國立交通大學

電子工程學系 電子研究所碩士班

碩 士 論 文

應用於查找表式場域可程式化閘陣列之壓縮樹延遲最佳化合成演算法

Delay Optimal Compressor Tree Synthesis for LUT-Based FPGAs

研 究 生：呂智宏

指導教授：周景揚 博士

黃俊達 博士

中 華 民 國 九 十 八 年 七 月

應用於查找表式場域可程式化閘陣列之壓縮樹延遲最佳化合成演算法

Delay Optimal Compressor Tree Synthesis for LUT-Based FPGAs

研 究 生：呂智宏　　Student: Jhig-Hong Lu

指導教授：周景揚 博士　　Advisor: Dr. Jing-Yang Jou

黃俊達 博士　　Advisor: Dr. Juinn-Dar Huang

國 立 交 通 大 學

電子工程學系電子研究所碩士班

碩 士 論 文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Electronics Engineering

July 2009

Hsinchu, Taiwan, Republic of China

中華民國 九十八年七月

應用於查找表式場域可程式化閘陣列之壓縮樹延遲最佳化合成演算法

研究生：呂智宏

指導教授：周景揚博士,黃俊達博士

國立交通大學

電子工程學系 電子研究所碩士班

摘要

在以查找表(Lookup table)為基礎的可程式邏輯陣列(FPGA)架構下，我們提出一個壓縮樹合成演算法(DOCT)。此演算法的主要目的是為了達到延遲最佳化。首先，在給定查找表的輸入限制之下，此演算法會先產生一組相對應的元素樣本集合，再藉著這些元素樣本，用整數線性規劃法(ILP)去合成出延遲最佳化的壓縮樹。並且在不失去延遲最佳化的特性下，更進一步用一套後製程序去降低壓縮數所需要的面積。在實驗部分，我們把結果跟另一個演算法(GPC)做比較。結果顯示，在現今的製程技術下，我們的延遲平均降低 32%，而面積平均降低 21%。

Delay Optimal Compressor Tree Synthesis for LUT-Based FPGAs

Student: Jhih-Hong Lu

Advisor: Dr. Jing-Yang Jou, Dr. Juinn-Dar Huang

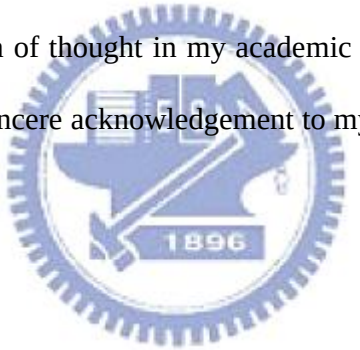
Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

Abstract

In this thesis, we present a *compressor tree* synthesis algorithm, named DOCT, which guarantees the delay optimal implementation in lookup-table (LUT) based FPGAs. Given a targeted *K*-input LUT architecture, DOCT firstly derives a finite set of prime patterns as essential building blocks. Then, it shows that a delay optimal compressor tree can always be constructed by those derived prime patterns via integer linear programming (ILP). Without loss of delay optimality, a post-processing procedure is invoked to reduce the number of demanded LUTs for the generated compressor tree design. DOCT has been evaluated over a broad set of benchmark circuits. Compared to the previous heuristic approach, the experimental results show that DOCT reduces the depth of the compressor tree by 32%, and the number of LUTs by 21% on average based on the modern 6-input LUT-based FPGA architecture.

Acknowledgment

At first, I deeply appreciate my advisor Professor Jing-Yang Jou. Not only did he made many beneficial suggestions for me, but also provide a resource-intensive environment. I am also really thankful to my co-guidance advisor Professor Juinn-Dar Huang for his guidance. He is a constant inspiration to me. I would like to thank Bu-Ching Lin and Wan-Hsien Lin. Without their support, I could not finish my research. Thanks to Yu-Shiang Wang, Ji-Huei Li, and Wan-Ling Shiu, for their friendship and encouragement. And thanks to all members of EDA laboratory. During the past two years, each one of them was never afraid to offer their opinions. This kind of favor enlivened my train of thought in my academic field. Finally and especially, I would like to express my sincere acknowledgement to my family and all my friends for their aid.



Content

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Technology Trend

As the manufacturing cost and time-to-market pressure of developing ASIC/SoC increase, the design and verification processes demand a better way to reduce the development cost. The advantages of low risk, low NRE cost, and fast time-to-market, have made FPGA a significant alternative for the electronic system design, and then FPGA is usually used for flexible and low-volume applications without regard to system performance. However, a high-performance system can be possibly implemented with modern FPGAs, since an FPGA device can have a very rich set of logic elements and very high-speed I/O interfaces under recent DSM technologies [1-4]. The arithmetic circuit, rather than the control-dominated circuit, is often the performance bottleneck for a high-performance system implemented with FPGAs [5, 6] and thus this thesis presents an algorithm for the delay optimal compressor tree synthesis.

1.2 Previous Works

A compressor tree is used to implement a multi-operand addition, which is one of the essential operations in most DSP applications, e.g., FIR/IIR filters [7], discrete cosine transform (DCT), multipliers, multiplier-accumulators (MAC) [8], motion estimation (ME) [9], and so on. In ASIC designs, the concept of the

compressor tree has been introduced by Wallace and Dadda for more than 40 years [10, 11]. An efficient method, called Three-Greedy Approach, is proposed in [12] for the delay-optimized compressor tree synthesis. A delay optimal algorithm without considering wire delay is further presented in [13]. All the building blocks for the above two researches are restricted to full-adders and half-adders. However, since the basic programmable logic block in a modern FPGA is a *K*-input LUT ($K = 5$ or 6), 3-input full-adders and 2-input half-adders are apparently not the appropriate building blocks for compressor tree synthesis from both area and delay perspectives.

Algorithms proposed in [14] and [15] have made significant progresses in reducing the delay of the compressor tree in FPGA designs. Although the GPC heuristic has achieved reasonably good results [14], its inherently heuristic nature cannot guarantee optimal solutions. Moreover, an algorithm which utilizes a set of GPC patterns is presented in [15] for the delay-optimized compressor tree construction via integer linear programming (ILP). However, this method does not consider all valid GPC patterns under a given input constraint (i.e., *K*); therefore, it cannot guarantee optimal solutions, either. According to [15], the compressor trees synthesized by the above two algorithms even have the same depth in many cases.

A hybrid architecture is presented to obtain advantages of both ASIC and FPGA technologies [16]. ASICs offer advantages of density and performance. On the other hand, FPGAs offer advantages of flexibility and fast time-to-market. By the same manner, hard configurable IP cores are developed to integrate into FPGAs for accelerating the speed of compressor trees [17, 18]. But this kind of approaches is beyond the scope of this thesis.

1.3 Contribution

In this thesis, we present a delay optimal compressor tree synthesis algorithm, named DOCT, for LUT-based FPGAs. It firstly derives a set of prime patterns as essential building blocks, and then utilizes them to construct the delay optimal compressor tree via ILP. Besides, a post-processing procedure is invoked to minimize the number of demanded LUTs without loss of delay optimality. Compared to the GPC heuristic [14], the experimental results show that DOCT reduces the depth of the compressor tree by 32% and the number of LUTs by 21% on average based on the modern 6-input LUT-based FPGA architecture.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Terminology, definitions, fundamental theorems, and problem formulation are introduced in Chapter 2. Chapter 3 details the proposed delay optimal compressor tree synthesis algorithm with ILP formulation. The experimental results are then presented in Chapter 4. Finally, Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

2.1 Compressor Trees

A compressor tree is a circuit dealing with a multi-operand addition. Before 1960s, the multi-operand addition was often accumulated by the carry-propagate adder (CPA). To minimize the delay of the carry chain produced by several CPAs, Wallace and Dadda proposed an efficient implementation in 1960s to reduce all partial products into two partial products by full-adders and half-adders, and to add the final two partial products by a CPA. Three reduction rules are used for constructing compressor trees: (i) any three dots with the same rank can be mapped onto a full adder, (ii) the remaining two dots with the same rank can be mapped onto a half adder or passed to the next stratum, and (iii) the last dots are directly passed to the next stratum. The full adder acts as a 3:2 counter to add as many dots as possible with the same rank. Figure 2.1 shows an example of a compressor tree

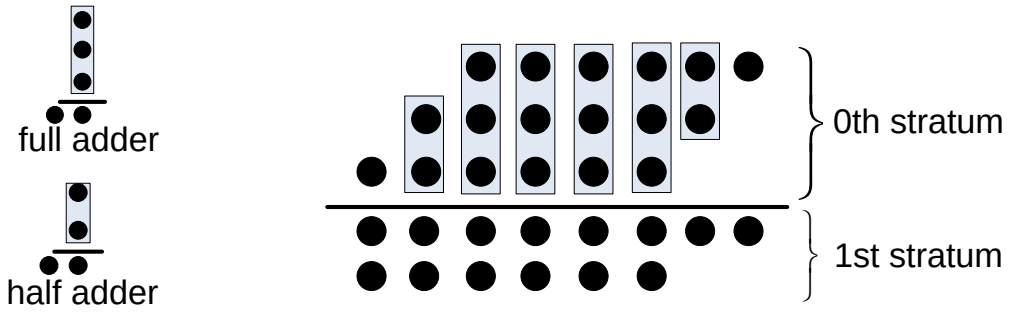


Figure 2.1 An example of a compressor tree on ASICs.

on ASICs, which reduces three partial products into two partial products.

