

國立臺灣大學電機資訊學院資訊工程學系  
碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

局部分群演算法用以合成低功率預先計算型

內容可定址記憶體上的低成本參數擷取器

Local Grouping Algorithm for Synthesizing Low-Cost  
Parameter Extractor of Low-Power Pre-computation-Based  
Content Addressable Memory

賴聰勝

Tsung-Sheng Lai

指導教授：賴飛羆 博士

Advisor: Feipei Lai, Ph.D.

中華民國 98 年 6 月

June, 2009

國立臺灣大學碩士學位論文  
口試委員會審定書

局部分群演算法用以合成低功率預先計算型內容可定  
址記憶體上的低成本參數擷取器

Local Grouping Algorithm for Synthesizing Low-Cost  
Parameter Extractor of Low-Power Pre-computation-Based  
Content Addressable Memory

本論文係賴聰勝君（學號 R96922071）在國立臺灣大學資訊工程  
學系完成之碩士學位論文，於民國 98 年 6 月 14 日承下列考試委  
員審查通過及口試及格，特此證明

口試委員：

賴聰勝

（指導教授）

伍大峰

莊紅輝

梁文耀

蔡坤霖

系主任

呂育道

## 誌謝

經過了許多時間的努力，終於完成此篇論文，而最感謝的當然是我的指導教授賴飛羆老師，老師不僅教導我學術上的知識，並且也讓我學到生活上的許多道理，此外也提供了良好的學習和實驗環境，使得我可以專心的將大部分時間放在研究上，努力完成了此篇論文，讓我在台大的這幾年收穫許多。

也感謝口試委員汪大暉教授、莊仁輝教授、梁文耀教授以及蔡坤霖教授在口試時的指導以及許多的建議，使我在口試的時候能夠更瞭解自己的論文以及報告還有哪些方面可以加強。

感謝我的家人，每次在我感到失望及沮喪時，陪我出外散心，紓解心中的不快，也讓我被問題困惑時，可以轉換情緒，使得我能夠想到解決的方法，將困惑我的問題一一迎刃而解，家人是我最大的精神支柱。

感謝李鴻璋教授對我的論文做建議以及指導和糾正，也感謝幫助我許多的彭治弘學長，每次我有問題問時，都能以我較能理解的方式解釋給我聽，此外也常常和我討論，教我許多的知識，讓我學到了許多的東西，而且此次論文题目的靈感也是來自於和學長的討論，此外學長也在此篇論文幫了許多大忙，非常感謝學長大力相挺。

也感謝實驗室的同窗江忠桓、詹承浩還有李育安等和我一起修課及寫論文，大家一同克服種種難關，以及 Low Power 組的學弟童顛叡、葉紘瑋、林志威、范聖欣、蕭錦濤、嚴天李，若不是有你們的幫忙以及大力的配合，計畫以及許多任務也無法順利的完成，且實驗室的歡笑都是來自於大家，我會將這份美好的回憶珍藏在心底的。

也非常感謝親切的周憲政學長對我們這些學弟的照顧，常常關心我們這些學弟的學業狀況以及生活。

## 摘要

因為內容可定址記憶體的高速特性，使得它在許多需要高速的設備中扮演著重要的角色，但是它的耗電量也非常的高。在這篇論文中我們提出一個合成演算法用來合成低功率預先計算型內容可定址記憶體上的參數擷取器，使得資料能夠被均勻的映射到每個參數，而且硬體的成本也較少。此外我們也提出一個方法去減少當一些資料在區塊中大部分是相同時，對參數擷取器所帶來的影響。實驗結果顯示，當和 Gate-Block Selection 演算法比較時，我們的方法可以減少 58.88% 的功率消耗，也可以省下 0.53% 的 CMOS 電晶體數目。如果用我們提出的捨去及交錯法去改善 Gate-Block Selection 演算法時，我們的方法仍然可以減少 13% 的功率消耗。

關鍵字：內容可定址記憶體、預先計算、低功率、低成本、合成演算法



# Abstract

Content addressable memory (CAM) plays an important role on the performance of some devices due to the high speed of CAM. But the power consumption of CAM is also high. In this work, we propose a synthesis algorithm to synthesize the parameter extractor for low-power pre-computation-base CAM (PB-CAM) such that the data can be mapped to parameters uniformly and the cost of the parameter extractor can also be lower. Moreover, we also propose a method to reduce the impact on mapping data to parameters when most data are identical in some data blocks. In the experimental results, the average reduction of the power consumption can achieve 58.88% and the number of CMOS transistors can save 0.53% when compared with Gate-Block Selection algorithm. If the Gate-Block Selection algorithm is also enhanced by our proposed discard and interlaced method (DAI method) then the power consumption can still be reduced by 13%.

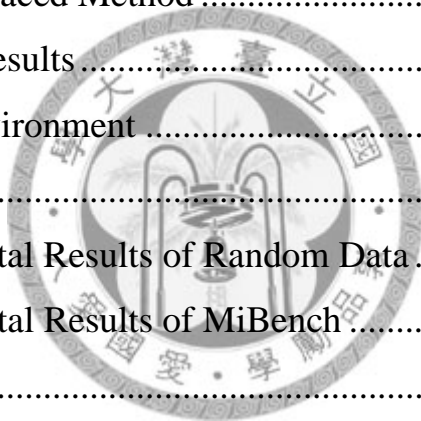
Keywords: content addressable memory (CAM), pre-computation, low power, low cost, synthesis algorithm



# Contents

|   |     |
|---|-----|
| 口試委員會審定書 .....  | i   |
| 誌謝 .....  | ii  |
| 摘要 .....  | iii |
| Abstract.....   | iv  |
| Contents.....   | v   |
| List of Figures.....                                  | vii |
| List of Tables.....                                   | ix  |
| Chapter 1 Introduction.....                           | 1   |
| 1.1 Power Dissipation in CMOS VLSI Circuit.....       | 1   |
| 1.1.1 Switching Power Dissipation .....               | 2   |
| 1.1.2 Short-Circuit Power Dissipation .....           | 2   |
| 1.1.3 Leakage Power Dissipation.....                  | 3   |
| 1.2 Concept of Content Addressable Memory.....        | 4   |
| 1.2.1 Content Addressable Memory.....                 | 4   |
| 1.2.2 Applications of Content Addressable Memory..... | 5   |
| 1.2.3 CAM Cell .....                                  | 6   |
| 1.2.4 Write Operation of a CAM Cell.....              | 7   |
| 1.2.5 Read Operation of a CAM Cell.....               | 9   |
| 1.2.6 Search Operation of a CAM Cell .....            | 10  |
| 1.2.7 Match Line Structure.....                       | 11  |
| Chapter 2 Related Work .....                          | 14  |
| 2.1 Selective Pre-charge Scheme .....                 | 16  |
| 2.2 Pre-computation Scheme.....                       | 17  |
| 2.2.1 Ones Count Scheme .....                         | 18  |
| 2.2.2 Block-XOR Scheme .....                          | 21  |
| 2.2.3 Gate-Block Selection Algorithm.....             | 22  |

|            |   |    |
|------------|---|----|
| 2.3        | Motivation and Objective.....                       | 25 |
| Chapter 3  | Proposed Approach.....                              | 26 |
| 3.1        | The Benefit of Distributing the Data Uniformly..... | 26 |
| 3.2        | Local Grouping Algorithm.....                       | 28 |
| 3.2.1      | Definition of the Variables.....                    | 28 |
| 3.2.2      | Top Level of Local Grouping Algorithm.....          | 29 |
| 3.2.3      | Grouping Function .....                             | 31 |
| 3.2.4      | Find Gate Function.....                             | 36 |
| 3.2.5      | Demonstration of Local Grouping Algorithm .....     | 40 |
| 3.2.6      | Time Complexity of Local Grouping Algorithm.....    | 43 |
| 3.3        | Discard and Interlaced Method .....                 | 45 |
| Chapter 4  | Experimental Results.....                           | 48 |
| 4.1        | Experimental Environment .....                      | 48 |
| 4.2        | Results .....                                       | 52 |
| 4.2.1      | Experimental Results of Random Data.....            | 52 |
| 4.2.2      | Experimental Results of MiBench.....                | 56 |
| Chapter 5  | Conclusion .....                                    | 63 |
| References | .....   | 64 |





## List of Figures

|             |   |    |
|-------------|---|----|
| Figure 1-1  | A simplified block diagram of a CAM. ....                                 | 5  |
| Figure 1-2  | A conventional 9T CAM cell.....   | 7  |
| Figure 1-3  | A 9T CAM cell performs a write operation.....                             | 8  |
| Figure 1-4  | A 9T CAM cell performs a read operation. ....                             | 9  |
| Figure 1-5  | (a) XOR type CAM cell. (b) XNOR type CAM cell.....                        | 10 |
| Figure 1-6  | The schematic of CAM (four word CAM cells).....                           | 11 |
| Figure 1-7  | A NOR type match line.....  | 12 |
| Figure 1-8  | A NAND type match line.....   | 12 |
| Figure 2-1  | The simplified architecture of the selective pre-charge scheme.....       | 16 |
| Figure 2-2  | The basic architecture of PB-CAM. ....                                    | 17 |
| Figure 2-3  | 7T PB-CAM cell. ....  | 19 |
| Figure 2-4  | Static pseudo-NMOS CAM word circuit. ....                                 | 20 |
| Figure 2-5  | Static parameter comparison circuit. ....                                 | 20 |
| Figure 2-6  | The 14 bits block-xor parameter extractor.....                            | 21 |
| Figure 2-7  | The Gate-Block Selection Algorithm.....                                   | 23 |
| Figure 3-1  | An example of the 2-level parameter extractor for the $n$ bits data. .... | 29 |
| Figure 3-2  | The top level of the local grouping algorithm. ....                       | 30 |
| Figure 3-3  | The general grouping function. ....                                       | 31 |
| Figure 3-4  | The simple grouping function for 2-bit block. ....                        | 32 |
| Figure 3-5  | The function for distinguishing all gate types. ....                      | 33 |
| Figure 3-6  | The status of reducing hardware cost of the parameter extractor. ....     | 35 |
| Figure 3-7  | The find gate function.....   | 37 |
| Figure 3-8  | The function of distinguishing gate types 5, 6 and 7.....                 | 39 |
| Figure 3-9  | The synthesized parameter extractor of the demonstrative example.....     | 42 |
| Figure 3-10 | The discard and interlaced method.....                                    | 46 |
| Figure 4-1  | The unique data distribution of three random data sets.....               | 55 |
| Figure 4-2  | The data distribution of the patricia_small.....                          | 59 |



Figure 4-3 The data distribution of the patricia\_large..... 60



## List of Tables

|            |  |    |
|------------|--|----|
| Table 1-1  | The comparison of the number of rules between CAM and TCAM. ....   | 6  |
| Table 1-2  | Search operations of XOR and XNOR type CAM cells. ....   | 11 |
| Table 1-3  | Comparison between NOR type and NAND type match lines. ....  | 13 |
| Table 2-1  | Number of data is related to the same parameter (ones count). ....   | 18 |
| Table 2-2  | Number of data is related to the same parameter (block-xor). ....  | 22 |
| Table 2-3  | The time complexity of gate-block selection algorithm. ....  | 24 |
| Table 3-1  | The average number of matched rows for four 2-bit parameters. ....   | 26 |
| Table 3-2  | The available synthesized gate types for 2-bit block. (a: msb, b: lsb) .   | 32 |
| Table 3-3  | An example of the benefit of using additional gate types. ....   | 34 |
| Table 3-4  | An example of the choice of the low-cost gate type. ....   | 34 |
| Table 3-5  | The method of finding the low-cost gate type. ....   | 36 |
| Table 3-6  | The relation of the logic gate type of the methods 2 and 4. ....   | 39 |
| Table 3-7  | An example to demonstrate local grouping algorithm. (First level) ...  | 40 |
| Table 3-8  | An example to demonstrate local grouping algorithm. (Second level)   | 41 |
| Table 3-9  | The time complexity of the local grouping algorithm. ....  | 43 |
| Table 3-10 | The comparison of the time complexity of the algorithms. ....  | 44 |
| Table 3-11 | An example of the problem of the identical data. ....  | 45 |
| Table 3-12 | An example to demonstrate the DAI method. (Steps 2 and 3).....   | 46 |
| Table 3-13 | An example to demonstrate the DAI method. (Result).....  | 47 |
| Table 4-1  | The test data in the experiment. ....  | 48 |
| Table 4-2  | The configurations for all schemes in the experiment. ....   | 49 |
| Table 4-3  | The configurations for each scheme in the experiment. ....   | 49 |
| Table 4-4  | The configuration for DAI method in the MiBench experiment. ....   | 50 |
| Table 4-5  | The standard deviation of each block in the MiBench experiment. ....   | 51 |
| Table 4-6  | The number of data in the random test data. ....   | 52 |
| Table 4-7  | The standard deviation on the number of unique data that are mapped<br>to each parameter in the random test data. .... | 52 |

|            |  |    |
|------------|--|----|
| Table 4-8  | The high level simulation result. ....   | 53 |
| Table 4-9  | The improvement rate of the high level simulation result.....  | 53 |
| Table 4-10 | The number of data in the MiBench.....   | 56 |
| Table 4-11 | The standard deviation on the number of unique data that are mapped<br>to each parameter in the MiBench..... | 57 |
| Table 4-12 | The high level simulation result in the MiBench. ....  | 58 |
| Table 4-13 | The average reduction rate of the high level simulation in MiBench..   | 58 |
| Table 4-14 | The average power consumption in MiBench.....  | 61 |
| Table 4-15 | The average reduction rate of the power consumption in MiBench. ...  | 61 |
| Table 4-16 | The average reduction rate of the power on the parameter extractor. ..                                       | 62 |
| Table 4-17 | The average reduction rate of the number of CMOS elements.....   | 62 |



# Chapter 1 Introduction

The performance of modern devices becomes faster than before, however the power consumption and thermal of devices also increase. So the reduction of the power consumption of a device becomes an important issue for many researches especially for embedded systems and portable devices. Content addressable memory (CAM) is one hot topic in these researches. CAM can search for content in parallel within it; so many devices use it to increase performance such as the translation look-aside buffer (TLB) in microprocessors and the tag memory of caches. In the network, the ternary content addressable memory (TCAM) also plays an important role because it performs the routing lookup and packet classification in the network router. Although CAM can operate in high frequency, it also consumes much power. The high power consumption is not suitable for portable devices and embedded systems. Therefore, the reduction of the power consumption of CAM while maintaining its high speed searching performance is required. In this thesis, we proposed an algorithm to synthesize the parameter extractor of Pre-computation-based CAM and the parameter extractor can more uniformly map the data to each parameter. We also proposed a method to reduce the impact on mapping data to parameters when some blocks have a lot of identical data.

## 1.1 Power Dissipation in CMOS VLSI Circuit

In the CMOS VLSI circuits, the power consumption can be separated into switching, short-circuit and leakage power consumption. So the average power consumption can be modeled by the following equation [1]:

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{leakage} \quad (\text{Equation 1-1})$$

### 1.1.1 Switching Power Dissipation

Switching power consumption is due to charging and discharging the parasitic capacitances when the transistors are switching. It is one of dominant sources of the power consumption in the CMOS circuits; the other is leakage power consumption. We can use the following equation to model the switching power consumption [2]:

$$P_{switching} = \alpha_{0 \rightarrow 1} \times C_L \times V_{dd}^2 \times f_{clk} \quad (\text{Equation 1-2})$$

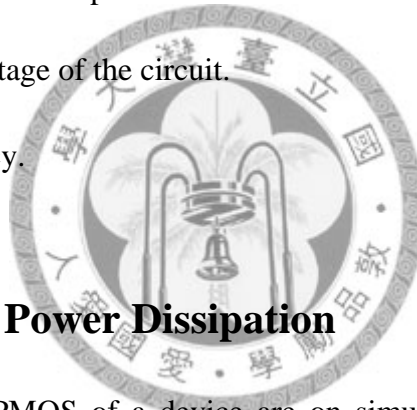
Where:

$\alpha_{0 \rightarrow 1}$  = The 0 to 1 transition probability per clock cycle.

$C_L$  = The sum of all load capacitance.

$V_{dd}$  = The supply voltage of the circuit.

$f_{clk}$  = Clock frequency.



### 1.1.2 Short-Circuit Power Dissipation

When the NMOS and PMOS of a device are on simultaneously, there is a direct current path between power supply and ground that causes the short-circuit power consumption. If the supply voltage is lower than the sum of the threshold voltage of the NMOS and PMOS in the device ( $V_{dd} < V_m + |V_{tp}|$ ) then the short-circuit current can be eliminated [1]. The short-circuit power consumption of a CMOS inverter can be estimated by the following equation [2]:

$$P_{short-circuit} = \frac{\beta}{12} \times (V_{dd} - 2V_t)^3 \times \frac{\tau}{T_{clk}} \quad (\text{Equation 1-3})$$

Where:

$\beta$  = An effective transistor strength that is a constant which depends on the transistor sizes and the technology.

$V_t$  = The threshold voltage of the NMOS and PMOS transistors.

$\tau$  = Input rising/falling time.

$T_{clk}$  = Clock cycle time.

### 1.1.3 Leakage Power Dissipation

Leakage power consumption can be separated into sub-threshold leakage and reverse-bias diode leakage as the following equation [2]:

$$P_{leakage} = (I_{sub-threshold} + I_{diode}) \times V_{dd} \quad (\text{Equation 1-4})$$

$$I_{sub-threshold} = Ke^{(V_{gs}-V_t)/(nV_T)} \left(1 - e^{-\frac{V_{ds}}{V_T}}\right) \quad (\text{Equation 1-5})$$

$$I_{diode} = I_S \left(e^{\frac{V}{V_T}} - 1\right) \quad (\text{Equation 1-6})$$

Where:

$k, n$  = A function of the technology.

$V_{gs}$  = Gate-source voltage.

$V_{ds}$  = Drain-source voltage.

$V_t$  = Threshold voltage.

$V_T = KT/q$  = Thermal voltage.

$I_S$  = Reverse saturation current.

The sub-threshold current flows from source to drain when the transistors are off and the gate to source voltage is still below the threshold voltage. Therefore the transistors still conduct by weak current. The reverse-bias current flows through the reverse-biased diodes that are formed between the diffusion regions and the substrate. These currents

are small for former process technologies, but they are no longer neglected due to the popularity of deep-submicron technology [3].

## **1.2 Concept of Content Addressable Memory**

In the following sections, we will introduce some concepts of CAM first. The topics include the architecture, operations and applications of CAM.

### **1.2.1 Content Addressable Memory**

CAM is one kind of fully associative memory so it can search for the data that are stored in the memory in parallel. CAM is different from the random access memory (RAM). Because RAM uses the access address as input and decodes the address to find the data within it then output the data. But CAM uses the search data as input and compares the search data with the stored data within it in parallel. If the data is found then CAM will output some addresses to access a RAM and the RAM stores some data which are related to the data in CAM. For example, Figure 1-1 shows the simplified block diagram of CAM. The size of it is eight words and each word is stored in eight CAM cells. The data in the CAM compare with the search data from the search line (SL) buffers (or called drivers) and there is a match in this example. So the voltage of match line 2 (ML2) remains high and the others are discharged low when the NOR type match line is used. Then the encoder outputs an address and decoder decodes this address to load data from the RAM.



