable during the timing interval defined by setup time and hold time of the flip-flop, causing the timing violations. Each situation is called setup time violation and hold time violation.

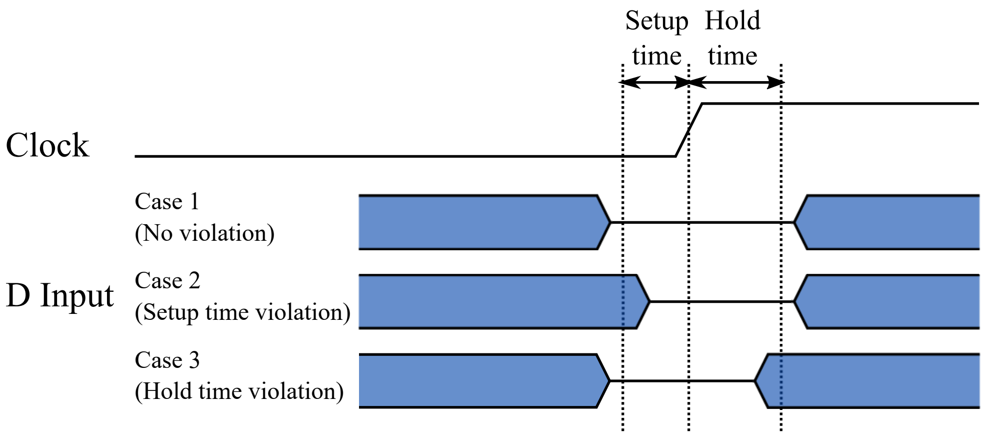


Figure 2.2: Timing waveforms showing three cases of flip-flop input timing, where timing constraints are satisfied, setup time constraint is violated, and hold time constraint is violated, respectively.

In circuit designs, the input data of flip-flops usually change depending on the output of the flip-flops connected through combinational logic cells. This implies that any pair of related flip-flops should be checked whether timing violations occur, to keep the circuit safe from the timing hazards. Figure 2.3 shows a pair of flip-flops, s_i and s_j, of a synchronous system. The maximum delay from the clock input of s_i to the data input of s_j is $D_{i,j}^{max}$ and the minimum is $D_{i,j}^{min}$. (The notation $D_{i,j}^{max}$ and $D_{i,j}^{min}$ in this paper includes the clock-to-Q delay of s_i, which is the time interval from the clock arrival on the clock pin and to the data launch from the data output pin.) Then, the interval in which data signal at the input pin of s_j is reliable begins at the time $D_{i,j}^{max}$ elapsed from the data launch, and ends at the time $D_{i,j}^{min}$ after the next clock arrival at s_i.

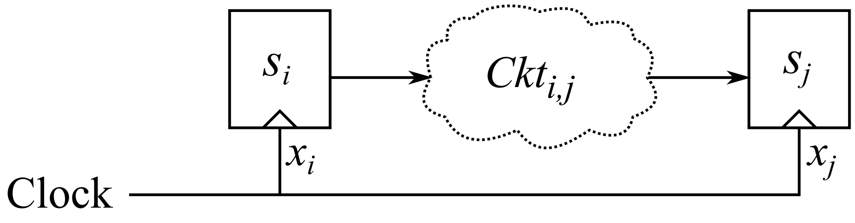


Figure 2.3: A datapath $Ckt_{i,j}$ from flip-flop s_i to s_j in a synchronous circuits.

To ensure the data signal arrives at the time it is required, in other words, to prevent setup time violation, x_i and x_j, the clock arrival time of s_i and s_j, should satisfy the following inequality:

$$x_i + D_{i,j}^{max} \leq x_j + T_{clk} - t_{setup,j} \tag{2.1}$$

where T_{clk} is the clock period and $t_{setup,j}$ is the setup time of s_j. Likewise, to prevent hold time violation, the data of the next cycle should not affect the current data capture. Thus, the following inequality:

$$x_i + D_{i,j}^{min} \geq x_j + t_{hold,j} \tag{2.2}$$

is also needed to be met. The former constraint is called *setup time constraint*, and the latter is called *hold time constraint*. Each constraint can be transformed as:

$$x_j - x_i \geq D_{i,j}^{max} - T_{clk} + t_{setup,j} \tag{2.3}$$

and

$$x_i - x_j \geq -D_{i,j}^{min} + t_{hold,j}. \tag{2.4}$$

The latter form of time constraints (Equation (2.3) and Equation (2.4)) will be used in this paper since we will consider only the difference between clock arrival times in this paper.

2.3 Clock Skew Optimization Objectives

As the satisfaction of setup and hold time constraints depends on the clock arrival times, delivering the clock signal to clock sinks at the desired time has been one of the main concerns in synchronous digital system designs. The optimization of clock signal arrival times can be performed based on either one of the two objectives: (1) meeting or minimizing *(global) clock skew* bound and (2) maximizing the exploitation of *useful clock skew* [22].

Because clock skew of a pair of clock sinks is defined by a time difference between the clock arrival on each of them, *global clock skew* refers to the difference of the latest and earliest arrival times of a clock signal to flip-flops. In other words, global clock skew is the maximum clock skew among all pairs of clock sinks of the whole circuit. If no confusion occurs, the global clock skew is simply referred to as *clock skew* in this presentation. As circuits are usually designed assuming zero clock skew, the tighter the clock skew is, the more setup and hold time constraints are likely to be met. For this reason, achieving zero or bounded clock skew for clock distribution networks can be an effective solution of the timing optimization problem, especially when the design size is huge so that considering every relationship between clock sinks requires enormous design time and effort.

On the other hand, since the setup and hold time constraints are looser for some pairs of sinks than the others, it is beneficial to intelligently schedule the arrival times of clock signal to distribute timing margins in a way to satisfy all the timing constraints. For instance, given three flip-flops s_i, s_j and s_k connected by combinational logics $Ckt_{i,j}$ and $Ckt_{j,k}$, increasing the clock path delay on s_j adds more margin to the setup constraint between s_i and s_j, and the hold constraint between s_j and s_k, and reduces the margin of the setup constraint between s_j and s_k, and the hold constraint between s_i and s_j. This method can be utilized to relax tight timing constraints in pipeline architecture by ‘stealing’ timing margin, called slack, from shorter datapaths and give them to longer ones. It is called *useful skew scheduling*. From the designers’ point of view, the problem of optimizing clock networks under useful skew scheduling is more complicated than the one limiting clock skew bounded by constant values since the former is required to examine the satisfaction of all the setup and hold time constraints during the skew scheduling.

2.4 Adjustable Delay Buffers

Adjustable delay buffer (ADB) is a buffer whose delay can be adjusted depending on the control signal. ADBs have been suggested as a solution to process variation problem that can be adopted in post-silicon tuning stage [23–26]. However, rather than fixing the delay to a constant value, they can generate varying delays when dynamically changing control signal is assigned [17]. This strategy enables ADBs to solve the timing problem in multiple power mode design, by changing clock arrival times during the operation.

Figure 2.4 shows a structure of a capacitor bank based adjustable delay buffer implementation [3]. An ADB is composed of two inverters, one at the input port and the other at the output port, and an array of capacitors with switch transistors is connected between them. The switches are activated by a capacitor bank controller to change the number of active capacitors depending on the value of control bits. Activating more capacitors increases the total capacitance between the two inverters, which in turn increases the signal propagation delay between the input and output ports.

As the delay generation of an ADB relies on the MOS capacitors in the capacitor bank, the size of capacitor bank is directly proportional to the maximum delay of

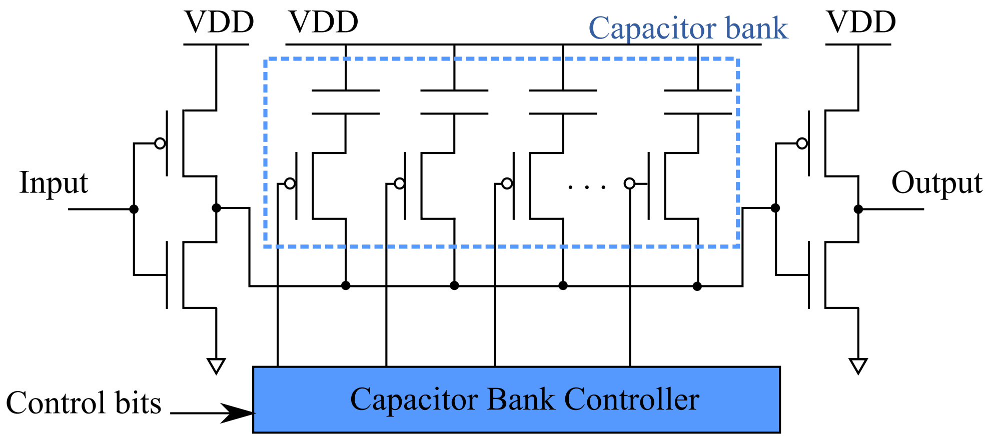


Figure 2.4: The implementation structure of a capacitor bank based ADB [3]. The ADB delay is dynamically adjusted by turning on and off the individual capacitors in the bank.

ADB. Thus, the ADB cells necessarily occupy significantly larger area than those of the ordinary clock buffers, to generate a sufficient amount of delay to control the clock arrival times. It adds non-negligible area overhead to the circuit, so reducing the number of ADBs to be inserted is an essential topic to be dealt to efficiently tackle timing problems in multiple power mode designs.

Chapter 3

TIMING CLOSURE IN MULTIPLE POWER MODE DESIGNS

3.1 ADB Allocation for Timing Correction

While many optimization methods were effective, advanced low power design techniques introduced new challenges to the clock skew control problem. Specifically, for multiple power mode designs, where the supply voltage to the circuit components changes dynamically depending on modes, the clock arrival time also varies accordingly. Even though the previous works can consider the clock skew constraints on every power mode, it would be highly likely that the resulting clock tree uses a substantially long wirelength or there exists no clock tree that satisfies the clock skew constraint on every power mode.

Meanwhile, allocating ADBs on the clock tree and assigning control signals for each mode can intentionally increase the clock path delays passing them, in order to resolve the timing violations. The idea of using ADBs in multiple power modes is to replace some of normal clock buffers with ADBs so that the clock skew constraints on each power mode can be met; when the power mode changes during execution, for example from power mode *mode-1* to power mode *mode-2*, the delays of ADBs in clock tree that have been adjusted under *mode-1* are readjusted to meet the clock skew constraints under *mode-2*.

The proposed clock timing correction flow is as follows. An initial clock tree is synthesized using ordinary clock buffers, which inevitably cause timing violations in multiple power mode. Based on the initial delays obtained by simulations under available power modes, the proposed algorithm finds the optimal set of clock buffers that should be replaced by ADBs. The algorithm also calculates the delay values that the ADBs should have in each mode, and converts them into the control signals. The calculation should take the change in applied voltage on ADBs in each power mode into account, to generate the intended delays in every case. The final ADB delay is implemented using a mode controller which stores and returns the calculated on/off signals to the capacitor bank of each ADB depending on the power mode.

Figure 3.1 shows an example of clock tree $\mathcal{T}$ that has four sinks s_1, s_2, s_3, and s_4. Assuming there are two power modes *mode-1* and *mode-2* in $\mathcal{T}$, the two numbers on each sink represent the clock signal arrival times to the sink on *mode-1* and *mode-2*. With the clock skew bound of 10, $\mathcal{T}$ has a clock skew violation between s_1 and s_4 in *mode-2* if the ADB is not used. To solve this problem, an ADB replaces a buffer on left which drives s_1 and s_2. The two numbers on ADB represent the values of delay increment in *mode-1* and *mode-2*. Precisely, the ADB adds delay of 3 in *mode-2*, increasing the signal arrival time to s_2 in *mode-2* to 6. Then, all the clock skews are within bound of 10.

3.2 Problem Definitions

Since ADBs add non-negligible area overhead to the circuit, minimizing the number of ADB cells to be allocated has been the most essential problem to solve for the effective use of ADBs. Meanwhile, as mentioned in Section 2.3, the clock timing optimization can have different kinds of objectives to eliminate the possibility of timing failure,

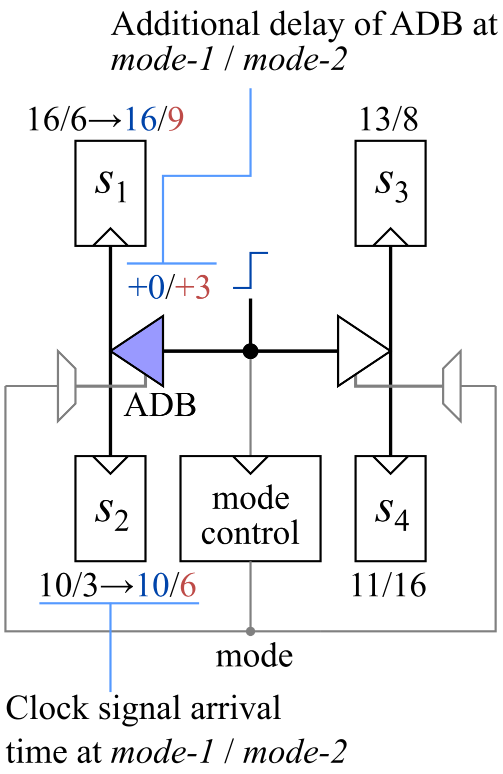


Figure 3.1: An example of clock tree $\mathcal{T}$ with the replacement of a clock buffer with ADB under bounded clock skew $(= 10)$ constraint.

meeting the constant global clock skew bound and utilizing useful skew scheduling.

Thus, the ADB-based clock tree optimization problem can be described as:

Problem 1 (ADB insertion problem under clock skew bound)**.** *Given a synthesized clock tree, arrival times of clock sinks in each power mode, and clock skew bounds κ_m, replace the least number of clock buffers with ADBs and assign delays to the ADBs to satisfy the bound κ_m in all power modes.*

Problem 2 (ADB insertion problem under useful skew)**.** *Given a synthesized clock tree, arrival times of clock sinks in each power mode, setup and hold time slacks, replace the least number of clock buffers with ADBs and assign delays to the ADBs to satisfy setup and hold time constraints in all power modes.*

Chapter 4

ADB ALLOCATION UNDER CLOCK SKEW BOUND

In this chapter, an optimal solution to Problem 1, the ADB insertion problem under constant clock skew bound is proposed.[1]

4.1 Related Works and Motivational Examples

Synchronous circuits are usually designed under the assumption of an ideal clock network, which drives the whole flip-flops to change their values at the same time. Thus, implementing clock trees to have zero or bounded clock skew is an efficient and safe solution to prevent timing violations. Several works have attempted to apply ADB insertion techniques for the timing closure in multiple power mode designs, while maintaining the area overhead reasonable. Su *et al.* [21] proposed a linear-time optimal algorithm for the delay assignment when the locations of ADBs are given. Then, they exploited the algorithm to find the appropriate points to allocate ADBs heuristically in a greedy manner. Lin *et al.* [29] proposed an efficient allocation algorithm of two-stage approach which performs a top-down ADB allocation followed by a bottom-up ADB elimination. Even though the approach reduced the runtime over that in [21], it still did not guarantee an optimality of ADB allocation. Lim and Kim [1] proposed a

[1]The content of this chapter is an extended version of [27, 28].

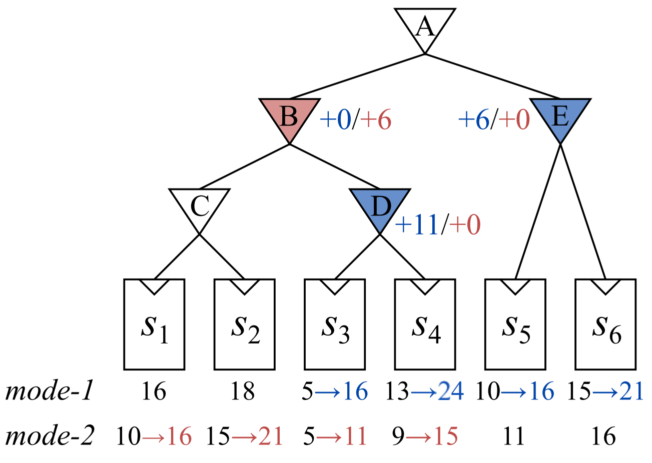
linear-time algorithm for the ADB allocation problem where they solved the problem optimally for *each* power mode.

Two common features of the previous ADB allocation algorithms ([1, 21, 29]) are that (1) they resolve the clock skew violation by synchronizing *the earliest arrival times* of subtrees of interest where they set the delay value of ADB on a root of one of the subtrees to the difference of the earliest arrival times of the subtrees; (2) the methods are applied *mode by mode*, independently.

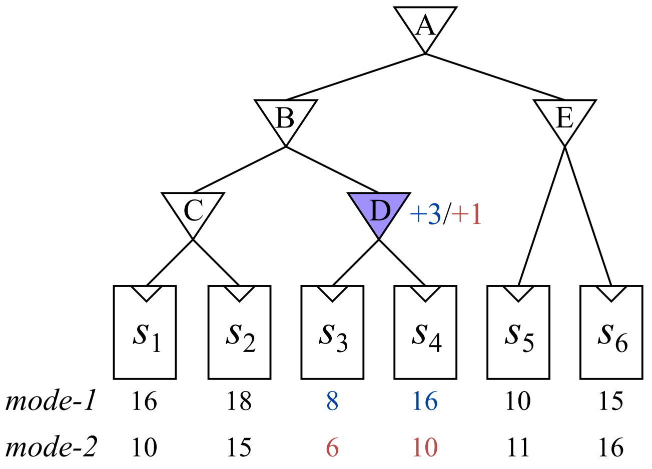
For example, consider the clock tree in Figure 4.1(a) with two operating modes *mode-1* and *mode-2*. The initial clock signal arrival times at sinks are shown at the bottom in black numbers. Suppose the clock skew bounds for each power mode, κ_1 and κ_2 is 10. Clearly, there are clock skew violations in both *mode-1* and *mode-2*; in *mode-1* the clock skew is 13, which is defined by sinks s_2 and s_3, and in *mode-2*, the clock skew is also 11, which is defined by s_3 and s_6.

The results of ADB allocation produced by the previous algorithm [1] for the clock tree is shown in Figure 4.1(a) where buffers B, D, and E are replaced with ADBs and the adjusted arrival times are shown in pairs of numbers next to the initial delays. Their delay adjustment procedure is as follows. In *mode-1*, the earliest arrival time $(= 5)$ of the subtrees rooted at D is synchronized to the earliest arrival time $(= 16)$ of the subtree rooted at C by assigning delay increment of 11 to the ADB in node D. However, the delay adjustment at D increases the arrival time at sink s_4 from 13 to 24, which causes another skew violation between the times in s_4 and s_5. The violation is then resolved by assigning delay increment of 6 to the ADB in node E. Likewise, in *mode-2* the clock skew violation due to the times at s_3 and s_6 is resolved by assigning delay increment of 6 to the ADB in node B.

From the ADB allocation and delay assignment, we observe that (1) synchronizing the subtree's earliest arrival times (e.g., time at s_3 in *mode-1*) introduces delay increases at the other sinks (e.g., s_4), so that additional ADB allocation with delay adjustment shall be needed; (2) even though the skew violation in *mode-2* requires



(a)



(b)

Figure 4.1: A motivational example for ADB allocation and delay assignment when the skew bound for each mode is given to 10 units of delay. (a) A clock tree with three ADBs allocated by the method of [1] to resolve clock skew violation. (b) An optimal allocation which uses one ADB.

one ADB to be allocated, node B is not the only position at which an ADB could be allocated. An alternative position is D, which coincides with the ADB allocation in *mode-1*.

An optimal ADB allocation is shown in Figure 4.1(b) in which only one ADB with delay increment of 3 in *mode-1* and 1 in *mode-2* is inserted to the tree. This example clearly shows that delay adjustment according to the synchronization of the earliest arrival times does not always yield optimal results. Furthermore, merely collecting the optimal results on individual power modes does not mean globally optimal for all power modes. In order to find optimal results, ADB allocation should consider all modes simultaneously.

4.2 ADB Allocation Algorithm Satisfying Clock Skew Bound

This section describes our proposed ADB allocation algorithm, ADB-PULLUP, to ensure the clock skew bounded by a given margin. The notations commonly used in the presentation is summarized in Table 4.1.

Table 4.1: Notations used in ADB-PULLUP

Symbol	Description
n_i	A node in a clock tree, which is either a buffer or a sink;
T_{n_i}	The subtree rooted at node n_i;
$arr_{n_i,m}$	Arrival time at sink node n_i at *mode-m*;
$lst_{n_i,m}$	The latest arrival time among the sinks on the subtree rooted at node n_i in *mode-m*;
κ_m	The given clock skew bound to meet in *mode-m*;
$\alpha_{n_i,m}$	Delay value (i.e., increment) of ADB located at node n_i in *mode-m*;
H_{n_i}	Set of child nodes of n_i not to be replaced by ADBs.

Firstly, we demonstrate the procedure of our algorithm for the allocation of ADBs under constant skew bound, called ADB-PULLUP, step-by-step using an example to see how the algorithm works. Then, we describe the flow of the algorithm and the properties of the algorithm.

Let us consider the clock signal arrival times shown in the clock tree in Figure 4.2(a). Let κ_m = 10 for all m. The numbers below the tree represent the clock delay from the clock source to the sinks in each mode. First, ADB-PULLUP initially assumes that each sink has a distinct fictitious ADB at the front of it. The numbers at the bottom of each sink s_i indicate the delay value increments of the ADB on s_i, $\alpha_{s_i,1}$ in *mode-1* and $\alpha_{s_i,2}$ in *mode-2*, for $i = 1, \cdots, 10$. We compute the delay value by

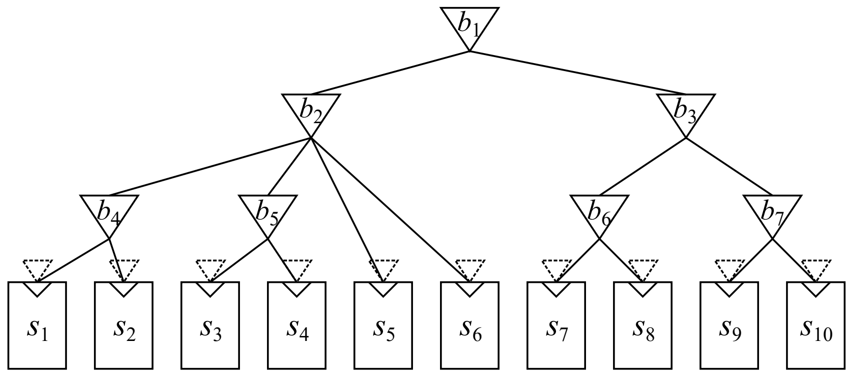
$$\alpha_{s_i,m} = \max\{0,\ lst_{root,m} - \kappa_m - arr_{s_i,m}\} \tag{4.1}$$

where *root* represents the clock source (root) node of the clock tree. Thus, $lst_{root,m}$ is the latest clock arrival time among those of all clock sinks in power mode m. For example, $\alpha_{s_1,1} = \max\{0, 20 - 10 - 7\} = 3$ and $\alpha_{s_1,2} = \max\{0, 20 - 10 - 9\} = 1$. Note that the value by Equation (4.1) for each sink s_i corresponds to the least increase of delay required on the fictitious ADB in s_i to meet the clock skew constraint. Then, ADB-PULLUP performs a bottom-up traversal on the clock tree to move up (i.e., pull up) the ADBs towards the root of the clock tree.