2.2 Definitions

Firstly, this subsection describes a formal expression to characterize the topology of the compressor tree. A compressor tree consists of a series of strata. Each stratum is represented by a dot plane. A *dot plane* with respect to the *s*-th stratum is denoted as an *n*-tuple $d_s = < t_{n-1}, t_{n-2}, ..., t_0 > \in Z^+ \times N^{n-2} \times Z^+$, where N is the set of non-negative integers, Z^+ is the set of positive integers, and t_i indicates the number of dots which is in the *i*-th column of the dot plane on the *s*-th stratum of the compressor tree. The set of dot planes is defined as D, and then the function

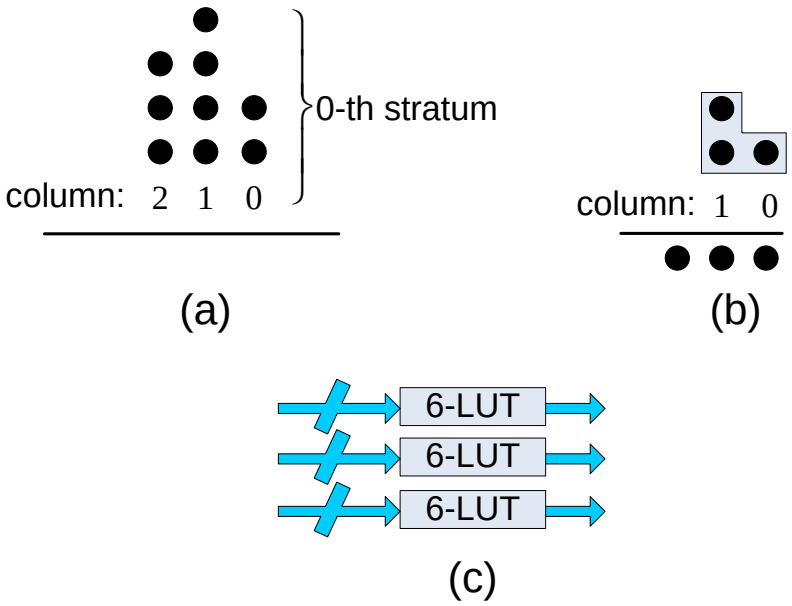


Figure 2.2 (a) A dot plane $d_0 = < 3,4,2 >$. (b) A pattern $< 2,1 >_p \in PS(3)$. (c) The pattern $< 2,1 >_p$ is mapped onto 3 6-input LUTs.

$r: D \times N \to N$ can indicate the *i*-th *element* of the dot plane d_s by $r(d_s, i) = t_i$. The function $w: D \to Z^+$ defines the *width* of each dot plane such that $w(d_s) = n$ iff d_s is an n-tuple; meanwhile, the function $h: D \to Z^+$ defines the *height* of the dot plane d_s according to $h(d_s) = \max\{r(d_s, i) \mid 0 \le i < w(d_s)\}$. Figure 2.2(a) provides an illustrative example as follows. The dot plane $d_0 = <3,4,2>$ is the input of a compressor tree consisting of three columns: two dots in the 0th column, four dots in the 1st column, and three dots in the 2nd column. Therefore, the height and the width of the dot plane d_0 are $h(d_0) = \max\{3,4,2\} = 4$ and $w(d_0) = 3$, respectively.

The following subsection describes a formal expression to characterize the *pattern*. A pattern is denoted as an m-tuple $p = <t_{m-1}, t_{m-2}, ..., t_0>_p \in Z^+ \times N^{m-2} \times Z^+$, where t_j indicates the number of dots which is in the *j*-th column of the pattern *p*. The set of patterns is denoted as *P*, and then the function $v: P \times N \to N$ can indicate the *j*-th element of the pattern *p* by $v(p, j) = t_j$. The function $iw: P \to Z^+$ defines the number of input columns of each pattern, i.e., $iw(p) = m$ iff $p = <t_{m-1}, t_{m-2}, ..., t_0>_p$. All patterns have the corresponding number of their

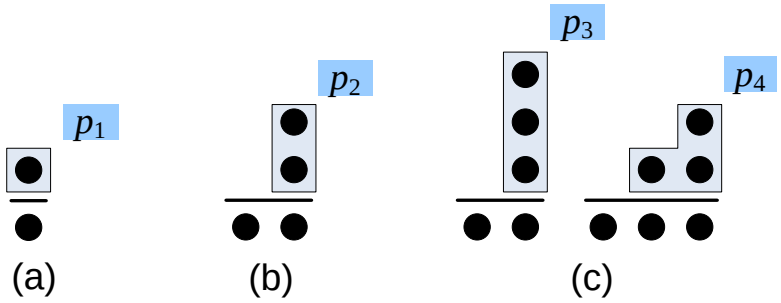


Figure 2.3 (a) *PPS*(1). (b) *PPS*(2). (c) *PPS*(3).

output bits. Thus the function $ow: P \to Z^+$ calculates the minimal number of the required output bits by $ow(p) = \lceil \log_2(\sum_{i=0}^{iw(p)-1} v(p,i) \times 2^i) \rceil$. A pattern is similar to a counter in functionality, but it can sum inputs with value 1 in different ranks. For example, a 3:2 counter like p_3 as shown in Figure 2.3(c) sums three rank-0 inputs while the pattern $<2,1>_p$ as shown in Figure 2.2(b) sums two rank-1 inputs and one rank-0 input. Furthermore, the number of input columns of the pattern $<2,1>_p$ is $iw(<2,1>_p) = 2$, and the number of its output bits is $ow(<2,1>_p) = 3$. The function $PS: Z^+ \to \mathcal{P}(P)$ points out the power set of patterns such that $p \in PS(k)$ iff $\sum_{i=0}^{iw(p)-1} v(p,i) = k$, e.g., a pattern p belongs to $PS(3)$, which implies $\sum_{i=0}^{iw(p)-1} v(p,i) = 3$. Moreover, the function $UPS: Z^+ \to \mathcal{P}(P)$ points out a union of pattern sets such that $UPS(k) = \bigcup_{i=1}^{k} PS(i)$.

Since a single LUT in the FPGA has the input constraint K ($K = 6$ for modern technologies), a pattern $p \in UPS(K)$ can be mapped onto $ow(p)$ copies of the K-input LUT. For example, the pattern $<2,1>_p$ can be mapped onto 3 copies of the 6-input LUT as shown in Figure 2.2(c). The delay is obviously equal to a LUT delay as all patterns belong to $UPS(K)$. The delay optimal compress tree can be constructed with $UPS(K)$, but $UPS(K)$ is an infinite set. In other words, we cannot determine the optimal solution with $UPS(K)$ unless there is a finite set to construct the compressor tree without loss of delay optimality.

This thesis shows that a finite subset of the infinite set $UPS(K)$ does exist to construct the compressor tree without loss of delay optimality. We denote the finite set as PP and take patterns in it as prime patterns. Therefore, $p \in PP$ iff $p = <1>_p \in PS(1)$, or p has the property $\sum_{i=0}^{j-1} v(p,i) \times 2^i \geq 2^j$ for $0 < j \leq iw(p)$. In other words, a prime pattern p with the property $iw(p) > 1$ possibly produces

the carry propagation in each of its columns. For example, the prime pattern $p_4 = <1,2>_p$ as shown in Figure 2.3(c) possibly produces two valid carries in the 0th and 1st columns due to $v(p_4,0)\times 2^0 \geq 2^1$ and $\sum_{i=0}^{1} v(p_4,i)\times 2^i \geq 2^2$, respectively. On the other hand, the pattern $<2,1>_p$ is not a prime pattern because it will never produce a valid carry in the 0th column due to $v(<2,1>_p,0)\times 2^0 < 2^1$. The function $PPS: Z^+ \rightarrow \mathcal{P}(PP)$ points out a power set of prime patterns such that $p \in PPS(k)$ iff $p \in \{PP \cap PS(k)\}$. For instance, a pattern p belongs to $PPS(3)$, which implies $p \in PP$ and $\sum_{i=0}^{iw(p)-1} v(p,i) = 3$, as shown in Figure 2.3(c). Moreover, Figure 2.3 illustrates three sets of prime patterns as follows: $PPS(1) = \{p_1\}$, $PPS(2) = \{p_2\}$, and $PPS(3) = \{p_3, p_4\}$. The function $UPPS: Z^+ \rightarrow \mathcal{P}(PP)$ points out a union of prime pattern sets such that $UPPS(k) = \bigcup_{i=1}^{k} PPS(i)$, and the function $UNPPS: Z^+ \rightarrow \mathcal{P}(PP)$ points out a set of non-prime patterns by $UNPPS(k) = UPS(k) - UPPS(k)$.