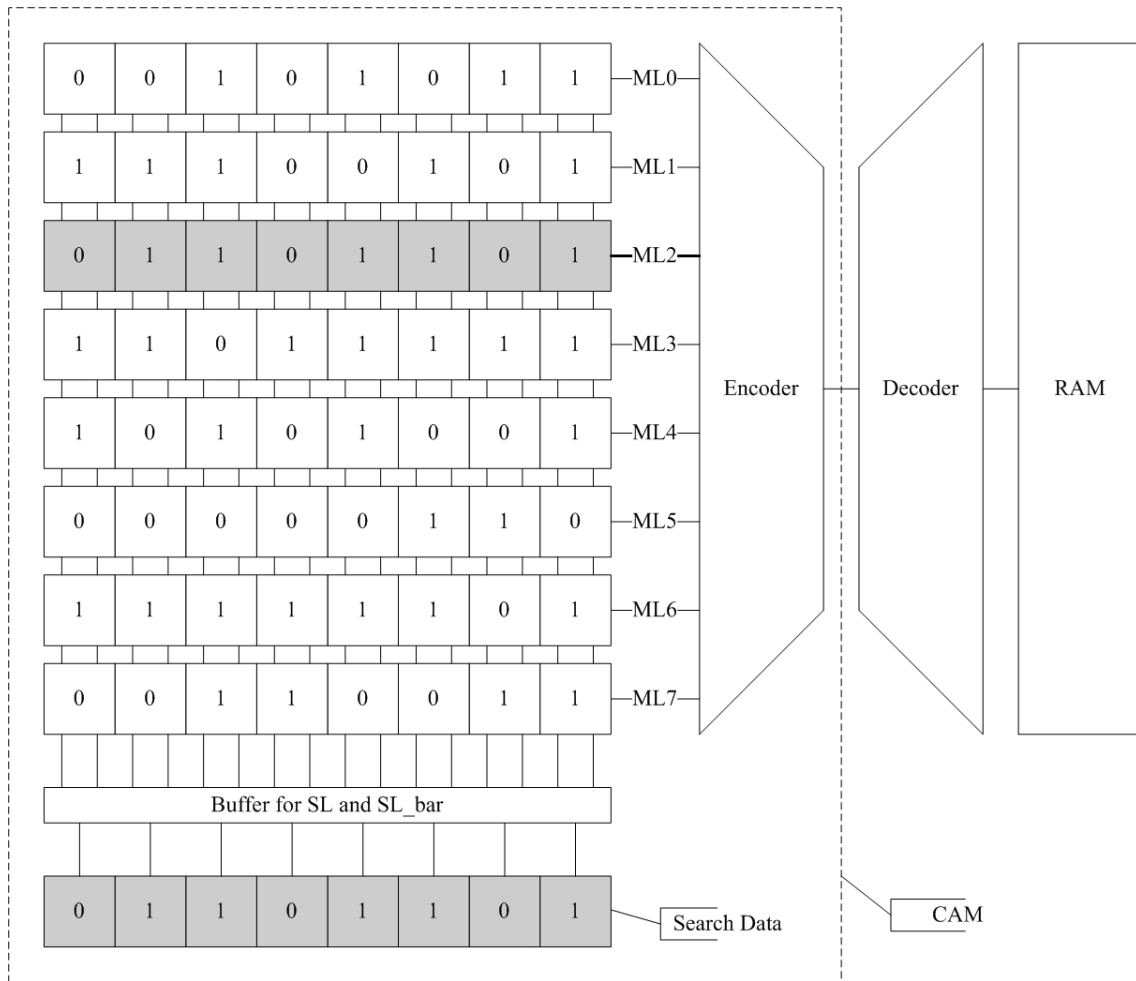


Figure 1-1 A simplified block diagram of a CAM.

## 1.2.2 Applications of Content Addressable Memory

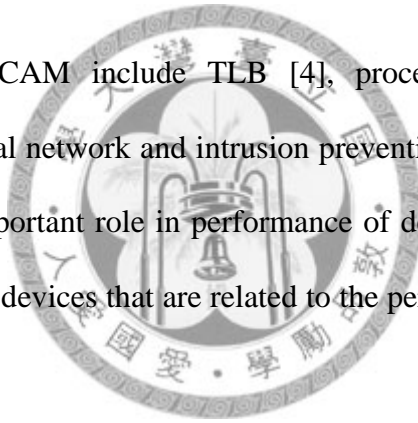
CAM is often used in many devices especially in computer networking devices. For example, the routing table of the network router and the policy table of the hardware firewall are implemented by CAM. It is used for packet forwarding and packet classification. CAM can be classified into two types. One is binary CAM as shown in Figure 1-1, the other is ternary CAM. Binary CAM is often used in computer devices but not network devices in recent years because it only can store two states “0” and “1” in a CAM cell. In this situation, it must contain many rules to gain high performance that causes the hardware cost become higher. So many network devices use ternary

CAM to store the rules. Ternary CAM can store additional one state that is “don’t care” state and therefore it can store more rules than binary CAM when the hardware cost is the same. For example, in Table 1-1 binary CAM needs four space to store the rules but the ternary CAM only needs one space.

Table 1-1 The comparison of the number of rules between CAM and TCAM.

| CAM Type    | Rules |
|-------------|-------|
| Binary CAM  | 0100  |
|             | 0101  |
|             | 0110  |
|             | 0111  |
| Ternary CAM | 01xx  |

Other applications of CAM include TLB [4], processor caches [5], database accelerators, artificial neural network and intrusion prevention system [6]. Thus we can see that CAM plays an important role in performance of devices in these applications. But it is only used in some devices that are related to the performance of systems due to the expensive cost of it.



### 1.2.3 CAM Cell

A CAM cell can search one bit data within it and the data can also be read from it or written into it through bit line by controlling the word line. A schematic of a conventional nine transistors CAM cell is shown in Figure 1-2. It uses six transistors static random access memory (SRAM) to store the data as shown in gray block of Figure 1-2. So the read and write operations of a CAM are the same as in a SRAM. The connection of the control transistor  $N_{ctrl}$  depends on which match line structure is used in the CAM. This transistor decides whether the match line of one word discharge or not. Detailed operations of a CAM cell will be introduced in next sections.



time, the P2 is turned off and N2 is turned on. A current  $I_{BLb}$  is generated from the  $V_{dd}$  node to ground through NW2 and N2. For a while, the Q will become logic 0 and Qb become logic 1 due to the bit line drivers are designed much stronger than the transistors in the CAM cell. Then P1 and N2 are turned off. N1 and P2 are turned on. The voltage of this CAM cell has become stable.

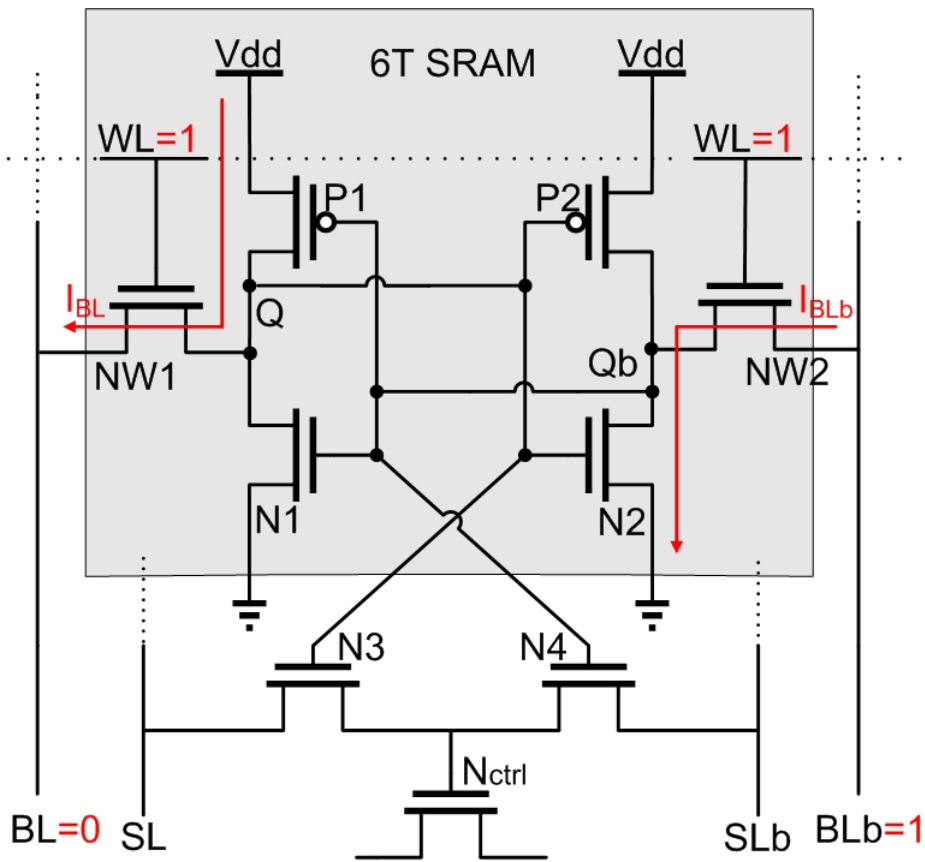


Figure 1-3 A 9T CAM cell performs a write operation.

In the explanation of the above example, we can know that the size of transistors and the drivers of CAM should be designed carefully to ensure that the write operations of a CAM are correct.

## 1.2.5 Read Operation of a CAM Cell

When CAM performs read operation of one word, the bit line and bit line bar should be charged to logic 1 first. Then the word line of this word also is charged to high. After charging these lines, the data can be read from the bit line. An example is shown in Figure 1-4.

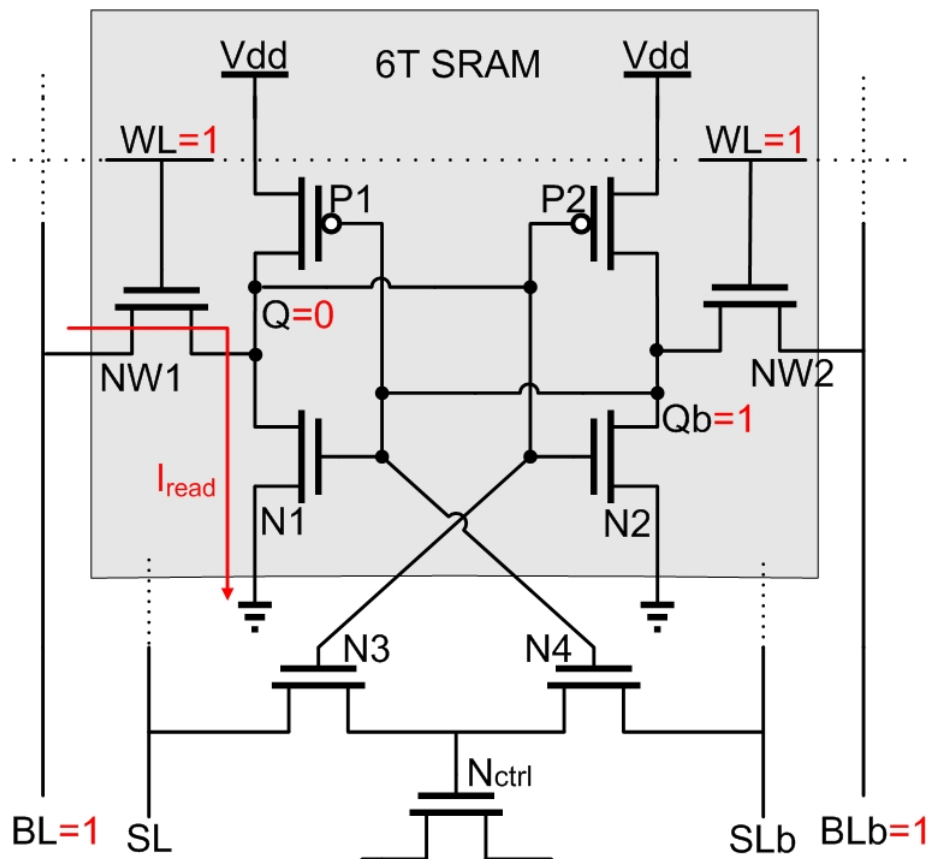


Figure 1-4 A 9T CAM cell performs a read operation.

In Figure 1-4, we assume that the stored data in this CAM cell is zero. The bit line and bit line bar is charged to high then so is the word line. After charging the word line, the NMOS NW1 and NW2 are turned on. A current  $I_{read}$  is generated from the  $V_{dd}$  node to ground through NW1 and N1. Note that when the read operation starts, the bit line drivers are turned off. Therefore the bit line will be discharged to logic zero when the size of N1 is larger than NW1. Because we need to ensure that the data will not flip when reading data from a CAM cell.

## 1.2.6 Search Operation of a CAM Cell

The 9T CAM cell [7] can be classified into two types according to its comparative method. One is XOR type CAM cell and the other is XNOR type CAM cell. The two CAM cells are shown in Figure 1-5 and the search line and the bit line are combined here. For instance, we assume that the stored data and the search data are logic 1 in these two CAM cells. Then N2 and N3 are turned on. N1 and N4 are turned off. Current flows through N2 and SLb to ground in Figure 1-5 (a) therefore  $M_{ctrl}$  is turned off. Moreover, the match line is pre-charged to high before searching data so the voltage of the match line will not be pulled down in this CAM cell. In Figure 1-5 (b), the  $M_{ctrl}$  is turned on because N3 is turned on and SL is logic 1. Therefore the CAM cell will pass the current to the next CAM cell that is connected with the same match line. If there is no next CAM cell then the match line will be connected to ground.

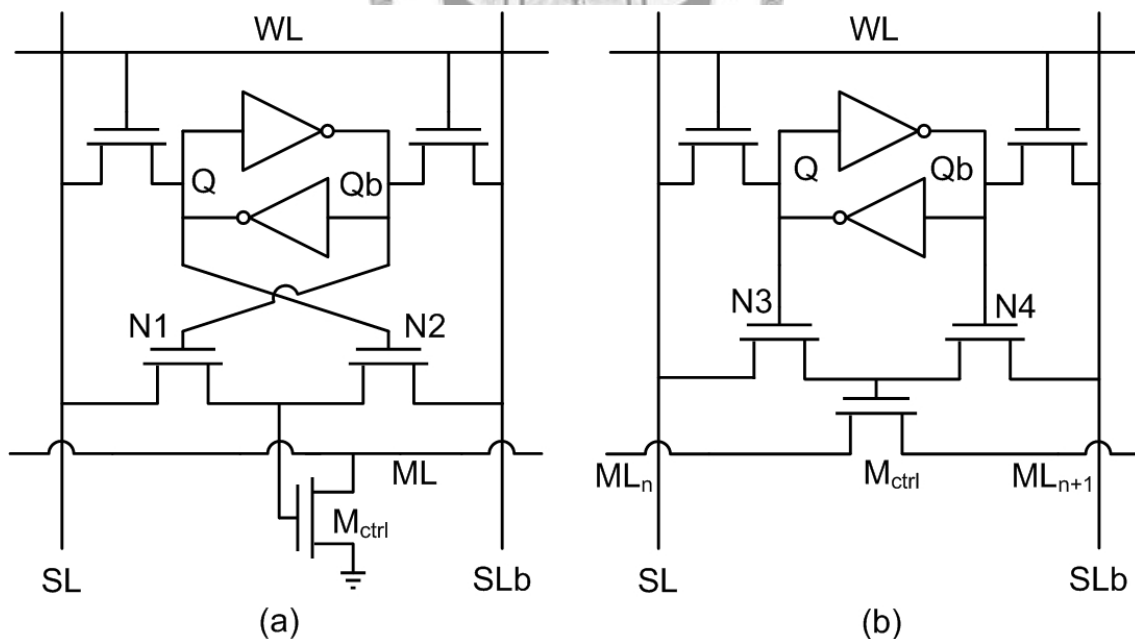


Figure 1-5 (a) XOR type CAM cell. (b) XNOR type CAM cell.

The search operation of these two CAM cells can be summarized in Table 1-2. If we replace the “on” state with logic 1 and the “off” state with logic 0 then the search operation is like the logic XOR and XNOR. This is why we name these two types as

XOR and XNOR types. Other structure of CAM cells can be found in [7, 8].

Table 1-2 Search operations of XOR and XNOR type CAM cells.

| <b>Q</b> | <b>SL</b> | <b>XOR type</b> | <b>XNOR type</b> |
|----------|-----------|-----------------|------------------|
|          |           | $M_{ctrl}$      | $M_{ctrl}$       |
| 0        | 0         | Off             | On               |
| 0        | 1         | On              | Off              |
| 1        | 0         | On              | Off              |
| 1        | 1         | Off             | On               |

### 1.2.7 Match Line Structure

The match line connects several CAM cells to store one word data. It is used to determine which data match the search data. The simplified schematic is shown in Figure 1-6. The match lines are pre-charged to high first and then search data are inputted into search line drivers. After data are inputted to the search line drivers, they drive the search line and search line bar to perform search operation in each CAM cell.

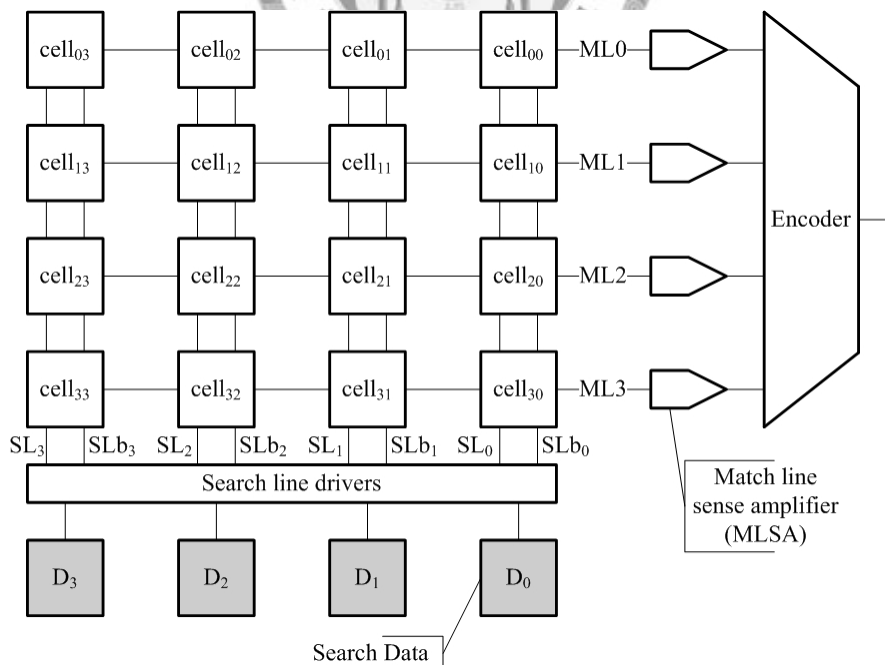


Figure 1-6 The schematic of CAM (four word CAM cells).

The match line structure [7] can also be classified into two types according to what kind of CAM cell we use in the CAM. If the XOR type CAM cells are used then the



match line structure is NOR-type match line. Otherwise, it is NAND-type match line when the XNOR type CAM cells are used. These two match line structures are shown in Figure 1-7 and Figure 1-8.

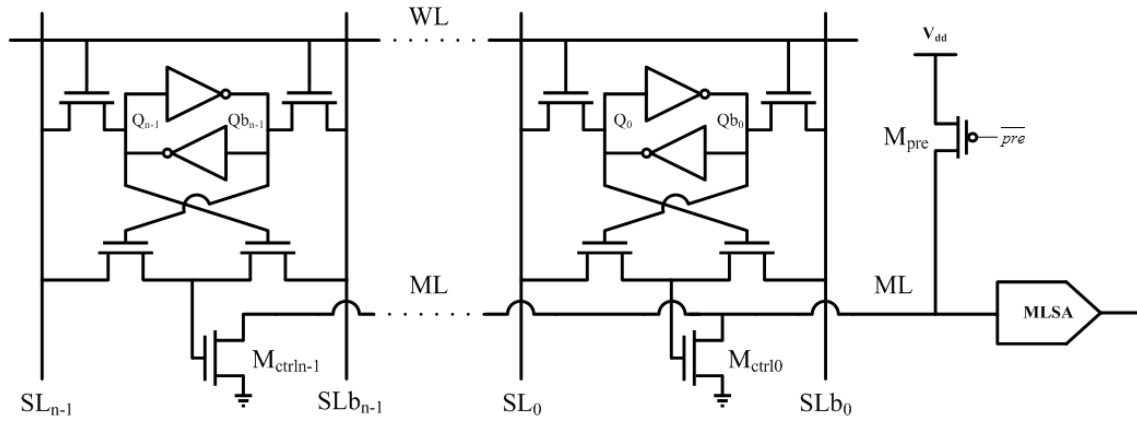


Figure 1-7 A NOR type match line.

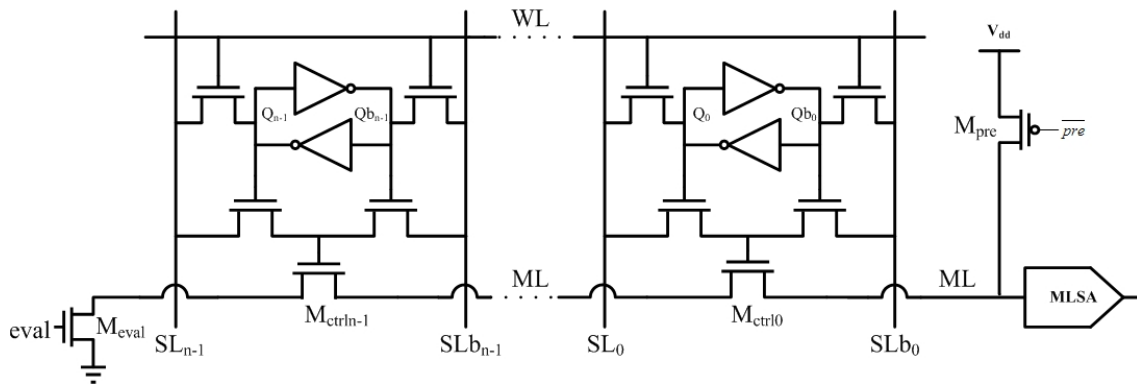


Figure 1-8 A NAND type match line.

A typical search operation of a NOR type match line has three phases. These three phases include search line pre-charge, match line pre-charge and match line evaluation. Before searching data, the search line is pre-charged to low in order to prevent the match line connecting to ground. After disconnecting the match line from ground, the signal *pre* is asserted to turn on the PMOS  $M_{pre}$ . Then the match line is pre-charged to high. In this time, the search data are inputted into the match line drivers and evaluate the match line. If the search data word matches all the stored bits of the CAM cell on

the same match line then the voltage of the match line will remain high, otherwise it will be pulled down to low. The match line sense amplifier (MLSA) is responsible for sensing the voltage of a match line and outputting the strong voltage that are corresponded to the sensing voltage.