The decision of allocating an ADB at n_k which is a non-sink and whose α value has been assigned is made according to the evaluation result of the inequality:

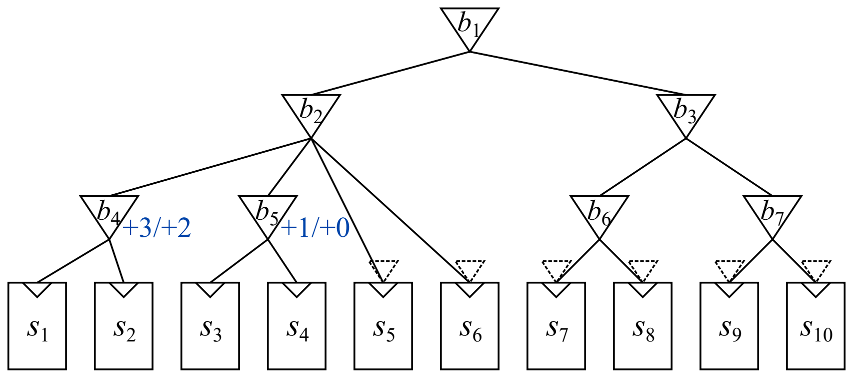
$$\alpha_{n_k,m} > lst_{root,m} - lst_{n_i,m} \tag{4.2}$$

where n_i is the parent node of n_k. If the inequality is true for at least one power mode, an ADB is allocated. For example, let us assume that $\alpha_{b_4,-}$, $\alpha_{b_5,-}$ have been calculated using the same method we are going to explain. The values are shown in Figure 4.2(b). Then, since $\alpha_{b_4,2}(= 2) > lst_{root,2} - lst_{b_2,2}(= 20 - 20 = 0)$, an ADB is inserted to b_4. However, since $\alpha_{b_5,1}(= 1) \leq lst_{root,1} - lst_{b_2,1}(= 20 - 16 = 4)$ and $\alpha_{b_5,2}(= 0) \leq lst_{root,2} - lst_{b_2,2}(= 20 - 20 = 0)$, no ADB is inserted to b_5.



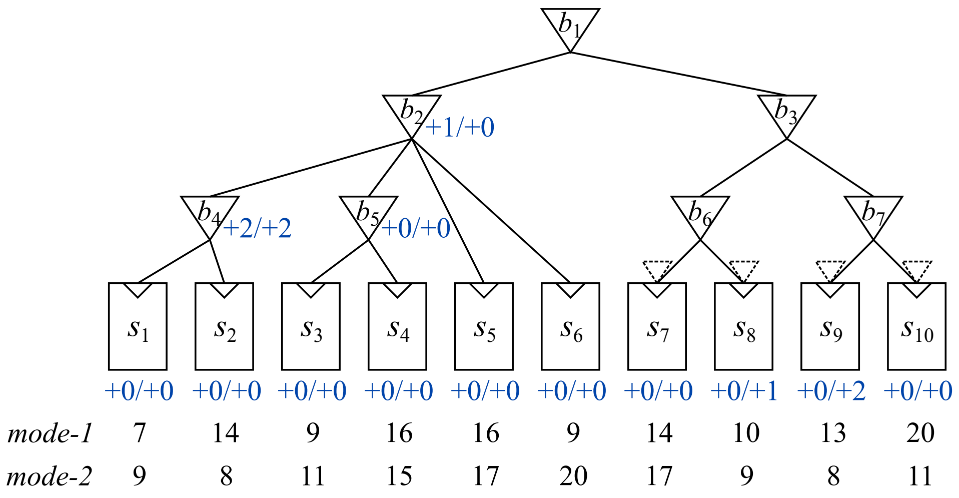
	+3/+1	+0/+2	+1/+0	+0/+0	+0/+0	+1/+0	+0/+0	+0/+1	+0/+2	+0/+0
mode-1	7	14	9	16	16	9	14	10	13	20
mode-2	9	8	11	15	17	20	17	9	8	11

(a) A clock tree T before the ADB insertion by ADB-PULLUP with $\kappa_1 = \kappa_2 = 10$; allocating $\alpha_{n_i,m}$ for each sink n_i and mode m.

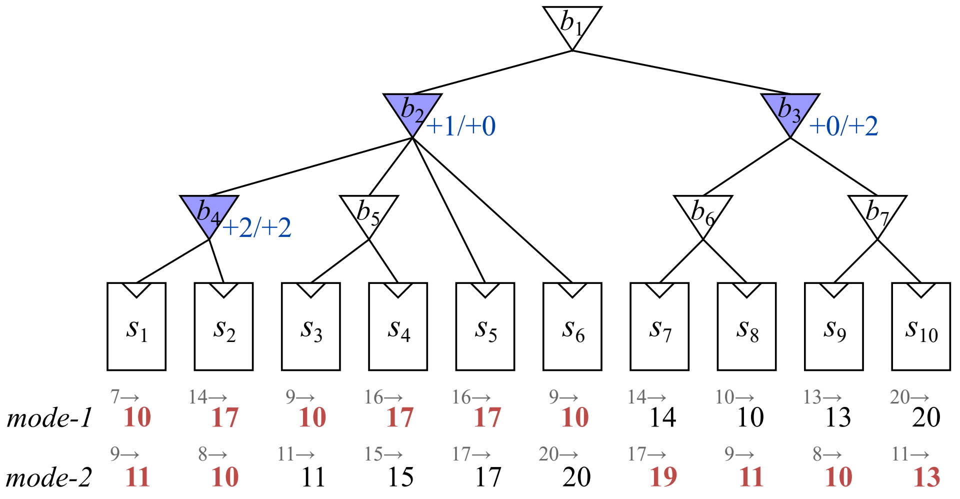


	+0/+0	+0/+0	+0/+0	+0/+0	+0/+0	+1/+0	+0/+0	+0/+1	+0/+2	+0/+0
mode-1	7	14	9	16	16	9	14	10	13	20
mode-2	9	8	11	15	17	20	17	9	8	11

(b) After the process of clock subtrees rooted at b_4 and b_5. (All children n_k of each subtree rooted at n_i satisfy $\alpha_{n_k,m} \leq lst_{root,m} - lst_{n_i,m}$ for all modes. Thus, ADBs are not inserted.)



(c) After the process of clock subtree rooted at b_2. ($\alpha_{b_4,2} > lst_{root,2} - lst_{b_2,2}$, thus, an ADB is inserted at b_4.)


b_1
b_2
+1/+0
b_3
+0/+2
b_4
+2/+2
b_5
b_6
b_7
s_1
s_2
s_3
s_4
s_5
s_6
s_7
s_8
s_9
s_{10}
mode-1 7→10 14→17 9→10 16→17 16→17 9→10 14→14 10→10 13→13 20→20
mode-2 9→11 8→10 11→11 15→15 17→17 20→20 17→19 9→11 8→10 11→13

(d) The complete subtree T after the ADB insertion by ADB-PULLUP.

Figure 4.2: Example showing step-by-step procedure of ADB-PULLUP.

Once the decision of allocating ADBs to all children of n_i is made, the α value of n_i is updated by

$$\alpha_{n_i,m} = \max\{\alpha_{n_k,m} : n_k \in H_{n_i}\} \tag{4.3}$$

where H_{n_i} represents the set of n_i's children on which ADBs are not allocated. If $H_{n_i} = \phi$, then $\alpha_{n_i,m}$ is set to 0 for every mode m. For example, in Figure 4.2(c), b_4 is not in H_{b_2} since $\alpha_{b_4,2} > lst_{root,2} - lst_{b_2,2}$, while α of b_5, s_5, s_6 is smaller than or equal to $lst_{root} - lst_{b_2}$ for all modes. Thus, $\alpha_{b_2,1} = \max\{\alpha_{b_5,1}, \alpha_{s_5,1}, \alpha_{s_6,1}\} = \max\{1, 0, 1\} = 1$ and $\alpha_{b_2,2} = \max\{\alpha_{b_5,2}, \alpha_{s_5,2}, \alpha_{s_6,2}\} = \max\{0, 0, 0\} = 0$ since $H_{b_2} = \{b_5, s_5, s_6\}$.

At this stage, from node n_i where its α values are set, we recursively perform delay-resetting on every descendant, n_k, of n_i by calling function READJUST described in Figure 4.3. READJUST subtracts $\alpha_{n_i,m}$ from the sum of delays on each path from a child of n_i to its descendent sinks, or set to 0 if $\alpha_{n_i,m}$ is bigger than the original sum of delays. For example, Figure 4.2(c) shows the results of delay readjustment when the delay value of b_2 is computed by Equation (4.3). For example, $\alpha_{b_4,1} = 3 - \min\{1, 3\} = 2$ and $\alpha_{b_4,2} = 2 - \min\{0, 2\} = 2$. Subtree T_{b_3} is processed likewise. After all the nodes are processed, ADB-PULLUP reports the result of ADB insertion with the updated arrival times as shown in Figure 4.2(d).

The flow of ADB-PULLUP is depicted in Figure 4.3. In the initialization phase, the $\alpha_{n_i,m}$ value of each sink n_i is assigned to the minimum value by which $arr_{n_i,m} + \alpha_{n_i,m}$ is not shorter than $lst_{root,m} - \kappa_m$. This fixes the skew violations by assuming the allocation of a fictitious ADB to each sink. The next phase is "pulling up" these ADBs to non-sink locations of the clock tree, by performing PULLUP operation in a topological order. Consider a non-sink node n_i to be processed in the flow. Each child, n_k, of n_i, is checked to see if an ADB is needed according to the evaluation of $\alpha_{n_k,m} > lst_{root,m} - lst_{n_i,m}$. If the evaluation is true, an ADB is inserted to n_k, otherwise, the maximum α value (initially 0) to be assigned to n_i is updated if needed. Once the process PULLUP at the bottom loop in Figure 4.3 is done, the α values at the descendants of n_i are recursively re-set according to function READJUST.

The time complexity of ADB-PULLUP is bounded by $O(mn \log n)$ where m is the number of power modes and n is the number of nodes of the input clock tree. Since m is usually very small, the complexity is reduced to $O(n \log n)$. The detailed derivation of the time complexity of ADB-PULLUP is the following: $O(n)$ time is taken to sort nodes in topological order and $O(mn)$ time to compute the lst values of all nodes. Likewise, $O(mn)$ time is taken to assign the α values for all leaf nodes and $O(mn)$ to the determination of ADB placement to nodes. Finally, total time of $O(mn \log n)$ is taken for READJUST function, which is the most dominant in the steps of ADB-PULLUP. It is because each node is visited by READJUST at most the number of their ancestors. All properties and theorems of ADB-PULLUP are summarized in Section 4.4.

4.3 Extensions

4.3.1 Supporting Discrete ADB Delay

By slightly updating the computation of α values in ADB-PULLUP, it is possible to support the ADB allocation with ADBs of discrete delay increments. (We call our updated ADB algorithm ADB-PULLUP-Q.)

At the stage of the decision made by Equation (4.2), $\alpha_{n_k,m}$ is replaced by the closest ADB delay which is larger than or equal to the original value. The quantized delay is used as $\alpha_{n_i,m}$ in Equation (4.3), and READJUST subtracts the new $\alpha_{n_i,m}$ value from the α of its descendent ADBs. The main difference of READJUST function in ADB-PULLUP and ADB-PULLUP-Q is that ADB-PULLUP-Q updates the α to the quantized value with the minimum increase from the original value at the same time of the subtraction. At the final stage of READJUST, the minimum delay increment occurs from the quantization is subtracted from $\alpha_{n_i,m}$, to balance the total clock delays.

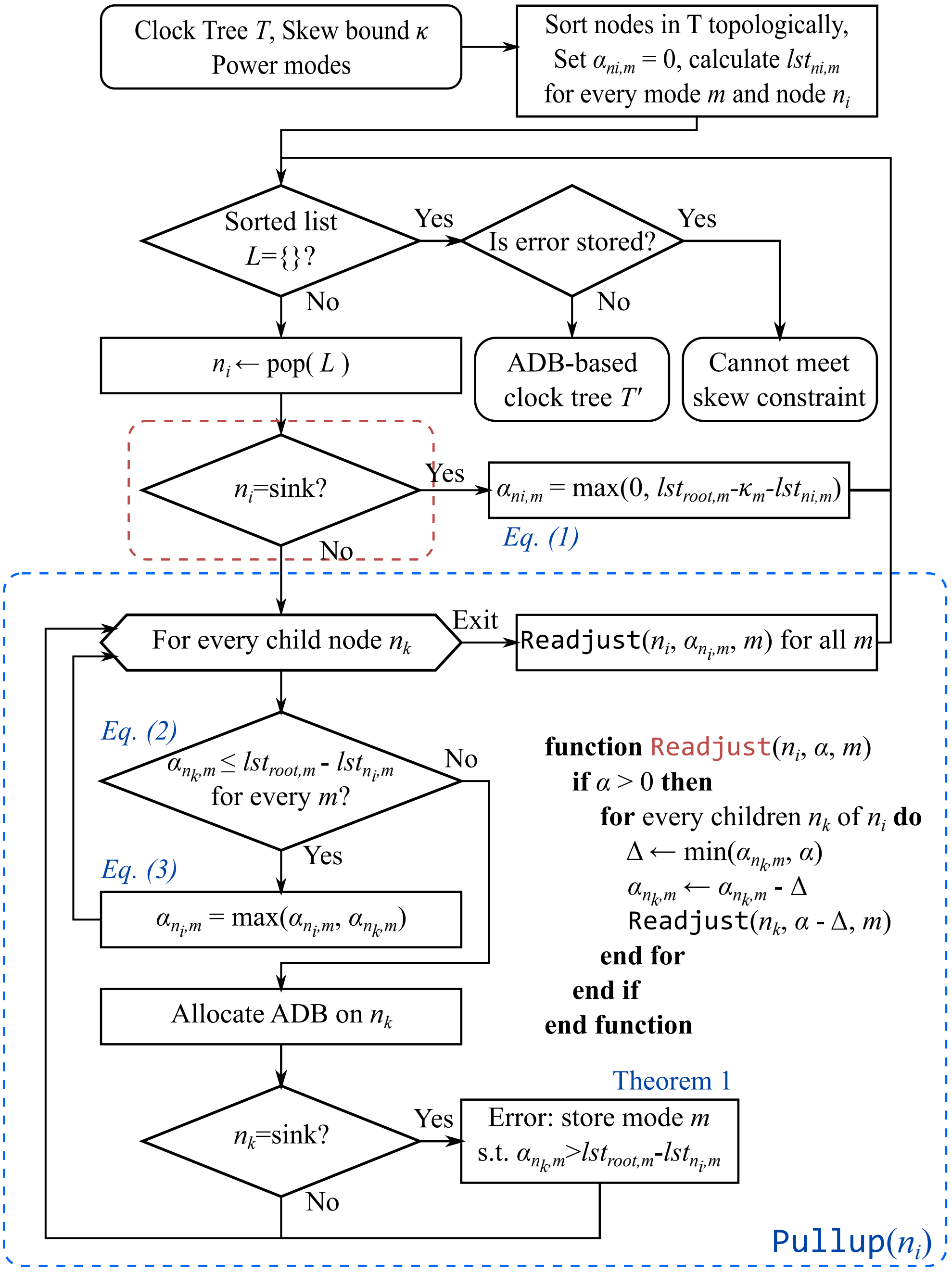

Clock Tree T, Skew bound κ
Power modes
Sort nodes in T topologically,
Set $\alpha_{ni,m} = 0$, calculate $lst_{ni,m}$
for every mode m and node n_i
Sorted list L={}?
Yes
No
Is error stored?
Yes
No
$n_i \leftarrow$ pop(L)
ADB-based clock tree T'
Cannot meet skew constraint
n_i=sink?
Yes
No
$\alpha_{ni,m} = \max(0, lst_{root,m}\text{-}\kappa_m\text{-}lst_{ni,m})$
Eq. (1)
For every child node n_k
Exit
Readjust(n_i, $\alpha_{n_i,m}$, m) for all m
Eq. (2)
$\alpha_{n_k,m} \leq lst_{root,m}$ - $lst_{n_i,m}$
for every m?
No
Yes
Eq. (3)
$\alpha_{n_i,m} = \max(\alpha_{n_i,m}, \alpha_{n_k,m})$
Allocate ADB on n_k
n_k=sink?
Yes
No
Theorem 1
Error: store mode m
s.t. $\alpha_{n_k,m} > lst_{root,m}\text{-}lst_{n_i,m}$
function Readjust(n_i, α, m)
if $\alpha > 0$ then
for every children n_k of n_i do
$\Delta \leftarrow \min(\alpha_{n_k,m}, \alpha)$
$\alpha_{n_k,m} \leftarrow \alpha_{n_k,m}$ - Δ
Readjust(n_k, α - Δ, m)
end for
end if
end function
Pullup(n_i)

Figure 4.3: The flow of ADB-PULLUP.

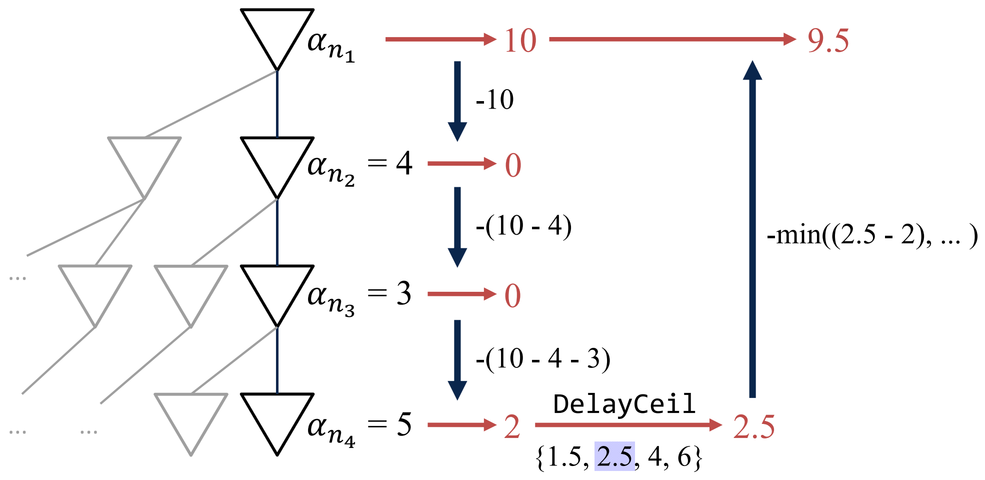


Figure 4.4: An example of executing modified READJUST function to handle discrete ADB delays. After the execution of original READJUST, additional procedure, DELAYCEIL(), is executed to select the delay among the available values.

4.3.2 Integration of Buffer Sizing

We can think of buffer sizing as an ADB allocation imposed by the restriction that the α values in power modes are *pre-defined*. For example, when a buffer b_i in the input clock tree is going to be replaced by a buffer buf_j in the buffer library $\mathcal{L}$ (rather than an ADB), the delay number in each power mode may be increased or decreased, but the number is fixed, which means uncontrollable, unlike ADB. Let $\beta^j_{n_i,m}$ be the delay increase or delay decrease in power mode m caused by the replacement of buffer b_i in the input clock tree by $buf_j \in \mathcal{L}$. We can compute all β values from the input clock tree and $\mathcal{L}$. Now, we want to substitute the minimal ADBs determined by ADB-PULLUP (or ADB-PULLUP-Q) with as many buffers in $\mathcal{L}$ as possible to further reduce the number of ADBs to be inserted in the clock tree while still meeting the clock skew constraint for every power mode. Since we have all the β and α values in every node of the clock tree in all power modes, a naive solution is to generate all the combinations of buffer sizing as well as ADB insertion for all nodes, and choose the one that uses the least number of ADBs while meeting the clock skew constraint. However, its computation time grows exponentially as the problem size increases. To be practically feasible, we propose a simple but effective iterative method:

1. For each node n_i in the clock tree, in which ADB-PULLUP (or ADB-PULLUP-Q) has decided that an ADB should be inserted in the node, for each buffer $buf_j \in \mathcal{L}$, we compute

$$\delta_{n_i}^{buf_j} = \sum_{m=1}^{K} (\alpha_{n_i,m} - \beta_{n_i,m}^{buf_j})^2 \tag{4.4}$$

   where K is the number of modes. For example, if $(\alpha_{n_1,1}, \alpha_{n_1,2}) = (+3, +1)$, $(\beta_{n_1,1}^{buf_1}, \beta_{n_1,2}^{buf_1}) = (+3, +2)$, $(\beta_{n_1,1}^{buf_2}, \beta_{n_1,2}^{buf_2}) = (+1, -1)$, then, $\delta_{n_1}^{buf_1} = (3 - 3)^2 + (1 - 2)^2 = 1$ and $\delta_{n_1}^{buf_2} = (3 - 1)^2 + (1 - (-1))^2 = 8$.

2. Select the pair of node and buffer sizing such that the corresponding δ value is minimal and it satisfies the clock skew and latency constraints. The buffer in the selected node is then resized accordingly. For the previous example, selecting buf_1 is preferred to that of buf_2 for resizing in node n_1 since $\delta_{n_1}^{buf_1} < \delta_{n_1}^{buf_2}$.

3. Update the arrival times at clock sinks according to the buffer resizing performed in Step 2, and iterate the procedure. The iteration stops when there is no pair that satisfies the skew and latency constraints or the resizing causes the number of ADBs to increase.

The rationale behind the use of δ is that as the smaller the value of δ in a node is, the more the corresponding buffer sizing is likely to close to the ADB that has been inserted to the node, thus, the buffer sizing taking over the role of the ADB with a minimal impact on the overall timing of the clock tree. We call the ADB allocation algorithm combined with buffer sizing ADB-PULLUP-B for the continuous delay of ADB and ADB-PULLUP-QB for the discrete delay of ADB.

4.4 Optimality Proofs of the Proposed Algorithm

This section provides proofs of the useful properties of the algorithm that (1) it always gets the answer unless the answer does not exist, and (2) it allocates the minimum number of ADBs to satisfy the constraint.

Property 1. *The arrival times at sinks produced by* ADB-PULLUP *never exceed* $lst_{root,m}$ *for every mode* m.

Proof. For simple notations, we drop the power mode symbol m in the presentation of the proofs if it is obvious.

For a power mode, let $L^{n_l}_{n_k \to s_j}$ be the sum of the α values of the nodes on the path from n_k (inclusive) to a sink s_j which is on the subtree rooted at n_k after READJUST is applied to n_l and $\alpha^{pre}_{n_k}$ be the α value of n_k before the application of READJUST to its parent. ($L^{s_j}_{s_j \to s_j} = \alpha^{pre}_{s_j}$ since READJUST is not applicable to sinks.)

We claim that the following inequality is hold:

$$L^{n_i}_{n_i \to s_j} + arr_{s_j} \leq lst_{root}. \tag{4.5}$$

We use induction in terms of the height, h, of the subtree rooted at n_i.

i. $h = 1$ corresponds to the case where n_i is a sink, which means s_j and n_i in Equation (4.5) are identical. Thus, $L^{n_i}_{n_i \to s_j} + arr_{s_j} = \alpha^{pre}_{s_j} + arr_{s_j}$. By Equation (4.1), $\alpha^{pre}_{s_j} + arr_{s_j} \leq lst_{root}$.

ii. For the induction step, we assume that the hypothesis holds for all $h \leq H$, and consider a node n_i with height $h = H + 1$. Then, all heights of its children n_k is less than or equal to H. By the induction hypothesis, for every n_k of n_i, it is true that

$$L^{n_k}_{n_k \to s_j} + arr_{s_j} \leq lst_{root} \tag{4.6}$$

where s_j is a sink in the subtree rooted at n_k.

Case 1. $L^{n_k}_{n_k \to s_j} \geq \alpha^{pre}_{n_i}$:

After the application of READJUST to n_i, $L^{n_i}_{n_k \to s_j}$ = $L^{n_k}_{n_k \to s_j} - \alpha^{pre}_{n_i}$. Thus, $L^{n_i}_{n_i \to s_j} + arr_{s_j} = L^{n_i}_{n_k \to s_j} + \alpha^{pre}_{n_i} + arr_{s_j} = L^{n_k}_{n_k \to s_j} - \alpha^{pre}_{n_i} + \alpha^{pre}_{n_i} + arr_{s_j}$ $= L^{n_k}_{n_k \to s_j} + arr_{s_j} \leq lst_{root}$ by Equation (4.6).