For the modern technology, there are only 37 distinct prime patterns in $UPPS(6)$; therefore, we can find the delay optimal compressor tree based on $UPPS(6)$. For simplicity, all examples are demonstrated under $K = 3$ in the rest of this thesis.

2.3 Problem Formulation

Before formulating the compressor tree problem, this thesis introduces the relationship between the dot plane and the pattern. In this thesis, a *match* is a subgraph indicating the relationship between a pattern and a collection of dots, and then the mapping is defined as $match: P \times N \rightarrow M$, where M is the set of matches.

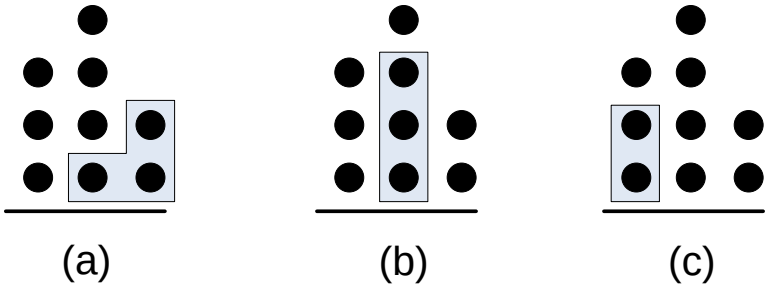


Figure 2.4 Matches under $d_0 =< 3,\ 4,\ 2 >$: (a) $match(p_4, 0) = m_1$. (b) $match(p_3, 1) = m_2$. (c) $match(p_2, 2) = m_3$.

Figure 2.4 demonstrates three feasible matches under the dot plane $d_0 =< 3,4,2 >$: $match(p_4, 0) = m_1$, $match(p_3, 1) = m_2$, and $match(p_2, 2) = m_3$. In other words, Figure 2.4(b) illustrates that three dots in the 1st column of d_0 are matched to the prime pattern p_3 by m_2. Furthermore, a *cover* is a set of matches such that each dot in the same dot plane is matched exactly once by a certain pattern, and then the mapping is defined as $cover : \mathcal{P}(M) \to C$, where C is the set of covers. Figure 2.5 demonstrates two feasible covers under the dot plane $d_0 =< 3,4,2 >$: the cover $cover(\{m_1, m_2, m_4\}) = c_1$ with $m_4 = match(p_3, 2)$; the cover $cover(\{m_1, m_3, m_5, m_6\}) = c_2$ with $m_5 = match(p_1, 1)$ and $m_6 = match(p_4, 1)$.

Since the dot plane d_{s+1} depends on the cover under the dot plane d_s, we can express d_{s+1} by the output derived from the cover under d_s. Therefore, the function $map : D \times C \to D$ determines the resultant dot plane d_{s+1} after the dot plane d_s is covered by the specific cover. For instance, Figure 2.5 shows two resultant dot planes, $map(d_0, c_1) =< 1,3,2,1 >$ and $map(d_0, c_2) =< 2,3,3,1 >$, derived from covers c_1 and c_2 under the dot plane $d_0 =< 3,4,2 >$, respectively.

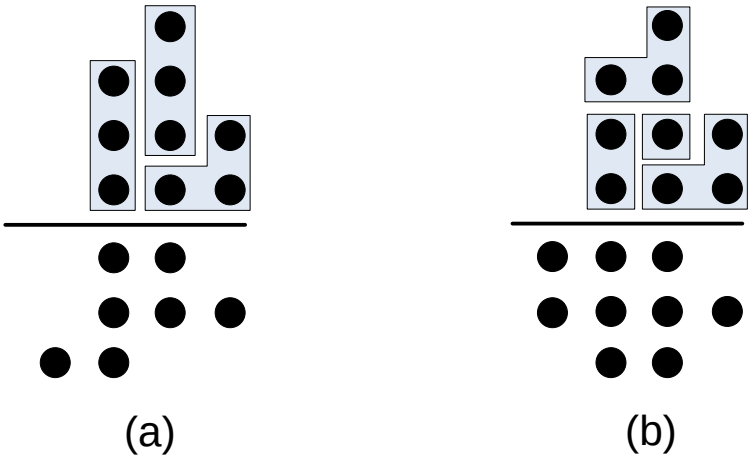


Figure 2.5 Illustrations for covers under $d_0 =< 3, 4, 2 >$.

On modern FPGAs, a ternary adder can sum three operands simultaneously. For flexibility, we denote the maximum quantity of operands as *H* for a CPA on the targeted FPGA. Therefore, *H* is equal to 3 in modern FPGAs. In the future, *H* may be increased to 4, 5, 6, and so on; therefore, we can construct the compressor tree by executing a sequence of covers until the height of the dot plane is less than or equal to *H*. After the compressor tree is constructed completely by the sequence of covers, at most *H* numbers are summed by a CPA. Since the delay of a pattern in $UPS(K)$ is equal to a LUT delay, the depth of the compressor is equal to the times of executing covers. Apparently, the fewer the times of executing covers is, the less the depth of the compressor tree is. This thesis describes a general pseudo code to construct compressor trees, as shown in Figure 2.6. Before we execute the loop body, the height of the dot plane will be checked whether it is larger than *H*. If the condition is true, the dot plane needs a specific cover to reduce its height. After the dot plane d_s is covered by the cover c_s, the resultant dot plane d_{s+1}, will be produced by $map(d_s, c_s)$. Therefore, the depth of the compressor tree is equal to the times of executing loop. The *unit delay model* is used such that the delay is

Procedure: Compressor Tree Synthesis
Input: d_0, H
Output: s, $\{c_{s-1}, c_{s-2}, \ldots, c_0\}$
1. $s = 0$;
2. **While** ($h(d_s) > H$)
3. Find a feasible cover c_s for the dot plane d_s;
4. $d_{s+1} = map(d_s, c_s)$;
5. $s = s+1$;
6. Return the depth s and the cover set $\{c_{s-1}, c_{s-2}, \ldots, c_0\}$;

Figure 2.6 General compressor tree synthesis pseudo code.

determined by the depth of the compressor tree. Thus, we can declare that a compressor tree is delay optimal if its depth is the minimum.

2.4 Properties of Prime Patterns

In order to synthesize the delay optimal compressor tree, the set of building blocks should contain patterns where the number of inputs (i.e., $\sum_{i=0}^{iw(p)-1} v(p,i)$) is equal to or less than K. In other words, the set of building blocks will be $UPS(K)$ exactly. Since $UPS(K)$ is an infinite pattern set, considering all combinations of the compressor tree with $UPS(K)$ is impossible. This thesis describes the truth

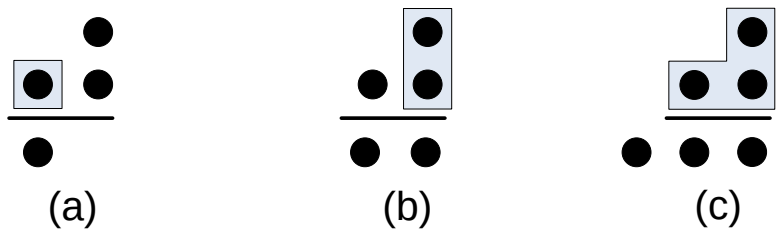


Figure 2.7 All subpatterns of the pattern $p_4 = <1,2>_p$.

that the delay optimal compressor tree can be constructed by the finite set $UPPS(K)$, rather than $UPS(K)$, without loss of delay optimality.