The search operation of a NAND type match line has two phases. These two phases include match line pre-charge and match line evaluation. Before searching data, the signal *pre* is asserted to turn on the PMOS  $M_{pre}$  and then the match line is pre-charged to high. After the match line is pre-charged, the search line can be driven by inputting the search data. Next, the signal *eval* is asserted to turn on the NMOS  $M_{eval}$  and the match line can be evaluated. If the search data matches the stored data then the voltage of the match line will be pulled down to low, otherwise it will remain high.

Table 1-3 Comparison between NOR type and NAND type match lines.

|                              |                      | NOR type | NAND type |
|------------------------------|----------------------|----------|-----------|
| <b>Voltage of match line</b> | <i>When match</i>    | High     | Low       |
|                              | <i>When mismatch</i> | Low      | High      |
| <b>Performance</b>           |                      | Fast     | Slow      |
| <b>Power consumption</b>     |                      | High     | Low       |

Table 1-3 shows the differences between NOR type and NAND type match lines. The performance of NOR type match line is faster than the NAND type due to the pull down path of the NAND type match line is too long when the stored data is matched. However, the power consumption of NOR type match line is higher than that of NAND type match line. Because in most applications, the search data matches only one data in CAM so the number of the discharged operations of the NOR type match line is more than that of NAND type.

## Chapter 2 Related Work

In this chapter, we will introduce some designs on low power CAM briefly. Moreover, we will also introduce some designs that are related to our work in the following sections.

Most researches of the CAM are organized well and introduced in [7]. These researches of low power CAM focus on some topics. They include how to reduce the power consumption of the match line, search line and architecture. Some designs will be introduced in the followings.

In low-swing scheme [9], each match line is added an additional injection capacitance to share the charge with the match line. Therefore, the voltage of the match line is less than the supply voltage and the power consumption is less than the conventional design in the missing situation. Another scheme for reducing the power consumption of the match line is current-race scheme [10]. The match lines are pre-charged low and evaluated by charging the match lines with a current  $I_{ML}$  from a current source. Furthermore, the search lines are also pre-charged to the search data during the match line is pre-charged. A sense amplifier with half-latch is used to fast sense the match result in each match line. In the missing state, the voltage of the match line is charged to  $I_{ML} \times R_{ML} / m$ , where  $m$  is the number of missing cells in one match line. Otherwise, the match line is charged to high voltage. Hence, this design can save the power consumption of the match line in the missing situation. Moreover, it can also save power on the search line because it eliminates the stage of the search line pre-charge low. The current-saving scheme [11, 12] is similar to the current-race scheme but a current control circuit is added on each match line to control the  $I_{ML}$ . In missing state, the current is less than in the matching state.

Another power consumption of the CAM is search line driving scheme. In the conventional NOR type match line structure, the search line must be pre-charged to low first. If we eliminate the search line pre-charge phase then the dynamic power consumption of the search line in the pre-charge phase can be reduced [7]. Another scheme is hierarchical search line scheme [13]. It is based on pipeline scheme. In the first segment, the match line and search line always are active. But the match line is not pre-charged and the search line is inactive in the following segments when the match line is mismatched in the previous segment. Thus it can reduce the power consumption of the match line and search line. The bank-selection scheme [7, 14, 15] is another design to reduce power consumption, but it focuses on architecture level. The CAM is divided logically into several banks and each bank contains one continuous address space of the CAM. When the CAM performs the search operation, the search data word is divided logically into stored bits and bank-select bits. Then the bank-select bits are used to decide which bank will be active in this search operation. After the bank is active, the stored bits are compared with the search data in the bank. Therefore this scheme can reduce power consumption because only some banks are active in one search operation. But the drawback of the bank-selection scheme is bank overflow. This situation happens when the capacity of the bank is smaller than the number of the stored data.

In the researches of the routing table and TCAM, the topics include encoding the rules and reducing the power consumption of the priority encoder. Most researches focus on encoding the table in TCAM. However, the tables are encoded for binary CAM with a simple scheme in [16]. If the table is encoded then it can store the number of rules as the original design with less space. Therefore this method can reduce the hardware cost and power consumption. Furthermore, the priority encoder is one of the

dominant power consumption and delay of TCAM. A power-optimized 64-bit priority encoder is proposed in [17]. It improves the conventional priority encoder on delay and power consumption and it also can be pipelined.

## 2.1 Selective Pre-charge Scheme

The data words are divided into two segments in the selective pre-charge scheme [18]. Some bits of data are stored in the first segment and the others in the second segment. While the CAM performs search operation, the first segment is active and the relative bits of data are compared. If the relative bits of some data words are matched in the first segment then the associated match line in the second segment will be pre-charged to high. After that, the remaining bits of data are compared in the second segment. If the remaining bits of some data are matched then it means the search data is matched in the CAM. Otherwise, the searching data is mismatched. Therefore, this scheme reduces the power consumption of the match lines. However, if many stored data in the first segment are identical then the power consumption of the CAM is still high.

The simplified architecture of the selective pre-charge scheme is shown in Figure 2-1. The XNOR type CAM cells are used in the first segment and the XOR type CAM cell in the second segment.

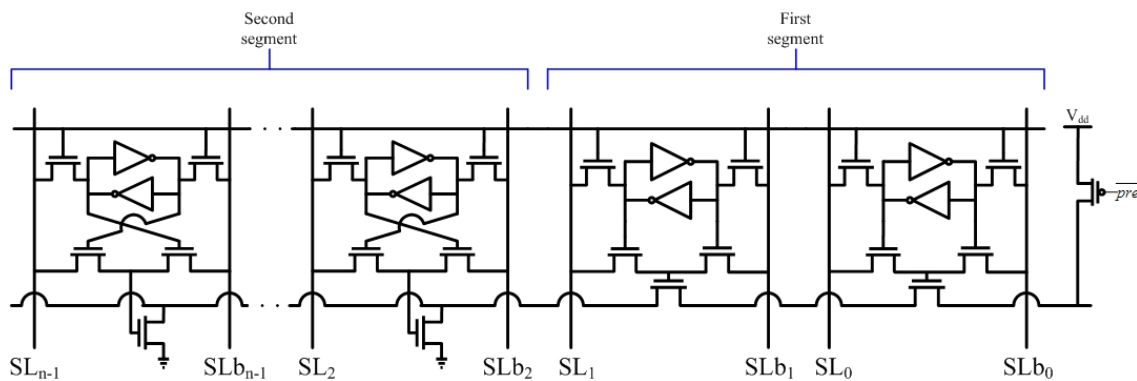


Figure 2-1 The simplified architecture of the selective pre-charge scheme.

## 2.2 Pre-computation Scheme

In this section, we will introduce the concept of pre-computation based CAM (PB-CAM). Then in the following sections, the three different designs of PB-CAM will be introduced.

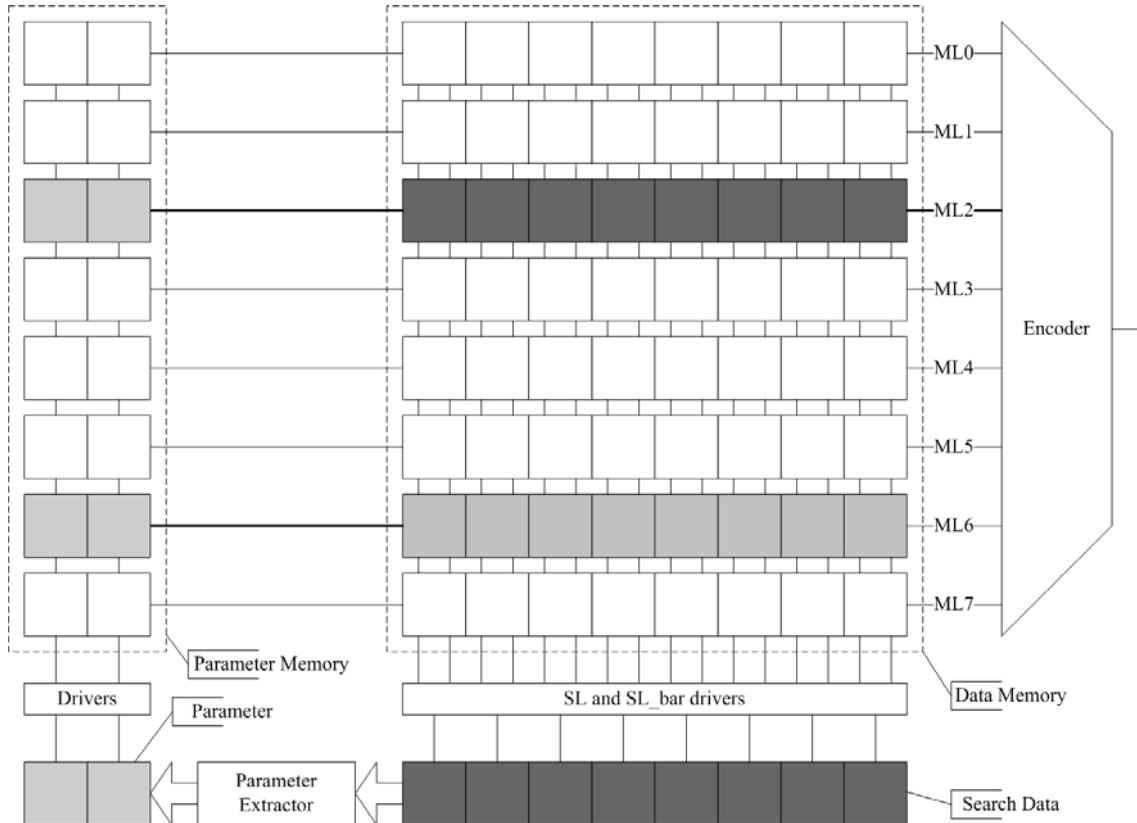


Figure 2-2 The basic architecture of PB-CAM.

The PB-CAM is shown in Figure 2-2 which contains one parameter extractor, drivers, parameter memory, data memory and encoder. The parameters in the parameter memory are calculated from the data by the parameter extractor. When the search operation is performed, the data is inputted into the search line drivers and the parameter extractor. Then the parameter is outputted from the parameter extractor and inputted into drivers. Afterward, the parameter memory can perform the search operation. If the stored parameters are matched then the relative match line in the CAM is pre-charged. After that, the match line can be evaluated. If the stored data is also

matched then the match line remains high. Otherwise, it will be discharged low. This scheme is similar to selective pre-charge scheme but the efficiency of reducing the power consumption depends on the parameter extractor and search data.

## 2.2.1 Ones Count Scheme

The PB-CAM is first proposed in [19]. The parameter extractor is implemented by the ones count function in [19]. The ones count parameter extractor counts the number of binary one which appears in one data word. However the hardware cost and delay of the ones count parameter extractor is expensive when the data word length grows. Furthermore, if we assume the data have a uniform distribution then the number of data that are mapped to parameters is not uniform. This situation is shown in Table 2-1 and the data word length is 14 bits. This distribution is not good for reducing the power consumption of the match line because many identical parameters may appear in the memory in one search operation.

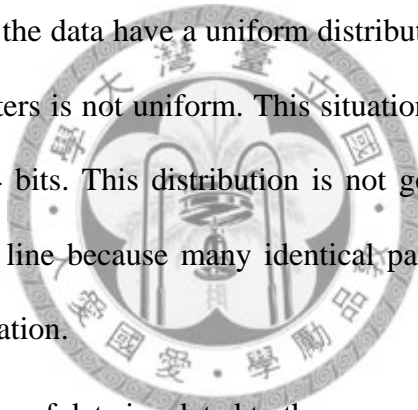


Table 2-1 Number of data is related to the same parameter (ones count).

| Ones count parameter | Relative data    | Average probability |
|----------------------|------------------|---------------------|
| 0                    | 1                | 0.0061035%          |
| 1                    | 14               | 0.0854492%          |
| 2                    | 91               | 0.5554199%          |
| 3                    | 364              | 2.2216797%          |
| 4                    | 1001             | 6.1096191%          |
| 5                    | 2002             | 12.2192383%         |
| 6                    | 3003             | 18.3288574%         |
| 7                    | 3432             | 20.9472656%         |
| 8                    | 3003             | 18.3288574%         |
| 9                    | 2002             | 12.2192383%         |
| 10                   | 1001             | 6.1096191%          |
| 11                   | 364              | 2.2216797%          |
| 12                   | 91               | 0.5554199%          |
| 13                   | 14               | 0.0854492%          |
| 14                   | 1                | 0.0061035%          |
| Total                | $2^{14} = 16384$ | 100%                |



However the ones count parameter extractor is used to cooperate with 7T PB-CAM cell. This PB-CAM cell only uses seven transistors such that the hardware cost and power consumption of data memory are less than the conventional design. The circuitry of 7T PB-CAM cell is shown in Figure 2-3.

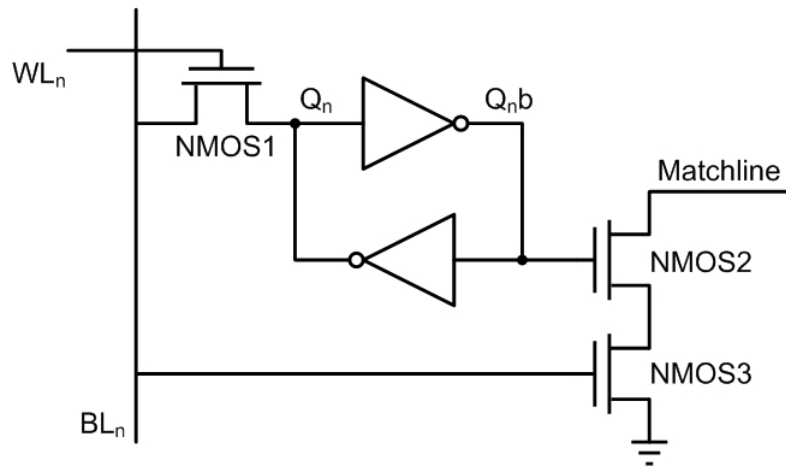


Figure 2-3 7T PB-CAM cell.

This 7T PB-CAM cell is not suitable for every parameter extractor because the match line remains high when  $Q_n = 1$  and  $BL_n = 0$ . However, the 7T PB-CAM cell cooperates with the ones count parameter extractor elegantly. We explain the operations in the following three conditions. First, if the parameter is matched and the data is also matched then the operation of the cell is correct. Second, if the parameter matches the incoming parameter but the data do not. It means the number of binary ones in the stored data and searching data are the same but they are different. Therefore there must be  $Q_i = 1$  and  $BL_i = 0$  in one cell and  $Q_j = 0$  and  $BL_j = 1$  in another cell of the same match line. So the match line is discharged by at least one cell. Third, if the parameter is mismatched and the data is also mismatched then the voltage of the match line is discharged by the static pseudo-NMOS CAM word circuit [19]. This circuit is shown in Figure 2-4. If the parameters are matched then the parameter comparison circuit outputs logic 0. Otherwise, it outputs logic 1. Therefore, the 7T PB-CAM cell can operate well

with the ones count parameter extractor.

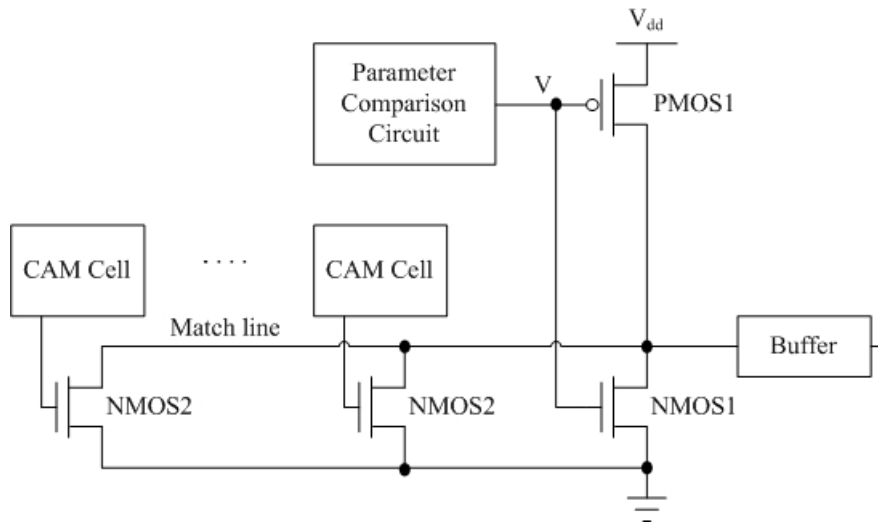


Figure 2-4 Static pseudo-NMOS CAM word circuit.

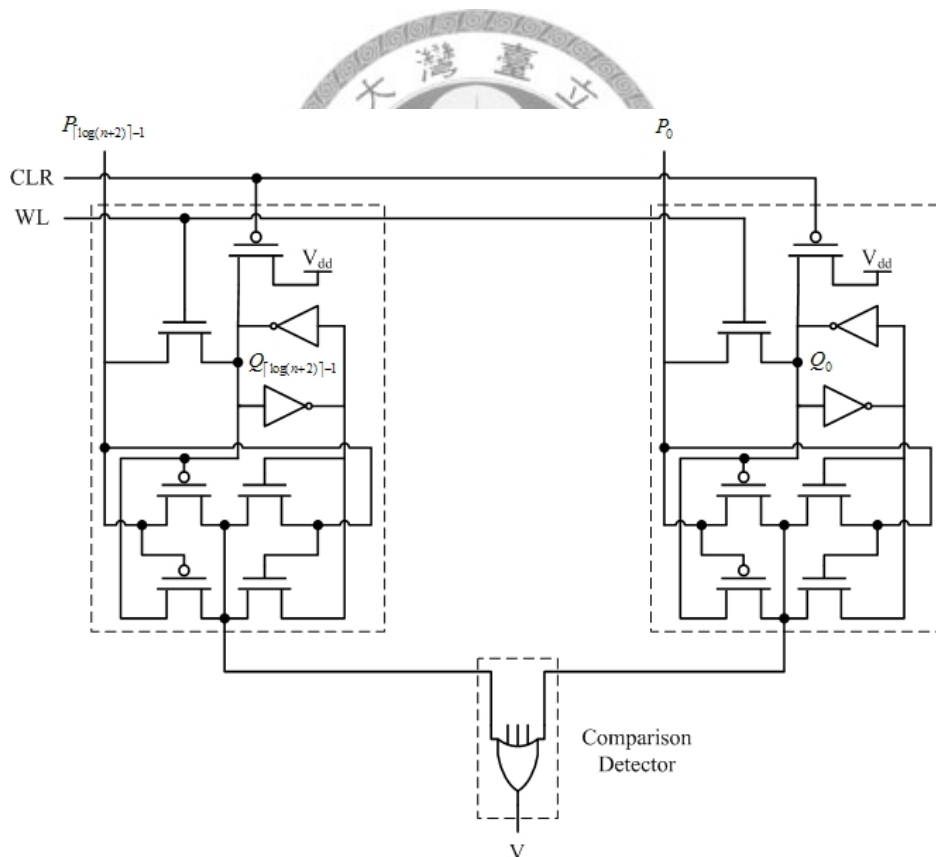


Figure 2-5 Static parameter comparison circuit.

The parameter comparison circuit is shown in Figure 2-5. It performs three operations. First, in parameter setting operation, the CLR is set to logic 0. Then the stored parameter  $Q$  is charged to logic 1. The max parameter means invalid flag in this

circuit because the decimal representation of the max value of one parameter word is larger than the bit length of data. Second, in the parameter writing operation, the WL is asserted such that the parameter P can be written into the storage unit Q. Third, in the parameter comparing operation, if the input parameters P match the stored parameters Q then the comparison detector will output logic 0. Otherwise, it will output logic 1.

### 2.2.2 Block-XOR Scheme

The block-xor scheme is proposed to improve the ones count parameter extractor in [20, 21] but it uses 9T CAM cell. The block-xor parameter extractor is composed of several two fan-in XOR gates. The block diagram of 14 bits block-xor parameter extractor is shown in Figure 2-6 and the design of the valid bit is in the below of Figure 2-6. If the parameters are all binary one then the 4 bits data in the *msb* are used as the parameter. Otherwise, the original parameters are used. So the data is invalid when the stored parameters are all binary one. Furthermore, when the 4 bits data in the *msb* are all binary one, the *msb* of the parameter will be 0. The multiplexer will select the parameter to output. In other words, this design does not have conflicting status.