Case 2. $L^{n_k}_{n_k \to s_j} < \alpha^{pre}_{n_i}$:

By Equation (4.2), after the application of READJUST to n_i,

$$L^{n_i}_{n_i \to s_j} = \alpha^{pre}_{n_i}, \tag{4.7}$$

and according to Equation (4.1) and the definition of lst_{n_i},

$$\alpha^{pre}_{n_i} \leq lst_{root} - lst_{n_i} \leq lst_{root} - arr_{s_j}. \tag{4.8}$$

Therefore, $L^{n_i}_{n_i \to s_j} + arr_{s_j}$ = $\alpha^{pre}_{n_i} + arr_{s_j} \leq lst_{root}$.

From Cases 1 and 2, Equation (4.6) holds for n_i with $h = H + 1$ if it holds for any node with $h \leq H$. Thus, from the induction, hypothesis $L^{n_i}_{n_i \to s_j} + arr_{s_j} \leq lst_{root}$ holds for every node in the tree. Then, because Property 1 holds for *root* as well, $L^{root}_{root \to s_j} + arr_{s_j}$, the clock arrival times at sink s_i after the whole execution of ADB-PULLUP, do not exceed lst_{root}.

□

In some cases, clock trees do not have any solution of ADB allocation. For example, consider a simple clock tree shown in Figure 4.5 with clock skew bound $\kappa = 10$. The clock arrival times at sink s_2 $(= 13)$ and sink s_3 $(= 2)$ cause the clock skew violation. However, it is not possible to resolve the skew violation in the figure whatever ADB allocations are attempted to A, B or both. We formally classify the input clock trees into ADB-solvable or ADB-unsolvable as follows:

Definition 1. It is said that a clock tree T with κ is **ADB-unsolvable** if there is a node $n_i \in T$ such that $lst_{n_i} - arr^{min}_{S(n_i)} > \kappa$ in which $S(n_i)$ is the set of sinks which are *directly* connected to n_i, $arr^{min}_{S(n_i)}$ is the minimum among the arrival times of sinks in $S(n_i)$. (It is ∞ if $S(n_i) = \phi$.) For the clock trees in which do not have any node $n_i \in T$ such that $lst_{n_i} - arr^{min}_{S(n_i)} > \kappa$, they are said to be **ADB-solvable**.

For example, the clock tree in Figure 4.5 is said to be ADB-unsolvable because $S(A) = \{s_3, s_4\}$ and $lst_A - arr^{min}_{S(A)} = 13 - 2 = 11 > \kappa(= 10)$. It can be easily seen that even allocating ADBs on every inner nodes cannot solve the problem.

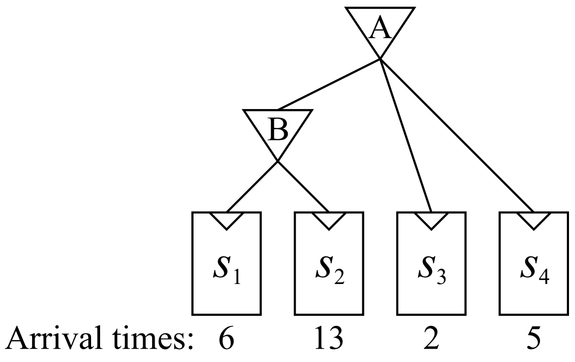


Figure 4.5: An example of clock tree that belongs to *ADB-unsolvable* when clock skew bound κ = 10.

Theorem 1. ADB-PULLUP *allocates ADBs on sinks if and only if the input clock tree is ADB-unsolvable.*

Proof. ($\Rightarrow$) If an ADB is allocated at sink s_k, which is a child of n_i,

$$\alpha^{pre}_{s_k} > lst_{root} - lst_{n_i} \tag{4.9}$$

for some mode m by Equation (4.2).

Since $\alpha^{pre}_{s_k} > lst_{root} - lst_{n_i} \geq 0$, Equation (4.1) implies

$$\alpha^{pre}_{s_k} = lst_{root} - arr_{s_k} - \kappa. \tag{4.10}$$

Clearly,

$$arr^{min}_{S(n_i)} \leq arr_{s_k}. \tag{4.11}$$

By Equation (4.11), $lst_{n_i} - arr^{min}_{S(n_i)} \geq lst_{n_i} - arr_{s_k}$ and by Equation (4.10), $lst_{n_i} - arr_{s_k} = lst_{n_i} + \alpha^{pre}_{s_k} - lst_{root} + \kappa$, which is greater than κ by Equation (4.9). Thus, $lst_{n_i} - arr^{min}_{S(n_i)} > \kappa$.

($\Leftarrow$) Since the clock tree is ADB-unsolvable, there is n_i such that $lst_{n_i} - arr^{min}_{S(n_i)} > \kappa$. Moreover, since $arr^{min}_{S(n_i)} < \infty$, $S(n_i) \neq \phi$. Let $s_k \in S(n_i)$ such that $arr^{min}_{S(n_i)} = arr_{s_k}$. Then, $lst_{n_i} - arr_{s_k} = lst_{n_i} - arr^{min}_{S(n_i)} > \kappa$.

Thus, by Equation (4.1), $\alpha^{pre}_{s_k} \geq lst_{root} - arr_{s_k} - \kappa > lst_{root} - lst_{n_i}$, which enables the allocation of ADB at s_k according to Equation (4.2).

□

Note that Property 1, which is a feature that enables to keep the total size of capacitor banks in ADBs within a certain limit, does not hold for the other ADB allocation algorithms proposed in previous works. In addition, Theorem 1 indicates that if there is at least one solution, ADB-PULLUP will always find an ADB allocation solution such that the α values of all sinks are 0.

To facilitate the proof of the optimality of our proposed algorithm, we define terms ADB-free-path and *est-dir*$_{i,m}$, and provide one lemma.

Definition 2. If the path from node n (exclusive) to a sink r in a clock tree does not contain ADBs, the path is called ADB-free-path and the sink is said to has ADB-free-path from n.

Although previous works ([1,21]) have used similar definitions, ours are stricter in that if an ADB is allocated because of some modes but $\alpha_{\cdot,m} = 0$ for the other modes, it is counted as ADB only in modes with $\alpha_{\cdot,m} > 0$. A good example is the clock tree shown in Figure 3.1. In *mode-2*, the ADB has $\alpha = 0$ and this is not counted as ADB in less strict version of ADB-free-path.

Definition 3. *est-dir*$_{n_i,m}$ represents the earliest arrival time among those at the sinks which have ADB-free-path from n_i in power mode m. *est-dir*$_{n_i,m} = \infty$ if such sink does not exist.

Lemma 1. *During the process of* ADB-PULLUP*, if* $\alpha^{pre}_{n_i,m} > 0$ *for a power mode* m*, there is a sink in subtree* T_{n_i} *that the arrival time at the sink is exactly* $lst_{root,m} - \kappa_m - \alpha^{pre}_{n_i,m}$ *and the sink has ADB-free-path from* n_i*.*

Proof. Let h denote the height of T_{n_i}. If $h = 1$, n_i only has sinks as its children, Thus, the lemma holds. Let us assume that the lemma is true for $h \leq H$. We now want to show that the lemma is true for $h = H + 1$. By Equation (4.3), if $\alpha^{pre}_{n_i,m} > 0$ for any n_i with its height of $H + 1$, there exists a child node n_{k_j} of H_{n_i} such that $\alpha^{pre}_{n_{k_j},m} = \alpha^{pre}_{n_i,m}(> 0)$. Since $T_{n_{k_j}}$ has a sink whose arrival time is $lst_{root,m} - \kappa_m - \alpha^{pre}_{n_{k_j},m}$ on ADB-free-path from n_i, T_{n_i} also has a sink on ADB-free-path passing through the child node n_{k_j} and its clock arrival time is $lst_{root,m} - \kappa_m - \alpha^{pre}_{n_i,m}$. □

Theorem 2. *After the application of* ADB-PULLUP *to* n_i *in clock tree* T*, the resulting subtree* T_{n_i} *has been allocated with a minimum number of ADBs while meeting the clock skew constraint for* T_{n_i}*.*

Proof. The proof of the optimality of ADB-PULLUP involves "cut-and-paste" argument. Let $N(T_{n_i})$ denote the number of ADBs in subtree T_{n_i} except the root n_i. Let $X_{n_i} = 1$ if node n_i has an ADB, and $X_{n_i} = 0$, otherwise.

We want to show that $N(T_{n_i})$ is the smallest number among those of all feasible ADB allocations on the subtree rooted at n_i, and it has the largest value of *est-dir*$_{n_i,m}$ for every power mode m among those of all feasible ADB allocations with the minimum number. For example, if $(N(T_{n_i}), \textit{est-dir}_{n_i,m}) = (4, 10)$, other feasible solutions could be $(5, 12)$, $(5, 8)$, $(4, 11)$, and $(4, 10)$, but will not be $(3, 12)$ or $(4, 9)$.

Let h be the height of T_{n_i}.

i. When $h = 1$, all children of n_i are sinks. Thus, $N(T_{n_i}) = 0$, which is trivially solvable, and *est-dir*$_{n_i,m} = \infty$ for every power mode.

ii. Let us assume this theorem holds for $h \leq H$. If the theorem is not true for $h = H + 1$, there is a subtree T'_{n_i} produced by an ADB allocation such that

(1) $N(T'_{n_i}) < N(T_{n_i})$, or

(2) $N(T'_{n_i}) = N(T_{n_i})$ and *est-dir*$'_{n_i,m}$ $>$ *est-dir*$_{n_i,m}$ for some mode m.

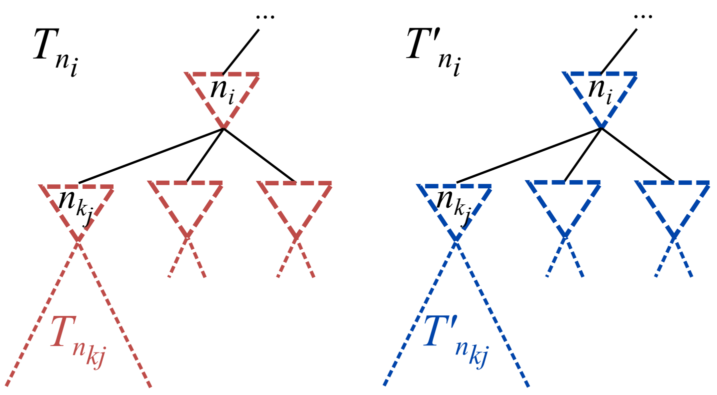
We want to prove that T'_{n_i}, which meets the above condition, does not exist, following the order illustrated in Figure 4.6. In Figure 4.6(a), the clock trees T_{n_i} and T'_{n_i} with ADBs are given. We will generate T''_{n_i} as shown in Figure 4.6(b) by replacing one child subtree $T'_{n_{k_j}}$ of T'_{n_i} corresponding to subtree of T_{n_i}. Also, because T'_{n_i} is the better solution, we can select at least one subtree $T'_{n_{k_j}}$ with less number of ADBs or larger *est-dir*. For all cases, we will show that **(*)** $N(T''_{n_i}) \leq N(T'_{n_i})$, and *est-dir*$''_{n_i,m}$ $\geq$ *est-dir*$'_{n_i,m}$ for every mode m if $N(T''_{n_i}) = N(T'_{n_i})$.

As shown in Figure 4.6(d), we can replace every child subtree $T'_{n_{k_j}}$ with $T_{n_{k_j}}$, including the node n_{k_j}. Let $T^f_{n_i}$ denote the ADB allocation tree produced by the process of replacement. Clearly, $T^f_{n_i}$ satisfies $N(T^f_{n_i}) \leq N(T'_{n_i})$, *est-dir*$^f_{n_i,m}$ $\geq$ *est-dir*$'_{n_i,m}$ for every mode m with the same number of ADBs. However, $T^f_{n_i}$ has the same values of N and *est-dir* as those of T_{n_i}, contradicting the assumption that $N(T'_{n_i}) < N(T_{n_i})$, or *est-dir*$'_{n_i,m}$ $>$ *est-dir*$_{n_i,m}$ for some power mode m if $N(T'_{n_i}) = N(T_{n_i})$.

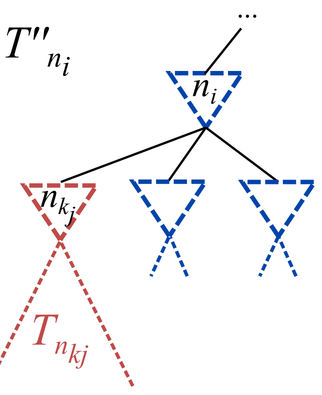
Now, we prove **(*)**. We use the fact that the theorem holds for $h \leq H$, (1) $N(T_{n_{k_j}}) \leq N(T'_{n_{k_j}})$ and (2) *est-dir*$_{n_{k_j},m}$ $\geq$ *est-dir*$'_{n_{k_j},m}$ for every power mode if $N(T_{n_{k_j}}) = N(T'_{n_{k_j}})$. In the proof, X_n denotes whether an ADB is allocated on node n. $X_n = 1$ means the node n has an ADB, and $X_n = 0$ means it does not.

Case 1. $N(T_{n_{k_j}}) < N(T'_{n_{k_j}})$:

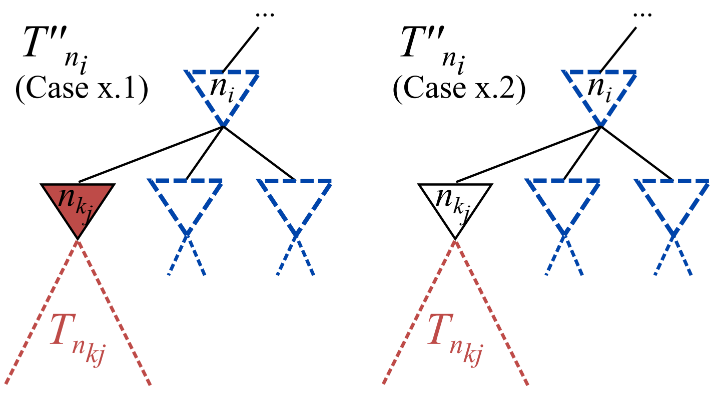
Case 1.1. n_{k_j} $(= n''_{k_j})$ has an ADB:



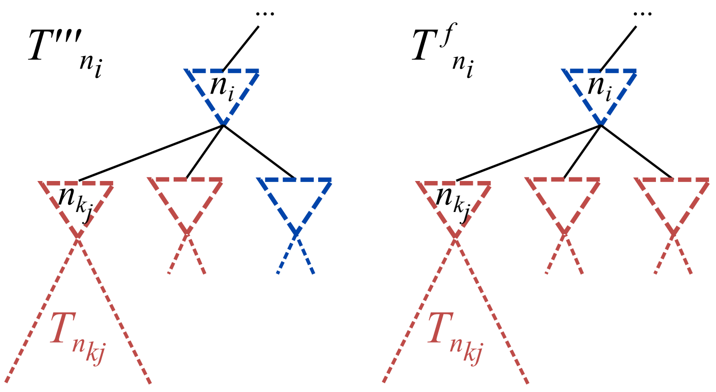
(a) Assume that (1) $N(T'_{n_i}) < N(T_{n_i})$, or (2) $N(T'_{n_i}) = N(T_{n_i})$ and *est-dir*$'_{n_i,m} >$ *est-dir*$_{n_i,m}$ for some mode m.



(b) Subtree $T'_{n_{k_j}}$ in T'_{n_i} is replaced by $T_{n_{k_j}}$ of T_{n_i}. The new tree is called T''_{n_i}.



(c) For Case 1 and 2, there are 2 subcases (Case x.1, Case x.2) in which n_{k_j} has an ADB or not. In all cases, (1) $N(T''_{n_i}) < N(T'_{n_i})$ or (2) $N(T''_{n_i}) = N(T'_{n_i})$ and *est-dir*$''_{n_i,m} \geq$ *est-dir*$'_{n_i,m}$ for every mode m.



(d) By replacing every subtree rooted on child node, we get $T^f_{n_i}$. Because $N(T^f_{n_i})$ and *est-dir*$_{n_i}$ are the same with those of T_{n_i}, it contradicts the assumption in (a).

Figure 4.6: The overall sketch of our optimality proof.

$N(T''_{n_i}) = N(T'_{n_i}) - (N(T'_{n_{k_j}}) + X'_{n_{k_j}}) + (N(T_{n_{k_j}}) + X_{n_{k_j}}) = N(T'_{n_i}) - (N(T'_{n_{k_j}}) - N(T_{n_{k_j}}) - (X'_{n_{k_j}} - X_{n_{k_j}}) \leq N(T'_{n_i}) - 1 - (X'_{n_{k_j}} - X_{n_{k_j}}) \leq N(T'_{n_i})$. Thus, $N(T''_{n_i}) \leq N(T'_{n_i})$. Also, since all sinks with ADB-free-path in T''_{n_i} are also in T'_{n_i}, *est-dir*$''_{n_i,m} \geq$ *est-dir*$'_{n_i,m}$ for every m.

Case 1.2. n_{k_j} does not have an ADB:

Since $X_{n_{k_j}} = 0$, $N(T''_{n_i}) = N(T'_{n_i}) - 1 - (X'_{n_{k_j}} - X_{n_{k_j}}) < N(T'_{n_i})$.

Case 2. $N(T_{n_{k_j}}) = N(T'_{n_{k_j}})$ and *est-dir*$_{n_{k_j},m} \geq$ *est-dir*$'_{n_{k_j},m}$ for every m:

Case 2.1. n_{k_j} has an ADB:

The ADB on n_{k_j} is allocated because $\alpha^{pre}_{n_{k_j},m} > 0$ for some power mode m, by Lemma 1, $lst_{n_i,m} -$ *est-dir*$'_{n_{k_j},m} \geq lst_{n_i,m} -$ *est-dir*$_{n_{k_j},m} = lst_{n_i,m} - (lst_{root,m} - \kappa_m - \alpha^{pre}_{n_{k_j},m}) > \kappa_m$. Also, the *est-dir* of n'_{k_j} if less than or equal to the one of n_{k_j}, so that n'_{k_j} should have an additional ADB. In addition, since the sinks with ADB-free-path are the same for T'_{n_i} and T''_{n_i}, $N(T''_{n_i}) = N(T'_{n_i}) - (N(T'_{n_{k_j}}) + X'_{n_{k_j}}) + (N(T_{n_{k_j}}) + X_{n_{k_j}}) = N(T'_{n_i})$ and *est-dir*$''_{n_i,m} =$ *est-dir*$'_{n_i,m}$ for every m.

Case 2.2. n_{k_j} does not have an ADB:

As seen from *Case* 2.1, n'_{k_j} has an ADB on it. $N(T''_{n_i}) = N(T'_{n_i}) - (N(T'_{n_{k_j}}) + X'_{n_{k_j}}) + (N(T_{n_{k_j}}) + X_{n_{k_j}}) = N(T'_{n_i}) - N(T'_{n_{k_j}}) + N(T_{n_{k_j}}) \leq N(T'_{n_i})$. Since T''_{n_i} and T'_{n_i} have the same sinks with ADB-free-path except the sinks in their subtrees rooted at n_{k_j}, *est-dir*$''_{n_{k_j},m} \geq$ *est-dir*$'_{n_{k_j},m}$ for every m. If n'_{k_j} has an ADB, $N(T''_{n_i}) < N(T'_{n_i})$.

□

Theorem 2 claims that T_{root} produced by the application of ADB-PULLUP uses the minimal number of ADBs while meeting the clock skew constraint.

Table 4.2: Benchmark circuits used in the experiment

Benchmark Circuit	#.FFs	#.Buffers	Original Skew (ps)	Latency (ps)
s35932	1728	97	264.1	545.1
s38417	1564	89	387.1	612.1
s38584	1168	66	299.8	552.8
B17	1312	89	287.7	654.7
B18	2752	173	405.1	825.1
B22	583	42	354.2	690.2
F31	273	345	268.8	1268.5
F34	157	218	211.2	1137.5

4.5 Experimental Results

The proposed algorithm ADB-PULLUP (continuous delay), ADB-PULLUP-Q (discrete delay), ADB-PULLUP-B (combining buffer sizing) and ADB-PULLUP-QB (combining buffer sizing with discrete delay) have been implemented in Python 3 language on a Linux machine with 8 cores of 3.50 GHz Intel i7 CPU and 16 GB memory. Table 4.2 shows the tested benchmark circuits used in the experiment. s35932, s38417 and s38584 are from ISCAS'89 benchmarks, B17, B18 and B17 are from ITC'99 benchmarks and F31 and F34 are from ISPD'09 benchmarks. ISCAS'95 and ITC'99 benchmarks were synthesized with *Synopsys IC Compiler* with 45 nm *Nangate Open Cell Library* [30]. ISPD'09 benchmarks were synthesized using the algorithm in [31]. Each benchmark was partitioned into 6 to 10 power domains which are able to operate in two different supply voltage levels, 0.95 V and 1.1 V. Each column represents the number of flip-flop, the number of clock buffers, the worst clock skew, and the worst clock latency in the four power modes of the input clock trees before the ADB allocation.

Table 4.3: Comparison of results produced by ADB-ESYNC [1], ADB-PULLUP and ADB-PULLUP-B

Circuit Name	Skew Bound (ps)	ADB-ESYNC [1]		ADB-PULLUP		ADB-PULLUP-B	
		#.ADBs	Area[a]	#.ADBs	Area	#.ADBs	Area
s35932	30	27	1180.2	25	1092.7	20	928.3
	40	25	1092.7	23	1005.3	19	871.2
	50	25	1092.7	23	1005.3	19	871.2
s38417	30	31	1355.0	27	1180.2	22	1076.0
	40	28	1223.9	25	1092.7	20	985.3
	50	26	1136.5	23	1005.3	18	960.9
s38584	30	22	961.6	20	874.2	13	686.9
	40	18	786.8	16	699.4	11	583.7
	50	18	786.8	16	699.4	11	580.7
B17	30	29	1267.6	25	1092.7	19	953.0
	40	26	1136.5	22	961.6	15	799.7
	50	26	1136.5	22	961.6	15	786.5
B18	30	150	6556.5	120	5245.2	105	5040.6
	40	147	6425.4	118	5157.8	99	4916.6
	50	144	6294.2	118	5157.8	94	5012.9
B22	30	32	1398.7	24	1049.0	21	986.8
	40	32	1398.7	24	1049.0	21	976.6
	50	31	1355.0	24	1049.0	21	971.5
F31	30	13	568.2	13	568.2	11	487.4
	40	13	568.2	13	568.2	7	325.8
	50	7	306.0	7	306.0	7	306.0
F34	30	30	1311.3	24	1049.0	21	965.4
	40	30	1311.3	24	1049.0	21	965.4
	50	30	1311.3	24	1049.0	18	844.2
Average (%)		100	100	86.16	86.16	67.73	75.29

[a]The columns indicated by "Area" represent the sum of the areas of ADBs, fixed ADBs and resized buffers in μm^2.