Before describing the fundamental theorem, we define the subpattern firstly. The function $sub: N \times N \times P \to P$ defines the subpattern $sub(j,i,p) =< v(p,j), v(p,j-1), ..., v(p,i) >_p \in P$ with the constraint $0 \le i \le j < iw(p)$. Figure 2.7 shows all subpatterns of the pattern $p_4 =< 1,2 >_p$: $sub(1,1,p_4) =< 1 >_p$, $sub(0,0,p_4) =< 2 >_p$, and $sub(1,0,p_4) =< 1,2 >_p$. In the following, this thesis defines pattern decomposition. The function $decompose: P \to \mathcal{P}(M)$ defines a list of feasible matches $\{(sub(j,i,p),k)\}$ such that the following conditions can be satisfied: (i) $\forall (x,i) \in decompose(p): x \in PP$, and (ii) $\forall ((x,i),(y,j)) \in decompose(p)^2, i > j: ow(y) + j - 1 < i$. Figure 2.8 shows that the pattern $< 3,1,0,1 >_p$ can be partitioned into $\{p_1, p_1, p_3\}$ because of the pattern decomposition $decompose(< 3,1,0,1 >_p) = \{(p_1,0),(p_1,2),(p_3,3)\}$. Then, this thesis shows that all patterns can be partitioned into a set of prime patterns.

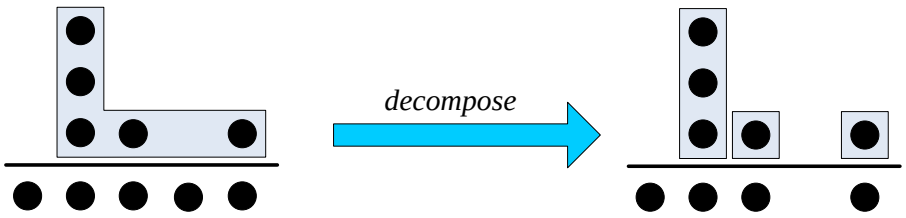


Figure 2.8 Illustration for $decompose(< 3,1,0,1 >_p)$.

Lemma 1: For each pattern $p \in UPS(k)$, p can be partitioned into a set of prime patterns $E_p = \{\hat{p} \mid (\hat{p}, i) \in decompose(p)\}$.

Proof: In the beginning, we identify whether p belongs to *UPPS*(*k*). If $p \in UPPS(k)$, $E_p = p$. Otherwise, we check the carry propagation possibility from the 0th column to ($iw(p)-1$)-th column in the pattern *p*. Because *p* belongs to *UNPPS*(*K*), there exists a set of non-negative integers $Q_p = \left\{q \mid 0 \le q < iw(p) \text{ and } \sum_{i=0}^{q} v(p,i) \times 2^i < 2^{q+1}\right\}$; otherwise, p should belong to $UPPS(K)$, which contradicts $p \in UNPPS(k)$. Based on $q = \min\{Q_p\}$, let the pattern $\hat{p}_0$ be the identified pattern with the property $iw(\hat{p}_0) \le q+1$ such that $v(\hat{p}_0, i) = v(p,i)$ for $0 \le i < iw(\hat{p}_0)$ and $v(p, j) = 0$ for $iw(\hat{p}_0) \le j \le q$. Due to $q = \min\{Q_p\}$, $\hat{p}_0$ is a prime subpattern of p. Similarly, let p' be the subpattern of p with the property $iw(p') < iw(p) - q$ such that $v(p', i) = v(p, iw(p) - iw(p') + i)$ for $0 \le i < iw(p')$, and $v(p, j) = 0$ for $q < j < iw(p) - iw(p')$. Obviously, $p' = sub(iw(p)-1, iw(p)-iw(p'), p) \in UPS(k)$ is true. If p' is prime, p can be partitioned into $E_p = \{\hat{p}_0, p'\}$; therefore, this lemma holds true. The above process, called cut, can extract another prime subpattern on p'. After we repeat cuts on p', p can be partitioned into $E_p = \{\hat{p}_0, \hat{p}_1, ..., \hat{p}_\ell\}$ such that $\hat{p}_i$ is a prime pattern for $0 \le i \le \ell$. Since p belongs to $UPS(k)$, all patterns in E_p do belong to $UPPS(k)$. Moreover, E_p is surely finite since each cut extracts a pattern $\hat{p}_i$ under $iw(\hat{p}_i) \ge 1$. In other words, the recursion of cuts always terminates. In summary, for each pattern $p \in UNPPS(k)$,

p can be partitioned into a set of prime patterns $E_p = \{\hat{p} \mid (\hat{p}, i) \in decompose(p)\}$.

According to Lemma 1, it is obvious that a non-prime pattern can be replaced by a set of prime patterns. Therefore, Lemma 2 can be deduced by Lemma 1. Due to pattern decomposition, the output of a non-prime pattern p may be different to that of E_p. For example, the pattern $<3,1,0,1>_p$ has the output $<1,1,1,1,1>$, but its decomposition $decompose(<3,1,0,1>_p) = \{(p_1,0),(p_1,2),(p_3,3)\}$ has the output $<1,1,1,0,1>$, as shown in Figure 2.8(b). The new cover $\hat{c}$ is derived after every non-prime pattern p is partitioned into E_p, where $match(p,i) \in c$, c is a feasible cover under the dot plane d_s, and $0 \le i < w(d_s)$. Thus, $map(d_s, \hat{c})$ never produces more dots than $map(d_s, c)$ does. In the following, Lemma 2 shows that a compressor tree constructed with non-prime patterns can be replaced by a compressor tree constructed with prime patterns, and then the latter has the same or less depth.

Lemma 2: If there exists a compressor tree *T* constructed with $UPS(k)$, there exists another compressor tree T' constructed with $UPPS(k)$ such that the depth of T' is equal to or less than that of *T*.

Proof: Let *T* be a compressor tree constructed with $UPS(k)$ and the depth of *T* be *z*. The cover c_s is assumed to be under the dot plane d_s on the *s*-th stratum of *T* for $0 \le s \le z$. According to Lemma 1, a non-prime pattern $p \in \{p' \mid \exists a : match(p', a) \in c_0 \text{ and } p' \in UNPPS(k)\}$ could be partitioned into a set of prime patterns $E_p \subseteq UPPS(k)$. After pattern decomposition, the new cover is

denoted as $\hat{c}_0$ and $map(d_0, \hat{c}_0)$ is denoted as $\hat{d}_1$. Since the decomposition may delete some dots of d_1, i.e., $r(\hat{d}_1, a) = r(d_1, a) - \Delta_a$ for $0 \le a < w(\hat{d}_1)$, where $\Delta_a \ge 0$. Thus, there is a new cover c_1' under $\hat{d}_1$ such that a non-empty set of dots Q' in $\hat{d}_1$ is matched by a certain match $m' \in c_1'$ iff $Q' \subseteq Q$, Q' and Q are maximal, and Q is matched by a certain match $m \in c_1$. In the same way, a non-prime pattern $p \in \{p' \mid match(p', b) \in c_1 \text{ and } p' \in UNPPS(k)\}$ can be partitioned into a set of prime patterns $E_p \subseteq UPPS(k)$. After pattern decomposition, the new cover is denoted as $\hat{c}_1$, and then the dot plane $\hat{d}_2 = map(\hat{d}_1, \hat{c}_1)$ is derived such that $w(\hat{d}_2) \le w(d_2)$ and $r(\hat{d}_2, b) = r(d_2, b) - \Delta_b$ for $0 \le b < w(\hat{d}_2)$, where $\Delta_b \ge 0$. This thesis calls the above process as transfer. By executing transfers from d_3 to d_{z-1}, we can partition each non-prime pattern in T into a set of prime patterns. In the end, there are two cases of $\hat{c}_s$ for $0 \le s < z$: (i) each dot in $\hat{d}_s$ is match to $p_1 = <1>_p$, and (ii) $\hat{c}_s$ contains at least one match $match(p, i)$, where $0 \le i < w(\hat{d}_s)$ and $p \ne <1>_p$. If $\hat{c}_s$ belongs to Case (i), $\hat{c}_s$ makes no delay; otherwise, $\hat{c}_s$ has a LUT delay. Therefore, T could be transferred recursively into T' with the depth z' such that $z' \le z$.

By Lemma 2, we acknowledge that every compressor tree containing non-prime patterns can be transformed into the compressor tree which only contains prime patterns. In the following, this thesis deduces the key theorem by Lemma 2. Theorem 1 illustrates that the compressor tree with the minimum depth can be archived by the set of prime patterns only.

Theorem 1: The minimum depth of compressor tree constructed with $UPS(k)$ is the same as that of the compressor tree constructed with $UPPS(k)$.

Proof: Let T be the compressor tree constructed with $UPS(k)$, and T' be the compressor tree constructed with $UPPS(k)$; meanwhile, T and T' have the minimum depth z and z'. Since $UPS(k)$ contains $UPPS(k)$, i.e., the solution space with $UPPS(k)$ is the subset of that with $UPS(k)$, we can derive that $z' \geq z$. According to Lemma 2, a compressor tree constructed with $UPPS(k)$ has the depth $z'' \leq z$. Since T' is the compressor tree constructed with $UPPS(k)$ to have the minimum depth, we can derive that $z' \leq z'' \leq z$. Due to $z' \geq z$ and $z' \leq z$, z is equal to z'.