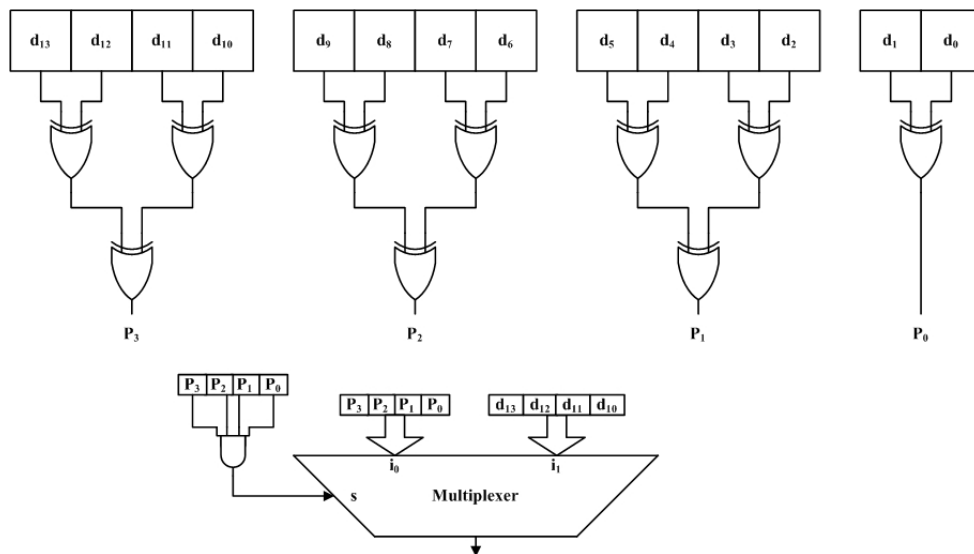


Figure 2-6 The 14 bits block-xor parameter extractor.

If we assume that the distribution of the data is uniform then the block-xor parameter extractor can distribute the data to parameters more uniformly than the ones count parameter extractor. The distribution is shown in Table 2-2 and the data word length is 14 bits.

Table 2-2 Number of data is related to the same parameter (block-xor).

| <b>Block-xor parameter</b> | <b>Relative data</b> | <b>Average probability</b> |
|----------------------------|----------------------|----------------------------|
| 0000                       | 1024                 | 6.25%                      |
| 0001                       | $1024 + (1024/8)$    | 7.03125%                   |
| 0010                       | $1024 + (1024/8)$    | 7.03125%                   |
| 0011                       | 1024                 | 6.25%                      |
| 0100                       | $1024 + (1024/8)$    | 7.03125%                   |
| 0101                       | 1024                 | 6.25%                      |
| 0110                       | 1024                 | 6.25%                      |
| 0111                       | $1024 + (1024/8)$    | 7.03125%                   |
| 1000                       | $1024 + (1024/8)$    | 7.03125%                   |
| 1001                       | 1024                 | 6.25%                      |
| 1010                       | 1024                 | 6.25%                      |
| 1011                       | $1024 + (1024/8)$    | 7.03125%                   |
| 1100                       | 1024                 | 6.25%                      |
| 1101                       | $1024 + (1024/8)$    | 7.03125%                   |
| 1110                       | $1024 + (1024/8)$    | 7.03125%                   |
| 1111                       | Valid bit            |                            |

### 2.2.3 Gate-Block Selection Algorithm

The block-xor can distribute the data to parameters more uniformly when the data have a uniform distribution. However, if the data do not have a uniform distribution and most 4-bit *msb* data are identical then the power consumption is still high. In [22], a gate-block selection algorithm is proposed to synthesize the proper parameter extractor when the data do not have a uniform distribution. This scheme is suitable for the embedded systems because the algorithm must analyze the trace of the system first. Then it tries to find a proper parameter extractor for this system. Furthermore, one equation is used to formulate the average number of comparison operation in the

algorithm when one 2 fan-in gate is used to extract a parameter. This equation is shown in Equation 2-1.

$$\begin{aligned}
 C_{avg} &= N_0 \times (1 - p) + N_1 \times p \\
 &= N_0 \times \left( \frac{N_0}{N_0 + N_1} \right) + N_1 \times \left( \frac{N_1}{N_0 + N_1} \right) \quad (\text{Equation 2-1}) \\
 &= \frac{N_0^2 + N_1^2}{N_0 + N_1}
 \end{aligned}$$

Where

$P =$  For all data, the probability of a two fan-in gate outputting the logic 1 in one block.

$N_0 =$  For all data, the number of zero entries in one parameter.

$N_1 =$  For all data, the number of one entries in one parameter.

#### Gate-Block Selection Algorithm

**Input Data** =  $(D_0, D_1, \dots, D_{n-1})$

$n$ : bit length of the input data,

$l$ : number of input bits for each partition block.

Step 1: Record

$$\text{NAND\_parameter}(k) = \overline{D_{2i} \bullet D_{2i+1}}$$

$$\text{NOR\_parameter}(k) = \overline{D_{2i} + D_{2i+1}}$$

$$\text{XOR\_parameter}(k) = \overline{D_{2i} \oplus D_{2i+1}}$$

**For**  $i, k = 0, 1, \dots, (n/2)-1, \forall$  input patterns

Step 2: Compute  $\text{NAND\_}C_{avg}(k), \text{NOR\_}C_{avg}(k), \text{XOR\_}C_{avg}(k)$

Using Equation 2-1,  $\forall k$

Step 3: Select a logic gate with the minimal  $C_{avg}(k), \forall k$

Step 4: **If** generated parameter bits  $> \lceil n/l \rceil$

Repeat step 1 to step 3 and use previous generated parameter as input data.

**Else**

Finish

Figure 2-7 The Gate-Block Selection Algorithm.

The gate-block selection algorithm is shown in Figure 2-7. In step 1, the output parameters of each gate are computed in each 2-bit block for all data. Then in step 2 and 3, the  $C_{avg}$  is computed for each gate. One logic gate which has minimal  $C_{avg}$  is selected for each 2-bit block. It means that the selected logic gate can make the average number of comparison operation lower than the others. In step 4, it determines whether the algorithm finishes or not.

Table 2-3 The time complexity of gate-block selection algorithm.

| Level | Time complexity of each level  |
|-------|--|
| 1     | Step 1: $m \times (n/2) \times g \times c_1$<br>Step 2: $(n/2) \times g \times c_2$<br>Step 3: $(n/2) \times c_3$<br>Step 4: $c_4$       |
| 2     | Step 1: $m \times (n/2^2) \times g \times c_1$<br>Step 2: $(n/2^2) \times g \times c_2$<br>Step 3: $(n/2^2) \times c_3$<br>Step 4: $c_4$ |
| ...   |  |
| k     | Step 1: $m \times (n/2^k) \times g \times c_1$<br>Step 2: $(n/2^k) \times g \times c_2$<br>Step 3: $(n/2^k) \times c_3$<br>Step 4: $c_4$ |

We analyze the time complexity of gate-block selection algorithm in Table 2-3 when the size of each block is two bits. We also define some variables in the following.

Let

$m$ : The number of the unique data. The unique data means that only one data are chosen and the other duplicate data is discarded.

$n$ : The word length of data.

$k$ : The value of  $k$  is the level that we want to synthesize. The range of  $k$  is  $1 \leq k \leq \lceil \log_2 n \rceil$ .

$g$ : The number of different gate is used in the gate-block selection algorithm. They

are NAND, NOR, XOR in Figure 2-7. Therefore, the value of  $g$  is 3.

$c$ : Constant value.

Therefore, the total time complexity is the sum of the time complexity of each level.

That is  $(c_1 \times m \times g \times n + c_2 \times g \times n + c_3 \times n) \times [1 - (\frac{1}{2})^k] + c_4 \times k$ . The time complexity is polynomial time  $O(mn)$ .

## 2.3 Motivation and Objective

The block-xor scheme can not distribute data to different parameters uniformly when the data do not have a uniform distribution. Furthermore, the execution time of gate-block selection algorithm will grow when the number of different gates that are used to synthesize the parameter extractor is increasing in the algorithm. Therefore, we want to use more different gates to synthesize the parameter extractor for PB-CAM in order to distribute the data more uniformly than block-xor scheme and gate-block selection algorithm. We also want to decrease the execution time when more different gates are used. Moreover, we want to reduce the impact on mapping data to parameters when some blocks have a lot of identical data.

## Chapter 3 Proposed Approach

We will introduce the benefit of distributing the data uniformly and our synthesizing algorithm for the parameter extractor of PB-CAM in the following sections. We will also introduce the method to reduce the impact on mapping data to parameters when some blocks have a lot of identical data.

### 3.1 The Benefit of Distributing the Data Uniformly

Table 3-1 The average number of matched rows for four 2-bit parameters.

| msb<br>0 | msb<br>1 | lsb<br>0 | lsb<br>1 | Case  | Parameter<br>example | Avg.<br>match<br>(case) | Prob. of<br>occurrence | Avg.<br>match<br>(distributi<br>on) |
|----------|----------|----------|----------|-------|----------------------|-------------------------|------------------------|-------------------------------------|
| 2        | 2        | 2        | 2        | best  | 00, 01<br>10, 11     | 1                       | 2/3                    | 1.33                                |
|          |          |          |          | worse | 00, 00<br>11, 11     | 2                       | 1/3                    |                                     |
| 2        | 2        | 3        | 1        | best  | 00, 00<br>10, 11     | 1.5                     | 1                      | 1.5                                 |
|          |          |          |          | worse | 00, 00<br>10, 11     | 1.5                     |                        |                                     |
| 3        | 1        | 3        | 1        | best  | 00, 00<br>10, 01     | 1.5                     | 3/4                    | 1.75                                |
|          |          |          |          | worse | 00, 00<br>00, 11     | 2.5                     | 1/4                    |                                     |
| 2        | 2        | 4        | 0        | best  | 00, 00<br>10, 10     | 2                       | 1                      | 2                                   |
|          |          |          |          | worse | 00, 00<br>10, 10     | 2                       |                        |                                     |
| 3        | 1        | 4        | 0        | best  | 00, 00<br>00, 10     | 2.5                     | 1                      | 2.5                                 |
|          |          |          |          | worse | 00, 00<br>00, 10     | 2.5                     |                        |                                     |
| 4        | 0        | 4        | 0        | best  | 00, 00<br>00, 00     | 4                       | 1                      | 4                                   |
|          |          |          |          | worse | 00, 00<br>00, 00     | 4                       |                        |                                     |



The average number of matched rows for four 2-bit parameters in the parameter memory is shown in Table 3-1 and each parameter bit is extracted from one data block. Although each parameter is related to one data, the data is not shown in Table 3-1. The first column to fourth column is the number of “zero” or “one” in the *msb* or *lsb* of the four 2-bit parameters. It means the distribution of each parameter bit. For example, in the second row, there are two binary ones and two binary zeros in the *msb* of these four 2 bits parameters as well as in the *lsb*. If we assume that these four parameters are all in the parameter memory and every parameter is searched one time. Then the average number of matched rows are  $(1 + 1 + 1 + 1) / 4 = 1$  in the best case and  $(2 + 2 + 2 + 2) / 4 = 2$  in the worse case. Moreover, the probability of the occurrence of the best case is  $4! / [4! / (2! \times 2!)]^2 = 2/3$  and  $2 \times [4! / (2! \times 2!)] / [4! / (2! \times 2!)]^2 = 1/3$ , the worse case. Therefore, the average number of matched rows under this distribution are  $1 \times (2/3) + 2 \times (1/3) = 4/3 \cong 1.33$ . We can see that the average matched rows are small when the data in the same block are mapped to parameters and the distribution of these parameters is uniform. Therefore, if we can map the data in the same block to parameters and the number of binary zeros in this parameter bit position is close to that of binary one. Then the average number of matched rows will decrease in the parameter memory. The power consumption will also be decreased in the PB-CAM because the number of the match lines being pre-charged in the data memory is decreasing. So our algorithm is based on this idea to distribute the data to the parameters such that the power consumption of PB-CAM is smaller than that of block-xor scheme and gate-block selection algorithm.

## 3.2 Local Grouping Algorithm

In this section, we will introduce our algorithm. Before introducing the algorithm, we will define some variables and terminologies first. And then we also analyze the time complexity of our algorithm and compare it with that of gate-block selection algorithm.

### 3.2.1 Definition of the Variables

- $d_i$ : One bit of the data in the bit position  $i$ .
- $p_i$ : One bit of the parameter in the bit position  $i$ .
- $n$ : The word length of the data. So the data is  $d_{n-1}d_{n-2}\dots d_1d_0$ .
- $m$ : The number of unique data in the trace of the system. The unique data means that only one data are chosen from original data and the other duplicate data are discarded.
- $S$ : The set of unique data. So  $|S| = m$ .
- $B_i$ : Each data is divided into several blocks logically. The block  $B_i$  contains all data that are in the same position  $i$ .
- $bs$ : The size of each block. The size of each block is two bits in this algorithm.
- $C_i$ : A set  $C_i = \{(0, c_0), (1, c_1), \dots, (2^{bs} - 1, c_{2^{bs}-1})\}$  contains several pairs for each  $bs$ -bit data block  $B_i$ . The first element of the pair is the decimal representation of the  $bs$ -bit data in the block  $B_i$ . The second element of the pair is the number of times the data appears in the data block  $B_i$  of set  $S$ , which is relative to the first element of the pair.
- $g_{i,j}$ : A synthesized logic gate type of the parameter extractor in the  $i$ -th level and the position  $j$ .

An example of the synthesized parameter extractor is shown in Figure 3-1. This parameter extractor has two levels for  $n$  bits data. The level of the parameter extractor begins at one.

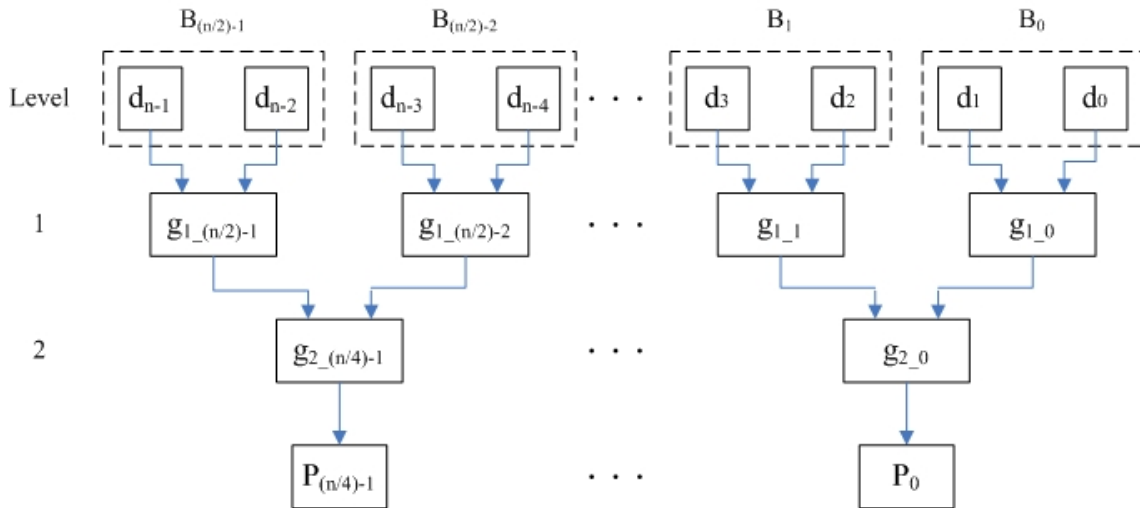


Figure 3-1 An example of the 2-level parameter extractor for the  $n$  bits data.

### 3.2.2 Top Level of Local Grouping Algorithm

The top level of the local grouping algorithm is shown in Figure 3-2. In the lines 11 to 16, each 2-bit data block of all unique input data is analyzed and statistics gathered because we need the information to synthesize the proper parameter extractor. In the lines 18 to 21, the elements of the  $C_i$  of each block are sorted by the count of each decimal number in descending order. Then the sorted  $C_i$  is inputted into the *FindGate* function that will be introduced later. This function will output the proper type of the logic gate that can distribute the data block uniformly to one bit parameter. Then the output type is recorded. In the lines 23 to 29, if the level of the synthesized parameter extractor is that we want then the algorithm is finished. Otherwise, the new data are computed by the new generating gates and old data for next level. The other variables are also prepared for next level.

```

1 Local Grouping Algorithm
2 Input:
3 S:      The set of unique data.
4 n:      The word length of data. So data is  $d_{n-1}d_{n-2} \dots d_1d_0$ .
5 level:   The number of levels of the parameter extractor that we want to
           synthesize.
6 Output: The parameter extractor.
7 Local variable:
8  $C_i$ : Set of counting pairs  $\{(0, c_0), (1, c_1), (2, c_2), (3, c_3)\}$  for 2-bit block  $B_i$ .
           Initial value of  $c_0, c_1, c_2$  and  $c_3$  is zero.
9  $li$ : The index of the level in the parameter extractor. Initial value is one.
10 {
11   for all data  $\in S$  {
12     for all 2-bit data block  $d_{2i+1}d_{2i}$ ,  $i = 0$  to  $\lfloor n/2 \rfloor - 1$  {
13        $j =$  decimal presentation of  $d_{2i+1}d_{2i}$  ;
14        $c_j = c_j + 1$  in the  $(j, c_j)$  of  $C_i$ ;
15     }
16   }
17
18   for  $i = 0$  to  $\lfloor n/2 \rfloor - 1$  {
19     Sort the elements  $(0, c_0), (1, c_1), (2, c_2)$  and  $(3, c_3)$  of  $C_i$  in descending
           order by  $c_j$ , where  $j \in \{0,1,2,3\}$ ;
20     FindGate (sorted  $C_i$ ) and record the gate type in the  $li$ -th level and
            $i$ -th position of the parameter extractor;
21   }
22
23   if ( $li < level$ ) {
24     Using the data set and the generating gates in the  $li$ -th level of the
           parameter extractor to compute the new data set and replace data set
           with the new data set;
25      $n =$  word length of new data;
26     Initialize  $C_i$  for new data; //  $i = 0$  to  $\lfloor n/2 \rfloor - 1$ 
27      $li = li + 1$ ;
28     Goto line 11;
29   }
30 }

```

Figure 3-2 The top level of the local grouping algorithm.

### 3.2.3 Grouping Function

Before introducing the *FindGate* function, we need to implement our idea first. As our mention before, we want to map the data in the same block to parameters and the number of binary zeros in this parameter bit position is close to that of binary ones. Therefore, we analyze the data and gather the statistics first in Figure 3-2. Then we use these statistics to group the data in the same block into two groups such that the difference of the relative count of these two groups is minimal. After that, we can map one group to zero and the other group to one. Therefore, the proper gates can be found by these two groups. The general version of the grouping function for two groups is shown in Figure 3-3. Then we modify this function such that it can run faster when the size of the block is two bits. This modified function is shown in Figure 3-4.

```

1  Grouping function // General version for two groups
2  Input:
3  Sorted  $C_i = \{(j_0, c_{j_0}), (j_1, c_{j_1}), \dots, (j_k, c_{j_k})\}$  and  $c_{j_0} \geq c_{j_1} \geq \dots \geq c_{j_k}$ .
   Where  $k = 2^{bs} - 1$  and bs is block size. The block size is two bits that are
   unnecessary in this function.
4  Output: Two groups  $G_0$  and  $G_1$ . Initial is empty.
5  Local variable:  $gc_0 = gc_1 = 0, p = 1, q=0$ 
6  {
7     Place  $j_0$  into  $G_0$  and  $j_1$  into  $G_1$ 
8      $gc_0 = gc_0 + c_{j_0}, \quad gc_1 = gc_1 + c_{j_1}$ 
9     for  $i = 2$  to  $2^{bs}-1$ 
10    {
11       place the  $j_i$  into  $G_p$ 
12        $gc_p = gc_p + c_{j_i}$ 
13       if ( $gc_p \geq gc_q$ ) swap ( $p, q$ )
14    }
15 }

```

Figure 3-3 The general grouping function.

```

1 SimpleGrouping function
2 Input:
3 Sorted  $C_i$ : The element  $(j, c_j)$  of  $C_i$  is sorted by  $c_j$  in descending order
   such that  $c_{j_0} \geq c_{j_1} \geq c_{j_2} \geq c_{j_3}$ . The block size is two bits.
4 Output: The groups  $G_0$  and  $G_1$ 
5 {
6     Place the  $j_0$  into  $G_0$  and the  $j_1$  and  $j_2$  into  $G_1$ ;
7     if  $((c_{j_1} + c_{j_2}) \geq c_{j_0})$  place the  $j_3$  into  $G_0$ ;
8     else place the  $j_3$  into  $G_1$ 
9 }

```

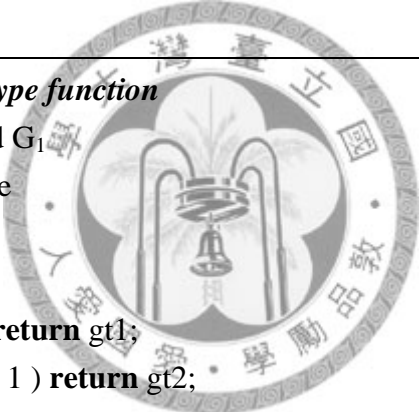
Figure 3-4 The simple grouping function for 2-bit block.

These two grouping functions choose one element from  $C_i$  in order and place this element into a group. After adding this element, if the total count of this group is greater than that of the other group then the next chosen element will be added into the other group. These functions use the greedy method to choose the elements. Therefore, these two functions can make the difference of the count of two groups is minimal.