Table 4.4: Comparison of results produced by ADB-ESYNC-Q [1], ADB-PULLUP-Q and ADB-PULLUP-QB

Circuit Name	Skew Bound (ps)	ADB-ESYNC-Q [1]		ADB-PULLUP-Q		ADB-PULLUP-QB	
		#.ADBs	Area	#.ADBs	Area	#.ADBs	Area
s35932	30	42	1835.8	29	1267.6	25	1143.6
	40	26	1136.5	24	1049.0	19	879.5
	50	25	1092.7	23	1005.3	19	871.2
s38417	30	36	1573.6	28	1223.9	23	1116.4
	40	31	1355.0	27	1180.2	20	991.9
	50	29	1267.6	25	1092.7	18	967.5
s38584	30	22	961.6	21	917.9	17	810.2
	40	21	917.9	20	874.2	16	723.5
	50	18	786.8	16	699.4	11	583.7
B17	30	35	1529.8	29	1267.6	24	1155.0
	40	30	1311.3	24	1049.0	16	828.5
	50	26	1136.5	22	961.6	15	794.7
B18	30	155	6775.0	122	5332.6	110	5125.0
	40	153	6687.6	119	5201.5	101	4937.7
	50	149	6512.8	118	5157.8	100	4942.8
B22	30	33	1442.4	24	1049.0	19	985.2
	40	32	1398.7	24	1049.0	21	981.7
	50	32	1398.7	24	1049.0	21	976.6
F31	30	13	568.2	13	568.2	12	527.8
	40	13	568.2	13	568.2	11	487.4
	50	7	306.0	7	306.0	7	306.0
F34	30	30	1311.3	24	1049.0	18	854.4
	40	30	1311.3	24	1049.0	21	965.4
	50	30	1311.3	24	1049.0	19	898.9
Average (%)		106.4	106.4	89.53	89.53	72.63	79.57

Table 4.3 summarizes the results produced by applying the algorithms which assume the continous delay values of ADBs, ADB-ESYNC [1], ADB-PULLUP and ADB-PULLUP-B. The experiments were done under the clock skew bound of 30, 40 and 50 ps, respectively, for all power modes. Table 4.4 shows the results of applying the algorithms ADB-ESYNC-Q [1], ADB-PULLUP-Q and ADB-PULLUP-QB which allocates ADBs having uniformly quantized delays with granularity of 10 ps. Though the experiments used the same values for the easy comparison, clock skew bounds of each power mode does not need to be the same. Also, ADB delay candidates do not required to have uniform intervals, neither the same between different positions or power modes. In ADB-PULLUP-B and ADB-PULLUP-QB, some of the ADBs were replaced with buffers and *fixed ADBs*, which are ADBs that have a fixed delay because its control logic is removed. It is effectively a large buffer, covering large additional delay that standard buffers in the library can not support. The columns denoted as "Area" are the area occupied by ADBs, fixed ADBs and resized buffers. By having reduced number of ADBs, the area overhead of ADB control logic which is proportional to the number of ADBs have decreased, resulting in the reduction of the total area. The last rows of both tables show the relative values compared to ADB-ESYNC.

It is observed that ADB-PULLUP uses consistently less number of ADBs compared to the previous work, since ADB-PULLUP considers multiple power modes simultaneously during optimization. Also, the results shown in Table 4.4 indicates that ADB-PULLUP-Q uses considerably less ADBs than ADB-ESYNC-Q. This is because ADB-ESYNC-Q relies on re-iteration with tighter skew bound when clock skew violation occurs after delay quantization while ADB-PULLUP-Q can use quantized delay directly during its bottom-up phase.

In addition, ADB-PULLUP-B and ADB-PULLUP-QB further reduce the number of ADBs over that of ADB-PULLUP and ADB-PULLUP-Q. We performed an experiment to check the effectiveness of the proposed algorithm combined with buffer sizing. Figure 4.7 shows the results on *ISCAS'89 s382* benchmark circuit, which was synthe-

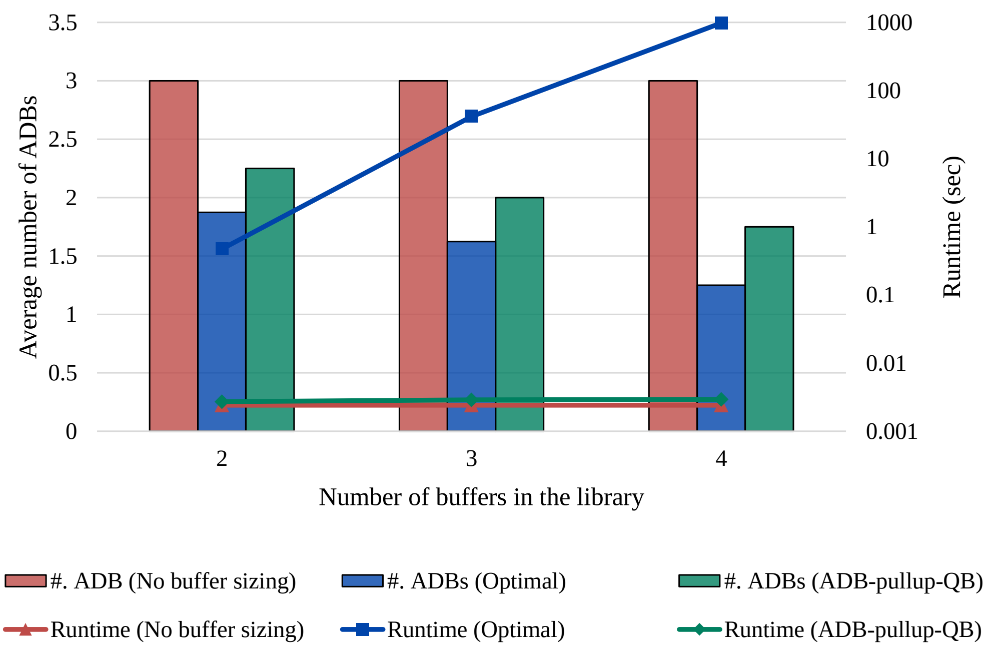


Figure 4.7: Comparison of the numbers of ADBs allocated (bar) and runtime (line) of ADB-PULLUP-Q (red), optimal exhaustive algorithm (blue), and ADB-PULLUP-QB (green). The runtime is in log-scale.

sized to have 10 clock tree buffers. It is shown that the proposed algorithm clearly uses much fewer number of ADBs over the original result, but uses a little more ADBs than that of the optimal allocation with buffer sizing. However, our runtime is very small compared to that of the exhaustive algorithm.

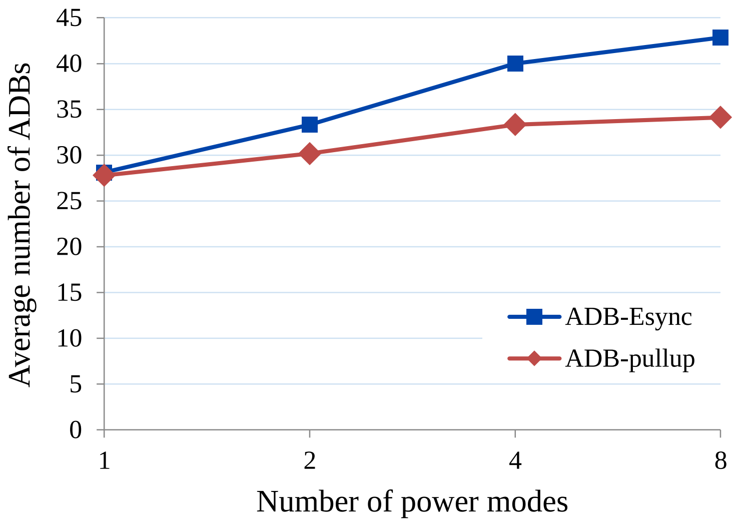


Figure 4.8: The changes of the average number of ADBs used by ADB-ESYNC and ADB-PULLUP by varying the number of power modes used.

Figure 4.8 shows the average numbers of ADBs allocated by ADB-ESYNC and our ADB-PULLUP when the number of modes varies. Clearly, ADB-PULLUP always uses less ADBs in all situations. The gap between the results increases as we increase the number of modes used since it is less likely that the ADB allocation in one mode coincides with the allocation in another mode.

Figure 4.9 shows the runtime of ADB-ESYNC [1], ADB-ESYNC-Q [1], and proposed algorithms. ADB-PULLUP takes comparable runtime with that of ADB-ESYNC, and ADB-PULLUP-Q takes shorter compared to that of ADB-ESYNC-Q because it does not rely on iterations. The runtimes of ADB-PULLUP and ADB-PULLUP-Q are

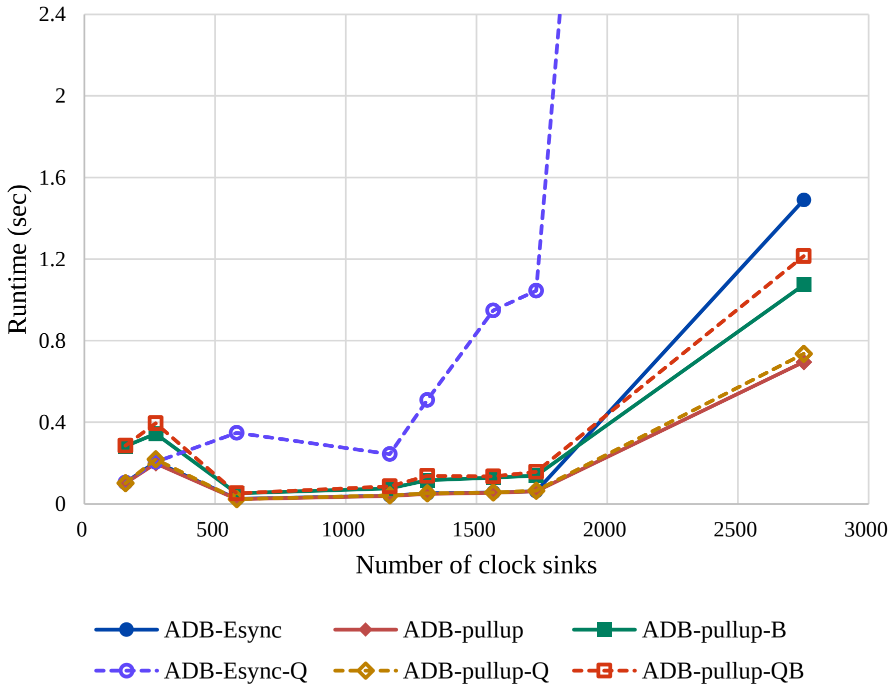


Figure 4.9: Runtime of ADB-ESYNC [1], ADB-ESYNC-Q [1], ADB-PULLUP, ADB-PULLUP-Q, ADB-PULLUP-B, and ADB-PULLUP-QB. ADB-ESYNC-Q took about 16 sec. for a circuit with 2752 sinks.

theoretically $O(n \log n)$, but they are arbitrarily shorter than it in practice. This might be because READJUST function does not traverse all the children when an ADB is not allocated and α becomes 0.

In summary, the comparison confirms that: (1) ADB-PULLUP finds the optimal solution with the minimum number of ADBs, reducing the average number of ADBs by 13.5% compared to ADB-ESYNC [1]; (2) ADB-PULLUP-Q allocates 15.8% less number of ADBs compared to ADB-ESYNC-Q [1] and its runtime is much shorter because it does not rely on iteration; (3) ADB-PULLUP-B and ADB-PULLUP-QB further reduces the area overhead.

Chapter 5

ADB ALLOCATION UNDER USEFUL SKEW

In this chapter, a solution to Problem 2, the ADB insertion problem utilizing useful skew is proposed.[1]

5.1 Related Works and Motivational Examples

While the most of the existing works on ADB allocation attempted to make clock trees to meet the bounded clock skew constraint, they contain a risk of the resulting clock trees being overdesigned owing to their attempts to reduce clock skew of irrelevant clock sink pairs. Hence, adopting useful skew scheduling, which is to determine the clock arrival time of each sink by carefully examining the timing relationships with other sinks, can help relaxing the design constraints so that the number of ADBs can be further reduced.

For instance, let us consider a datapath connecting from the output of s_1 to the input of s_3 in Figure 5.1(a), having maximum propagation delays of 45 and 38 and minimum delays of 30 and 27 in *mode-1* and *mode-2*, respectively. Then, given a clock period $T_{clk} = 50$, setup time $t_{setup} = 1.8$, and hold time $t_{hold} = 1.6$, the setup time constraints at s_3 from s_1 are (*constraint-1*) $x_{s_3,1} - x_{s_1,1} \geq 45 - 50 + 1.8$ in *mode-1* and

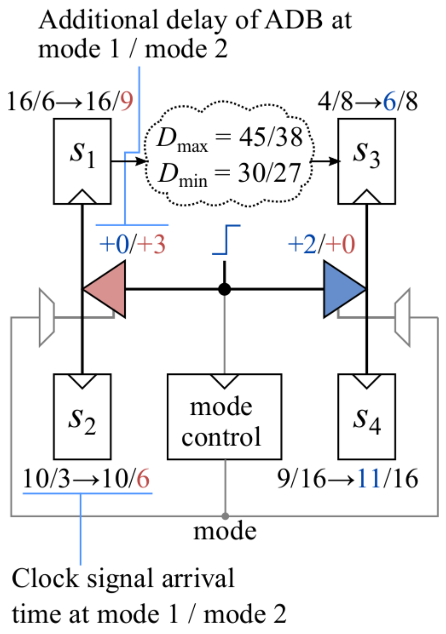
[1]The content of this chapter is an extended version of [32, 33].

(*constraint-2*) $x_{s_3,2} - x_{s_1,2} \geq 38 - 50 + 1.8$ in *mode-2* and the hold time constraints at s_3 from s_1 are (*constraint-3*) $x_{s_1,1} - x_{s_3,1} \geq -30 + 1.6$ in *mode-1* and (*constraint-4*) $x_{s_1,2} - x_{s_3,2} \geq -27 + 1.6$ in *mode-2*.

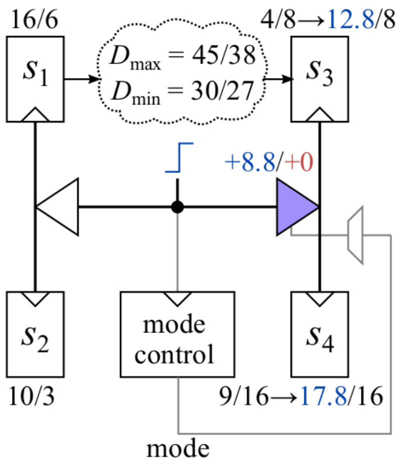
Under the clock skew bounds of 10 for each power mode, *two* clock buffers should be replaced by ADBs. However, even after the allocation, the signal arrival times to s_1 are 16 and 9 and times to s_3 are 6 and 8 in *mode-1* and *mode-2*, respectively, in which *constraint-1* is not satisfiable as shown in Figure 5.1(a). Note that the violations would not be resolved even with the tightest skew bound feasible $(= 6)$. In addition, if the maximum delay of the datapath from s_1 to s_3 in *mode-1* was larger than 48.2, even a clock tree with a complete zero skew would not resolve the violation. On the other hand, Figure 5.1(b) shows the ADB allocation result under the use of useful clock skew for the same initial clock tree in Figure 5.1(a). In this case, only *one* ADB with delay (increment) assignment of 8.8 and 0 in *mode-1* and *mode-2* suffices to satisfy all the setup and hold time constraints.

From this example, we can observe that (1) the (red-colored) ADB on the left in Figure 5.1(a) is inserted unnecessarily, to reduce the clock skew between s_2 and s_4 which does not affect the operation of the circuit. In other words, using the global skew bound would lead the constraints among clock sinks in the design be tighter than are required, which means more ADBs will be allocated. Moreover, (2) utilizing the useful clock skew enables the satisfaction of timing constraints for some cases which was not solvable by reducing the clock skew. This example clearly shows that ADB allocation problem should be addressed under the context of useful skew scheduling if we really want to utilize ADBs effectively in multiple power mode designs. In addition, it should be mentioned that meeting the clock skew bound in multiple power modes or meeting the setup and hold time constraints by using useful skew in multiple power modes entails a lot of complication and very difficult to apply previous methods (e.g., [22, 34–38]), which have originally targeted to the designs of single power mode.

The work by Chou *et al.* [2] addressed the ADB allocation problem under useful



(a)



(b)

Figure 5.1: An example of clock tree T with the replacement of clock buffers with ADBs to meet the clock skew constraints. ($T_{clk} = 50$, $t_{setup} = 1.8$, $t_{hold} = 1.6$) (a) A clock tree with two ADBs allocated to satisfy the bounded clock skew ($\kappa_1 = 3.2$, $\kappa_2 = 10.2$) constraint. (b) Useful clock skew leads to allocate only one ADB while meeting the setup time constraint.

skew scheduling. For a given allocation of m ADBs in a clock tree, they formulated the problem of assigning the values of delay adjustment for each ADB with the objective of minimizing the clock period into a linear programming (LP) and solved it optimally. More precisely, setup and hold time constraints are converted to LP constraints considering the ADB delays and clock period T as variables. Then, the LP solver tries to solve the problem with an objective of minimizing T. In this method, the positions of clock buffers to be replaced by ADBs should be provided to formulate the LP problem. An iterative, greedy algorithm was adopted in the work, in which they try inserting the first ADB on every position and choose the best one, and repeat the same step to find the next position.

One of the weaknesses of this algorithm is that it does not guarantee the allocation of a minimum number of ADBs, due to its greedy nature which does not consider the other ADBs to be allocated later. The other is that it tries to find a minimal clock period by allocating ADBs, one ADB at a time, until using m target ADBs. To ensure the algorithm works, the formulated LP problem should have solution every time it has to choose the next position to allocate the ADB. Thus, the algorithm will not work if all hold time violations in the initial clock tree cannot be completely resolved when only a single ADB is available during the first iteration of the algorithm. (Setup time violations are always able to be solved by increasing the clock period.)

This work overcomes the drawbacks of the prior ADB allocation under useful skew scheduling. Our graph based algorithm solves the ADB allocation problem in multiple power modes under useful skew scheduling optimally, meaning that both of the allocation of a minimum number of ADBs and 100% elimination of setup and hold time violations are guaranteed in every power mode. In some cases, 100% elimination of time violation may not be possible if the time constraints or clock period is too tight. If then, our algorithm reports that any ADB allocation cannot completely resolve the time violation. Thus, by incrementally relaxing the clock period, the repeated application of our algorithm can find a minimal clock period that leads to a 100% elimination of the time violation.

To sum up, the technical contributions of the work are (1) optimal ADB allocation algorithm under useful clock skew scheduling, (2) graph traversal based speeding up computation without losing optimality, and (3) dealing with practical issues: trade-off between quality and runtime, supporting ADBs with discrete values, and ADBs with delay upper bound.

5.2 Observations

A difference graph $G = (E, V)$ is a directed graph where the weight of edge $(u, v) \in E$ means the difference between the values of vertices associated with $u, v \in V$. For example, an edge (u, v) with the weight of $w(u, v)$ denotes an inequality

$$x_v - x_u \geq w(u, v), \tag{5.1}$$

where x_u, x_v are the values associated with vertices u and v.

Since our proposed algorithm plays with a difference constraint graph, we start with the procedure of transforming an input clock tree with time information into a constraint graph followed by introducing our graph-based ADB allocation algorithm. Figure 5.2 shows the whole procedure.

We define some terms on a clock tree $\mathcal{T}$ to be used throughout the description.

- $\alpha_{i,m}$: the value of delay increment, in mode m, of an ADB which is allocated to replace a buffer in node v_i in $\mathcal{T}$. ($\alpha_{i,m}$ is *not* the delay of the ADB in m. The ADB delay is the buffer delay in m *plus* α_{i_m}.)

- $\delta_{i,m}$: the sum of α values, in mode m, of all ADBs on the path from the root of $\mathcal{T}$ to node v_i. From the definition, $\alpha_{i,m} = \delta_{i,m} - \delta_{p,m}$ where v_p is the parent of v_i.

- $\beta(s_k)$: the buffer node in $\mathcal{T}$ that directly drives sink s_k.

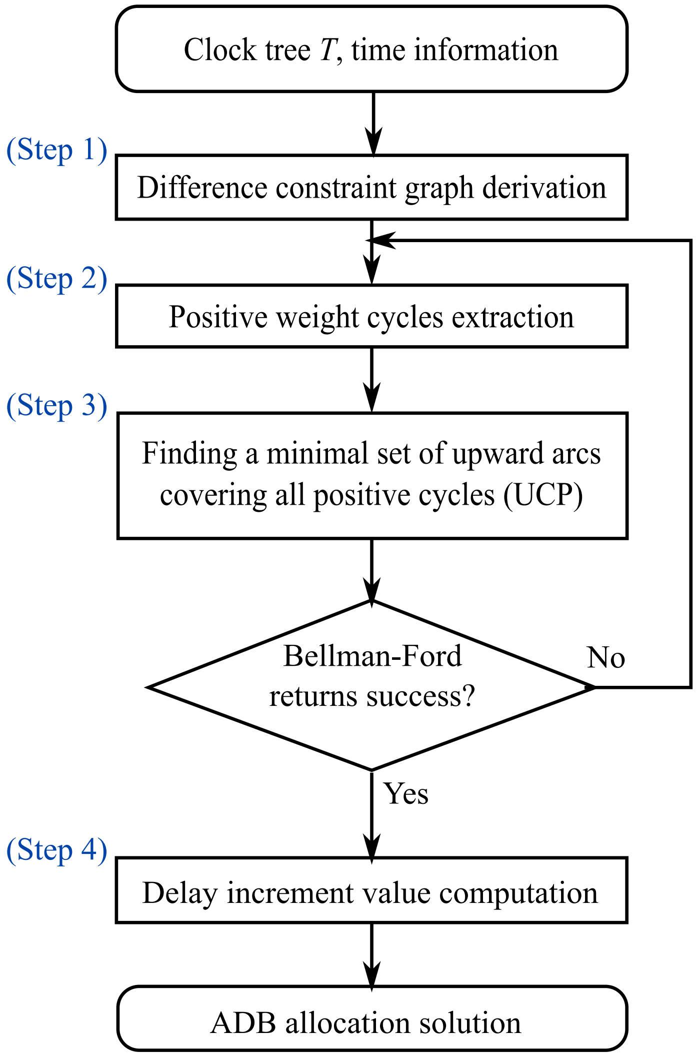
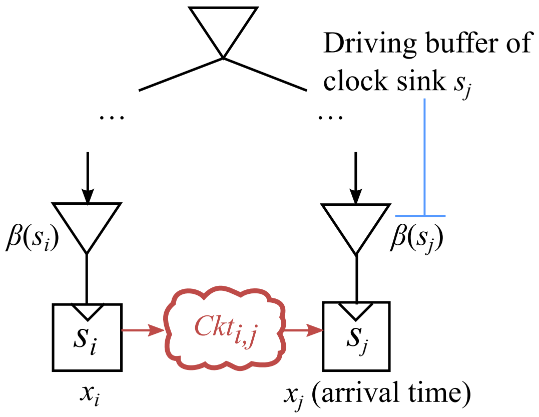
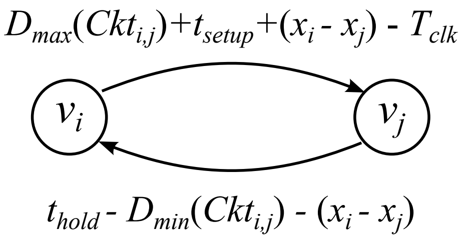

Clock tree T, time information
(Step 1)
Difference constraint graph derivation
(Step 2)
Positive weight cycles extraction
(Step 3)
Finding a minimal set of upward arcs covering all positive cycles (UCP)
Bellman-Ford returns success?
No
Yes
(Step 4)
Delay increment value computation
ADB allocation solution

Figure 5.2: A flow diagram showing the process of the proposed algorithm.

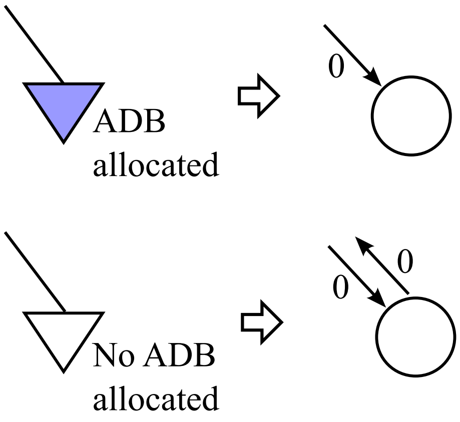


(a) Timing relation between two sinks.