According to the theorem, the delay optimal compressor can be constructed with $UPPS(K)$, rather than $UPS(K)$. In other words, $UPPS(K)$ is a compact set of basic building blocks for compressor trees. Throughout the rest of this thesis we deal the compressor tree problem with only $UPPS(K)$.

Chapter 3

Proposed Algorithm

In this chapter, this thesis describes a delay optimal compressor tree synthesis algorithm, DOCT, to synthesis compressor trees. The detail processes are shown in Figure 3.1. Step 1 generates all prime patterns in $UPPS(K)$. Step 2 determines the upper bound of the minimum depth, denoted as *UB*, under the given input of the compressor tree. Under *UB*, Step 3 unrolls the loop as shown in Figure 2.6 to

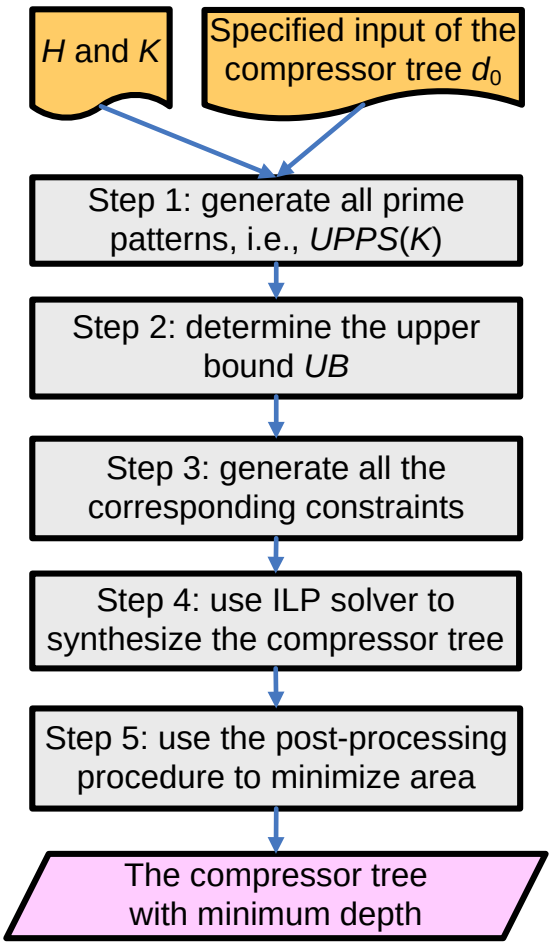


Figure 3.1 DOCT flow.

generate all the corresponding constraints and the objective function. Step 4 gains the delay optimal compressor tree via the ILP solver. Furthermore, Step 5 uses a post-processing procedure to minimize area overhead.

3.1 Upper Bound Determination

Since this thesis unrolls the loop as shown in Figure 2.6 to get the delay optimal compressor tree, the upper bound of the minimum depth needs to be determined in advance. Therefore, this subsection describes how to determine *UB*. Given a dot plane d_0 which is the input of the compressor tree, we can construct another dot plane d_0' such that $w(d_0) = w(d_0')$ and $r(d_0', i) = h(d_0)$ for $0 \le i < w(d_0)$. We call this process as *extend*. For example, we can extend $d_0 =< 3,4,2 >$ to $d_0' =< 4,4,4 >$, as shown in Figure 3.2(a). The prime pattern

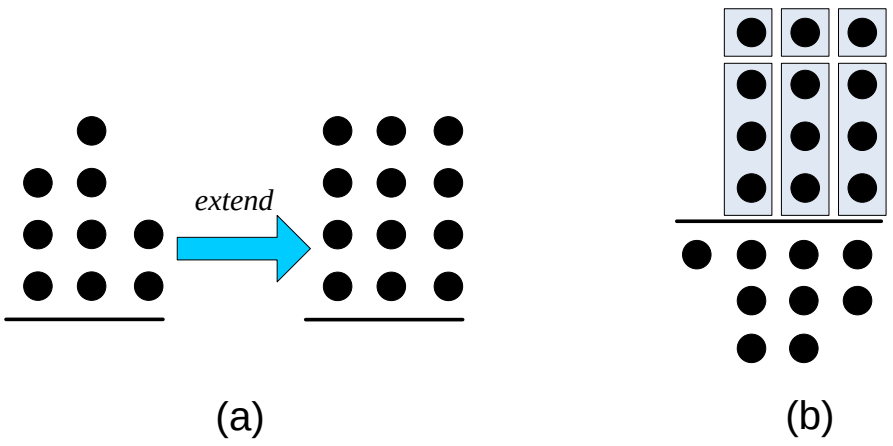


Figure 3.2 (a) Extending $d_0 =< 3,4,2 >$ to $d_0' =< 4,4,4 >$. (b) The resultant dot plane $d_1' = map(d_0', c)$ with d_0' covered by a collection of p_1 and p_3 such that $h(d_1') = 3$.

$x =< K >_p$ matches as many dots in the dot plane d_0' as possible if *K* is equal to or less than $h(d_0')$. Then, the prime pattern $y =< h(d_0') \bmod K >_p$ matches the remaining dots, where *K* is the given input constraint of a LUT. Since d_0' is regular, we can determine $h(d_1')$ precisely by the equation (1), where $d_1' = map(d_0', c)$ is a dot plane derived from the cover $c = \{\bigcup_{i=0}^{w(d_0')-1} match(x,i)\} \bigcup \{\bigcup_{j=0}^{w(d_0')-1} match(y,j)\}$. Therefore, the upper bound of the estimation, $\min\{h(d_1') \mid \exists c \in C : d_1' = map(d_0', c)\}$, can be determined by $h(d_1')$.

$$h(d_{s+1}') = \lfloor h(d_s')/K \rfloor \times ow(<K>_p) + ow(<h(d_s') \bmod K>_p) \quad (1)$$

In all cases of this thesis we assume $H = 2$. To determine *UB*, we execute the equation (1) recursively until $h(d_{s+1}')$ is equal to or less than *H*. As the times of executing the equation (1) is z', z' and *UB* should be equal. For example, based on $K = 3$ and $d_0' =< 4,4,4 >$, nine dots in d_0' are matched to $p_3 =< 3 >_p$ firstly, and then the others are matched to $p_1 =< 4 \bmod 3 >_p$, as shown in Figure 3.2(b). Therefore, we can derive the upper bound of the minimum height of the dot plane d_1' as $(\lfloor 4/3 \rfloor \times ow(<3>_p) + ow(<4 \bmod 3>_p)) = 3$. Moreover, we can determine the upper bound of the minimum height of the dot plane d_2' as 2 by the equation (1). Since $h(d_2')$ is equal to *H*, the determination process for *UB* is terminated. In this example, the times of executing the equation (1) is 2. Hence, *UB* is equal to 2.

3.2 Variables

This subsection introduces the variables used in ILP formulation.

$x_{s,i,j}$: the count of the match $match(p_j, i)$ occurring on d_s

$h_{s,i}$: the number of dots in the i-th column of d_s, i.e., $r(d_s, i)$

$$c_{s,i} = \begin{cases} 1, & r(d_s, i) > H \\ 0, & r(d_s, i) \le H \end{cases}$$

$$q_s = \begin{cases} 1, & h(d_s) > H \\ 0, & h(d_s) \le H \end{cases}$$

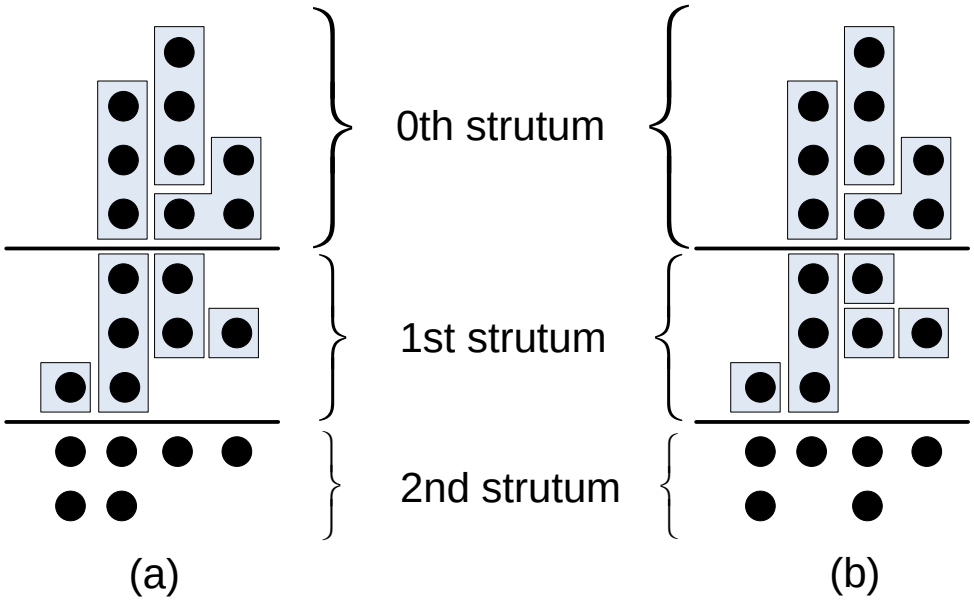


Figure 3.3 (a) The compressor tree before area minimization. (b) The compressor tree after Phase I of the post-processing procedure.