Table 3-2 The available synthesized gate types for 2-bit block. (a: msb, b: lsb)

| <b>Mapping<br/>Grouping</b>          | <b><math>G_0</math> map to 0<br/><math>G_1</math> map to 1</b> | <b><math>G_0</math> map to 1<br/><math>G_1</math> map to 0</b> | <b>Identity of gate<br/>type</b> |
|--------------------------------------|--|--|----------------------------------|
| $G_0 = \{0\}$<br>$G_1 = \{1, 2, 3\}$ | a or b =<br>not (a nor b)                                      | a nor b  | gt1                              |
| $G_0 = \{1\}$<br>$G_1 = \{0, 2, 3\}$ | a or (not b) =<br>(not a) nand b                               | (not a) and b =<br>a nor (not b)                               | gt2                              |
| $G_0 = \{2\}$<br>$G_1 = \{0, 1, 3\}$ | (not a) or b =<br>a nand (not b)                               | a and (not b) =<br>(not a) nor b                               | gt3                              |
| $G_0 = \{3\}$<br>$G_1 = \{0, 1, 2\}$ | a nand b   | a and b =<br>not (a nand b)                                    | gt4                              |
| $G_0 = \{0, 1\}$<br>$G_1 = \{2, 3\}$ | a  | not a  | gt5                              |
| $G_0 = \{0, 2\}$<br>$G_1 = \{1, 3\}$ | b  | not b  | gt6                              |
| $G_0 = \{0, 3\}$<br>$G_1 = \{1, 2\}$ | a xor b  | a xnor b   | gt7                              |

Then we want to find the synthesized gate from the groups so we analyze all groups first. The total combinations of groups are shown in the first column of Table 3-2 when the size of each block is two bits. Moreover, the fourteen gate types under two mapping status are shown in the second and third columns. We also mark each row an identity because we only choose one gate type from each row in our algorithm. Besides, all the gate types in Table 3-2 can be distinguished by the function in Figure 3-5. So the number of the logic gate types is used in our algorithm that is more than the gate-block selection algorithm. We show the benefit of using these additional gate types in the following example.



```

1 DistinguishAllGateType function
2 Input: Groups  $G_0$  and  $G_1$ 
3 Output: The gate type
4 {
5     if(  $|G_0| = 1$  ){
6         if (  $j_0 = 0$  ) return gt1;
7         else if (  $j_0 = 1$  ) return gt2;
8         else if (  $j_0 = 2$  ) return gt3;
9         else return gt4;
10    }
11    else if(  $|G_0| = 2$  ) {
12        Bitwise xor two elements of  $G_0$  and store the decimal presentation of
13        the result in the variable i;
14        if (  $i = 1$  ) return gt5;
15        else if (  $i = 2$  ) return gt6;
16        else if (  $i = 3$  ) return gt7;
17    }

```

Figure 3-5 The function for distinguishing all gate types.

Table 3-3 An example of the benefit of using additional gate types.

| 2-bit data | Appearance times (count) |
|------------|--------------------------|
| 00         | 3                        |
| 01         | 10                       |
| 10         | 3                        |
| 11         | 3                        |

(a) 2-bit data distribution.

| Gate type     | Parameter 0 | Parameter 1 |
|---------------|-------------|-------------|
| a nor b       | 16          | 3           |
| a nand b      | 3           | 16          |
| a xor b       | 6           | 13          |
| a nor (not b) | 9           | 10          |

(b) The number of data that are related to the parameter.

The data distribution is shown in Table 3-3(a) and the mapping status for four logic gate types is shown in Table 3-3(b). We can see that the gate-block selection algorithm will select the XOR gate as the synthesized gate but the gate type that is in the last row of Table 3-3(b) can perform better than XOR gate. Therefore, we can know that the additional four gate types are necessary, if we want to map the data to the parameter uniformly.

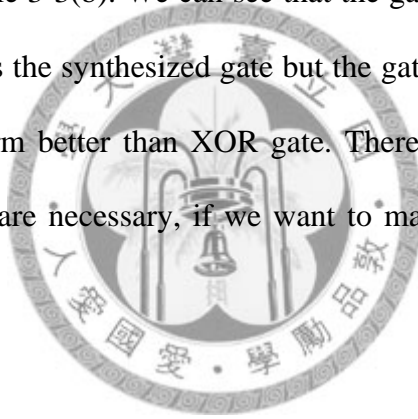


Table 3-4 An example of the choice of the low-cost gate type.

| 2 bits data | Appearances times |
|-------------|-------------------|
| 00          | 3                 |
| 01          | 1                 |
| 10          | 3                 |
| 11          | 3                 |

(a) 2-bit data distribution.

| Gate Type | Parameter 0 | Parameter 1 |
|-----------|-------------|-------------|
| a         | 4           | 6           |
| b         | 6           | 4           |
| a xor b   | 6           | 4           |

(b) The number of data that are related to the parameter.



Although we can use the *SimpleGrouping* function and *DistinguishAllGateType* function as *FindGate* function in our algorithm. However we can see that we have more than one choice when some relative counts of two groups are equal in the last three rows of Table 3-2. In this status, we should choose a low-cost gate type as the synthesized gate instead of depending on the sorted order. An example is shown in Table 3-4. The number of data that are related to the parameters is similar in Table 3-4(b) in these three gate types. Therefore, these three gate types can distribute the data to the parameters uniformly. We can choose the gate type *a* or *b* that are better than *xor* because the number of the CMOS transistors of these two gate types is less and the fan-in is only one. If *a* or *b* gate type appears in the level that is more than one then the hardware cost can be saved. An example is shown in Figure 3-6. Three gate types are saved in the most significant part of the parameter extractor. Therefore, we analyze the order of the counts in the groups in Table 3-5 in order to construct our *FindGate* function.

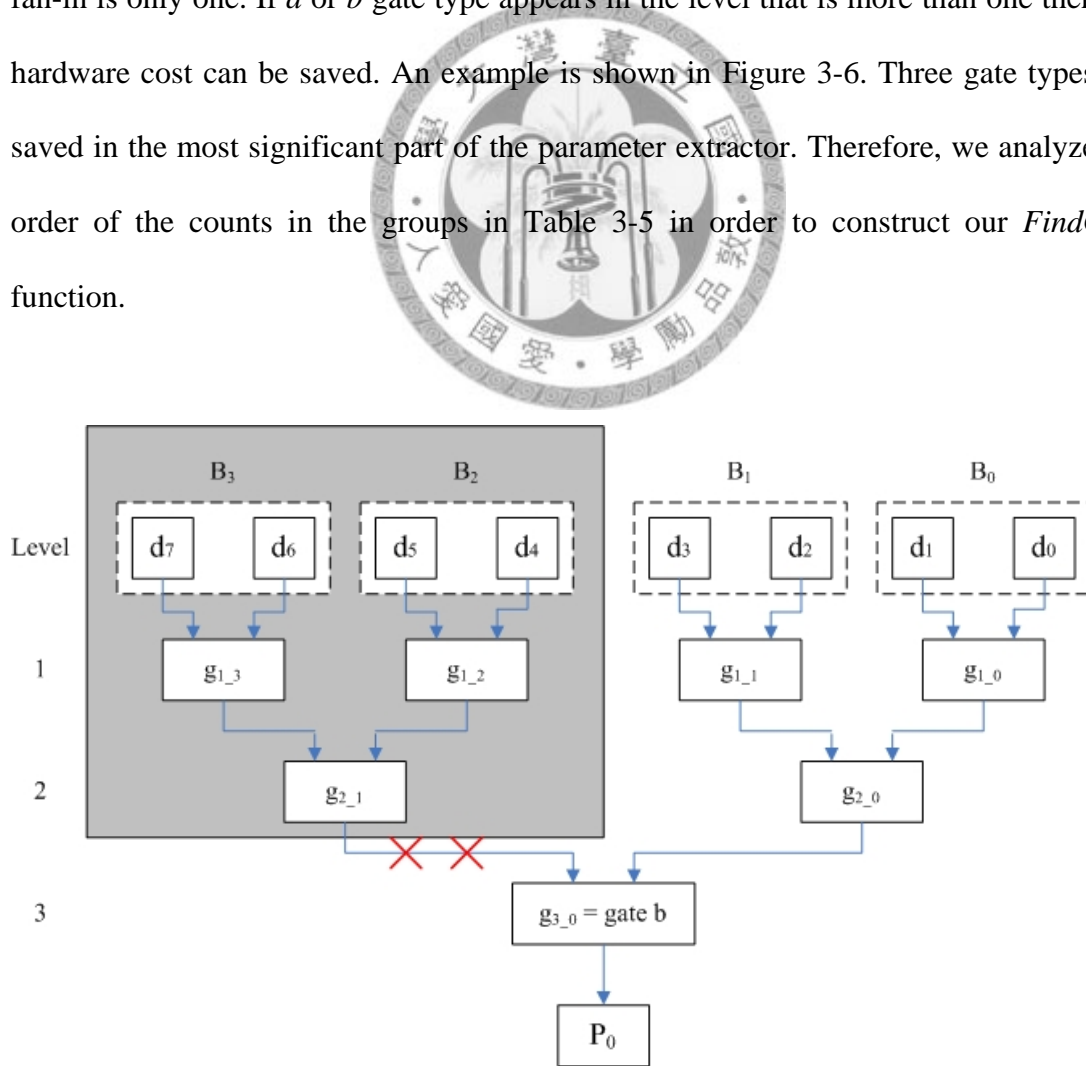


Figure 3-6 The status of reducing hardware cost of the parameter extractor.

### 3.2.4 Find Gate Function

Table 3-5 The method of finding the low-cost gate type.

| Order of counts<br>( $j_0, j_1, j_2, j_3 \in \{0,1,2,3\}, j_0 \neq j_1 \neq j_2 \neq j_3$ ) | Groups<br>( $p, q \in \{0,1\}, p \neq q$ )   | Method of finding gates<br>in the comment of the<br>FindGate function |
|---|--|---|
| $c_{j_0} = c_{j_1} = c_{j_2} = c_{j_3}$<br>$c_{j_0} = c_{j_1} = c_{j_2} > c_{j_3}$          | $G_p = \{j_0, j_1\}$<br>$G_q = \{j_2, j_3\}$ | Method 1  |
|   | $G_p = \{j_0, j_2\}$<br>$G_q = \{j_1, j_3\}$ |   |
|   | $G_p = \{j_0, j_3\}$<br>$G_q = \{j_1, j_2\}$ |   |
| $c_{j_0} = c_{j_1} > c_{j_2} > c_{j_3}$<br>$c_{j_0} = c_{j_1} > c_{j_2} = c_{j_3}$          | $G_p = \{j_0, j_2\}$<br>$G_q = \{j_1, j_3\}$ | Method 2  |
|   | $G_p = \{j_0, j_3\}$<br>$G_q = \{j_1, j_2\}$ |   |
| $c_{j_0} > c_{j_1} = c_{j_2} = c_{j_3}$   | $G_p = \{j_0, j_1\}$<br>$G_q = \{j_2, j_3\}$ | Method 3  |
|   | $G_p = \{j_0, j_2\}$<br>$G_q = \{j_1, j_3\}$ |   |
|   | $G_p = \{j_0, j_3\}$<br>$G_q = \{j_1, j_2\}$ |   |
|   | $G_p = \{j_0\}$<br>$G_q = \{j_1, j_2, j_3\}$ | Method 5  |
|   |  |   |
| $c_{j_0} > c_{j_1} > c_{j_2} = c_{j_3}$   | $G_p = \{j_0, j_2\}$<br>$G_q = \{j_1, j_3\}$ | Method 4  |
|   | $G_p = \{j_0, j_3\}$<br>$G_q = \{j_1, j_2\}$ |   |
|   | $G_p = \{j_0\}$<br>$G_q = \{j_1, j_2, j_3\}$ | Method 5  |
| $c_{j_0} > c_{j_1} > c_{j_2} > c_{j_3}$<br>$c_{j_0} > c_{j_1} = c_{j_2} > c_{j_3}$          | $G_p = \{j_0, j_3\}$<br>$G_q = \{j_1, j_2\}$ | Method 6  |
|   | $G_p = \{j_0\}$<br>$G_q = \{j_1, j_2, j_3\}$ |   |

In the first column of Table 3-5, all possible orders of the count of one 2-bit block are listed and the relative groups are also shown in the second column. For example, in the third row of Table 3-5,  $c_{j_0} = c_{j_1} > c_{j_2} > c_{j_3}$ , so  $c_{j_1} + c_{j_2} > c_{j_0}$ . Each group must have two elements according to the grouping function. Furthermore,  $j_0$  and  $j_1$  can be exchanged in these two groups because  $c_{j_0} = c_{j_1}$ . Therefore, there are two combinations in the third row. Based on Table 3-5 we can implement the *FindGate*

function in Figure 3-7.

```

1  FindGate function
2  Input: Sorted  $C_i = \{(j_0, c_{j_0}), (j_1, c_{j_1}), (j_2, c_{j_2}), (j_3, c_{j_3})\}$  and
       $c_{j_0} \geq c_{j_1} \geq c_{j_2} \geq c_{j_3}$ 
3  Output: One gate type.
4  {
5      bool  j3toG0 = false;
6      if ( $c_{j_1} + c_{j_2} \geq c_{j_0}$ )  j3toG0 = true;
7      if ( $c_{j_0} = c_{j_1}$ ) {
8          if ( $c_{j_1} = c_{j_2}$ )  return one of gt5, gt6 and gt7;  // Method 1
9          else { // Method 2
10             temp =  $j_0 + j_1$ 
11             return DistinguishGateType567 ( temp );
12         }
13     } // end if ( $c_{j_0} = c_{j_1}$ )
14     else if ( $c_{j_2} = c_{j_3}$ ) {
15         if ( j3toG0 = true ) {
16             if ( $c_{j_1} = c_{j_2}$ )  return one of gt5, gt6 and gt7;  // Method 3
17             else { // Method 4
18                 temp =  $j_2 + j_3$ 
19                 return DistinguishGateType567 ( temp );
20             }
21         }
22         else {
23             Place  $j_0$  into  $G_0$ ;  Place  $j_1, j_2$  and  $j_3$  into  $G_1$ ;  // Method 5
24         }
25     } // end else if ( $c_{j_2} = c_{j_3}$ )
26     else { // Method 6
27         //SimpleGrouping function
28         Place  $j_0$  into  $G_0$ ;  Place  $j_1$  and  $j_2$  into  $G_1$ ;
29         if ( j3toG0 = true ) Place  $j_3$  into  $G_0$ ;
30         else Place  $j_3$  into  $G_1$ ;
31     } // end else
32     return DistinguishAllGateType ( $G_0, G_1$ );
33 }

```

Figure 3-7 The find gate function.

This function is easy to construct, for example, if our condition matches the method 2 then we can determine whether  $c_{j_0}$  is equal to  $c_{j_1}$  or not in order to separate the methods 1 and 2 from the others in Table 3-5. After that, we can also determine whether  $c_{j_1}$  is equal to  $c_{j_2}$  or not to separate the methods 1 and 2. So we can distinguish these methods easily. After distinguishing these methods, we need to choose the gate type as the synthesized gate. Therefore, we classified these methods under three situations. First, the methods 1 and 3 cover all combinations of the groups when each group has two elements so we can choose one gate type from *gt5*, *gt6* and *gt7*. Second, the methods 2 and 4 cover some combinations of the groups so we need another function to choose the logic gate type that will be explained later. Third, the methods 5 and 6 can use the idea of the *SimpleGrouping* function to group the elements and then the *DistinguishAllGateType* function can be used to choose a proper gate type.

The total combinations of the groups of methods 2 and 4 are arranged in Table 3-6. We can find the relation between different statuses of the groups in this table. For example, if  $c_1 = c_2 > c_0 > c_3$  then each group has two elements that is based on the grouping idea. Moreover, numbers 1 and 2 can be exchanged. This status is shown in the 7<sup>th</sup> row of the method 2 of Table 3-6 and the relative gate type is *gt5* and *gt6*. So we can choose one gate type to output. We also find the characteristic value of the different groups of these two methods in the first column of Table 3-6 such that we can implement the distinguishing function easily. The characteristic value of the method 2 is  $j_0 + j_1$  and that of method 4 is  $j_2 + j_3$  in our algorithm. But it is easy to find that the characteristic value of the methods 2 and 4 can also be  $j_0 + j_1$  or  $j_2 + j_3$ . Then we can implement the *DistinguishGateType567* function as in Figure 3-8 that is based on the characteristic value.

Table 3-6 The relation of the logic gate type of the methods 2 and 4.

| Method   | 2                                       |                |                |                |           | 4                                       |                |                |                |           |
|--|---|----------------|----------------|----------------|-----------|---|----------------|----------------|----------------|-----------|
| Order of counts  | $c_{j_0} = c_{j_1} > c_{j_2} > c_{j_3}$ |                |                |                |           | $c_{j_0} > c_{j_1} > c_{j_2} = c_{j_3}$ |                |                |                |           |
| Order of counts  | $c_{j_0} = c_{j_1} > c_{j_2} = c_{j_3}$ |                |                |                |           |   |                |                |                |           |
| cv = j <sub>i</sub> + j <sub>i+1</sub><br>(i=0 in Method2)<br>(i=2 in Method4) | j <sub>0</sub>                          | j <sub>1</sub> | j <sub>2</sub> | j <sub>3</sub> | Gate type | j <sub>0</sub>                          | j <sub>1</sub> | j <sub>2</sub> | j <sub>3</sub> | Gate type |
| 1  | 0                                       | 1              | 2              | 3              | gt6,<br>7 | 2                                       | 3              | 0              | 1              | gt6,<br>7 |
|  |   |                | 3              | 2              |           | 3                                       | 2              |                |                |           |
|  | 1                                       | 0              | 2              | 3              |           | 2                                       | 3              | 1              | 0              |           |
|  |   |                | 3              | 2              |           | 3                                       | 2              |                |                |           |
| 2  | 0                                       | 2              | 1              | 3              | gt5,<br>7 | 1                                       | 3              | 0              | 2              | gt5,<br>7 |
|  |   |                | 3              | 1              |           | 3                                       | 1              |                |                |           |
|  | 2                                       | 0              | 1              | 3              |           | 1                                       | 3              | 2              | 0              |           |
|  |   |                | 3              | 1              |           | 3                                       | 1              |                |                |           |
| 3  | 0                                       | 3              | 1              | 2              | gt5,<br>6 | 1                                       | 2              | 0              | 3              | gt5,<br>6 |
|  |   |                | 2              | 1              |           | 2                                       | 1              |                |                |           |
|  | 3                                       | 0              | 1              | 2              |           | 1                                       | 2              | 3              | 0              |           |
|  |   |                | 2              | 1              |           | 2                                       | 1              |                |                |           |
| 3  | 1                                       | 2              | 0              | 3              | gt5,<br>6 | 0                                       | 3              | 1              | 2              | gt5,<br>6 |
|  |   |                | 3              | 0              |           | 3                                       | 0              |                |                |           |
|  | 2                                       | 1              | 0              | 3              |           | 0                                       | 3              | 2              | 1              |           |
|  |   |                | 3              | 0              |           | 3                                       | 0              |                |                |           |
| 4  | 1                                       | 3              | 0              | 2              | gt5,<br>7 | 0                                       | 2              | 1              | 3              | gt5,<br>7 |
|  |   |                | 2              | 0              |           | 2                                       | 0              |                |                |           |
|  | 3                                       | 1              | 0              | 2              |           | 0                                       | 2              | 3              | 1              |           |
|  |   |                | 2              | 0              |           | 2                                       | 0              |                |                |           |
| 5  | 2                                       | 3              | 0              | 1              | gt6,<br>7 | 0                                       | 1              | 2              | 3              | gt6,<br>7 |
|  |   |                | 1              | 0              |           | 1                                       | 0              |                |                |           |
|  | 3                                       | 2              | 0              | 1              |           | 0                                       | 1              | 3              | 2              |           |
|  |   |                | 1              | 0              |           | 1                                       | 0              |                |                |           |

```

1 DistinguishGateType567 function
2 Input: cv is the characteristic value of groups.
3 Output: The gate type
4 {
5     if ( cv = 3 ) return one of gt5 and gt6;
6     else if ( cv is odd ) return one of gt6 and gt7;
7     else if ( cv is even ) return one of gt5 and gt7;
8 }
```

Figure 3-8 The function of distinguishing gate types 5, 6 and 7.