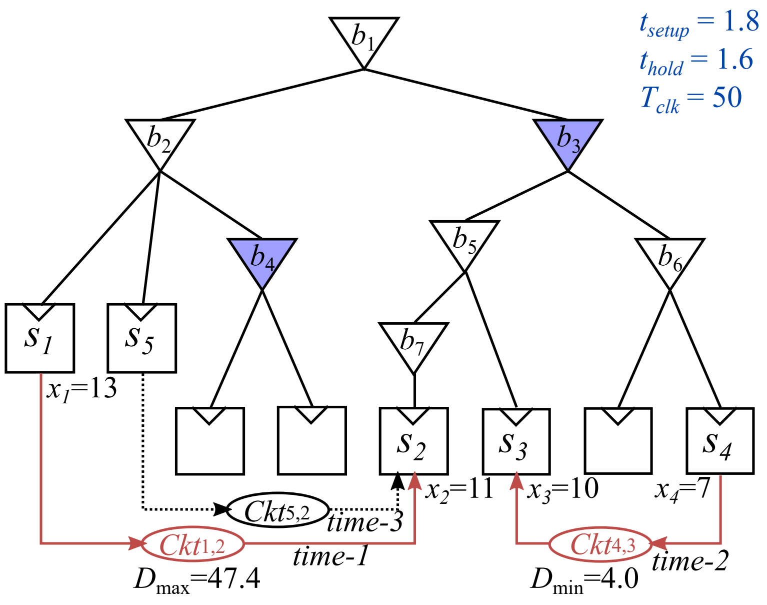


$D_{max}(Ckt_{i,j})+t_{setup}+(x_i - x_j) - T_{clk}$
v_i
v_j
$t_{hold} - D_{min}(Ckt_{i,j}) - (x_i - x_j)$

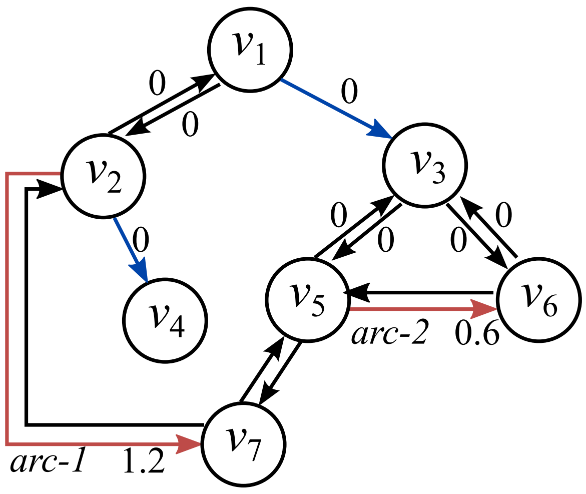
(b) A part of a constraint graph that shows a setup time constraint from v_i (corresponding to $\beta(s_i)$) to v_j (corresponding to $\beta(s_j)$) and a hold time constraint from v_j to v_i.



(c) Nodes corresponding to an ADB and a buffer in a constraint graph.



(d) A clock tree $\mathcal{T}$.



(e) The difference constraint graph $G(V, A)$ for $\mathcal{T}$ where each node represents a distinct buffer in $\mathcal{T}$, and *arc-1* and *arc-2* correspond to the *time-1* and *time-2* in (b), respectively. The weight $(= 1.2)$ on *arc-1* indicates the value of $t_{hold} - D_{min}(Ckt_{4,3}) - (x_4 - x_3)$ (i.e., the right term in Equation (5.5)). If there are multiple arcs with the same direction between two nodes, we select the arc with the largest weight and remove the others.

Figure 5.3: Derivation of a time difference constraint graph from a clock tree $\mathcal{T}$.

- x_i (x_j) denotes the clock signal arrival time at sink s_i (s_j) in a clock tree with no ADBs, and $Ckt_{i,j}$ represents the datapath that connects s_i and s_j. (An illustration is in Figure 5.3(a).) $D_{max}(Ckt_{i,j})$ and $D_{min}(Ckt_{i,j})$ denote the maximum and minimum delays from the output of s_i to the input of s_j through circuit $Ckt_{i,j}$, respectively. Note that x_i, x_j, $D_{max}(Ckt_{i,j})$, $D_{min}(Ckt_{i,j})$, t_{setup}, t_{hold}, and T_{clk} will change their values depending on the power modes used. For simplification, it is assumed that the values implicitly vary in different power modes.

Let T_{clk} be the clock period, and t_{setup} and t_{hold} indicate the setup and hold times of a sink, respectively. Then, *the setup and hold time constraints* at s_j directly driven from s_i in a power mode can be expressed as:

$$x_j - x_i \geq D_{max}(Ckt_{i,j}) + t_{setup} - T_{clk}, \tag{5.2}$$

$$x_i - x_j \geq t_{hold} - D_{min}(Ckt_{i,j}). \tag{5.3}$$

Setup time constraint disallows zero-clocking at sinks caused by overly long data propagation in the datapath while the hold time constraint avoids double-clocking caused by fast data propagation.

When we name nodes corresponding to $\beta(s_i)$ and $\beta(s_j)$ as v_i and v_j, δ_i and δ_j are respectively the amounts of the increase of signal arrival times at s_i and s_j resulting from an allocation of ADB(s) to the clock tree. Thus, the updated signal arrival times become $x'_i = x_i + \delta_i$ and $x'_j = x_j + \delta_j$. Accordingly, the time constraints in Equation (5.2) and Equation (5.3) are updated as:

$$\delta_j - \delta_i \geq D_{max}(Ckt_{i,j}) + t_{setup} + (x_i - x_j) - T_{clk}, \tag{5.4}$$

$$\delta_i - \delta_j \geq t_{hold} - D_{min}(Ckt_{i,j}) - (x_i - x_j). \tag{5.5}$$

By the definition of δ, for two nodes v_i and v_p in a clock tree such that v_i is a child of v_p, $\delta_i \geq \delta_p$. Furthermore, $\delta_i = \delta_p$ if v_i does not use ADB for delay adjustment.

Our aim is to minimally allocate ADBs to the internal nodes of clock tree, thereby controlling the arrival times, so that all the setup and hold time constraints (e.g., Equation (5.4) and Equation (5.5)) in the circuit should be satisfied.

Definition 4. The *difference constraint graph* $G(V, A)$ for an input clock tree $\mathcal{T}$ is constructed as follows.

- A *node* $v_i \in V$ is created for each buffer node B_i in $\mathcal{T}$ and $|V|$ amounts to the total number of buffer nodes in $\mathcal{T}$. Thus, in the following description, we use notation v_i to indicate either a node in $\mathcal{T}$ or the corresponding node in $G(V, A)$ if it does not cause a confusion.

- For a setup time constraint at s_j from s_i, there exists an arc $v_i \rightarrow v_j \in A$ such that v_i and v_j correspond to $\beta(s_i)$ and $\beta(s_j)$ in $\mathcal{T}$, respectively. The value of arc weight, denoted by $w(v_i \rightarrow v_j)$, is set to $D_{max}(Ckt_{i,j})+t_{setup}+(x_i-x_j)-T_{clk}$, which is the right term in Equation (5.4). (See Figure 5.3(b).)

- For a hold time constraint at s_j from s_i, there exists an arc $v_j \rightarrow v_i \in A$ such that v_i and v_j correspond to $\beta(s_i)$ and $\beta(s_j)$ in $\mathcal{T}$, respectively. The value of arc weight $w(v_j \rightarrow v_i)$ is set to $t_{hold} - D_{min}(Ckt_{i,j}) - (x_i - x_j)$, which is the right term in Equation (5.5). The arc direction is reversed as opposed to that in the setup time constraint. (See Figure 5.3(b).)

- For a node v_i and its parent v_j in an initial clock tree $\mathcal{T}$, there exist two arcs $v_i \rightarrow v_j$ and $v_j \rightarrow v_i$, each of which has a weight of 0. We will use the two arcs to impose the constraints $\delta_j - \delta_i \leq 0$ and $\delta_i - \delta_j \leq 0$, namely $\delta_i = \delta_j$ in $\mathcal{T}$.

- Later on, if an ADB is allocated to v_i, the constraint $\delta_i \leq \delta_j$ $(= \delta_i - \delta_j \leq 0)$ does not hold any more. Thus, only the downward arc $v_j \rightarrow v_i$ will be retained to impose the constraint $\delta_j \leq \delta_i$ $(= \delta_j - \delta_i \leq 0)$. For example, Figure 5.3(c) shows the change of arcs on v_i when an ADB is allocated to v_i and no ADB is allocated to v_i, respectively.

An arc $e \in A$, say $v_i \to v_j$, created by a setup or hold time constraint is called a *constraint arc*. We use notations $f(e)$ and $r(e)$ denote v_i and v_j, respectively, and $w(e)$ the arc weight of e.

Figure 5.3(e) shows the difference constraint graph $G(V, A)$ for the clock tree in Figure 5.3(d). For example, for the setup time constraint at s_2 from s_1 in Figure 5.3(d), an arc from v_2, corresponding to $\beta(s_1) = b_2$ in Figure 5.3(d), to v_7, corresponding to $\beta(s_2) = b_7$ in Figure 5.3(d), is created in Figure 5.3(e). The arc weight is $D_{max}(Ckt_{i,j}) + t_{setup} + (x_i - x_j) - T_{clk} = 47.4 + 1.8 + (13 - 11) - 50 = 1.2$. On the other hand, for the hold time constraint at s_3 from s_4 in Figure 5.3(b), an arc from v_5, corresponding to $\beta(s_3) = b_5$, to v_6, corresponding to $\beta(s_4) = b_6$, is created. The arc weight is $t_{hold} - D_{min}(Ckt_{i,j}) - (x_i - x_j) = 1.6 - 4.0 - (7 - 10) = 0.6$. We can see that every non-root node with no ADB has two connecting arcs, one from the node to its parent i.e., the upward arc and the other from its parent to the node i.e., the downward arc, while every non-root node with ADB does have the downward arc only.

5.3 ADB Allocation Algorithm Utilizing Useful Skew

This section proposes a solution to the ADB allocation problem under useful skew scheduling. Given an initial buffered clock tree $\mathcal{T}$, power modes m_1, m_2, $\cdots$, m_K, clock signal arrival time to each sink in $\mathcal{T}$ on every power mode, together with the values of T_{clk}, t_{setup} and t_{hold} of sinks, and D_{max} and D_{min} values of the datapaths between sinks, the algorithm finds a subset of buffers in $\mathcal{T}$ to be replaced by ADBs and assign delay values to the ADBs for every power mode such that the number of ADBs used is minimized while the setup and hold time constraints are satisfied for all power modes.

We have the following properties for the difference constraint graph. The proofs of the properties are provided in Section 5.5.

Lemma 2. *Let G be the difference constraint graph constructed by* ADB-UCP *for a clock tree $\mathcal{T}$ with an ADB allocation instance I. Then, G contains no positive weight cycle if and only if there exists a delay assignment to the ADBs in I that meets all the setup and hold time constraints.*

Lemma 2 and the fact that an allocation of ADB to a node in $\mathcal{T}$ removes its upward arc in G suggest that the problem can be solved by minimally allocating ADBs to the nodes in $\mathcal{T}$ in a way that the resultant G has no positive weight cycle.

Theorem 3. *Any instance of useful skew scheduling problem can be transformed into the instance of problem of removing a minimum number of upward arcs from the constraint graph of the input clock tree. In addition, the starting node of every upward arc, say $v_i \to v_p$, removed is the location where an ADB is allocated, and its delay value of ADB in v_p can be set to $\delta_i - \delta_p$.*

Based on Theorem 3, we propose an ADB allocation algorithm, called ADB-UCP, to solve the problem. ADB-UCP performs the following four steps: (Step 1) generating a difference constraint graph $G(V, A)$ of an input clock tree $\mathcal{T}$; (Step 2) extracting positive weight cycles in G; (Step 3) finding a minimum number of upward arcs in G that cut all the cycles extracted in Step 2; (Step 4) removing the upward arcs found in Step 3 (replacing the corresponding buffers with ADBs) and determining their delay increments.

(Step 1) Derivation of difference constraint graph from $\mathcal{T}$: Difference constraint graph $G(V, A)$ introduced in Section 5.2 provides the time relation in a way that each node in V represents a distinct buffer in $\mathcal{T}$ and each arc in A imposes a time constraint. The derivation of difference constraint graph follows the steps in Definition 4. Figure 5.4(a) shows an example of G with two power modes *mode-1* and *mode-2*. For example, $+2/-1$ indicate that its arc weight in *mode-1* is $+2$ and -1 in *mode-2*.

ADB-UCP completes G by abstracting multiple arcs into a single representative arc. Multiple arcs whose sources as well as destinations are identical, but with different arc weights are abstracted into a single one and the arc weight is set to the largest one. For example, two arcs with weight of $+3/-2$ and $-1/-1$ is abstracted to an arc having weight of $\max(+3, -1)/\max(-2, -1) = +3/-1$ The rationale for the selection of the largest weight is that its time relation is the tightest, meaning that in the course of ADB allocation trials, the time relation on each arc that were removed can never be tighter than that of the arc with the largest weight we chosen.

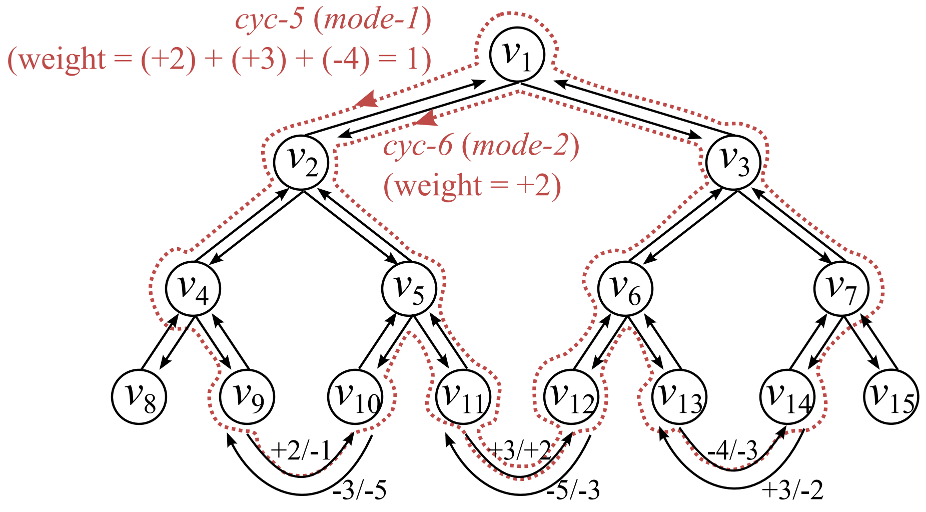
(Step 2) Extracting positive weight cycles from G: By Lemma 2, ADB-UCP finds all positive cycles[2] of each mode in G. We devise a very efficient graph traversal algorithm that is suited for a fast cycle enumeration, specialized in the context of our problem while ensuring the exhaustiveness. The details are presented in Section 5.4.1. On the other hand, if a positive cycle without any upward arcs (i.e., composed of only downward arcs and constraint arcs) exists, the timing constraints cannot be met even when all buffers are replaced with ADBs. In this case, ADB-UCP reports that problem is unsolvable and stops. Otherwise, ADB-UCP moves on Step 3.

(Step 3) Finding a minimal set of upward arcs covering all positive cycles: For a set C of all positive cycles obtained in Step 2, we are required to find a set R of upward arcs in G to resolve all time violations such that (i) for each $c_i \in C$, there is an upward arc $e_j \in R$ in c_i and (ii) $|R|$ is minimum. Our ADB-UCP solves the problem by transforming it into the *unate covering problem* (UCP).

The UCP [39] is, given a matrix M of m rows and n columns, for which $M_{i,j}$ is either 0 or 1, the problem of finding a minimum cardinality column subset U that satisfies

$$\exists_{j \in U} M_{i,j} = 1, \forall i \in \{1, \cdots, m\}. \tag{5.6}$$

[2]Our algorithm cuts all positive cycles, but finding a minimal number of cut points could be very expensive for large graphs. In practice, the algorithm repeatedly finds a set of limited number of positive cycles and cuts them until there is no positive cycle in the graph. We use a control parameter *limit* to set the upper bound of positive cycles. If *limit* = ∞, simply one iteration is performed, and guarantees the optimality. The details of the iterative strategy is described in Section 5.4.1.



Positive cycles in *mode-1*

☐: upward arcs

cyc-1: $v_9 \rightarrow \boxed{v_{10} \rightarrow v_5 \rightarrow} v_2 \rightarrow v_4 \rightarrow v_9$

cyc-2: $v_{11} \rightarrow \boxed{v_{12} \rightarrow v_6 \rightarrow v_3 \rightarrow} v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_{11}$

cyc-3: $v_{14} \rightarrow \boxed{v_{13} \rightarrow v_6 \rightarrow} v_3 \rightarrow v_7 \rightarrow v_{14}$

cyc-4: $v_9 \rightarrow \boxed{v_{10} \rightarrow} v_5 \rightarrow v_{11} \rightarrow \boxed{v_{12} \rightarrow v_6 \rightarrow v_3 \rightarrow} v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_9$

cyc-5: $v_9 \rightarrow \boxed{v_{10} \rightarrow} v_5 \rightarrow v_{11} \rightarrow \boxed{v_{12} \rightarrow} v_6 \rightarrow v_{13} \rightarrow \boxed{v_{14} \rightarrow v_7 \rightarrow v_3 \rightarrow} v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_9$

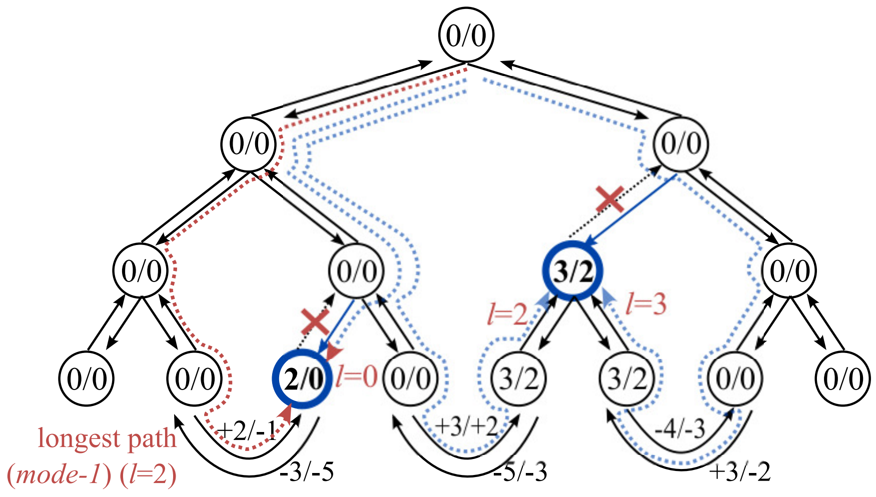
Positive cycles in *mode-2*

cyc-6: $v_{11} \rightarrow \boxed{v_{12} \rightarrow v_6 \rightarrow v_3 \rightarrow} v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_{11}$

(a) The extraction of all positive cycles with upward arcs in Step 2.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
cyc-1					×					×					
cyc-2			×			×						×			
cyc-3						×							×		
cyc-4			×			×				×		×			
cyc-5			×				×			×		×		×	
cyc-6			×			×						×			

(b) The construction of constraint matrix for (a) and the UCP solution in Step 3.



(c) The ADB insertion and delay assignment for (b) in Step 4.

Figure 5.4: An example illustrating the steps of ADB-UCP.

That is, the columns in the set U cover M in the sense that every row of M contains an 1-entry in at least one of the columns of U, and there is no smaller set which also covers M. The matrix M is called constraint matrix.

Thus, our transformation into UCP is to construct a constraint matrix by letting columns with all upward arcs in G, rows with all positive cycles in G, and $M_{i,j} = 1$ if the arc in the j^{th} column is an arc of the cycle in the i^{th} row, and $M_{i,j} = 0$, otherwise. For example, Figure 5.4(b) shows the constraint matrix of G in Figure 5.4(a), in which the matrix has six rows, one for each cycle extracted in G, and fifteen columns, one for each upward arcs in G. (Since every upward arc uniquely matches the tree node, we will denote the upward arcs with the name of nodes from which the upward arc originates.) The columns (i.e., v_6 and v_{10}) enclosed by the boxes indicate the minimal buffer locations for ADB replacement. The two bold circles in Figure 5.4(c) indicate ADB replacement of the two buffers b_6 and b_{10} in v_6 and v_{10}.

(Step 4) Computing delay increments of ADBs: We can use the following Lemma to assign delay increments of ADBs.

Lemma 3. *If the value of δ_i of node v_i in the constraint graph G produced in Step 3 of* ADB-UCP *is set to the length of the longest path from the root to v_i, all the timing violations in G can be resolved.*

Once the locations in G for ADB placement are determined in Step 3, ADB-UCP assigns value of delay increment to every ADB for each power mode. The value assignment is performed in two sub-steps:

4.1 (*Remove arcs originating from ADB nodes in G*): Recall that the zero weight on two arcs between a node v_i and its parent v_j in G is used to constrain $\delta_j - \delta_i \geq 0$ and $\delta_j - \delta_i \leq 0$, i.e., $\delta_j = \delta_i$ when no ADB is placed on v_i. If an ADB is placed on v_i, $\delta_j - \delta_i \geq 0$ will not hold any more. Thus, ADB-UCP removes the upward arc originating from each node in G with ADB.

4.2 (*Find the longest path from the clock source to every node in G*): ADB-UCP computes, for every node v_i in G, the length of longest path from clock source to v_i and set the computed value to δ_i. Thus, according to Lemma 3, if an ADB allocated at v_i is assigned with delay increment of $\delta_i - \delta_p$ where v_p is the parent of v_i, all time violations associated with v_i will be resolved. ADB-UCP applies the Bellman-Ford algorithm [40] to find, for each power mode, the longest path length to every node in G and compute the delay increments of the nodes with ADB. (Note that since Step 4.1 completely cleans all positive cycles out from G, the problem of finding a longest path in G can be transformed into the problem of finding a shortest path by multiplying every arc weight in G by -1.)

5.4 Extensions

5.4.1 Acceleration for Extracting Positive Cycles in G

ADB-UCP significantly reduces the number of positive cycles to be extracted from G, thereby reducing the total time of cycle extraction as well as the number of rows in the constraint matrix M, by utilizing *dominance relation* (in Definition 5) among cycles while maintaining the optimality of ADB-UCP.

Definition 5. A positive cycle *cyc-1* in a constraint graph G is said to be **dominating** a positive cycle *cyc-2* in G if every upward arc in *cyc-1* is also an upward arc in *cyc-2*.

ADB-UCP is interested in extracting a cycle set that at least one of its elements dominates any positive cycle in the graph, since a set of upward arcs that covers all cycles in the set also covers their dominated cycles, in other words, all positive cycles. Because we will convert the problem to UCP, extracting only essential elements from the set of all positive cycles is crucial for reducing runtime and memory consumption. We call such sets *dominating cycle sets*, and propose an efficient method that finds a dominating cycles set.

The key player in the procedure is a function EXPANDABLECONSTARCS(*cycle*) which returns every arc e_{K+1} when cycle *cycle* is $cyc(e_1, \cdots, e_K)$ and a constraint arc e_{K+1} satisfies the following five conditions. Note that $cyc(e_1, \cdots, e_N)$ denotes a shortest cycle passing constraint arcs from e_1 to e_N. For example, in Figure 5.4, *cyc-5* is $cyc(v_9 \rightarrow v_{10}, v_{11} \rightarrow v_{12}, v_{13} \rightarrow v_{14})$. On the other hand, $path(e_1, \cdots, e_N)$ denotes a shortest path passing constraint arcs from e_1 to e_N. The difference between $cyc(e_1, \cdots, e_N)$ and $path(e_1, \cdots, e_N)$ is that $cyc(e_1, \cdots, e_N)$ includes the shortest path from $r(e_N)$ to $f(e_1)$ in addition to $path(e_1, \cdots, e_N)$ where $f(e)$ and $r(e)$ represent the starting and ending nodes of arc e in *G*, respectively. The conditions every arc e_{K+1} in EXPANDABLECONSTARCS($cyc(e_1, \cdots, e_K)$) should satisfy are:

(1) $w(cycle) + w(e_{K+1}) > 0$,

(2) $l(NCA(f(e_1), r(e_K))) < l(NCA(r(e_K), f(e_{K+1})))$,

(3) $l(NCA(f(e_{K+1}), r(e_{K+1}))) < l(NCA(r(e_K), f(e_{K+1})))$ or $w(e_{K+1}) > 0$,

(4) $path(e_1, \cdots, e_{K+1})$ is an elementary path,

(5) A path exist (upward arcs were not covered) from $r(e_K)$ to $f(e_{K+1})$,

where $l(v_i)$ represents the level, which is the distance from root, of buffer B_i in the clock tree corresponding to *G*, and $NCA(v_i, v_j)$ is the nearest common ancestor of v_i and v_j in *G*. (Refer to Definition 4 to check other notations.)