Figure 3.3(a) shows that the dot plane $d_0 =< 3,4,2 >$ has a cover $c_0 = cover(\{m_1, m_2, m_3\})$ which is constructed from $m_1 = match(p_4 =< 1,2 >_p, 0)$, $m_2 = match(p_3 =< 3 >_p, 1)$, and $m_3 = match(p_3, 2)$; therefore, $x_{0,0,4} = x_{0,1,3} = x_{0,2,3} = 1$ and other variables $x_{0,i,j}$ on the 0th stratum are equal to zero. Furthermore, the dot plane $d_1 =< 1,3,2,1 >$ has a cover $c_1 = cover(\{m_3, m_4, m_5, m_6\})$ which is constructed from $m_4 = match(p_1 =< 1 >_p, 0)$, $m_5 = match(p_2 =< 2 >_p, 1)$, and $m_6 = match(p_1, 3)$; therefore, $x_{1,0,1} = x_{1,1,2} = x_{1,2,3} = x_{1,3,1} = 1$, and other variables $x_{1,i,j}$ on the 1st stratum are equal to zero.

3.3 Covering and Succeeding Constraints

The constraint (2) is called as *covering constraint* used to enforce a feasible cover under the dot plane d_s. We use the covering constraint to ensure that every dot is matched exactly once. The inner summation of the constraint (2) sums the amount of dots in the i-th column of d_s, and they are matched to the prime pattern p_j by the match $match(p_j, i-k)$ for $0 \le k < \min\{iw(p_j), i+1\}$. Further, the outer summation of the constraint (2) sums all the results of the inner summation for all prime patterns. Figure 3.3(a) shows that the covering constraint enforces a feasible cover $cover(\{m_1, m_2, m_3\})$ under the dot plane $d_0 =< 3,4,2 >$ such that $2x_{0,0,4} = 2 = h_{0,0}$, $3x_{0,1,3} + x_{0,0,4} = 4 = h_{0,1}$, and $3x_{0,2,3} = 3 = h_{0,2}$.

$$\sum_{j=1}^{|UPPS(K)|}\sum_{k=0}^{iw(p_j)-1} v(p_j,k)\times x_{s,i-k,j} = h_{s,i}, \forall 0 \le i < w(d_s), \forall 0 \le s \le UB \qquad (2)$$

Since the dot plane d_{s+1} depends on how the preceding dot plane d_s is covered, we need the constraint (3) to construct the dot plane d_{s+1} such that $h_{s+1,i} = r(d_{s+1}, i)$ for $0 \le i < w(d_{s+1})$. This thesis calls the constraint (3) as *succeeding constraint*. Firstly, the inner summation of the constraint (3) sums the number of matches $match(p_j, i-k)$ for $0 \le k < \min\{ow(p_j), i+1\}$ to get the dots in the *i*-th column of the dot plane d_{s+1}; meanwhile, the dots are contributed by the pattern p_j. Further, the outer summation of the constraint (3) sums all the results of the inner summation for all prime patterns to get $h_{s+1,i}$. Figure 3.3(a) shows $h_{1,0} = x_{0,0,4} = 1$, $h_{1,1} = x_{0,1,3} + x_{0,0,4} = 2$, $h_{1,2} = x_{0,1,3} + x_{0,2,3} + x_{0,0,4} = 3$, and $h_{1,3} = x_{0,2,3} = 1$, where the dot plane $d_0 = <3,4,2>$ has the cover $cover(\{(p_4,0),(p_3,1),(p_3,2)\})$.

$$\sum_{j=1}^{|UPPS(K)|}\sum_{k=0}^{ow(p_j)-1} x_{s,i-k,j} = h_{s+1,i}, \forall 0 \le i < w(d_s),\ \forall 0 \le s \le UB \qquad (3)$$

3.4 Column and Stratum Constraints

The union of the constraints (4) and (5) is used to compute the correct $c_{s,i}$. This thesis calls the union of the two constraints as *column constraint*. If the *i*-th element of the dot plane d_s is more than *H*, the column constraint should enforce $c_{s,i}$ to be

1. On the other hand, $c_{s,i}$ will be enforced to be 0 as $h_{s,i}$ is equal to or less than H. In Figure 3.3(a), due to $h_{1,1} = 2$ and $h_{1,2} = 3$, $c_{1,1}$ is set as 0 and $c_{1,2}$ is set as 1 via the column constraint. Besides, the term *Inf* used in the constraints (5) and (7) can be set as $\sum_{i=0}^{w(d_0)-1} r(d_0, i)$.

$$(H+2) \times c_{s,i} - 1 \le h_{s,i}, \forall 0 \le i < w(d_s), \ \forall 0 \le s \le UB \quad (4)$$

$$Inf \times c_{s,i} + H \ge h_{s,i}, \forall 0 \le i < w(d_s), \ \forall 0 \le s \le UB \quad (5)$$

When the compressor tree is constructed completely, $q_s = 0$ and $h(d_s) \le H$ should be satisfied iff the depth is equal to s. Therefore, a CPA can be used. Otherwise, the dot plane d_s still need be covered to reduce its height. This thesis uses $c_{s,i}$ to obtain q_s via the constraints (6) and (7). The union of these two constraints is called as *stratum constraint*. Figure 3.3(a) shows an illustrative example. The variable q_1 is set as 1 via the stratum constraint due to $h(d_1) = 3 > H$. On the other hand, the variable q_2 is set as 0 due to $h(d_2) = 2 = H$.

$$q_s \le \sum_{i=0}^{w(d_s)-1} c_{s,i}, \forall 0 \le s \le UB \quad (6)$$

$$Inf \times q_s \ge \sum_{i=0}^{w(d_s)-1} c_{s,i}, \forall 0 \le s \le UB \quad (7)$$

3.5 Objective Function

As stated above, the summation of q_s will be the depth of the compressor tree and it is equal to the equation (8). When we minimize the equation (8), the depth of the compressor tree is minimized.

$$Minimize: \sum_{s=0}^{UB} q_s \tag{8}$$

For example, the depth of the compressor tree is 2 when the dot plane $d_0 =< 3,4,2 >$ has the specific cover $c_0 = cover(\{(p_4,0),(p_3,1),(p_3,2)\})$, and $d_1 =< 1,3,2,1 >$ has the cover $c_1 = cover\{(p_1,0),(p_2,1),(p_3,2),(p_1,3)\}$ as shown in Figure 3.3(a).

3.6 Complexity Analysis

Here, this thesis analyzes the complexity of the number of variables and constrains in our ILP formulation. Firstly, the number of variables is proportional to the number of patterns (i.e., $|UPPS(K)|$); the number of columns in every dot plane; the upper bound of the minimum depth *UB*. This thesis denotes the number of patterns as $|P|$. Besides, the minimum depth upper bound *UB* is proportional to $\log(h(d_0))$. Therefore, the complexity of the number of variables is $O(\log(h(d_0)) \times w(d_0) \times |P|)$. Secondary, this thesis makes the analysis of the number

of the constraints in the following. Similar to the complexity of the number of variables, the number of the constraints is proportional to the number of columns in every dot plane and the minimum depth upper bound *UB*. Therefore, the complexity of the number of the constraints is $O(\log(h(d_0)) \times w(d_0))$.