### 3.2.5 Demonstration of Local Grouping Algorithm

Table 3-7 An example to demonstrate local grouping algorithm. (First level)

| Data |
|------|
| 0011 |
| 1101 |
| 0100 |
| 1100 |
| 1001 |

(a)

| Block $B_1$ | Block $B_0$ |
|-------------|-------------|
| 00          | 11          |
| 11          | 01          |
| 01          | 00          |
| 11          | 00          |
| 10          | 01          |

(b)

|        | $C_1$                      | $C_0$                      |
|--------|----------------------------|----------------------------|
| Binary | (Decimal, Count of $B_1$ ) | (Decimal, Count of $B_0$ ) |
| 00     | (0, 1)                     | (0, 2)                     |
| 01     | (1, 1)                     | (1, 2)                     |
| 10     | (2, 1)                     | (2, 0)                     |
| 11     | (3, 2)                     | (3, 1)                     |

(c)

| Sorted $C_1$               | Sorted $C_0$               |
|----------------------------|----------------------------|
| (Decimal, Count of $B_1$ ) | (Decimal, Count of $B_0$ ) |
| (3, 2)                     | (0, 2)                     |
| (0, 1)                     | (1, 2)                     |
| (1, 1)                     | (3, 1)                     |
| (2, 1)                     | (2, 0)                     |

(d)

After introducing the local grouping algorithm, we use an example to demonstrate the local grouping algorithm. The five 4-bit data are shown in Table 3-7(a). The data are divided into two 2-bit data block logically in Table 3-7(b). Then we count the 2-bit data in each block in Table 3-7(c). For example, the 2-bit binary data  $(11)_2 = (3)_{10}$  appears two times in the block  $B_1$ . So  $C_1$  has an element (3, 2) in Table 3-7(c). After counting

the 2-bit data, the elements of  $C_0$  and  $C_1$  are sorted by the count in descending order in Table 3-7(d). Then the sorted  $C_0$  and  $C_1$  are inputted into the *FindGate* function. First, the sorted  $C_0$  matches the method 2 in Table 3-5 because  $(c_0 = 2) = (c_1 = 2) > (c_3 = 1) > (c_2 = 0)$ . So the characteristic value is  $(0 + 1) = 1$  that is inputted into the *DistinguishGateType567* function. One of the gate types 6 and 7 can be chosen, we choose gate type 6 because the number of the CMOS transistors is fewer. Second, the sorted  $C_1$  matches the method 3 and each group has two elements in Table 3-5 because  $(c_3 = 2) > (c_0 = 1) = (c_1 = 1) = (c_2 = 1)$  and  $c_0 + c_1 \geq c_3$ . So we can choose gate types 5, 6 and 7 in the *FindGate* function. We choose gate type 6 here because the number of CMOS transistors of the gate types 5 and 6 is fewer than gate type 7. Moreover, the fan-in of the gate type 6 is connected to the least significant bit of the 2-bit data. Therefore, the first level of the parameter extractor has two gate types 6.

Table 3-8 An example to demonstrate local grouping algorithm. (Second level)

| <b>Block <math>B_0</math> of new data</b> |    |
|---|----|
|   | 01 |
|   | 11 |
|   | 10 |
|   | 10 |
|   | 01 |

(a)

|               | <b><math>C_0</math></b>                     |
|---------------|---|
| <b>Binary</b> | <b>(Decimal, Count of <math>B_0</math>)</b> |
| 00            | (0, 0)                                      |
| 01            | (1, 2)                                      |
| 10            | (2, 2)                                      |
| 11            | (3, 1)                                      |

(b)

| <b>Sorted <math>C_0</math></b>              |        |
|---|--------|
| <b>(Decimal, Count of <math>B_0</math>)</b> |        |
|   | (1, 2) |
|   | (2, 2) |
|   | (3, 1) |
|   | (0, 0) |

(c)

After synthesizing the first level of the parameter extractor, if we implement gt6 with a buffer then the new data are calculated by the gates in the first level and the old data in Table 3-8(a). Then the number of 2-bit data are counted and sorted for each block in Table 3-8(b) and (c). The sorted  $C_0$  in Table 3-8(c) also matches the method 2 and each group has two elements. So we also choose gate type 6. Therefore, we need four inverters to construct the parameter extractor when the buffer is implemented by two inverters. However the gate-block selection algorithm needs one NAND, NOR and XOR. The synthesized parameter extractor is shown in Figure 3-9. So the hardware cost of our synthesized parameter extractor is fewer than that of the gate-block selection. Moreover, the data are also distributed to the parameters uniformly in these two parameter extractors.

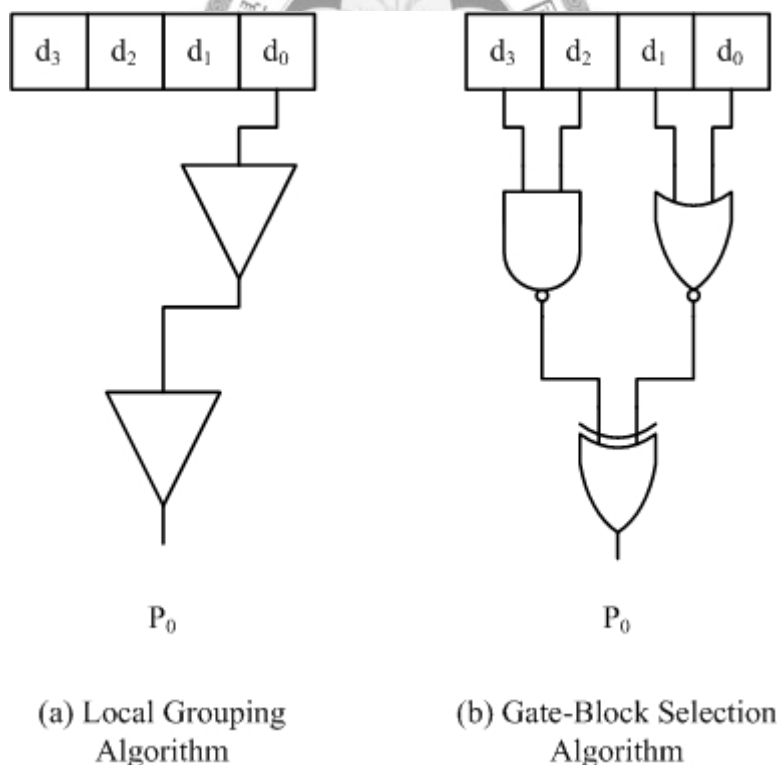


Figure 3-9 The synthesized parameter extractor of the demonstrative example.



### 3.2.6 Time Complexity of Local Grouping Algorithm

Table 3-9 The time complexity of the local grouping algorithm.

| Level | Time complexity of each level   |
|-------|---|
| 1     | Line 11~16: $m \times (n/2) \times c_5$<br>Line 18~21: $(n/2) \times c_6$<br>Line 23: $c_7$<br>Line 24: $m \times (n/2) \times c_8$<br>Line 25, 27, 28: $c_9$<br>Line 26: $(n/2^2)c_{10}$       |
| 2     | Line 11~16: $m \times (n/2^2) \times c_5$<br>Line 18~21: $(n/2^2) \times c_6$<br>Line 23: $c_7$<br>Line 24: $m \times (n/2^2) \times c_8$<br>Line 25, 27, 28: $c_9$<br>Line 26: $(n/2^3)c_{10}$ |
| ...   |   |
| k     | Line 11~16: $m \times (n/2^k) \times c_5$<br>Line 18~21: $(n/2^k) \times c_6$<br>Line 23: $c_7$   |

$m$ : The number of the unique data.

$n$ : The word length of data.

$k$ : The value of  $k$  is the level that we want to synthesize. The range of  $k$  is  $1 \leq k \leq \lceil \log_2 n \rceil$ .

$c$ : Constant value.

The total time complexity of the local grouping algorithm is  $m \times n \times \{c_5 \times [1 - (\frac{1}{2})^k] + c_8 \times [1 - (\frac{1}{2})^{k-1}]\} + n \times \{c_6 \times [1 - (\frac{1}{2})^k] + c_{10} \times [\frac{1}{2} - (\frac{1}{2})^{k-1}]\} + k \times c_7 + (k-1) \times c_9$  in Table 3-9. So the time complexity is  $O(mn)$ . Hence the local grouping algorithm is also a polynomial time algorithm. Then we compare the time complexity of the local grouping algorithm with that of gate-block selection algorithm in Table 3-10.

Table 3-10 The comparison of the time complexity of the algorithms.

| <b>Time complexity of the gate-block selection algorithm</b>  |  |
|---|--|
| $c_1 \times m \times g \times n \times [1 - (\frac{1}{2})^k] + c_2 \times g \times n \times [1 - (\frac{1}{2})^k] + c_3 \times n \times [1 - (\frac{1}{2})^k] + c_4 \times k$   |  |
| $c_1$ : The time to calculate the output of one gate and counting the appearance times.   |  |
| $c_2$ : The time to calculate the $C_{avg}$ of one gate in one block.   |  |
| $c_3$ : The time to select one gate that has minimal $C_{avg}$ in one block.  |  |
| $c_4$ : The time to determine whether the algorithm is finish or not.   |  |
| <b>Time complexity of the local grouping algorithm</b>  |  |
| $m \times n \times \{c_5 \times [1 - (\frac{1}{2})^k] + c_8 \times [1 - (\frac{1}{2})^{k-1}]\} + n \times \{c_6 \times [1 - (\frac{1}{2})^k] + c_{10} \times [\frac{1}{2} - (\frac{1}{2})^{k-1}]\} + k \times c_7 + (k-1) \times c_9$ |  |
| $c_5$ : The time to count the appearance times of one block of one data.  |  |
| $c_6$ : The time to sort four elements and find one proper gate type.   |  |
| <p>The time complexity of the <i>FindGate</i> function is constant and the most times of the comparison are nine. Moreover, the time to sort four elements is also constant.</p>  |  |
| $c_7$ : The time to determine whether the algorithm is finish or not.   |  |
| $c_8$ : The time to calculate the output of one gate.   |  |
| $c_9$ : The time to prepare the local variables for next level.   |  |
| $c_{10}$ : The time to initialize the counting set $C_i$ of one block $B_i$ for next level.   |  |
| $c_1 \cong c_5 + c_8$   |  |
| $c_4 \cong c_7$   |  |

The time complexity of these two algorithms is polynomial time  $O(mn)$ . Furthermore, the dominant source of the time complexity is the number of the unique data. So if the number of the unique data increases and it is far more than the word length of the data then the execution time of the gate-block selection algorithm will be larger than that of the local grouping algorithm. We can also see that in Table 3-10 because  $c_1 \cong c_5 + c_8$  and the first product term of the time complexity of the gate-block selection is multiplied by the number of gates that is used to synthesize. Therefore, the execution time of the local grouping algorithm can be smaller than that of the gate-block selection algorithm.

### 3.3 Discard and Interlaced Method

The drawback of the selective pre-charge scheme is the power consumption still remains high when the most data in the first segment are identical [7]. Some parameter extractors of the PB-CAM also meet this problem. For example, in Table 3-11, the most data are identical in the first five columns. If we use the block-xor parameter extractor without valid bit design to map the data to 2-bit parameters and the size of each block is four. Then we can see that the  $p_0$  is uniform distribution but the  $p_1$  is identical in the last second column of Table 3-11. This situation causes the data to be centralized in some parameters such that the average match times are still high. So we introduce a method to reduce the impact on mapping data in the following. Before introducing the method, we will introduce the standard deviation [23] first because we use the standard deviation to measure the variability of the data.

Table 3-11 An example of the problem of the identical data.

| Original data |       |       |       |       |       |       |       | XOR   |       |
|---------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $d_7$         | $d_6$ | $d_5$ | $d_4$ | $d_3$ | $d_2$ | $d_1$ | $d_0$ | $p_1$ | $p_0$ |
| 0             | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0             | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 0     | 1     |
| 0             | 0     | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 1     |
| 0             | 0     | 0     | 0     | 0     | 0     | 1     | 1     | 0     | 0     |
| 0             | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0             | 0     | 0     | 0     | 0     | 1     | 0     | 1     | 0     | 0     |
| 0             | 1     | 0     | 1     | 0     | 1     | 1     | 0     | 0     | 0     |
| 0             | 1     | 0     | 1     | 0     | 1     | 1     | 1     | 0     | 1     |

If there are  $N$  values  $x_1, \dots, x_N$  then the standard deviation of these  $N$  values is in the Equation 3-1.

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (\text{Equation 3-1})$$

Where:  $\bar{x}$  is the mean of the  $N$  values. That is  $\bar{x} = (x_1 + x_2 + \dots + x_{N-1} + x_N) / N$ .

We can see that if the most values are close to the mean then the standard deviation is

small. Otherwise, the standard deviation is high. Therefore, when the standard deviation on the appearance times of the data of one block is high, most data of this block are most likely identical or the data are centralized in some values. So our method is based on this idea that is shown in Figure 3-10. It discards some blocks those have high standard deviation and interlaces some blocks those have high and some blocks those have small standard deviation in order to break the identical data in some blocks.

- 1 ***Discard and Interlaced Method***
  - 2 Divide unique data into  $t$  blocks.
  - 3 Compute the standard deviation of each block.
  - 4 Choose some blocks  $BS_0$  which have high standard deviation and discard them.
  - 5 Choose other blocks  $BS_1$  which have high standard deviation.
  - 6 Choose and copy other blocks  $BS_2$  which have small standard deviation.
  - 7 Interlace the blocks  $BS_1$  and  $BS_2$  as new blocks  $BS_{new0}$ .
- Use the blocks  $BS_{new0}$  and the remaining part of original data as new input data of the parameter extractor.

Figure 3-10 The discard and interlaced method.

Table 3-12 An example to demonstrate the DAI method. (Steps 2 and 3)

| Block                     | $B_3$      |       | $B_2$      |       | $B_1$ |       | $B_0$ |       |
|---------------------------|------------|-------|------------|-------|-------|-------|-------|-------|
| Position                  | $d_7$      | $d_6$ | $d_5$      | $d_4$ | $d_3$ | $d_2$ | $d_1$ | $d_0$ |
| <b>Data</b>               | 0          | 0     | 0          | 0     | 0     | 0     | 0     | 0     |
|                           | 0          | 0     | 0          | 0     | 0     | 0     | 0     | 1     |
|                           | 0          | 0     | 0          | 0     | 0     | 0     | 1     | 0     |
|                           | 0          | 0     | 0          | 0     | 0     | 0     | 1     | 1     |
|                           | 0          | 0     | 0          | 0     | 0     | 1     | 0     | 0     |
|                           | 0          | 0     | 0          | 0     | 0     | 1     | 0     | 1     |
|                           | 0          | 1     | 0          | 1     | 0     | 1     | 1     | 0     |
|                           | 0          | 1     | 0          | 1     | 0     | 1     | 1     | 1     |
| <b>Standard deviation</b> | $\sqrt{6}$ |       | $\sqrt{6}$ |       | 2     |       | 0     |       |

We use the data in Table 3-11 to demonstrate the discard and interlaced method (DAI method). First, the data are divided into four 2-bit data blocks and the standard deviation on the appearance times of each block is calculated in Table 3-12. Second, the standard deviations of the  $B_3$  and  $B_2$  are high so we choose  $BS_0 = \{B_3\}$  and  $BS_1 = \{B_2\}$ .

We also choose  $BS_2 = \{B_0\}$  because it has small standard deviation. After choosing, we discard the  $BS_0$  then data =  $d_5d_4d_3d_2d_1d_0$ . Next, the  $BS_1$  and  $BS_2$  are interlaced and combined with the remaining data so data =  $d_5d_0d_4d_1d_3d_2d_1d_0$ . The reconstructed data are shown in Table 3-13 and the output result of the block-xor parameter extractor without valid bit design is in the last two columns. The average match times in Table 3-13 are smaller than that in Table 3-11. Therefore, we can know that this method can reduce the impact on mapping data when most data are identical or centralized in some values.

Note that the reconstructed data are used as the input data of the parameter extractor instead of storing them in the data memory because we still need the original data to make the comparison result is correct. So the wires are connected with parameter extractor as the bit positions of the reconstructed data in the hardware design. Moreover, this method can also apply to the synthesized algorithm. We should only remove the duplicate data on the reconstructed data and then use them as the input unique data of the algorithm.

Table 3-13 An example to demonstrate the DAI method. (Result)

| Reconstructed data |       |       |       |       |       |       |       | XOR   |       |
|--------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $d_5$              | $d_0$ | $d_4$ | $d_1$ | $d_3$ | $d_2$ | $d_1$ | $d_0$ | $p_1$ | $p_0$ |
| 0                  | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0                  | 1     | 0     | 0     | 0     | 0     | 0     | 1     | 1     | 1     |
| 0                  | 0     | 0     | 1     | 0     | 0     | 1     | 0     | 1     | 1     |
| 0                  | 1     | 0     | 1     | 0     | 0     | 1     | 1     | 0     | 0     |
| 0                  | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0                  | 1     | 0     | 0     | 0     | 1     | 0     | 1     | 1     | 0     |
| 0                  | 0     | 1     | 1     | 0     | 1     | 1     | 0     | 0     | 0     |
| 0                  | 1     | 1     | 1     | 0     | 1     | 1     | 1     | 1     | 1     |

## Chapter 4 Experimental Results

In this section, we will introduce our experimental environment and compare our algorithm and method with the gate-block selection algorithm and block-xor without valid bit design. The experimental results will also be shown in the following sections.

### 4.1 Experimental Environment

Table 4-1 The test data in the experiment.

| Test data (The word length is 32 bits) | Annotation   |
|--|--|
| MiBench [24]                           | Modify the sim-cahce.c in the SimpleScalar [25] and dump the data address that are used to access data TLB as test data.   |
| Random data                            | The data words are separated into 4-bit blocks. First, 15 numbers are generated randomly between 0 and 1000 for each block. Then the 15 numbers divide the area that is from 0 to 1000 into 16 areas. Second, the sizes of these 16 areas are used as the probability of generating data in the blocks. After that, the test data are generated. |

We use two kinds of test data in our experiment. One is MiBench [24] and the other is random data. The method that is used to obtain these data is shown in the second column of Table 4-1. Furthermore, the other configurations for all schemes and the simulation tools are shown in Table 4-2. The high-level simulation tool is used to count the times of all status in the CAM when the test data are searched. For example, the dominant source of the power consumption on the match line of CAM is the match lines are pre-charged and then discharged. So we can count the number of rows of one data is mismatched and the relative parameter is matched in total search. We also count the

number of rows of that one data and relative parameter are mismatched in this search but the parameter and data in the same position are matched in the previous search or replacement. Therefore, we can sum these two counts and use them as the estimative power consumption on the match line of CAM. Moreover, we also pre-fetch data and store them in the CAM before searching.

Table 4-2 The configurations for all schemes in the experiment.

|  |                                  |
|--|----------------------------------|
| <b>Spice simulation tool</b>                               | Synopsys NanoSim                 |
| <b>High level simulation tool</b>                          | Design by ourselves              |
| <b>Technology</b>  | CIC 0.18um SPICE MODEL           |
| <b>Supply voltage</b>                                      | 1.8 V                            |
| <b>CAM cell</b>  | Conventional 9T XOR type         |
| <b>Parameter comparison circuit</b>                        | As in the ones count scheme [19] |
| <b>Match line structure of data memory</b>                 | NOR type match line              |
| <b>Word structure</b>                                      | Static [19]                      |
| <b>Capacity of parameter memory</b>                        | 4 bits × 128 words               |
| <b>Capacity of data memory</b>                             | 32 bits × 128 words              |
| <b>Replacement Policy</b>                                  | Counter-based LRU                |
| <b>Pre-fetch data</b>                                      | yes                              |
| <b>Size of the data block for each parameter extractor</b> | 2 bits                           |
| <b>Number of levels of the parameter extractor</b>         | 3 levels                         |

Table 4-3 The configurations for each scheme in the experiment.

| <b>Schemes</b>                 | <b>Configurations</b>  |
|--------------------------------|--|
| Block-XOR                      | Without valid bit design   |
| Gate-block selection algorithm | The priority of selecting gates is NAND > NOR > XOR when the $C_{avg}$ is equal.   |
| Local grouping algorithm       | gt1 = a nor b, gt2 = a nor (not b),<br>gt3 = (not a) nor b, gt4 = a nand b,<br>gt5 = a, gt6 = b, gt7 = a xor b<br>The gt5 and gt6 are implemented by two inverters |
|                                | The priority of selecting gate types is gt6 > gt5 > gt7 when we need to choose a gate type to return.  |

The other configurations for each scheme are shown in Table 4-3. In the second row, the block-xor parameter extractor without valid bit design is used in our experiment because the pure xor function can distribute data more uniformly than the block-xor scheme with valid bit design when data are uniform distribution. We also define the gate types that are used in the experiment and the priority of selecting gates for gate-block selection algorithm and local grouping algorithm. Moreover, the configuration of DAI method is shown in Table 4-4. The block 3 is discarded. Besides, block 2 and copied block 0 are interlaced as new blocks 3 and 2 when MiBench is used because we divided the unique data of the benchmark into four blocks and analyzed them. We found that the standard deviation of most significant two blocks is high and least significant one block is low. This analytic result is shown in Table 4-5.