Figure 5.5 shows our cycle search algorithm that extracts all positive cycles. The condition checking in the dotted box indicates the control of the upper limit of the number of cycles to be used as an input to one-time application of UCP solver, which should be set to an infinite number for the optimal solution. Parameter *limit* will be adjusted to finite number in Section 5.4.2, to efficiently handle runtime problem.

An example of dominating cycle set extraction is shown in Figure 5.6, in which a queue (*Q*) data structure is used to incrementally generate (i.e., expand) positive cycles, starting from the smallest cycles to the largest.

Theorem 4. *The cycle search algorithm in* ADB-UCP *that uses function* EXPANDABLECONSTARCS() *guarantees to extract a cycle set that dominates all the*

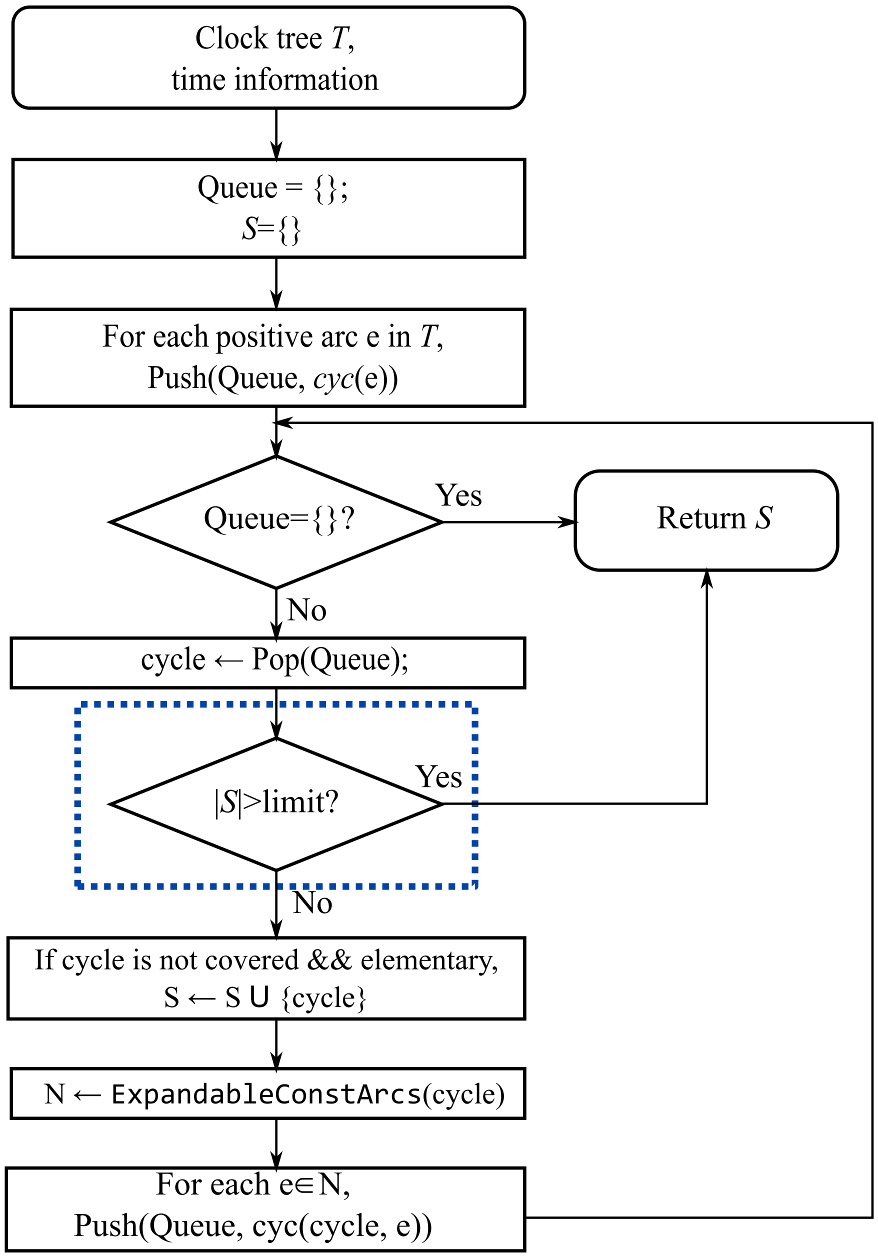
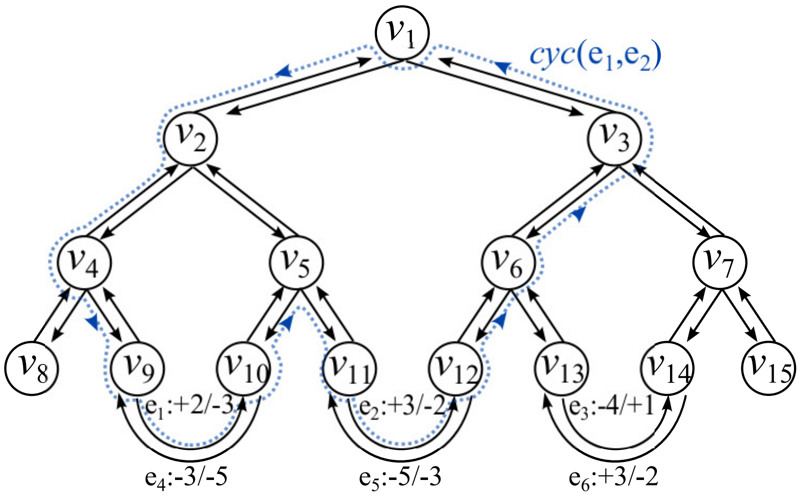
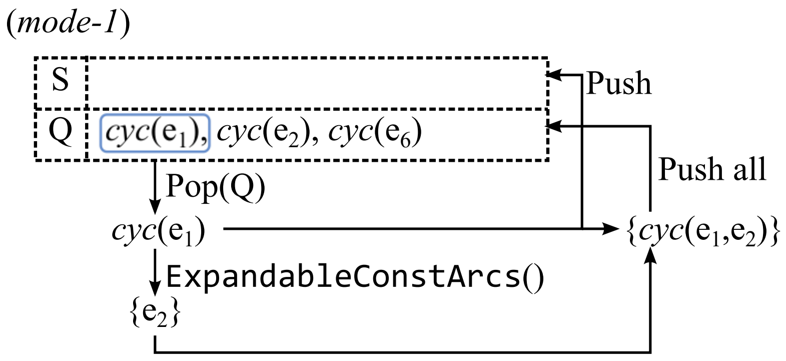


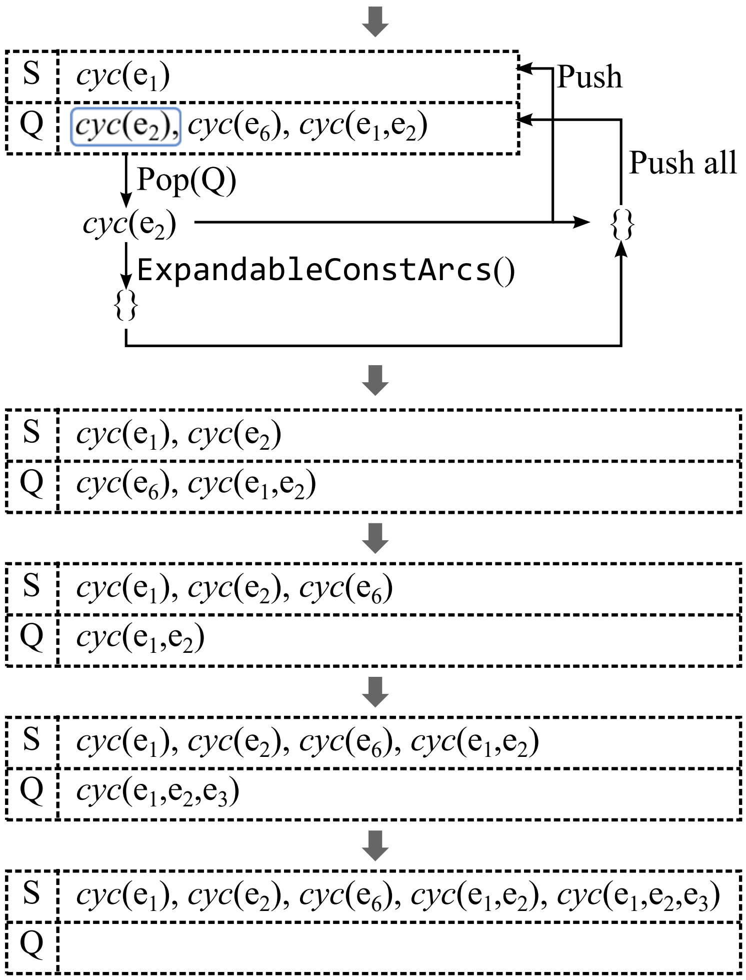
Figure 5.5: Flow diagram of incrementally generating a dominating cycle set S.



(a) Given constraints. $cyc(e_1, e_2)$ means the smallest cycle which contains e_1 and e_2 in the written order, as shown with dotted line.



(b) Extraction procedure in *mode-1*. Pop a cycle $cyc(e_1)$ from Q, push cycle $cyc(e_1, e_i)$ ($e_i \in \text{EXPANDABLECONSTARCS}(cyc(e_1))$) to Q, and push $cyc(e_1)$ to S.


S
cyc(e1)
Q
cyc(e2), *cyc*(e6), *cyc*(e1,e2)
Push
Push all
Pop(Q)
cyc(e2)
{}
ExpandableConstArcs()
{}
S
cyc(e1), *cyc*(e2)
Q
cyc(e6), *cyc*(e1,e2)
S
cyc(e1), *cyc*(e2), *cyc*(e6)
Q
cyc(e1,e2)
S
cyc(e1), *cyc*(e2), *cyc*(e6), *cyc*(e1,e2)
Q
cyc(e1,e2,e3)
S
cyc(e1), *cyc*(e2), *cyc*(e6), *cyc*(e1,e2), *cyc*(e1,e2,e3)
Q

(c) Repeat until $Q = \{\}$.

A dominating cycle set of all positive cycles :

(*mode-1*): *cyc*(e_1), *cyc*(e_2), *cyc*(e_6), *cyc*(e_1,e_2), *cyc*(e_1,e_2,e_3)

(*mode-2*): *cyc*(e_2)

(d) Repeat the extraction process for *mode-2*. The final cycle set will cover all positive cycles.

Figure 5.6: An example of systematic extraction of a dominating cycle set.

positive cycles in G.

The runtime for ADB allocation is significantly improved by devising an efficient graph traversal technique for positive cycle extraction. For example, our algorithm uses 81 seconds for benchmark s38417 while the prior LP based method spends 2322 seconds for the same circuit. Another merit is that our algorithm is able to explore design space to trade off between the number of ADBs allocated and the amount of time violations. This is possible due to the fast runtime and the easy evaluation of intermediate results in our graph formulation with incremental manipulation of ADB allocations.

5.4.2 Handling Scalability Problem

Even though our proposed method in Section 5.4.1 is able to accelerate the process of extracting all positive cycles, as the problem size increases, the method would suffer from the long computation time. As an alternative viable solution for coping with the runtime problem, we employ a sequential, greedy divide-and-conquer approach by introducing a control parameter *limit* in Figure 5.5 to limit the number of positive cycles to be extracted by ADB-UCP in one step of iteration. Once a subset of positive cycles is extracted by ADB-UCP, a minimal number of ADBs that cut the cycles are allocated. Then, the cycle extraction process with the value of *limit* repeats for the updated constraint graph. The iteration stops when the resultant constraint graph has no positive cycle.

This procedure notably reduces the runtime, due to the fact that the problem size shrinks enormously after each iteration. As can be seen from the dominating positive cycle set extraction technique in Section 5.4.1, positive cycles share a large portion of their upward arcs with smaller cycles. Thus, even if the upward arcs cover a limited number of positive cycles, removing them highly likely eliminates most of the unfound positive cycles, too.

Note that the result of the modified algorithm is optimal when all the positive cycles were removed after only one iteration. In other words, if Bellman-Ford algorithm does not return error after one execution of UCP, the number of allocated ADBs is minimal no matter how many positive cycles were exploited. It is due to the property of UCP that the optimal solution for a subset of constraints cannot be worse than the solution satisfying the whole original constraints to be covered.

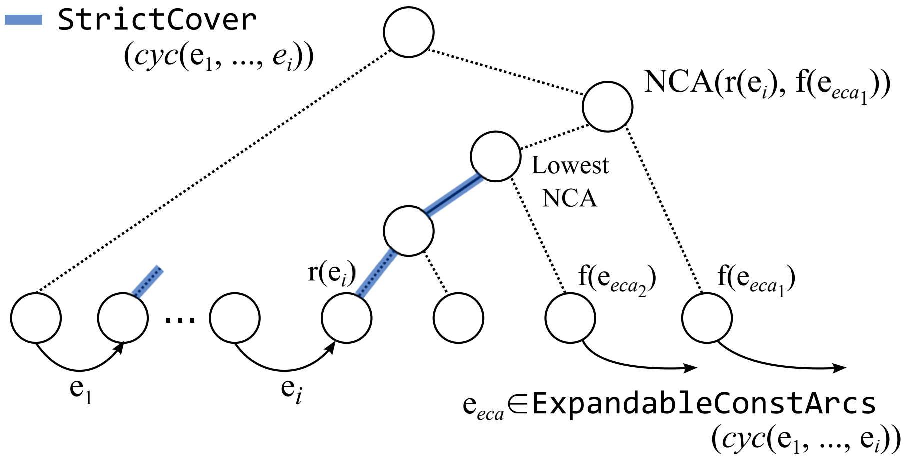


Figure 5.7: Finding a set of upward arcs who covers all positive cycles that can be found by expanding $cyc(e_1, \cdots, e_i)$. The set is marked with bold lines.

With the proper *limit* value, the algorithm usually runs within acceptable time. However, to constrain the runtime more strictly, i.e., limit the number of iteration, another technique can be additionally used. The key idea of this method is to modify the input of UCP at the last iteration to ensure that the output of UCP covers all positive cycles, even the undiscovered ones, at the expense of optimality. Figure 5.7 shows an intuitive example of extracting the upward arc set that covers all positive cycles that can be found by expanding $cyc(e_1, \cdots, e_i)$. The set is generated by joining the upward arc sets on two different paths, a subpath of the original cycle from $f(e_1)$ to $r(e_i)$, and the one from $r(e_i)$ to the lowest $NCA(r(e_i), f(e_{eca}))$

where $e_{eca} \in$ EXPANDABLECONSTARCS$(cyc(e_1, \cdots, e_i))$. The newly generated set dominates all cycles that can be generated from $cyc(e_1, \cdots, e_i)$. The overall flow of dominating cycle set generation with the iteration count limitation is presented in Figure 5.8. (STRICTCOVER(cyc) is a function that returns a set of upward arcs which dominates all positive cycles that can be found by adding more constraint arcs on cyc, as described in Figure 5.7.)

5.4.3 Supporting Discrete ADB Delay

Practically, the delay increment of ADBs is not continuous and the delay increment discrete values may vary depending on the power modes applied. For example, let us assume that an ADB on node v_i in a clock tree can only have delay increments of $\{\alpha_{i,m,1}, \alpha_{i,m,2}, \cdots, \alpha_{i,m,B}\}$ where $\alpha_{i,m,k}$ indicates k-th smallest delay increment value which the ADB can have. Then, our ADB allocation solution can be extended to solve such ADB allocation problem with the delay increments by repeatedly quantizing delays for each mode and adding ADBs until all timing violations are resolved. Here, the key procedure is to quantize delay values which non-seriously change the δ value of each sink. We use a bottom-up traversal strategy to find the best legal delay value for a node, and readjust ADB delay values of its subtree. It utilizes function DELAYCEIL(α, i, m), which returns the smallest delay increment value of an ADB located on v_i in power mode m and is larger or equal to α.

Our quantization procedure is illustrated in Figure 5.9. We assume the set of α values available to use is $\{1, 2, 3\}$ for every node and power mode for convenience. (Our algorithm is still valid for arbitrary values.) In addition, since the procedure repeats every power mode, we simply omit the power mode denotation m. Δ_i represents the 'error' caused by the delay quantization. For example, Δ_5, which was $1 - 0.6 = 0.4$ before the application of function READJUST in Figure 5.9, is updated to -0.6 after READJUST updates α values. The completion of the quantization produces a clock tree with legal ADBs, satisfying almost all timing constraints in most cases. If there is

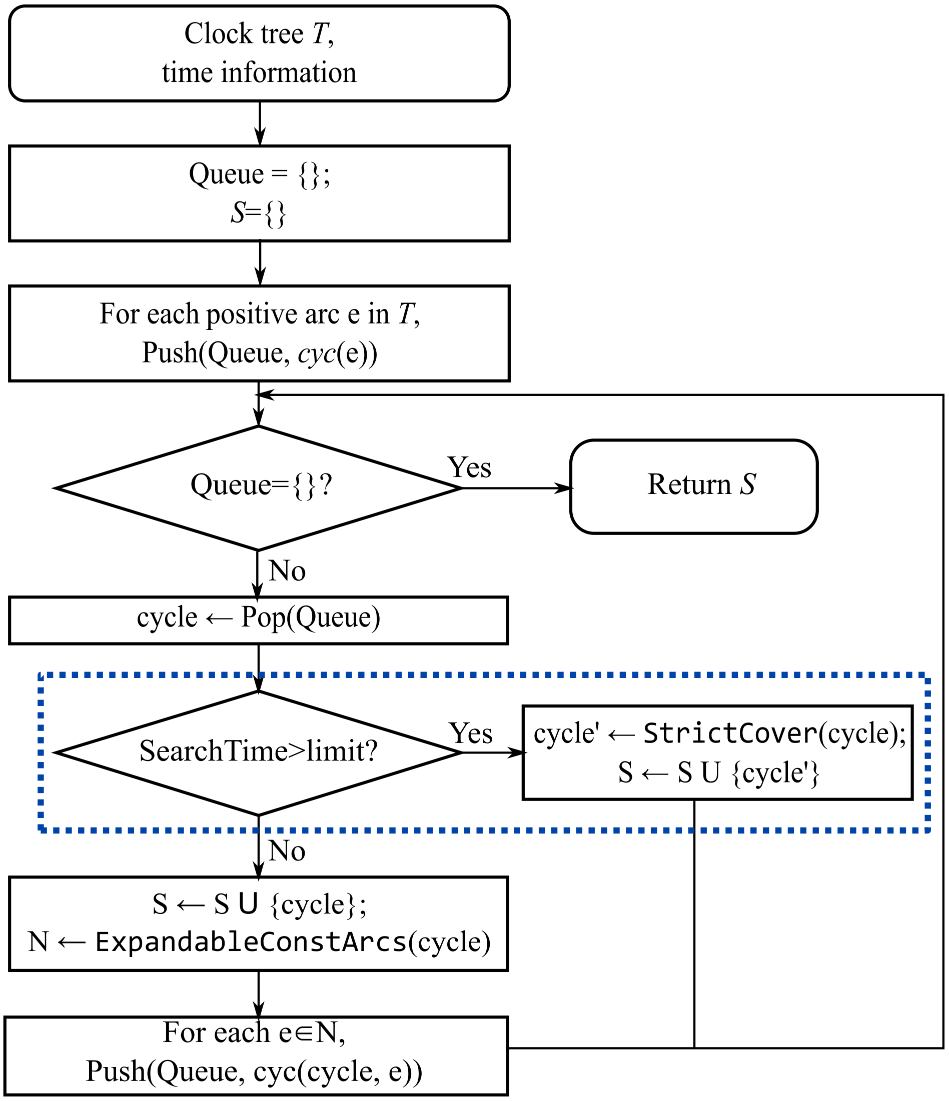
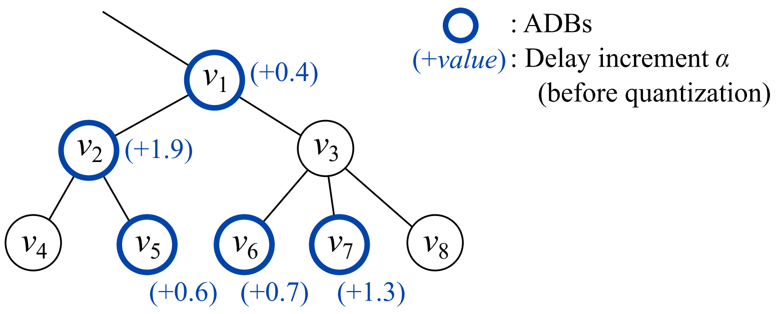
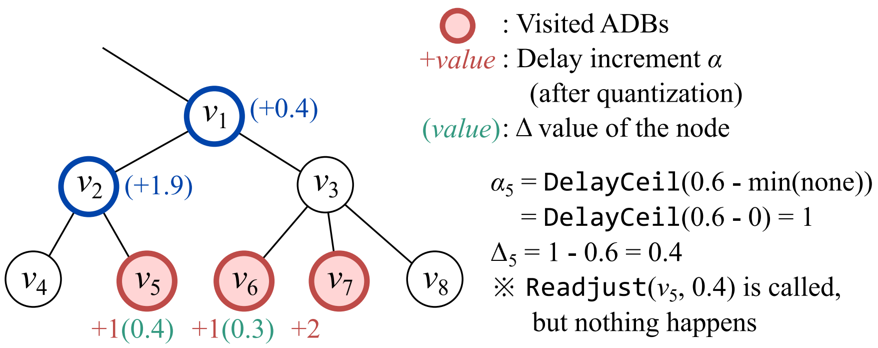

Clock tree *T*, time information
Queue = {}; *S*={}
For each positive arc e in *T*, Push(Queue, *cyc*(e))
Queue={}?
Yes
Return *S*
No
cycle ← Pop(Queue)
SearchTime>limit?
Yes
cycle' ← StrictCover(cycle); S ← S U {cycle'}
No
S ← S U {cycle}; N ← ExpandableConstArcs(cycle)
For each e∈N, Push(Queue, cyc(cycle, e))

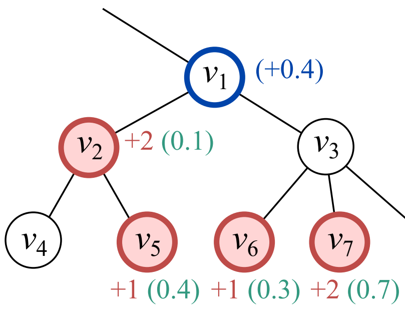
Figure 5.8: Flow diagram of incrementally generating a dominating cycle set with the hard constraint on the number of iteration.



(a) ADBs are allocated according to the application of ADB-UCP.

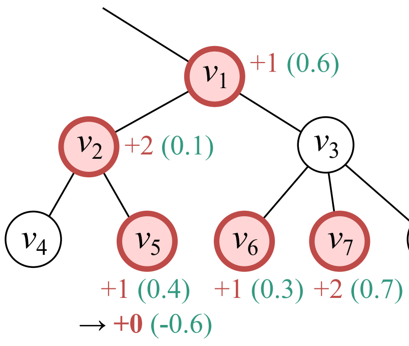

: Visited ADBs
+value : Delay increment α (after quantization)
(value): Δ value of the node
α_5 = DelayCeil(0.6 - min(none))
= DelayCeil(0.6 - 0) = 1
Δ_5 = 1 - 0.6 = 0.4
※ Readjust(v_5, 0.4) is called, but nothing happens
v_1 (+0.4)
v_2 (+1.9)
v_3
v_4
v_5
v_6
v_7
v_8
+1(0.4) +1(0.3) +2

(b) Delay values of v_5, v_6, and v_7 are quantized, and their Δ values are updated accordingly.