3.7 Post-processing for Area Minimization

This thesis describes a post-processing procedure to reduce the area overhead without losing delay optimality. This post-processing procedure is described as two phases. Before detailing this two phases, we would define *redundant* matches firstly. A match *m* on the dot plane d_s is redundant iff $h(d_{s+1})$ does not increase while all dots matched by *m* can be matched by p_1 instead. In Phase I, we would delete all redundant matches under the dot plane d_{z-1} on the penultimate stratum when the minimum depth of the delay optimal compress tree is *z*. Figure 3.3(a) shows that a redundant match $match(p_2,1)$ exists on the dot plane $d_1 = <1,3,2,1>$, based on the specific cover $cover(\{(p_1,0),(p_2,1),(p_3,2),(p_1,3)\})$. Figure 3.3(b) shows that the depth of the compressor tree does not increase after deleting the redundant match $match(p_2,1)$ on d_1. According to this phenomenon, this thesis presents Phase I of the post-processing procedure for area minimization. Firstly, we check the existence of redundant matches under the dot plane d_{z-1}. If there is a redundant match *m*, it will be deleted from the compressor tree, e.g., the two dots of the match $match(p_2,1)$ on the dot plane d_1 can be matched by p_1 instead such that they can be passed through from the dot plane d_1 to the dot plane d_2, as shown in Figure 3.3(b). Otherwise, Phase I

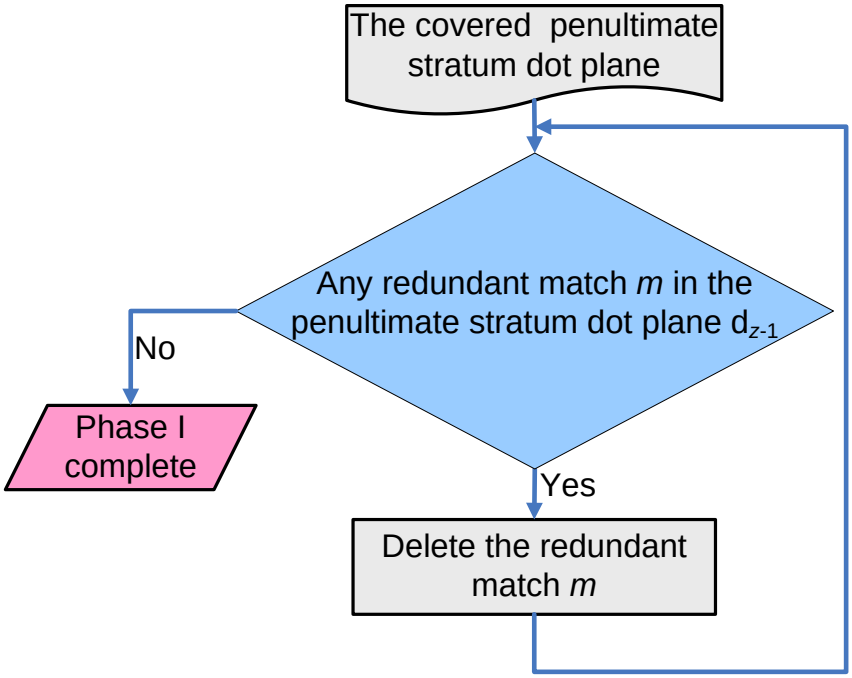


Figure 3.4 Phase I of the post-processing procedure.

is finished. As shown in Figure 3.4, the process stated above is one of the iterations in Phase I; therefore, this post-processing procedure will repeat the process until there is no redundant match on the penultimate stratum.

Practically, basic logic cells on modern FPGAs are flexible. In other words, modern FPGAs employ two single-output LUTs with shared inputs as shown in Figure 3.5. In general, this kind of circuits is called two-output LUTs. We observe that two-output LUTs can map two single-output Boolean functions simultaneously if the two functions satisfy two conditions: (i) the summation of the two function's distinct variables should be fewer than or equal to the *physical-input* constraint denoted as *PIC* ($PIC = 8$ on Altera Stratix IV, and $PIC = 5$ in Xilinx Vertex V, e.g., *PIC* is equal to 6 in the example as shown in Figure 3.5), and (ii) the summation of the LUT size of the two functions should be fewer than or equal to the

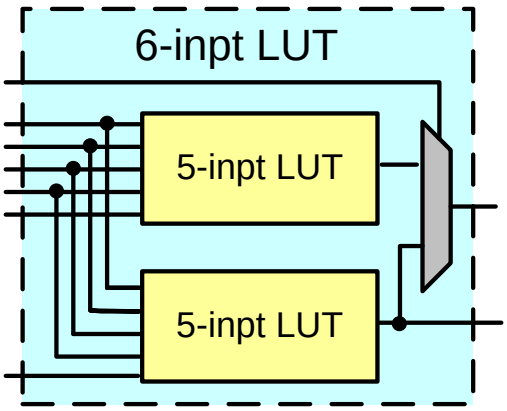


Figure 3.5 A two-output LUT with shared inputs.

physical-capacity constraint denoted as *PCC* ($PCC = 64$ on Altera Stratix IV, and $PCC = 64$ in Xilinx Vertex V [1, 2]) . Actually, *PCC* is equal to 2^K, where *K* is the input constraint of a LUT. Besides, a two-output LUT can map a single output function if the number of variables of the function is *K* (*K* = 6) as shown in Figure 3.5. Moreover, we can merge the two distinct LUTs among all strata if the two functions mapped by these two distinct cells satisfy *PIC* and *PCC*. Suppose we want to map the two prime patterns $p_4 = <1,2>_p$ as shown in Figure 3.6(a) (i.e., $PIC = 6$, $PCC = 64$), and then we can map the two patterns onto four two-output LUTs as shown in Figure 3.6(a). Obviously, the summation of the number of inputs of LUT 2 and 4 is equal to 6 which is fewer than *PIC*, and the summation of the LUT size is equal to $2^3 + 2^3$ which is fewer than *PCC*. Hence, we can merge LUT 2 and LUT 4 into a single LUT as shown in Figure 3.6(b). In Phase II, we merge the distinct LUTs to map different patterns if these two functions mapped by them satisfy *PIC* and *PCC*.

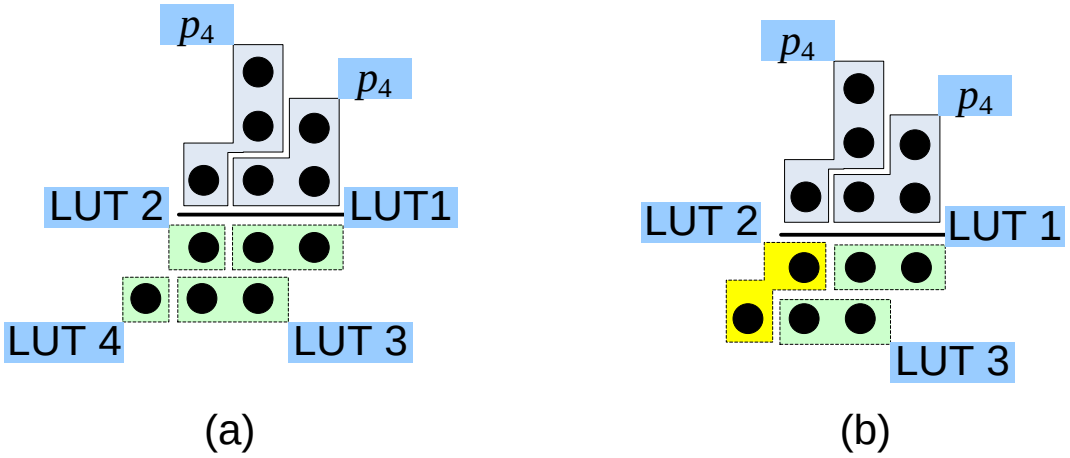


Figure 3.6 (a) The mapping before Phase II. (b) The mapping after Phase II.

Chapter 4

Experimental Results

4.1 Experimental Information

We implement DOCT and the GPC heuristic [14] in C/C++ language on a workstation with an Intel Xeon 2-GHz processor and 16 GB main memory under the Centos 5.2 operating system. Besides, an open source package, lp_solve 5.5.13, is used to solve the linear formulations. A set of benchmark circuits is evaluated including three Radix-4 unsigned Booth-encoded multipliers (8 by 8 and 16 by16), multiplier accumulators (MAC), discrete cosine transformation (DCT) [20], finite impulse response filters (FIR), and motion estimations (ME). The input of each compressor trees is extracted from the simulation result produced by MATLAB Simulink toolbox. All compressor trees in our experiments are directly synthesized without pipelined.

Table I illustrates the detail information of the benchmark circuits. In Table I, Column 1 shows the variety of our benchmark circuits. Column 2 and Column 3 show the width and the height of the input dot plane, respectively. Column 4 shows the number of dots in the input dot plane.

TABLE I

CIRCUIT INFORMATION

Circuit	Width	Height	Total
8 by 8	16	5	56
16 by 16	32	9	176
DCT_1	22	10	150
DCT_2	20	6	83
DCT_3	20	5	82
DCT_4	20	8	116
DCT_5	22	7	105
FIR_1	15	9	72
FIR_2	23	13	167
FIR_3	39	21	374
ME_1	10	10	100
ME_2	14	14	196
MAC_1	9	6	34
MAC_2	11	7	47

4.2 Parameters Setup

We implement two compressor tree synthesis algorithms DOCT and the GPC heuristic. The following is the setting of parameters in our experiment.