Table 4-4 The configuration for DAI method in the MiBench experiment.

|                   | <b>Block 3</b>                          | <b>Block 2</b>                          | <b>Block 1</b>             | <b>Block 0</b>       |
|-------------------|---|---|----------------------------|----------------------|
| <b>Original</b>   | $d_{31}d_{30}\dots d_{25}d_{24}$        | $d_{23}d_{22}\dots d_{17}d_{16}$        | $d_{15}d_{14}\dots d_9d_8$ | $d_7d_6\dots d_1d_0$ |
|                   | <b>New block 3</b>                      | <b>New block 2</b>                      | <b>Block 1</b>             | <b>Block 0</b>       |
| <b>DAI method</b> | $d_{23}d_0d_{22}d_1 d_{21}d_2d_{20}d_3$ | $d_{19}d_4d_{18}d_5 d_{17}d_6d_{16}d_7$ | $d_{15}d_{14}\dots d_9d_8$ | $d_7d_6\dots d_1d_0$ |



Table 4-5 The standard deviation of each block in the MiBench experiment.

| <b>MiBench</b>        | <b>Standard deviation</b> |                       |                       |                       |
|-----------------------|---------------------------|-----------------------|-----------------------|-----------------------|
|                       | <b><i>Block 3</i></b>     | <b><i>Block 2</i></b> | <b><i>Block 1</i></b> | <b><i>Block 0</i></b> |
| bf_small_decode       | 511.18                    | 511.18                | 85.90                 | 0.63                  |
| bf_small_encode       | 511.17                    | 511.17                | 85.89                 | 0.63                  |
| bitcnts_large         | 10.09                     | 10.09                 | 7.90                  | 0.65                  |
| bitcnts_small         | 10.06                     | 10.06                 | 7.86                  | 0.64                  |
| crc_small_encode      | 255.69                    | 255.69                | 64.22                 | 0.62                  |
| dijkstra_large        | 258.18                    | 258.18                | 62.13                 | 0.70                  |
| dijkstra_small        | 258.18                    | 258.18                | 62.13                 | 0.70                  |
| fft_large             | 64.01                     | 64.01                 | 31.87                 | 0.45                  |
| fft_large_inv         | 64.01                     | 64.01                 | 31.87                 | 0.45                  |
| fft_small             | 64.01                     | 64.01                 | 31.87                 | 0.45                  |
| fft_small_inv         | 64.01                     | 64.01                 | 31.87                 | 0.45                  |
| patricia_large        | 257.49                    | 257.49                | 62.05                 | 0.74                  |
| patricia_small        | 257.43                    | 257.43                | 62.05                 | 0.74                  |
| qsort_large           | 259.31                    | 259.22                | 63.36                 | 0.67                  |
| qsort_small           | 257.19                    | 257.10                | 62.14                 | 0.69                  |
| rijndael_small_decode | 511.14                    | 511.14                | 84.65                 | 0.68                  |
| rijndael_small_encode | 511.16                    | 511.16                | 84.68                 | 0.64                  |
| sha_small_encode      | 258.49                    | 258.49                | 63.29                 | 0.82                  |
| susan_large_corners   | 7154.87                   | 4671.03               | 111.21                | 1.00                  |
| susan_large_edges     | 7154.87                   | 4671.03               | 111.19                | 1.00                  |
| susan_large_smoothing | 7154.87                   | 4671.03               | 111.21                | 0.99                  |
| susan_small_corners   | 706.85                    | 705.91                | 97.32                 | 1.24                  |
| susan_small_edges     | 706.84                    | 705.91                | 97.30                 | 1.24                  |
| susan_small_smoothing | 706.85                    | 706.73                | 97.34                 | 1.23                  |
| toast_small_encode    | 303.11                    | 303.11                | 66.06                 | 1.04                  |
| untoast_small_decode  | 326.30                    | 326.30                | 68.84                 | 0.74                  |

## 4.2 Results

In this section, we will show the experimental results on the random data first then that on the MiBench.

### 4.2.1 Experimental Results of Random Data

Table 4-6 The number of data in the random test data.

| Random data | Number of data |                 |
|-------------|----------------|-----------------|
|             | <i>unique</i>  | <i>original</i> |
| newData1    | 982,514        | 1,000,000       |
| newData2    | 988,613        | 1,000,000       |
| newData3    | 990,763        | 1,000,000       |
| newData4    | 993,294        | 1,000,000       |
| newData5    | 989,177        | 1,000,000       |
| data1       | 49,986         | 50,000          |
| data2       | 49,962         | 50,000          |

Table 4-7 The standard deviation on the number of unique data that are mapped to each parameter in the random test data.

| Random data | Standard deviation |             |            |
|-------------|--------------------|-------------|------------|
|             | <i>group</i>       | <i>gsel</i> | <i>xor</i> |
| newData1    | 524.97             | 2085.12     | 6224.66    |
| newData2    | 257.48             | 4294.38     | 17547.60   |
| newData3    | 554.34             | 4013.53     | 5610.75    |
| newData4    | 269.23             | 1647.30     | 3935.91    |
| newData5    | 272.28             | 3199.01     | 3849.80    |
| data1       | 50.80              | 264.47      | 266.61     |
| data2       | 47.59              | 140.18      | 289.65     |

The number of unique data and original data is shown in Table 4-6. We can see that the difference of them is small because the random data does not have locality. So the most data will be stored in the CAM but the miss rate will be high. Moreover, we calculate the standard deviation on the number of unique data that are mapped to

parameters in Table 4-7. The lower the standard deviation is, the more uniform the number of data that is relative to each parameter is. So we can see that the parameter extractor generated by our algorithm in the “group” column can distribute the data more uniformly than the gate-block selection algorithm and pure xor function in the “gsel” and “xor” columns. But this method only can measure the global scope because only some parts of data are stored in the CAM. So we use the high level simulation tool that is designed by ourselves to count the status of the CAM when each operation is performed. The results of the high level simulation are shown in Table 4-8 and the improvement rate is shown in Table 4-9.

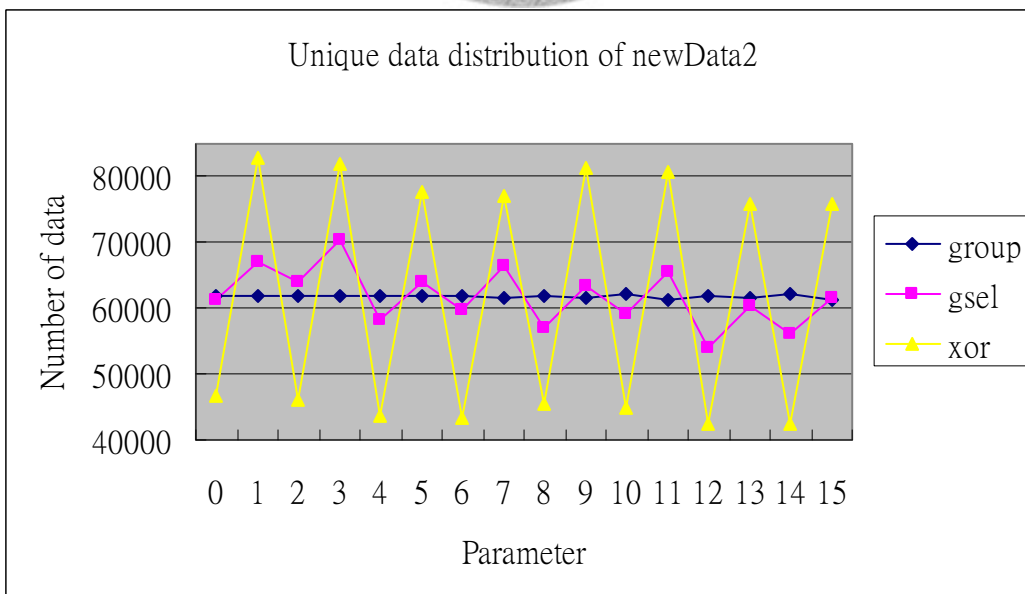
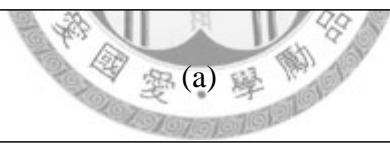
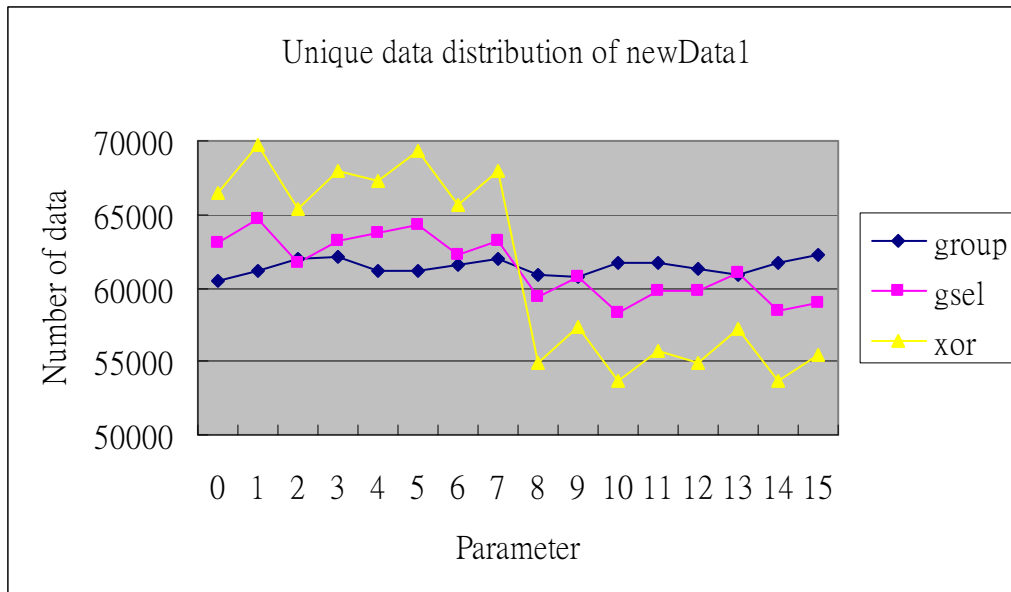
Table 4-8 The high level simulation result.

| Random data | mat_mis + (mat_mat -> mis_mis) |             |            |
|-------------|--------------------------------|-------------|------------|
|             | <i>group</i>                   | <i>gsel</i> | <i>xor</i> |
| newData1    | 8,934,661                      | 8,946,933   | 9,022,533  |
| newData2    | 8,939,674                      | 8,972,140   | 9,594,843  |
| newData3    | 8,936,071                      | 8,965,914   | 9,006,159  |
| newData4    | 8,934,051                      | 8,945,856   | 8,973,156  |
| newData5    | 8,939,205                      | 8,956,204   | 8,968,062  |
| data1       | 446,491                        | 449,224     | 449,692    |
| data2       | 447,135                        | 448,026     | 450,661    |

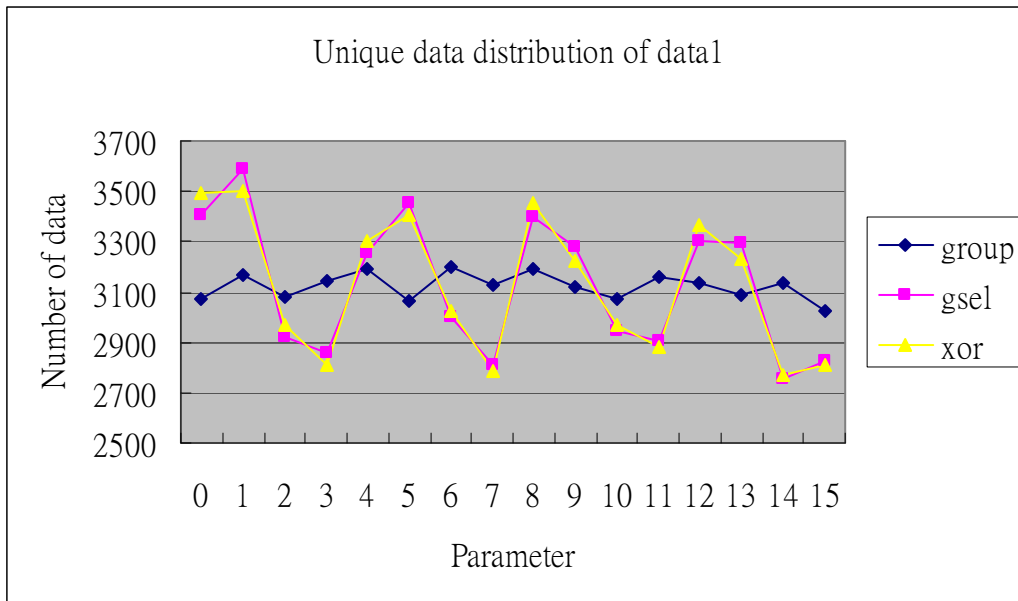
Table 4-9 The improvement rate of the high level simulation result.

| Random data    | mat_mis + (mat_mat -> mis_mis) (%) |                |
|----------------|------------------------------------|----------------|
|                | <i>group</i>                       |                |
|                | <i>Vs. gsel</i>                    | <i>Vs. xor</i> |
| newData1       | 0.1372%                            | 0.9739%        |
| newData2       | 0.3619%                            | 6.8283%        |
| newData3       | 0.3328%                            | 0.7782%        |
| newData4       | 0.1320%                            | 0.4358%        |
| newData5       | 0.1898%                            | 0.3218%        |
| data1          | 0.6084%                            | 0.7118%        |
| data2          | 0.1989%                            | 0.7824%        |
| <b>Average</b> | 0.2801%                            | 1.5475%        |

We can see that the improvement rate is small when the random data are used because the random data do not have locality. Therefore, most search operations are missed in the data memory and data are often replaced such that the counts are close in these three schemes. In next section, we will see that the improvement rate will increase when the data have locality.



(b)



(c)

Figure 4-1 The unique data distribution of three random data sets.

The number of data that are mapped to each parameter is shown in Figure 4-1. We only show three data sets in these figures. The more uniform the data distribution is, the smoother the line is. Therefore, our synthesized parameter extractors are better than the other two schemes in our experiment when the random data are used. Moreover, the most data of each block are different so we do not need to use the discard and interlaced method on the random data. We did not either use the NanoSim to run the simulation when the random data are used because the result of the high level simulation is close on these three schemes.

## 4.2.2 Experimental Results of MiBench

Table 4-10 The number of data in the MiBench.

| MiBench               | Number of data |                 |
|-----------------------|----------------|-----------------|
|                       | <i>unique</i>  | <i>original</i> |
| bf_small_decode       | 8,442          | 623,941         |
| bf_small_encode       | 8,441          | 623,938         |
| bitcnts_large         | 220            | 1,724           |
| bitcnts_small         | 219            | 1,729           |
| crc_small_encode      | 4,271          | 1,369,097       |
| dijkstra_large        | 4,308          | 36,247          |
| dijkstra_small        | 4,308          | 30,658          |
| fft_large             | 1,094          | 962,149         |
| fft_large_inv         | 1,094          | 688,480         |
| fft_small             | 1,094          | 116,383         |
| fft_small_inv         | 1,094          | 172,663         |
| patricia_large        | 4,297          | 3,265,344       |
| patricia_small        | 4,296          | 542,003         |
| qsort_large           | 4,332          | 3,145,099       |
| qsort_small           | 4,298          | 107,078         |
| rijndael_small_decode | 8,418          | 623,970         |
| rijndael_small_encode | 8,433          | 623,969         |
| sha_small_encode      | 4,315          | 312,043         |
| susan_large_corners   | 114,998        | 221,568         |
| susan_large_edges     | 114,996        | 221,566         |
| susan_large_smoothing | 115,000        | 221,570         |
| susan_small_corners   | 11,624         | 14,820          |
| susan_small_edges     | 11,622         | 14,818          |
| susan_small_smoothing | 11,626         | 14,822          |
| toast_small_encode    | 4,992          | 25,937          |
| untoast_small_decode  | 5,356          | 25,945          |

The number of unique data and original data is shown in Table 4-10. The difference of them is large because these data sets have locality and most data are duplicate in each data set. The standard deviations on the number of unique data that are mapped to each parameter are shown in Table 4-11. The difference of the standard deviation of each

scheme is small in the last three columns when the DAI method is not used in these schemes; because data are centralized in some parameters due to most data are identical in some blocks. After using the DAI method, the standard deviations of each scheme are decreased that are shown in the “dai\_group”, “dai\_gsel” and “dai\_xor” columns. Furthermore, most standard deviations in the “dia\_group” column are smaller than the others.

Table 4-11 The standard deviation on the number of unique data that are mapped to each parameter in the MiBench.

| MiBench               | Standard deviation |                 |                |              |             |            |
|-----------------------|--------------------|-----------------|----------------|--------------|-------------|------------|
|                       | <i>dai_group</i>   | <i>dai_gsel</i> | <i>dai_xor</i> | <i>group</i> | <i>gsel</i> | <i>xor</i> |
| bf_small_decode       | 527.63             | 527.97          | 527.97         | 878.16       | 878.57      | 914.27     |
| bf_small_encode       | 1.84               | 527.91          | 527.91         | 878.19       | 878.60      | 914.16     |
| bitcnts_large         | 4.38               | 9.44            | 15.67          | 25.12        | 25.12       | 25.64      |
| bitcnts_small         | 4.43               | 16.19           | 15.59          | 25.06        | 25.06       | 25.49      |
| crc_small_encode      | 1.09               | 267.20          | 267.28         | 437.46       | 437.73      | 462.70     |
| dijkstra_large        | 3.60               | 247.82          | 247.82         | 442.07       | 441.94      | 441.94     |
| dijkstra_small        | 3.60               | 247.82          | 247.82         | 442.07       | 441.94      | 441.94     |
| fft_large             | 68.66              | 59.63           | 59.63          | 108.56       | 108.56      | 108.56     |
| fft_large_inv         | 68.66              | 59.63           | 59.63          | 108.56       | 108.56      | 108.56     |
| fft_small             | 68.66              | 59.63           | 59.63          | 108.56       | 108.56      | 108.56     |
| fft_small_inv         | 68.66              | 59.63           | 59.63          | 108.56       | 108.56      | 108.56     |
| patricia_large        | 2.34               | 268.58          | 268.58         | 440.74       | 440.74      | 465.17     |
| patricia_small        | 268.52             | 268.52          | 268.52         | 440.63       | 440.63      | 465.06     |
| qsort_large           | 8.98               | 250.72          | 254.09         | 443.84       | 443.95      | 449.91     |
| qsort_small           | 268.67             | 248.41          | 251.92         | 439.97       | 439.97      | 446.16     |
| rijndael_small_decode | 526.13             | 526.31          | 526.13         | 878.96       | 879.17      | 911.28     |
| rijndael_small_encode | 1.95               | 527.29          | 527.06         | 878.46       | 878.71      | 912.90     |
| sha_small_encode      | 3.18               | 269.80          | 270.19         | 442.55       | 442.59      | 467.67     |
| susan_large_corners   | 2.85               | 3054.54         | 3054.54        | 7295.45      | 8366.22     | 8385.63    |
| susan_large_edges     | 1.92               | 3052.78         | 3052.78        | 7295.58      | 8366.33     | 8384.24    |
| susan_large_smoothing | 8.46               | 3054.42         | 3054.42        | 7295.33      | 8366.12     | 8385.65    |
| susan_small_corners   | 3.69               | 724.88          | 725.17         | 1214.27      | 1214.29     | 1256.79    |
| susan_small_edges     | 3.12               | 724.76          | 725.04         | 1214.34      | 1214.36     | 1256.57    |
| susan_small_smoothing | 726.65             | 726.64          | 726.95         | 1216.11      | 1216.11     | 1258.93    |
| toast_small_encode    | 5.34               | 305.56          | 312.33         | 521.43       | 521.48      | 540.77     |
| untoast_small_decode  | 2.38               | 326.82          | 334.76         | 561.78       | 561.78      | 579.81     |

Table 4-12 The high level simulation result in the MiBench.