$\alpha_2 =$ `DelayCeil`(1.9 - min(0, 0.4))=2
$\Delta_2 = 2 - 1.9 = 0.1$
※ `Readjust`(v_2, 0.1) is called,
but nothing happens

(c) Delay value of v_2 is quantized.



```
function Readjust($n_i$, $\Delta$)
   $prev\_\alpha_i \leftarrow \alpha_i$
   $\alpha_i \leftarrow$ DelayCeil($\alpha_i - \Delta$)
   $\Delta_i \leftarrow \Delta_i$ - ($prev\_\alpha_i - \alpha_i$)
   $\Delta \leftarrow \Delta$ - ($prev\_\alpha_i - \alpha_i$)
   for every child $n_k$ of $n_i$ do
      Readjust($n_k$, $\Delta + \Delta_k$)
   end for
end function
```

(d) Delay value of v_1 is quantized, and function READJUST updates the delay increment value of v_5.

Figure 5.9: An example of delay quantization procedure under arbitrarily given delay increment values.

still a timing violation, we recalculate the constraint edge weights using the new clock arrival time and repeat the procedure, in which the set of available α values is updated to $\{\max(0, \alpha_{i,m,k} - \alpha_i) : 1 \leq k \leq B\}$ for each node n_i that has previously been allocated with an ADB.

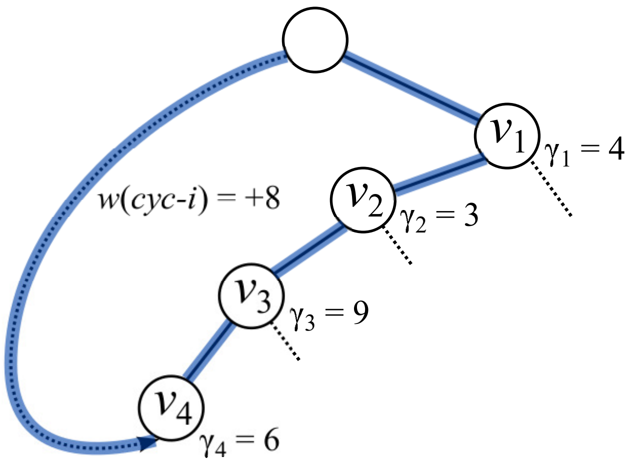
5.4.4 Supporting Bounded ADB Delay

Because of the area restriction on ADB cells, ADB's adjustable delay range is bounded from 0 to a certain number, say $\gamma_{n_i,m}$, i.e., $0 \leq \alpha_{n_i,m} \leq \gamma_{n_i,m}$. To support this, ADB-UCP updates the constraint matrix M by replacing the every row to the newly generated rows with the purpose of enabling UCP solver to produce a cover such that the resulting delay increments of ADBs are all within the value of γ.

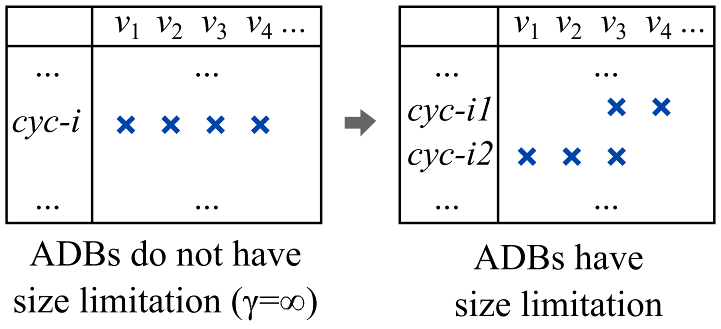
The algorithm is modified as follows. For a set of upward arcs C_i in a positive cycle, find every combination of upward arcs in C_i whose total γ does not exceed the weight of the original cycle. Then, generate constraint matrix rows consisted with sets $C_i \setminus S_{i,j}$ for each combination j and replace the row corresponding to C_i with the newly generated rows Note that if $S_{i,x}$ is a subset of $S_{i,y}$, the constraint matrix do not have to include $C_i \setminus S_{i,x}$ because its existence does not affect the final result. In addition, in Step 4, for each node v_i found for ADB replacement, weight of the upward arc from v_i is set to $-\gamma_{i,m}$ instead of removing the arc.

For example, Figure 5.10(a) shows a cycle *cyc-i* of $w($*cyc-i*$) = +8$ with upward arcs starting from v_1, v_2, v_3 and v_4. Let us assume $\gamma_1 = 4$, $\gamma_2 = 3$, $\gamma_3 = 9$, $\gamma_4 = 6$. Then, $\{\{v_1\}, \{v_2\}, \{v_4\}, \{v_1, v_2\}\}$ are the set of combinations whose total sum is smaller than 8. Because $\{v_1\}$ and $\{v_2\}$ are subsets of $\{v_1, v_2\}$, replacing the original row with two sets $\{v_1, v_2, v_3, v_4\} \setminus \{v_1, v_2\} = \{v_3, v_4\}$ and $\{v_1, v_2, v_3, v_4\} \setminus \{v_4\} = \{v_1, v_2, v_3\}$ is enough.

Figure 5.10(b) shows the original and newly generated constraint matrix in which the row of *cyc-i* is replaced by two rows labeled *cyc-i1*, *cyc-i2*. We can easily check that the rows *cyc-i1* and *cyc-i2* all together constrain UCP solver to select the to resolve

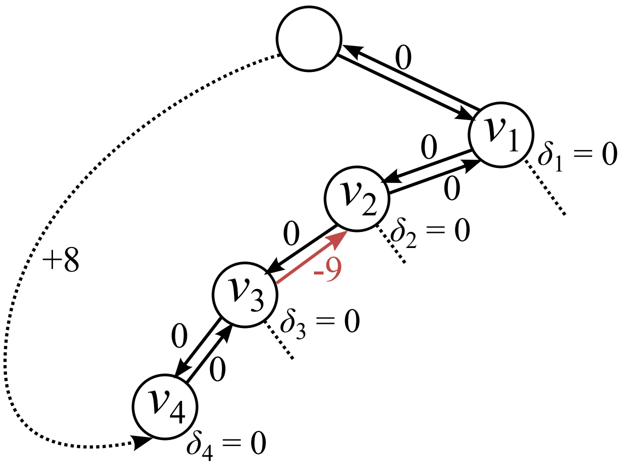


(a) A positive cycle in G with $w = +8$ and upward arcs = $\{v_1, v_2, v_3, v_4\}$.

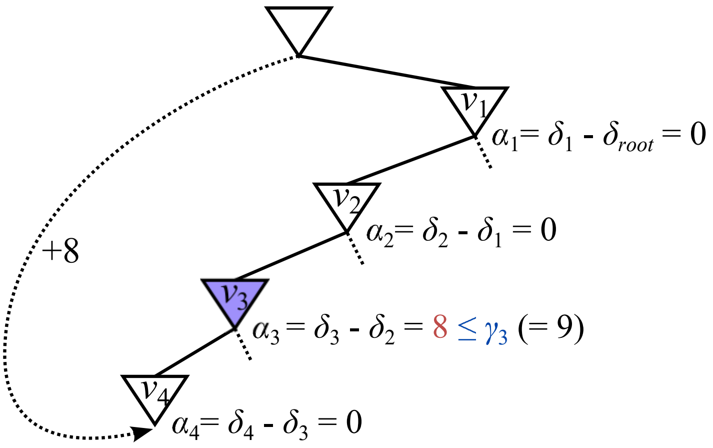


v_1 v_2 v_3 v_4 …
…
cyc-i
…
ADBs do not have size limitation (γ=∞)
v_1 v_2 v_3 v_4 …
…
cyc-i1
cyc-i2
…
ADBs have size limitation

(b) The constraint matrix whose covering solution satisfies the delay increment bounds. $\{v_3\}$ covers *cyc-i1* and *cyc-i2*.



(c) The weight of upward arc starting from v_3 is set to $-\gamma_3 = -9$, instead of being removed.


v_1
$\alpha_1 = \delta_1 - \delta_{root} = 0$
v_2
$\alpha_2 = \delta_2 - \delta_1 = 0$
+8
v_3
$\alpha_3 = \delta_3 - \delta_2 = 8 \le \gamma_3 (= 9)$
v_4
$\alpha_4 = \delta_4 - \delta_3 = 0$

(d) The resulting delay increments satisfy the size limitation.

Figure 5.10: An example showing how ADBs with range $[0, \gamma]$ of delay increments are supported by ADB-UCP.

the time violation corresponding to *cyc-i* with weight of 8 while the delay increments of resulting ADBs never exceed γ values. In Figure 5.10(c), the final constraint graph is shown. The longest path from the root to each node corresponds to δ, whose difference between it and the one of parent node is the α value of ADB, as showin in Figure 5.10(d).

The key idea of this approach is that (1) the delay upper bound of ADBs can be constrained using upward arcs whose weight is $-\gamma$, and (2) a cycle cannot be non-positive if the total weight is still positive even after changing the weight of selected upward arcs from 0 to $-\gamma$. We use ADB-UCP-LM to refer to this extended version of ADB-UCP. Note that ADB-UCP-LM preserves the optimality under the bounded range of delay increment of ADBs.

5.5 Property Proofs of the Proposed Algorithm

Lemma 2. Let G be the difference constraint graph constructed by ADB-UCP for a clock tree $\mathcal{T}$ with an ADB allocation instance I. Then, G contains no positive weight cycle if and only if there exists a delay assignment to the ADBs in I that meets all the setup and hold time constraints.

Proof. δ_i is defined as the sum of α values from the root to v_i, and this can be changed to a definition of α_i which is $\delta_i - \delta_p$ when v_p is the parent of v_i. Thus, when there does not exist a solution of α, there cannot exist a solution of δ, and vice versa.

Let us assume that the constraint graph has a positive weight cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. Then, if we add LHS and RHS of the inequalities $\delta_2 \geq \delta_1 + w(v_1 \to v_2)$, $\delta_3 \geq \delta_2 + w(v_2 \to v_3)$, $\cdots$, $\delta_1 \geq \delta_n + w(v_n \to v_1)$, we get $0 \geq 0 + w(v_1 \to v_2) + w(v_2 \to v_3) + \cdots + w(v_n \to v_1)$. From the assumption that $v_1 \to v_2 \to \cdots \to v_n \to v_1$ is a positive cycle, the RHS of the inequality is positive. Then, the inequality can not be true, so there can not exist any value of δ that satisfies the condition. Thus, if the constraint graph has a positive cycle, there does not exist a feasible assignment solution of α with the current ADB allocation.

On the other hand, if the graph does not have any positive cycle, we can get the value of δ with the longest path length from the root, and we can derive α_i by subtracting δ_p from δ_i when v_p is the parent of v_i. The proof that longest path length can be a solution of δ is provided in Lemma 3.

□

Theorem 3. Any instance of useful skew scheduling problem can be transformed into the instance of problem of removing a minimum number of upward arcs from the constraint graph of the input clock tree. In addition, the starting node of every upward arc, say $v_i \rightarrow v_p$, removed is the location where an ADB is allocated, and its delay value of ADB in v_p can be set to $\delta_i - \delta_p$.

Proof. By Lemma 2, keeping away from making positive cycles in the constraint graph is equal to allocating ADBs to meet all the time constraints. Also, the number of upward arcs is the same as the number of buffers which is not replaced by ADBs. Thus, maximizing the number of upward arcs while preserving the non-existence of positive cycles is maximizing the number of buffers not allocated by ADBs while meeting the time constraints. In other words, minimizing the number of removed upward arcs is minimizing the number of ADBs used. Also, by Lemma 3, δ values can be determined from the longest path length from the root, and by the definition of δ, α value can be calculated as $(\delta_i - \delta_p)$.

□

Lemma 3. If the value of δ_i of node v_i in the constraint graph G produced in Step 3 of ADB-UCP is set to the length of the longest path from the root to v_i, all the timing violations in G can be resolved.

Proof. Let us assume that there is an inequality which is not satisfied when we assign the value of longest path length to δ. We will let the unsatisfied inequality $\delta_j \geq \delta_i +$

$w(v_i \to v_j)$. (Thus, $\delta_j < \delta_i + w(v_i \to v_j)$.) Because there exists an arc from v_i to v_j, there is a path from the root to v_j, passing through the longest path from the root to v_i and arc $v_i \to v_j$. Then the weight of the path is $\delta_i + w(v_i \to v_j)$, which contradicts the assumption that longest path from the root to v_j is δ_j. Thus, all the inequalities are satisfied when we assign the δ value with the longest path length.

□

Theorem 4. The cycle search algorithm in ADB-UCP that uses function EXPANDABLECONSTARCS() guarantees to extract a cycle set that dominates all the positive cycles in G.

Proof. To prove this theorem, we need to use Lemma 4.1, 4.2 and Lemma 5. Let N be the number of constraint arcs in a positive cycle in G. We use induction.

i. For $N = 1$, we can find all positive cycles with only one constraint arc.

ii. Suppose the theorem is correct for $N \leq m$. For a positive cycle with $N = m+1$, by Lemma 5, there is a strongly positive path, P, on the cycle. (A definition of strongly positive path is in Definition 6.) Let e_1, e_2, $\cdots$, e_N in this order be the constraint arcs in P. For $K = 1, \cdots, N$, we check if arc e_{K+1} satisfies the condition: (a) $l(NCA(f(e_1), r(e_K))) < l(NCA(r(e_K), f(e_{K+1})))$, (b) $l(NCA(f(e_{K+1}), r(e_{K+1}))) < l(NCA(r(e_K), f(e_{K+1})))$ or $w(e_{K+1}) > 0$ If an arc does not satisfy the condition, stop.

   Case 1. e_{K+1} $(K < N)$ is the first arc that does not satisfy the condition and the unsatisfied condition is (a):

   Cycle $cyc(e_1, \cdots, e_K)$ dominates the initially given cycle $cyc(e_1, \cdots, e_N)$ by Lemma 4.1. Meanwhile, the given search algorithm in ADB-UCP finds the cycle $cyc(e_1, \cdots, e_K)$. Thus, the algorithm already found the positive cycle that dominates $cyc(e_1, \cdots, e_N)$.

Case 2. e_{K+1} $(K < N)$ is the first arc that does not satisfy the condition but (a) is satisfied (Only (b) is unsatisfied.):

By Lemma 4.2, cycle $cyc(e_1, \cdots, e_K, e_{K+2}, \cdots, e_N)$ dominates the given cycle. Also, because the weight of e_{K+1} is non-positive, the weight of $cyc(e_1, \cdots, e_K, e_{K+2}, \cdots, e_N)$ is positive. Thus, the algorithm is dominated by a smaller cycle whose dominating cycle is found by the algorithm.

Case 3. Every arc in P satisfies the condition:

The given search algorithm in ADB-UCP finds the cycle $cyc(e_1, \cdots, e_N)$.

□

Notations: $f(e)$ and $r(e)$ represent the starting and ending nodes of arc e in G, respectively; $l(v)$ denotes the level of v in G; $NCA(v_1, v_2)$ denotes the nearest common ancestor of v_1 and v_2 in G; an *upward_arc* in G indicates an arc of $v_i \rightarrow v_j$ such that v_j is the parent of v_i.

If $l(v_i) < l(v_j)$, v_i is closer to the root node than v_j does. Because $NCA(v_i, v_j)$ is the ancestor of v_i, it is always true that $l(NCA(v_i, v_j)) < l(v_i)$.

Lemma 4.1. *For* $cyc(e_1, e_2, \cdots, e_N)$*, which represents a cycle in* G *that contains a sequence of constraint arcs* $e_1, e_2, \cdots, e_N$ *in this order and a constraint arc* e_K *(*$2 \leq K \leq N$*) in the cycle,* $l(NCA(f(e_1), r(e_{K-1}))) \geq l(NCA(r(e_{K-1}), f(e_K)))$*,* $\Rightarrow$ $cyc(e_1, \cdots, e_{K-1})$ *dominates the given cycle.*

Proof. The upward arcs of $cyc(e_1, \cdots, e_{K-1})$ are divided into the ones in the path from $f(e_1)$ to $r(e_{K-1})$, and the path from $r(e_{K-1})$ to $f(e_1)$. Any upward arc in the first set is in the given cycle. Also, any element of the second set also dominates in the given cycle, because $NCA(r(e_{K-1}), f(e_K))$ is the ancestor of $NCA(f(e_1), r(e_{K-1}))$ from the assumption. Figure 5.11 shows the case. Thus, under the given condition, $cyc(e_1, \cdots, e_{K-1})$ dominates $cyc(e_1, \cdots, e_K)$. □

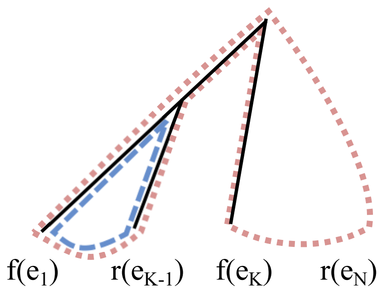


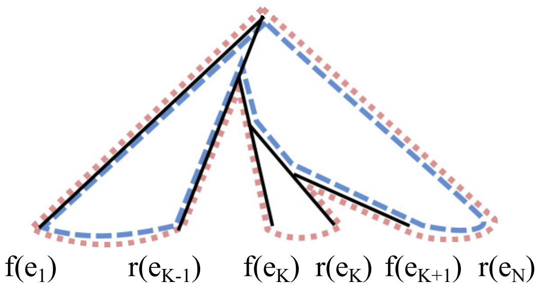
Figure 5.11: A diagram showing the dominance relationship between cycles, when $l(NCA(f(e_1), r(e_{K-1}))) \geq l(NCA(r(e_{K-1}), f(e_K)))$. The given cycle $cyc(e_1, \cdots, e_N)$, marked with (red) dotted line, is dominated by $cyc(e_1, \cdots, e_{K-1})$, which is marked with (blue) dashed line.

Lemma 4.2. *$l(NCA(f(e_1), r(e_{K-1}))) < l(NCA(r(e_{K-1}), f(e_K)))$ and $l(NCA(f(e_K), r(e_K))) \geq l(NCA(r(e_{K-1}), f(e_K)))$,*
$\Rightarrow cyc(e_1, \cdots, e_{K-1}, e_{K+1}, \cdots, e_N)$ dominates the given cycle.

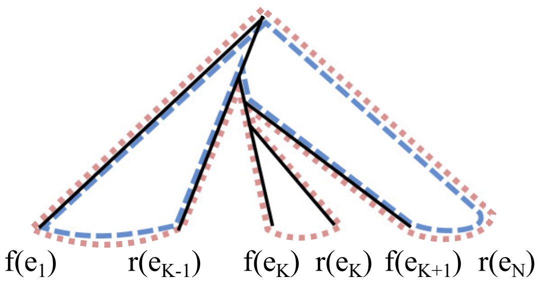
Proof. The upward arcs of $cyc(e_1, \cdots, e_{K-1}, e_{K+1}, \cdots, e_N)$ are divided into two, the ones on the path from $f(e_{K+1})$ to $r(e_{K-1})$, and the others on the path from $r(e_{K-1})$ to $NCA(r(e_{K-1}), f(e_{K+1}))$. Any upward arc in the first set can cut the given cycle. We will also prove that **(*)** every upward arc in the second set cuts the path from $r(e_{K-1})$ to $NCA(r(e_{K-1}), f(e_K))$ of the given cycle, by considering three cases shown in Figure 5.12. By checking the three possible positions $NCA(r(e_K), f(e_{K+1}))$ can have, we can conclude that the assumption **(*)** holds for every case. Thus, with the given condition, $cyc(e_1, \cdots, e_{K-1}, e_{K+1}, \cdots, e_N)$ dominates the given cycle. $\square$

Definition 6. A path P in G is called *strongly positive* if when we perform additions of arc weights from the starting node of P to the ending node, all intermediate values produced are positive.

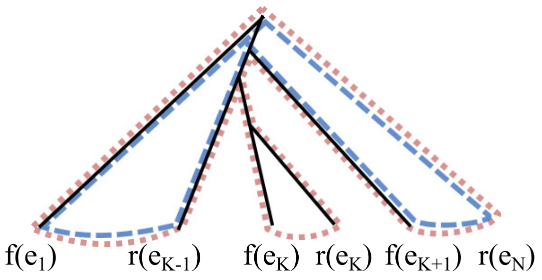
For example, the path marked by arrows in Figure 5.13 is strongly positive because



(a) $NCA(r(e_K), f(e_{K+1}))$ is located below $NCA(f(e_K), r(e_K))$.



(b) $NCA(r(e_K), f(e_{K+1}))$ is located between $NCA(f(e_K), r(e_K))$ and $NCA(r(e_{K-1}), f(e_K))$.



(c) $NCA(r(e_K), f(e_{K+1}))$ is located above $NCA(r(e_{k-1}), f(e_k))$.

Figure 5.12: Diagrams showing the dominance relationship between cycles, when $l(NCA(f(e_1), r(e_{K-1}))) < l(NCA(r(e_{K-1}), f(e_K)))$ and $l(NCA(f(e_K), r(e_K))) \geq l(NCA(r(e_{K-1}), f(e_K)))$. The given cycle $cyc(e_1, \cdots, e_N)$, marked with (red) dotted line, is dominated by $cyc(e_1, \cdots, e_{K-1}, e_{K+1}, \cdots, e_N)$, which marked with (blue) dashed line.

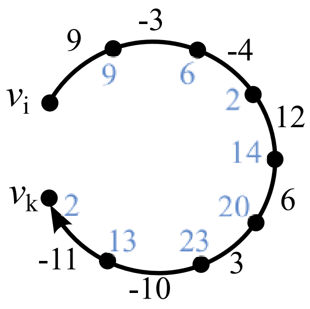


Figure 5.13: An example of strongly positive path. Every node on the path has positive distance from the starting node, v_i.

$9 > 0$, $(9 + (-3)) > 0$, $(9 + (-3) + (-4)) > 0$, $\cdots$, and $(9 + (-3) + (-4) + \cdots + (-11)) > 0$.

Lemma 5. *For every positive cycle in G, it always has a strongly positive path such that all vertices in the cycle appear in the path.*

Proof. A cycle can be decomposed into two types of paths: the maximal length paths whose individual arc weights are either 0 or negative and the rest of paths. We denote the former as $(-)$ and the latter as $(+)$. Then a positive cycle in G can be expressed as $(+) \to (-) \to (+) \to \cdots$. Let n be the number of patterns of $(+) \to (-)$ in the expression. We prove the lemma by induction.

i. If $n = 1$, the path from the starting node in $(+)$ to the ending node in $(-)$ is a strongly positive path.

ii. Suppose the lemma is true for positive cycles with $n \leq N$ and let us assume a positive cycle has $n = N + 1$. Because the cycle has positive total weight, at least one of the patterns of $(+) \to (-)$ has a positive total weight. Thus, we concatenate the pattern $(+) \to (-)$ with the next $(+)$ to the right and replace them with a new $(+)$. Now, we have a cycle with m number of patterns of $(+) \to (-)$. Since there is a strongly positive path for any positive cycle with

$n = N$ by the induction hypothesis, there exists a strongly positive path in the positive cycle.

□

5.6 Experimental Results

Our proposed algorithms ADB-UCP, ADB-UCP-Q (supporting ADBs with discrete delay), ADB-UCP-LM (supporting ADBs with delay upper bound), and the previous linear programming based algorithm ADB-LP [2] have been implemented in C on a Linux machine with 8 cores of 3.50 GHz Intel i7 CPU and 16 GB memory. In the experiments, we used only one core out of the 8 cores. ISCAS'89 benchmarks were synthesized with *Synopsys IC Compiler* with 45 nm *NanGate Open Cell library* [30] and partitioned into 5 to 8 power domains. Power domains operate in different supply voltage levels ranging from 0.85 V to 1.05 V. LP problems were solved using *GLPK* library, and UCP solver was implemented in C using dominance relationship and Petrick's method. Table 5.1 shows the number of clock buffers, clock sinks, and timing constraints of the benchmark along with T_{clk} used for the experiment.

Table 5.2 shows the number of ADBs allocated, the number of time violations, and runtime used by ADB-LP [2] and ADB-UCP. ADB-LP repeatedly applies LP solver, adding one ADB at a time in a greedy manner, to reduce the clock period while meeting the setup and hold time constraints. Note that the ADB allocation by ADB-LP does not inherently guarantee complete elimination of hold time violations, while ADB-UCP and ADB-UCP-Q do. Thus, for meaningful comparison purpose, we used two versions of our algorithms, the original one and the modified version which ignores hold time constraints that were violated in ADB-LP.