DOCT: Compressor tree synthesis using DOCT described in preceding sections under $K = 6$ and $H = 3$. This thesis supposes that DOCT is evaluated on Altera Stratix IV. Thus, the physical input constraint (*PIC*) is set to 8 and the physical capacity constraint (*PCC*) is set to 64. The compressor tree produces three outputs summed by ternary adder.

GPC: Compressor tree synthesis using the generalized parallel counter (GPC) heuristic. In the GPC heuristic, there are three parameters: (i) *M* is the input

constraint of GPC patterns (the input constraint of LUTs in the targeted FPGA, e.g., 6 for Altera Stratix IV, and Xilinx Vertex V), (ii) *N* is the output constraint of GPC patterns, and (iii) *k* is the number of inputs of the final CPA (i.e., *k* is equal to *H*). In our experiments, *M* is set as 6; *N* is set as 4; *k* is set as 3. The setting is the same as [14].

4.3 Experimental Results

In our experiment, we compare both the depth and area produced by DOCT to that by the GPC heuristic. Table II, III, and IV show the experimental results under

TABLE II

SYNTHESIS RESULT UNDER *K* = 5

K = 5					
Circuit	*UB*	delay		LUTs	
		DOCT	GPC	DOCT	GPC
8 by 8	1	1	2	29	45
16 by 16	3	2	4	149	222
DCT_1	3	3	3	160	171
DCT_2	2	2	2	103	109
DCT_3	1	1	1	42	47
DCT_4	2	2	2	101	107
DCT_5	2	2	3	86	132
FIR_1	3	2	4	61	97
FIR_2	3	3	4	177	222
FIR_3	4	4	4	536	580
ME_1	3	3	3	107	110
ME_2	4	3	4	222	240
MAC_1	2	1	3	17	40
MAC_2	2	2	3	34	50
Avg.	0.83	0.73	1	0.83	1

different input constraints of LUTs: $K=5$, $K=6$, and $K=7$, respectively. In Table II, III, and IV, Column 2 illustrates upper bounds of all benchmark circuits. Column 3 and Column 4 illustrate the depth of compressor trees produced by DOCT and the GPC heuristic. Meanwhile, Column 5 and Column 6 illustrate the area in terms of LUTs on Altera Vertex Stratix IV produced by DOCT and the GPC heuristic. Compared to the GPC heuristic, DOCT has 27% less depth with 17% fewer LUTs under $K=5$; 32% less depth with 21% fewer LUTs under $K=6$; and 20% less depth with 2% fewer LUTs under $K=7$. For all benchmark circuits, the GPC heuristic was finished in few seconds; meanwhile, DOCT was finished in 500 seconds.

TABLE III

SYNTHESIS RESULT UNDER *K* = 6

K = 6					
Circuit	*UB*	delay		LUTs	
		DOCT	GPC	DOCT	GPC
8 by 8	1	1	1	24	30
16 by 16	2	2	3	132	174
DCT_1	2	2	3	127	152
DCT_2	2	2	3	98	120
DCT_3	1	1	2	35	70
DCT_4	2	2	3	90	118
DCT_5	2	2	3	66	104
FIR_1	2	2	2	52	58
FIR_2	3	2	3	131	160
FIR_3	3	3	4	400	459
ME_1	2	2	3	78	105
ME_2	3	3	4	174	193
MAC_1	1	1	2	16	26
MAC_2	2	1	2	19	38
Avg.	0.73	0.68	1	0.79	1

It is evident that DOCT always have better or the same result in depth compared to the GPC heuristic. The reason is that DOCT consider all combinations of all prime patterns for constructing a compressor tree. Although DOCT does not outperform the GPC heuristic in area for every case, it provides smaller area on average.

TABLE IV

SYNTHESIS RESULT UNDER $K = 7$

$K = 7$					
Circuit	*UB*	delay		LUTs	
		DOCT	GPC	DOCT	GPC
8 by 8	1	1	2	26	40
16 by 16	2	2	2	125	119
DCT_1	2	2	3	109	127
DCT_2	2	2	2	83	78
DCT_3	1	1	2	40	62
DCT_4	2	2	2	82	78
DCT_5	1	1	1	44	44
FIR_1	2	2	2	51	48
FIR_2	2	2	3	125	135
FIR_3	3	3	3	397	334
ME_1	2	2	2	78	69
ME_2	2	2	3	123	153
MAC_1	1	1	1	12	16
MAC_2	1	1	2	22	33
Avg.	0.8	0.8	1	0.98	1

Chapter 5

Conclusions and Future Work

A delay optimal compressor tree synthesis algorithm, DOCT, has been presented in this thesis. Since the infinite set of patterns can be superseded by the finite set of prime patterns without loss of delay optimality, DOCT adopts an ILP-based methodology to map prime patterns onto the compress tree with the minimum depth and utilizes a post-processing procedure to minimize area overhead. Therefore, DOCT can authentically archive compressor trees with minimum depths by all prime patterns under the input constraint of a LUT. On average, compressor trees produced by DOCT have 32% less depth and 21% fewer LUTs than those produced by the GPC heuristic on modern technologies.

Although DOCT has made a progress in reducing area overhead compared to the GPC heuristic, we believe that there is still room for improvement. In the beginning, we have put the area cost in the cost function of ILP formulation. Unfortunately, the run time of DOCT is too long and unacceptable. But according to the result of some smaller case, DOCT considering area cost in the cost function could archive around 50% fewer LUTs than that does not consider. Yet, we believe that the research of reducing area optimally is worth being performed in the future.

Reference

[1] Altera Corporation, Stratix IV device handbook. [Online]. Available: http://www.altera.com/

[2] Xilinx Corporation, Vertex-5 FPGA user guide. [Online]. Available: http://www.xilinx.com/

[3] Altera Corporation, Stratix III device handbook. [Online]. Available: http://www.altera.com/

.

[4] Xilinx Corporation, Vertex-4 FPGA user guide. [Online]. Available: http://www.xilinx.com/

[5] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *Computer*, vol. 40, no. 3, pp. 50–57, March 2007.

[6] Altera Corporation, FPGAs provide reconfigurable DSP solutions. [Online]. Available: http://www.altera.com/literature/wp/wp_dsp_fpga.pdf

[7] S. Mirzaei, A. Hosangadi, and R. Kastner, "FPGA implementation of high speed FIR filters using add and shift method," *International Conference on Computer Design*, October 2006, pp. 308–313.

[8] O. Kwon, K. Nowka, and Jr. Swartzlander, "A 16-bit by 16-bit MAC design using fast 5:3 compressor cells," *Journal* of *VLSI Signal Processing Systems*, Vol. 31, No. 2, pp. 77-89, June 2002.

[9] C.-Y. Chen, S.-Y. Chien, Y.-W. Huang, T.-C. Chen, T.-C. Wang, and L.-G. Chen, "Analysis and architecture design of variable block-size motion estimation for H.264/AVC," *IEEE Transaction on Circuits and Systems*, vol. 53, no 2, pp. 578-583, February 2006.

[10] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, May 1965.

[11] C. S. Wallace, “A suggestion for a fast multiplier,” *IEEE Transaction on Electronic computers*, vol. 13, no. 1, pp. 14–17, February 1964.

[12] V. G. Oklobdzija, D. Villeger, and S. S. Liu, “A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach,” *IEEE Transaction on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

[13] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, “Optimal circuits for parallel multipliers,” *IEEE Transaction on Computers*, vol. 47, no. 3, pp. 273–285, March 1998.

[14] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Efficient synthesis of compressor trees on FPGAs,” *Asia South Pacific Design Automation Conference*, March 2008, pp. 138–143.

[15] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Improving synthesis of compressor trees on FPGAs via integer linear programming,” *Design Automation and Test in Europe*, March 2008, pp. 1256–1261.

[16] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, "A hybrid ASIC and FPGA architecture," *International Conference of Computer-Aided Design*, November 2002 , pp. 187-194.

[17] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. K. Verma, P. Brisk, F. K. Gurkaynak, Y. Leblebici, and P. Ienne, “Architectural improvements for field programmable counter arrays: enabling efficient synthesis of fast compressor trees on FPGAs,” *International Symposium on Field Programmable Gate Arrays*, February 2008, pp. 181–190.

[18] P. Brisk, A. K. Verma, P. Ienne, and H. Parandeh-Afshar, “Enhancing FPGA performance for arithmetic circuits,” *Design Automation Conference*, June 2007, pp. 334–337.

[19] J. Cong and Y. Ding, “FlowMap: an optimal technology mapping algorithm for delay optimization in Lookup-Table based FPGA designs,” *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, January 1994.

[20] W. Pan, A. Shams, and M. A. Bayoumi, “NEDA: a new distributed arithmetic architecture and its application to one dimensional discrete cosine transform,” *IEEE Signal Processing Systems*, October 1999, pp. 159–168.