| Mibench               | mat_mis + (mat_mat -> mis_mis) |            |            |             |             |             |
|-----------------------|--------------------------------|------------|------------|-------------|-------------|-------------|
|                       | dai_group                      | dai_gsel   | dai_xor    | group       | gsel        | xor         |
| bf_small_decode       | 18,015,217                     | 16,644,821 | 16,644,821 | 35,478,702  | 33,171,997  | 33,185,089  |
| bf_small_encode       | 9,280,497                      | 16,644,828 | 16,644,828 | 35,478,719  | 33,171,936  | 33,185,054  |
| bitcnts_large         | 14,314                         | 28,001     | 29,054     | 57,135      | 56,596      | 54,664      |
| bitcnts_small         | 14,313                         | 30,117     | 29,276     | 57,219      | 56,690      | 55,239      |
| crc_small_encode      | 20,364,290                     | 37,193,389 | 37,193,676 | 77,857,529  | 74,150,863  | 74,159,018  |
| dijkstra_large        | 403,649                        | 854,593    | 854,593    | 1,561,272   | 1,714,874   | 1,714,874   |
| dijkstra_small        | 373,477                        | 804,033    | 804,033    | 1,450,283   | 1,604,999   | 1,604,999   |
| fft_large             | 31,151,734                     | 27,005,873 | 27,005,873 | 46,837,508  | 54,023,813  | 54,023,813  |
| fft_large_inv         | 22,331,308                     | 19,349,879 | 19,349,879 | 33,535,963  | 38,665,089  | 38,665,089  |
| fft_small             | 3,765,030                      | 3,262,801  | 3,262,801  | 5,660,491   | 6,529,231   | 6,529,231   |
| fft_small_inv         | 5,599,548                      | 4,848,193  | 4,848,193  | 8,407,287   | 9,686,612   | 9,686,612   |
| patricia_large        | 49,726,827                     | 53,009,762 | 53,009,762 | 194,911,337 | 105,808,393 | 105,868,310 |
| patricia_small        | 16,189,035                     | 8,855,006  | 8,855,006  | 32,088,174  | 17,588,741  | 17,648,889  |
| qsort_large           | 48,761,156                     | 64,978,205 | 64,977,862 | 141,398,770 | 130,253,771 | 130,252,556 |
| qsort_small           | 3,246,987                      | 1,862,427  | 2,425,654  | 4,261,816   | 3,520,477   | 4,432,758   |
| rijndael_small_decode | 18,012,230                     | 16,645,439 | 16,638,821 | 35,467,704  | 33,174,326  | 33,173,548  |
| rijndael_small_encode | 7,955,024                      | 16,641,668 | 16,637,048 | 30,411,328  | 33,172,656  | 33,169,248  |
| sha_small_encode      | 4,971,033                      | 8,475,166  | 8,475,944  | 19,005,237  | 16,893,203  | 16,899,213  |
| susan_large_corners   | 3,295,462                      | 5,978,766  | 5,978,766  | 12,589,608  | 11,916,070  | 11,920,096  |
| susan_large_edges     | 3,295,699                      | 5,978,690  | 5,978,690  | 12,479,185  | 11,915,964  | 11,920,017  |
| susan_large_smoothing | 3,240,315                      | 5,978,828  | 5,978,828  | 12,479,356  | 11,917,063  | 11,920,207  |
| susan_small_corners   | 212,522                        | 396,645    | 396,660    | 819,060     | 783,145     | 790,586     |
| susan_small_edges     | 217,831                        | 396,608    | 396,561    | 826,315     | 783,093     | 790,468     |
| susan_small_smoothing | 410,282                        | 396,796    | 397,433    | 814,511     | 783,222     | 792,275     |
| toast_small_encode    | 345,488                        | 684,781    | 685,449    | 1,331,370   | 1,360,472   | 1,366,557   |
| untoast_small_decode  | 376,650                        | 718,738    | 714,377    | 1,437,708   | 1,429,930   | 1,424,436   |

Table 4-13 The average reduction rate of the high level simulation in MiBench.

| Average reduction rate of the high level simulation |        |          |        |         |        |        |        |
|---|--------|----------|--------|---------|--------|--------|--------|
| dai_group   |        | dai_gsel |        | dai_xor |        | group  |        |
| gsel  | xor    | gsel     | xor    | gsel    | xor    | gsel   | xor    |
| 59.39%  | 60.11% | 49.62%   | 49.99% | 49.00%  | 49.49% | -7.08% | -6.20% |

The result of the high level simulation is shown in Table 4-12 and the average reduction rate is shown in Table 4-13. When the DAI method is not used in each



scheme, most data are centralized in some parameters such that the result of each scheme is high, especially for the local grouping algorithm. This situation can be seen in the last three columns of Table 4-12 and the last column of Table 4-13. However, each scheme can be improved when the DAI method is used in each scheme. This situation can be seen in Table 4-12 and Table 4-13.

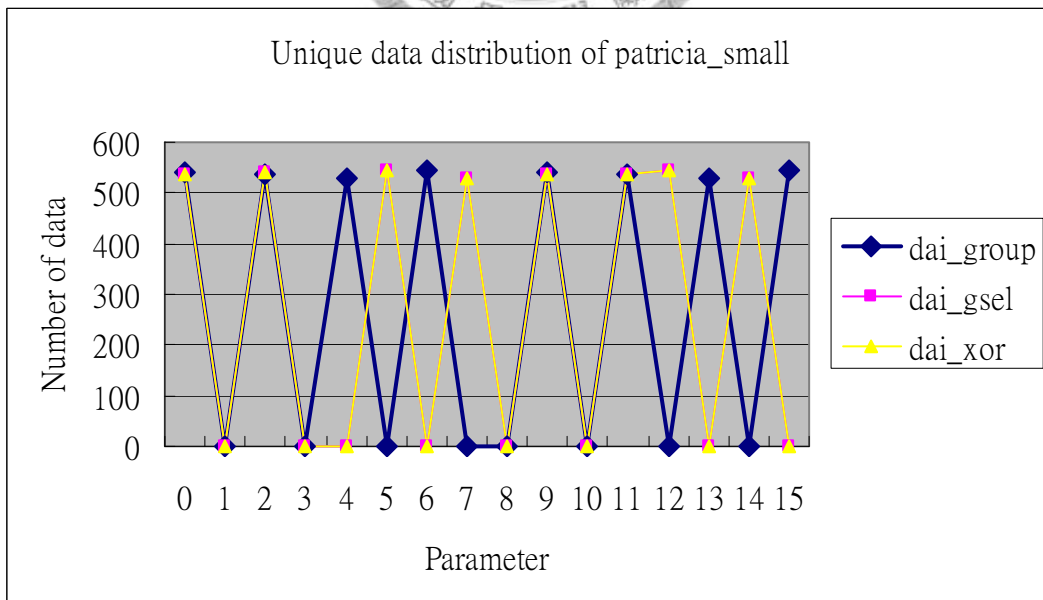
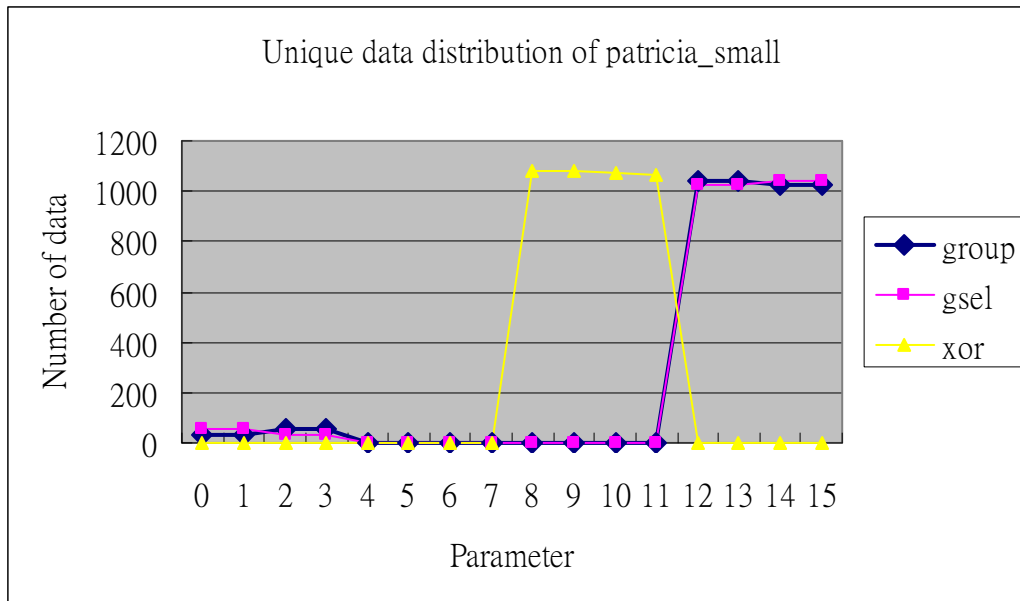


Figure 4-2 The data distribution of the patricia\_small.

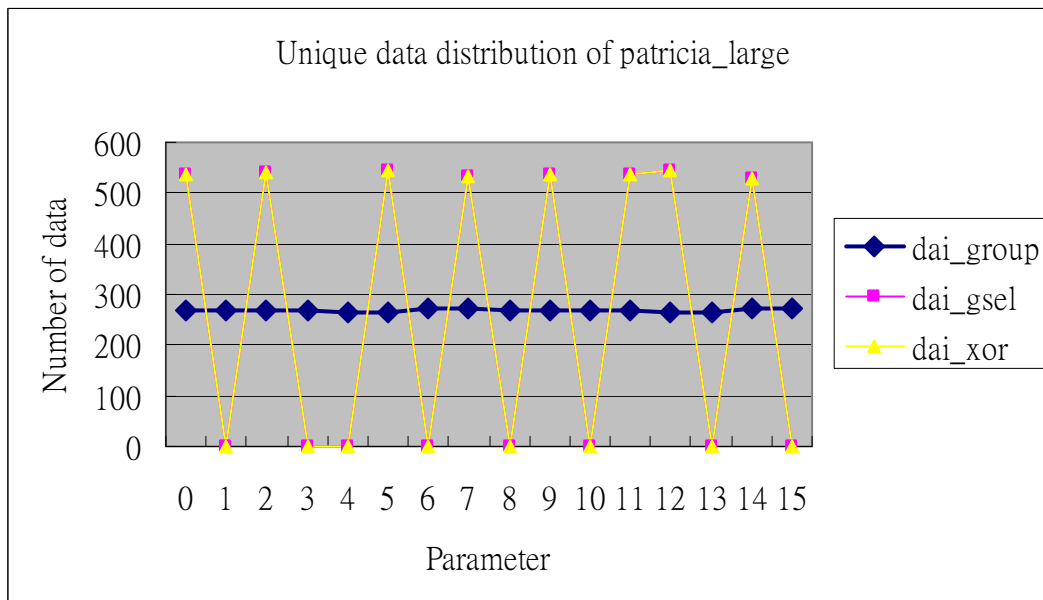
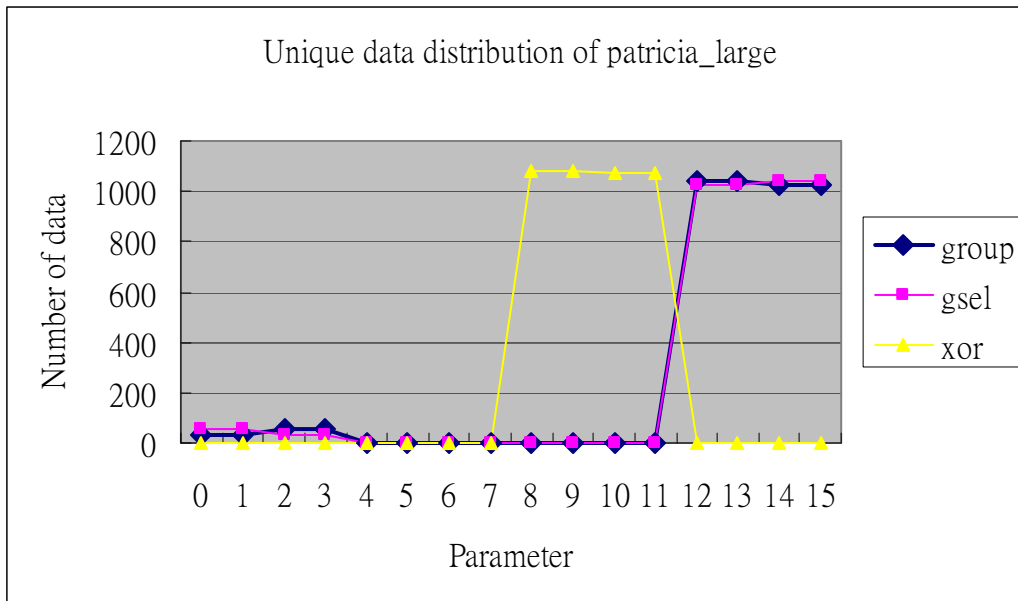


Figure 4-3 The data distribution of the patricia\_large.

The data distributions of patricia are shown in Figure 4-2 and Figure 4-3. When the DAI method is not used, the most data are distributed to some parameters. But the impact on mapping is reduced when the DAI method is used. We can also see that the data are still centralized in some parameters in the patricia\_small but the number of data that are related to one parameter is reduced, because the DAI method can only reduce the impact of some blocks. It can not break all identical data of all data blocks.

Table 4-14 The average power consumption in MiBench.

| MiBnech               | Average Power Consumption on total circuit (uW) |                 |                |              |             |            |
|-----------------------|---|-----------------|----------------|--------------|-------------|------------|
|                       | <i>dai_group</i>                                | <i>dai_gsel</i> | <i>dai_xor</i> | <i>group</i> | <i>gsel</i> | <i>xor</i> |
| bitcnts_large         | 4150.33   | 7437.91         | 7674.15        | 13436.19     | 13331.50    | 12490.76   |
| bitcnts_small         | 4139.43   | 7914.27         | 7693.84        | 13428.88     | 13304.16    | 12579.35   |
| crc_small_encode      | 6266.96   | 11487.83        | 11487.85       | 22545.01     | 21763.89    | 21765.89   |
| dijkstra_large        | 5455.15   | 11021.99        | 11029.75       | 18636.03     | 20736.97    | 20734.16   |
| fft_small             | 13528.01  | 11822.05        | 11822.67       | 19399.59     | 22511.40    | 22510.41   |
| fft_small_inv         | 13558.05  | 11841.11        | 11841.53       | 19415.87     | 22514.30    | 22513.74   |
| patricia_small        | 12643.04  | 9680.81         | 9680.81        | 23092.27     | 17543.30    | 17570.47   |
| qsort_small           | 12164.48  | 9944.28         | 10973.30       | 18383.02     | 17819.36    | 19490.27   |
| sha_small_encode      | 6646.77   | 11486.07        | 11486.70       | 24090.53     | 21755.78    | 21761.76   |
| susan_small_corners   | 6028.83   | 11344.96        | 11353.23       | 22026.17     | 21269.45    | 21443.44   |
| susan_small_edges     | 6320.37   | 11344.58        | 11351.71       | 22160.86     | 21272.35    | 21444.37   |
| susan_small_smoothing | 11280.94  | 11347.65        | 11372.37       | 21892.86     | 21269.41    | 21485.58   |
| toast_small_encode    | 5956.30   | 11202.83        | 11226.05       | 20437.40     | 21135.14    | 21207.09   |
| untoast_small_decode  | 6169.06   | 11693.39        | 11637.49       | 21986.08     | 22111.57    | 22024.57   |

Table 4-15 The average reduction rate of the power consumption in MiBench.

| Average reduction rate of the power consumption |            |                 |            |                |            |              |            |
|---|------------|-----------------|------------|----------------|------------|--------------|------------|
| <i>dai_group</i>                                |            | <i>dai_gsel</i> |            | <i>dai_xor</i> |            | <i>group</i> |            |
| <i>gsel</i>                                     | <i>xor</i> | <i>gsel</i>     | <i>xor</i> | <i>gsel</i>    | <i>xor</i> | <i>gsel</i>  | <i>xor</i> |
| 58.88%  | 59.10%     | 46.00%          | 45.93%     | 45.57%         | 45.54%     | -1.42%       | -1.49%     |

Due to the simulation time of the SPICE code, we only simulated some benchmarks. The results are shown in Table 4-14 and the reduction rate is shown in Table 4-15. Although some benchmarks are not simulated, the reduction rate of our scheme is still higher than the others. The power consumption of the gate-block selection scheme and block-xor scheme can also be reduced when the DAI method is used on them. Furthermore, the power consumption of the parameter extractor and the number of CMOS elements are also obtained from the NanoSim. The average reduction rates of them are shown in Table 4-16 and Table 4-17. The result shows that our parameter extractors not only reduce the power consumption but also save some hardware cost.

Table 4-16 The average reduction rate of the power on the parameter extractor.

| <b>Average reduction rate of the power on the parameter extractor</b> |                |              |            |
|---|----------------|--------------|------------|
| <i>dai_group</i>  |                | <i>group</i> |            |
| <i>dai_gsel</i>   | <i>dai_xor</i> | <i>gsel</i>  | <i>xor</i> |
| 63.60%  | 63.62%         | 21.85%       | 22.17%     |

Table 4-17 The average reduction rate of the number of CMOS elements.

| <b>Average reduction rate of the number of CMOS elements on total circuit</b> |                |              |            |
|---|----------------|--------------|------------|
| <i>dai_group</i>  |                | <i>group</i> |            |
| <i>dai_gsel</i>   | <i>dai_xor</i> | <i>gsel</i>  | <i>xor</i> |
| 0.53%   | 0.59%          | 0.20%        | 0.48%      |



## Chapter 5 Conclusion

In this work, we propose a local grouping algorithm to synthesize a proper parameter extractor such that the power consumption of the PBCAM can be reduced. Moreover, the cost of the parameter extractor is also lower than the others. We also propose the DAI method to reduce the influence of the identical data in some data blocks. This method can improve the efficiency of the parameter extractor. The experiment results also show that our schemes can reduce the power consumption and the number of CMOS elements. Moreover, the DAI method can also improve the gate-block selection algorithm and block-xor scheme. Therefore, our schemes are suitable for embedded systems when the applications of a system are known in advance.



## References

- [1] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498-523, Apr. 1995.
- [2] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*, 1 ed. Norwell, MA and AH Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995.
- [3] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, *et al.*, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68-75, Dec. 2003.
- [4] L. T. Clark, C. Byungwoo, and M. Wilkerson, "Reducing translation lookaside buffer active power," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 10-13.
- [5] V. Chaudhary, T. H. Chen, F. Sheerin, and L. T. Clark, "Critical race-free low-power nand match line content addressable memory tagged cache memory," *IET Computers & Digital Techniques*, vol. 2, no. 1, pp. 40-44, Jan. 2008.
- [6] C.-C. Wu, S.-H. Wen, N.-F. Huang, and C.-N. Kao, "A pattern matching coprocessor for deep and large signature set in network security system," in *IEEE Global Telecommunications Conference (GLOBECOM)*, 2005, p. 5.
- [7] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712-727, Mar. 2006.
- [8] K. J. Schultz, "Content-addressable memory core cells: a survey," *Integration, the VLSI Journal* vol. 23, no. 2, pp. 171-188, Nov. 1997.
- [9] G. Kasai, Y. Takarabe, K. Furumi, and M. Yoneda, "200MHz/200MSPS 3.2W at 1.5V V<sub>dd</sub>, 9.4Mbits ternary CAM with new charge injection match detect circuits and bank selection scheme," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2003, pp. 387-390.

- [10] I. Arsovski, T. Chandler, and A. Sheikholeslami, "A ternary content-addressable memory (TCAM) based on 4T static storage and including a current-race sensing scheme," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 1, pp. 155-158, Jan. 2003.
- [11] I. Arsovski and A. Sheikholeslami, "A current-saving match-line sensing scheme for content-addressable memories," in *IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, 2003, pp. 304-494 vol.1.
- [12] I. Arsovski and A. Sheikholeslami, "A mismatch-dependent power allocation technique for match-line sensing in content-addressable memories," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1958-1966, Nov. 2003.
- [13] K. Pagiamtzis and A. Sheikholeslami, "A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 9, pp. 1512-1519, Sep. 2004.
- [14] J.-H. Lee, G.-h. Park, S.-B. Park, and S.-D. Kim, "A selective filter-bank TLB system [embedded processor MMU for low power]," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 312-317.
- [15] P. Echeverria, J. L. Ayala, and M. Lopez-Vallejo, "A banked precomputation-based CAM architecture for low-power storage-demanding applications," in *IEEE Mediterranean Electrotechnical Conference (MELECON)*, 2006, pp. 57-60.
- [16] S. Hanzawa, T. Sakata, K. Kajigaya, R. Takemura, and T. Kawahara, "A large-scale and low-power CAM architecture featuring a one-hot-spot block code for IP-address lookup in a network router," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 4, pp. 853-861, Apr. 2005.
- [17] K. Cheong, Q. Shaolei, and A. Mason, "A power-optimized 64-bit priority encoder utilizing parallel priority look-ahead," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, 2004, pp. II-753-6

Vol.2.

- [18] C. A. Zukowski and S.-Y. Wang, "Use of selective precharge for low-power content-addressable memories," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 1997, pp. 1788-1791 vol.3.
- [19] C.-S. Lin, J.-C. Chang, and B.-D. Liu, "A low-power precomputation-based fully parallel content-addressable memory," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 4, pp. 654-662, Apr. 2003.
- [20] S.-J. Ruan, C.-Y. Wu, and J.-Y. Hsieh, "Low Power Design of Precomputation-Based Content-Addressable Memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 3, pp. 331-335, Mar. 2008.
- [21] C.-Y. Wu, S.-f. Ruan, C.-K. Cheng, and M.-B. Lin, "A new Block-XOR precomputation-based CAM design for low-power embedded system," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2005, pp. 1-4.
- [22] J.-Y. Hsieh and S.-J. Ruan, "Synthesis and design of parameter extractors for low-power pre-computation-based content-addressable memory using gate-block selection algorithm," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2008, pp. 316-321.
- [23] *Standard deviation*. Available: [http://en.wikipedia.org/wiki/Standard\\_deviation](http://en.wikipedia.org/wiki/Standard_deviation)
- [24] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization (WWC-4)*, 2001, pp. 3-14.
- [25] *SimpleScalar LLC*. Available: <http://www.simplescalar.com/>