Table 5.3 includes the allocation results produced by the modified version of ADB-UCP, which ignores the hold time constraints that were violated in results generated by ADB-LP. Our algorithm consistently allocates less ADBs than ADB-LP, reducing

Table 5.1: Benchmark circuits used in the experiment

Circuit Name	#.Clock Bufs	#.Clock Sinks	#.Timing CSTs
s382	12	21	348
s386	3	6	84
s1196	9	18	46
s1238	8	18	52
s1423	36	74	4256
s1494	2	6	110
s5378	89	163	2868
s13207	186	330	2830
s15850	66	134	1452
s38417	913	1564	74106
s38584	683	1168	21750
s35932	996	1728	9018

Table 5.2: Comparison of results produced by ADB-LP [2] that considers useful skew scheduling, our ADB-UCP without allowing timing violations

Circuit	T_{clk}	ADB-LP [2]			ADB-UCP without violation		
		#.ADBs	#.Vio.	Runtime	#.ADBs	#.Vio.	Runtime
s382	1378 ps	6	0	0.5 s	6	0	< 0.1 s
s386	1585 ps	2	0	< 0.1 s	2	0	< 0.1 s
s1196	1395 ps	6	0	0.1 s	6	0	< 0.1 s
s1238	1436 ps	7	0	< 0.1 s	6	0	< 0.1 s
s1423	2681 ps	5	1	8.6 s	5	0	< 0.1 s
s1494	2072 ps	2	0	< 0.1 s	2	0	< 0.1 s
s5378	2288 ps	6	114	22.1 s	6	0	< 0.1 s
s13207	1852 ps	3	231	92.2 s	15	0	0.9 s
s15850	1903 ps	10	9	24.7 s	11	0	0.3 s
s38417	3365 ps	6	3528	2322.1 s	24(I)[a]	0	80.5 s
s38584	2544 ps	10	851	1566.2 s	33	0	9.7 s
s35932	2137 ps	5	583	2162.6 s	34	0	11.8 s
Average (%)		100	100	-	165.5	0	-

[a] 'I' means that the solution is a result of iterative algorithm, so its optimality is not guaranteed.

Table 5.3: Comparison of results produced by ADB-LP [2] that considers useful skew scheduling, our ADB-UCP allowing timing violations

Circuit	T_{clk}	ADB-LP [2]			ADB-UCP with violation		
		#.ADBs	#.Vio.	Runtime	#.ADBs	#.Vio.	Runtime
s382	1378 ps	6	0	0.5 s	6	0	< 0.1 s
s386	1585 ps	2	0	< 0.1 s	2	0	< 0.1 s
s1196	1395 ps	6	0	0.1 s	6	0	< 0.1 s
s1238	1436 ps	7	0	< 0.1 s	6	0	< 0.1 s
s1423	2681 ps	5	1	8.6 s	4	1	< 0.1 s
s1494	2072 ps	2	0	< 0.1 s	2	0	< 0.1 s
s5378	2288 ps	6	114	22.1 s	3	64	< 0.1 s
s13207	1852 ps	3	231	92.2 s	1	226	0.1 s
s15850	1903 ps	10	9	24.7 s	9	9	0.3 s
s38417	3365 ps	6	3528	2322.1 s	6	2134	26.8 s
s38584	2544 ps	10	851	1566.2 s	5	825	3.4 s
s35932	2137 ps	5	583	2162.6 s	4	544	13.9 s
Average (%)		100	100	-	76.7	90.7	-

the number by 23.3% on average.

Table 5.4: The number of ADBs allocated by ADB-UCP on ISCAS'89 S38417, S38584 and S35932 with varying values of $limit$

Circuit	$limit_s$ / $limit_l$	Number of ADBs			
		1000	4000	16000	64000
s38417 (T_{clk}=3764 ps)	1	24	24	24	24
	4	24	24	24	24
	16	24	24	24	24
	64	24	24	24	24
s38584 (T_{clk}=2761 ps)	1	33	33	33	33
	4	33	33	33	33
	16	33	33	33	33
	64	33	33	33	33
s35932 (T_{clk}=2257 ps)	1	36	35	35	34
	4	36	35	35	34
	16	36	35	35	34
	64	36	35	35	34

Table 5.4 and Table 5.5 summarizes the results by ADB-UCP performed with the change of $limit$ value. $limit_s$ denotes the number of positive cycles aimed to find in the first iteration, and $limit_l$ is the number used in the subsequent iterations. While the algorithm allocate the same number of ADBs on S38417 and S38584 no matter what the value of $limit$ is, more ADBs were allocated on S35932 when the less effort was imposed. It is shown that $limit_l$ does not seriously affect the quality of the solution but increases the number of iteration and runtime sharply, as shown in Table 5.5. Thus,

Table 5.5: Runtime and the number of iterations of ADB-UCP on ISCAS'89 $\mathrm{s}38417$, $\mathrm{s}38584$ and $\mathrm{s}35932$ with varying values of $limit$

Circuit	$limit_s$ \ $limit_l$	Runtime (sec) / Number of iterations			
		1000	4000	16000	64000
s38417 (T_{clk}=3764 ps)	1	29.1 / 5	31.6 / 3	59.4 / 3	187.5 / 2
	4	35.9 / 4	83.0 / 2	110.4 / 2	195.8 / 2
	16	32.6 / 3	83.4 / 2	110.4 / 2	196.1 / 2
	64	46.4 / 3	83.5 / 2	110.7 / 2	195.4 / 2
s38584 (T_{clk}=2761 ps)	1	5.8 / 3	6.0 / 1	17.9 / 1	70.2 / 1
	4	4.6 / 2	6.0 / 1	17.9 / 1	70.1 / 1
	16	4.7 / 2	6.0 / 1	17.9 / 1	70.1 / 1
	64	5.8 / 2	6.0 / 1	17.9 / 1	70.1 / 1
s35932 (T_{clk}=2257 ps)	1	13.5 / 7	8.8 / 2	25.4 / 2	126.1 / 1
	4	124.9 / 4	102.1 / 2	138.2 / 2	126.3 / 1
	16	127.3 / 4	102.3 / 2	138.1 / 2	126.4 / 1
	64	125.5 / 3	102.1 / 2	138.4 / 2	126.2 / 1

one guideline is to set $limit_s$ to a large value and then set $limit_l$ to a value, which is much smaller than $limit_s$.

Table 5.6: The number of ADBs allocated by ADB-UCP-Q, under different settings of quantization resolution

Circuit	T_{clk}	ADB-UCP	ADB-UCP-Q (Q: unit of delay increment)			
			100 ps	200 ps	300 ps	400 ps
s382	1723 ps	0	0	0	0	0
s386	1982 ps	0	0	0	0	0
s1196	1744 ps	3	3	3	3	-
s1238	1795 ps	2	3	4	4	-
s1423	3103 ps	2	2	3	5	9
s1494	2590 ps	0	0	0	0	0
s5378	2586 ps	6	6	8	-	-
s13207	2062 ps	15	25	-	-	-
s15850	2251 ps	4	5	-	-	-
s38417	3764 ps	24	38	-	-	-
s38584	2761 ps	33	-	-	-	-
s35932	2257 ps	34	36	-	-	-

Table 5.6 and Table 5.7 summarizes the results produced by our ADB-UCP, ADB-UCP-Q and ADB-UCP-LM. The experiments showed that the results of ADB-UCP-Q and ADB-UCP-LM satisfy the given constraints, quantized delay value and the size limitation. ADB-UCP-Q uses more ADBs as the delay quantization unit increases, and sometimes cannot produce the result that meets all the constraints and the delay quantization constraint. The application of ADB-UCP-LM by varying the upper delay limit (i.e., γ) from 200 ps to 1000 ps gradually reduces the number of

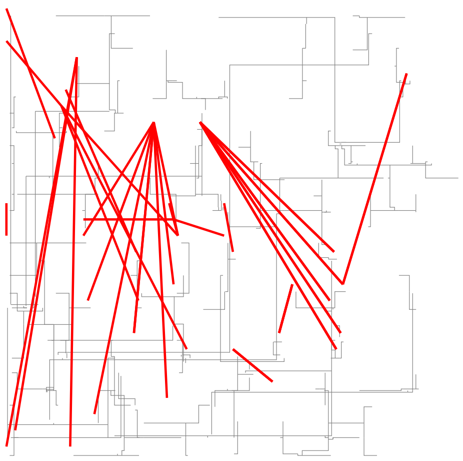
Table 5.7: The number of ADBs allocated by ADB-UCP-LM, under different settings of delay upper limit

Circuit	T_{clk}	ADB-UCP	ADB-UCP-LM (γ: upper limit)				
			200 ps	400 ps	600 ps	800 ps	1000 ps
s382	1723 ps	0	0	0	0	0	0
s386	1982 ps	0	0	0	0	0	0
s1196	1744 ps	3	-	4	3	3	3
s1238	1795 ps	2	-	2	2	2	2
s1423	3103 ps	2	2	2	2	2	2
s1494	2590 ps	0	0	0	0	0	0
s5378	2586 ps	6	8	6	6	6	6
s13207	2062 ps	15	18	15	15	15	15
s15850	2251 ps	4	4	4	4	4	4
s38417	3764 ps	24	34	25	24	24	24
s38584	2761 ps	33	39	33	33	33	33
s35932	2257 ps	34	40	34	34	34	34

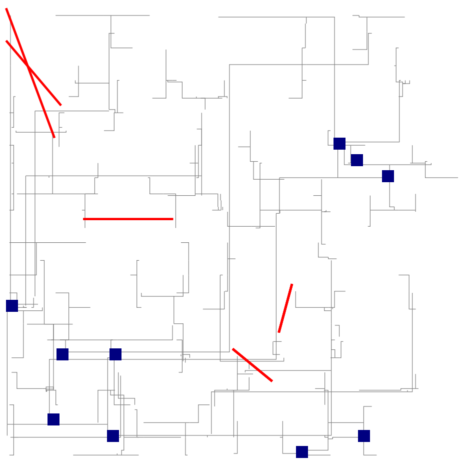
ADBs allocated. Notation '-' indicates that ADB-UCP-Q or ADB-UCP-LM reports any of the ADB allocation cannot resolve all the time violation. As the constraints on available delay increment become tighter, the algorithms fail to find the solution more frequently.

Figure 5.14 shows layout comparison of the benchmark circuit *s15850* for the initial tree, the result produced by ADB-LP [2], and by ADB-UCP with and without violation. Blue dots indicate ADBs and red lines indicate time violation. It is confirmed that the result in Figure 5.14(b) has no timing violations at all, and Figure 5.14(c) has less timing violations than that of Figure 5.14(a) while using less number of ADBs.

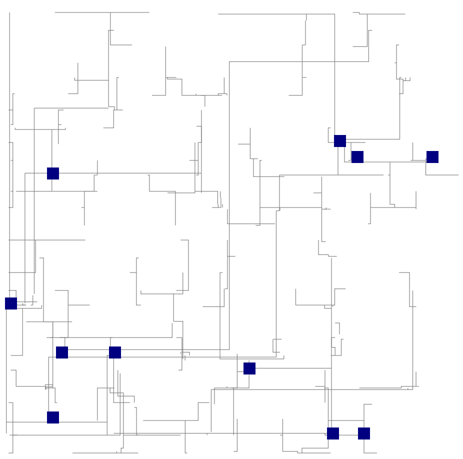
In summary, the comparison confirms a set of results: (1) ADB-UCP *never produces time violation* while ADB-LP [2] and the algorithms under clock skew bounds do; (2) ADB-UCP partially allowing the hold time violations that occur in ADB-LP *reduces the number of ADBs by 23.3%* on average over that of ADB-LP; (3) ADB-UCP runs *30∼460 times faster* for large designs than ADB-LP does; (4) ADB-UCP-Q and ADB-UCP-LM find the solutions which support the ADBs with discrete delay increment and maximum delay limitation, respectively.



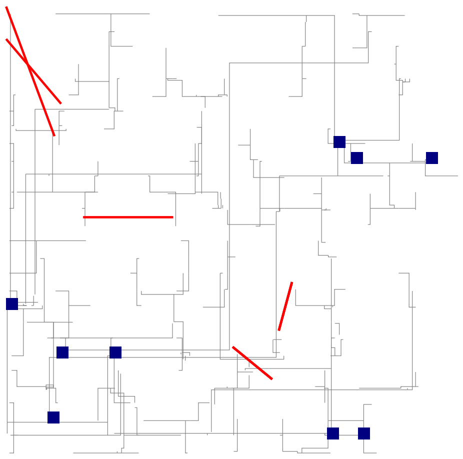
(a) The initial violation.



(b) ADB allocation result produced by ADB-LP [2]. (#. ADB= 10)



(c) ADB allocation result produced by ADB-UCP without violation. (#. ADB= 11)



(d) ADB allocation result produced by ADB-UCP with violation allowed. (#. ADB= 9)

Figure 5.14: ADB allocation result on ISCAS'89 s15850 design. Blue dots indicate ADBs and red lines indicate time violation between sinks.

Chapter 6

CONCLUSION

This dissertation presents solutions of adjustable delay buffer (ADB) allocation problem which is to correct timing violations under varying operating environment, especially in multiple power mode designs.

The work proposes a polynomial-time optimal algorithm to reduce the area overhead while ensuring the clock skew bounded by the given constant value, as well as providing the way of handling practical issues of supporting quantized delay values and reducing more area by replacing the ADBs with other buffers. From the experimental results on benchmarks, it was shown that our proposed algorithm uses, under 30 ps$\sim$50 ps clock skew bound, 13.5% and 15.8% fewer numbers of ADBs for continuous and discrete ADB delays on average, respectively, compared to the results by the best known ADB allocation algorithm. In addition, when buffer sizing is integrated, our algorithm reduces the area of ADBs and buffers by 15.0% and 16.3% for continuous and discrete ADB delays, respectively.

We also suggest a graph based algorithm for solving the ADB allocation problem effectively under useful clock skew scheduling, along with the acceleration techniques to enable the trade-off between the runtime and the optimality. The algorithm can be extended to support quantized delay values of the ADBs or the case ADBs have limitation on maximum delay increment. The experiments with benchmark circuits showed that our algorithm reduces the number of ADBs by 23.3% on average over the results produced by the conventional ADB allocation under useful clock skew scheduling under the same constraints. In addition, our optimal algorithm runs 30$\sim$460 times faster than the prior work.

The proposed methods will enable timing correction in multiple power mode design utilizing ADBs adding small overhead. The theoretical outcomes of this work can also be applied usefully to the diverse environments, for instance, non-uniform thermal effect with the dynamically varying clock skew.

Bibliography

[1] K.-H. Lim, D. Joo, and T. Kim, “An optimal allocation algorithm of adjustable delay buffers and practical extensions for clock skew optimization in multiple power mode designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 392–405, Mar. 2013.

[2] H.-M. Chou, H. Yu, and S.-C. Chang, “Useful-skew clock optimization for multi-power mode designs,” in *Proceedings of the 2011 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2011, pp. 647–650.

[3] N. J. A. Kapoor and S. P. Khatri, “A novel clock distribution and dynamic de-skewing methodology,” in *Proceedings of the 2004 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2004, pp. 626–631.

[4] R. S. Tsay, “Exact zero skew,” in *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, Nov. 1991, pp. 336–339.

[5] T.-H. Chao, Y.-C. Hsu, J.-M. Ho, and A. B. Kahng, “Zero skew clock routing with minimum wirelength,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 11, pp. 799–814, Nov. 1992.

[6] M. Edahiro, “A clustering-based optimization algorithm in zero-skew routings,” in *Proceedings of the 30th IEEE/ACM Design Automation Conference*, Jun. 1993, pp. 612–616.

[7] A. B. Kahng and C.-W. A. Tsao, “Planar-dme: a single-layer zero-skew clock tree router,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 1, pp. 8–19, Jan. 1996.

[8] J. Cong, A. B. Kahng, C.-K. Koh, and C.-W. A. Tsao, “Bounded-skew clock and steiner routing,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 3, pp. 341–388, Jul. 1998.

[9] C.-W. A. Tsao and C.-K. Koh, “Ust/dme: a clock tree router for general skew constraints,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 359–379, Jul. 2002.

[10] J. Cong, C.-K. Koh, and K.-S. Leung, “Simultaneous buffer and wire sizing for performance and power optimization,” in *Proceedings of the 1996 ACM/IEEE International Symposium on Low Power Electronics and Design*, Aug. 1996, pp. 271–276.

[11] T. Okamoto and J. Cong, “Buffered steiner tree construction with wire sizing for interconnect layout optimization,” in *Proceedings of the 1996 IEEE/ACM International Conference on Computer Aided Design*, Nov. 1996, pp. 44–49.

[12] C. J. Alpert, A. Devgan, and S. T. Quay, “Buffer insertion with accurate gate and interconnect delay computation,” in *Proceedings of the 36th IEEE/ACM Design Automation Conference*, Jun. 1999, pp. 479–484.

[13] C. C. N. Chu and D. F. Wong, “An efficient and optimal algorithm for simultaneous buffer and wire sizing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 9, pp. 1297–1304, Sep. 1999.

[14] I.-M. Liu, T.-L. Chou, A. Aziz, and D. F. Wong, “Zero-skew clock tree construction by simultaneous routing, wire sizing and buffer insertion,” in *Proceedings of the 2000 ACM International Symposium on Physical Design*, May 2000, pp. 33–38.

[15] J.-L. Tsai, T.-H. Chen, and C.-P. Chen, "Zero skew clock-tree optimization with buffer insertion/sizing and wire sizing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 565–572, Apr. 2004.

[16] K. Wang, Y. Ran, H. Jiang, and M. Marek-Sadowska, "General skew constrained clock network sizing based on sequential linear programming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, pp. 773–782, May 2005.

[17] S. Tam, S. Rusu, U. Nagarji Desai, R. Kim, J. Zhang, and I. Young, "Clock generation and distribution for the first IA-64 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1545–1552, Nov. 2000.

[18] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, Apr. 1992.

[19] M. Weiser, B. Welch, A. Demers, and S. Shenker, *Scheduling for reduced CPU energy*. Boston, MA: Springer US, 1996, pp. 449–471.

[20] Y.-S. Su, W.-K. Hon, C.-C. Yang, S.-C. Chang, and Y.-J. Chang, "Value assignment of adjustable delay buffers for clock skew minimization in multi-voltage mode designs," in *Proceedings of the 2009 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2009, pp. 535–538.

[21] ——, "Clock skew minimization in multi-voltage mode designs using adjustable delay buffers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 12, pp. 1921–1930, Dec. 2010.

[22] J. P. Fishburn, "Clock skew optimization," *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 945–951, Jul. 1990.

[23] S. Hu and J. Hu, "Unified adaptivity optimization of clock and logic signals," in *Proceedings of 2007 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2007, pp. 125–130.

[24] V. Khandelwal and A. Srivastava, "Variability-driven formulation for simultaneous gate sizing and postsilicon tunability allocation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 610–620, Apr. 2008.

[25] E. Takahashi, Y. Kasai, M. Murakawa, and T. Higuchi, "A post-silicon clock timing adjustment using genetic algorithms," in *2003 Symposium on VLSI Circuits Digest of Technical Papers*, Jun. 2003, pp. 13–16.

[26] J.-L. Tsai and L. Zhang, "Statistical timing analysis driven post-silicon-tunable clock-tree synthesis," in *Proceedings of the 2005 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2005, pp. 575–581.

[27] J. Kim, D. Joo, and T. Kim, "An optimal algorithm of adjustable delay buffer insertion for solving clock skew variation problem," in *Proceedings of the 50th IEEE/ACM Design Automation Conference*, Jun. 2013, pp. 1–6.

[28] ——, "Optimal utilization of adjustable delay clock buffers for timing correction in designs with multiple power modes," *Integration, the VLSI journal*, vol. 52, no. Supplement C, pp. 91–101, Jan. 2016.

[29] K.-Y. Lin, H.-T. Lin, and T.-Y. Ho, "An efficient algorithm of adjustable delay buffer insertion for clock skew minimization in multiple dynamic supply voltage designs," in *Proceedings of the 2011 IEEE Asia and South Pacific Design Automation Conference*, Jan. 2011, pp. 825–830.

[30] "Nangate 45nm open cell library," http://www.nangate.com/.

[31] T.-Y. Kim and T. Kim, "Clock tree synthesis for TSV-based 3D IC designs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 4, pp. 48:1–48:21, Oct. 2011.

[32] J. Kim and T. Kim, "Useful clock skew scheduling using adjustable delay buffers in multi-power mode designs," in *Proceedings of the 2015 IEEE Asia and South Pacific Design Automation Conference*, Jan. 2015, pp. 466–471.

[33] ——, "Adjustable delay buffer allocation under useful clock skew scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 641–654, Apr. 2017.

[34] R. B. Deokar and S. S. Sapatnekar, "A graph-theoretic approach to clock skew optimization," in *Proceedings of the 1994 IEEE International Symposium on Circuits and Systems*, May 1994, pp. 407–410.

[35] J. G. Xi and W. W.-M. Dai, "Useful-skew clock routing with gate sizing for low power design," in *Proceedings of the 33rd IEEE/ACM Design Automation Conference*, Jun. 1996, pp. 51–67.

[36] X. Liu, M. C. Papaefthymiou, and E. G. Friedman, "Maximizing performance by retiming and clock skew scheduling," in *Proceedings of the 36th IEEE/ACM Design Automation Conference*, Jun. 1999, pp. 231–236.

[37] J.-L. Tsai, D. H. Baik, and C.-P. Chen, "A yield improvement methodology using pre- and post-silicon statistical clock scheduling," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2004, pp. 611–618.

[38] V. Nawale and T. W. Chen, "Optimal useful clock skew scheduling in the presence of variations using robust ilp formulations," in *Proceedings of the 2006 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2006, pp. 27–32.

[39] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Boston, MA: Springer US, 1996.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.

초 록

클락의 타이밍은 동기 회로의 성능에 큰 영향을 끼치므로, 클락 분배 네트워크를 최적화하기 위해 많은 합성 기술들이 제안되어 왔다. 특히, 클락 스큐 제한조건을 만족시키는 것은 회로를 성공적으로 동작시키기 위해 고려해야 할 중요한 문제 중 하나이다. 한편, 다중 전압 환경에서는 인가 전압에 따라 클락 지연 시간이 계속해서 변화하므로, 이를 사용하는 것은 클락 타이밍 문제를 더욱 해결하기 어렵게 하고 있다. 가변 지연 시간 버퍼를 클락 트리에 삽입하고 그 지연 시간을 조절함으로써 이 문제를 해결할 수 있으나, 이는 회로의 면적과 제어에 무시할 수 없는 추가 부담을 발생시킨다. 이 연구는 모드 변화에 따라 클락 지연 시간이 변하는 환경인 다중 전압 모드 설계에서 가변 지연 시간 버퍼를 최소화하는 해결 방안을 제안한다. 세부적으로는, (1) 클락 스큐를 주어진 값 이내로 줄이기 위한 O($n \log n$) 시간에 동작하는 최적 알고리즘과, (2) 유용한 스큐를 얻을 수 있도록 계획하는 그래프 기반의 알고리즘을 제시한다. 또한 (3) 가변 지연 시간 버퍼가 생성 가능한 지연 시간이 제한되어 있을 때 대응하는 방법이나 다른 버퍼 크기 조절을 통해 가변 지연 시간 버퍼의 수를 더욱 줄이는 등의 실용적인 확장 방법 또한 제안하였다. 기존 알고리즘과의 비교 실험에서는 클락 스큐의 크기 제한 조건 하에서, 혹은 유용한 클락 스큐를 사용할 수 있도록 하는 조건 하에서 제안된 알고리즘은 각각 평균적으로 13.5%, 23.3% 더 적은 수의 가변 지연 시간 버퍼를 할당하였다.

주요어: 가변 지연 시간 버퍼, 클락 네트워크 디자인, 다중 전압 환경, 클락 스큐
학번: 2013-20776