Architecture for Memristor-based Storage Structures

by

Yang Liu

Department of Computer Science
Duke University

Date:_______________________

Approved:

___________________________
Alvin R. Lebeck, Supervisor

___________________________
Chris Dwyer

___________________________
Bruce M. Maggs

___________________________
Daniel J. Sorin

___________________________
Xiaowei Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2011
ABSTRACT
ARCHITECTURES FOR MEMRISTOR-BASED STORAGE STRUCTURES

by

Yang Liu

Department of Computer Science
Duke University

Date:_______________________
Approved:

___________________________
Alvin R. Lebeck, Supervisor

___________________________
Chris Dwyer

___________________________
Bruce M. Maggs

___________________________
Daniel J. Sorin

___________________________
Xiaowei Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2011
Copyright by
Yang Liu
2011
Abstract

Rapid data growth nowadays makes it more critical to reduce search time to improve the performance of search-intensive applications. However, huge data size makes it more difficult to efficiently perform search operations. Representative conventional approaches to reduce search time, such as CAM and in-memory databases, are no longer efficient because of the data explosion: CMOS-based CAM has low capacity which cannot be increased through CMOS scaling, and in-memory databases have performance degradation as data size increases. As a result, we have to exploit emerging nanotechnologies to accelerate search.

Among emerging nanotechnologies, memristors have become promising candidates to build storage structures because of high capacity, short switching time and low power consumption. However, the benefit we can obtain from these storage structures is limited by low endurance of memristors. In order to utilize the computation ability of memristors and deal with the endurance problem, we explore the design space of memristor-based storage structures.

We first propose MemCAM/MemTCAM, a configurable memristor-based CAM/TCAM design, in which we use memristors as both memory latches and logic gates. Computation ability of memristors makes it possible to perform range search and high density of memristors provides an opportunity to build MemCAM/MemTCAM
with large capacity and small area. We use SPICE to model the memristor and analyze power and performance at different temperatures. The results show that it is feasible to build MemCAM and MemTCAM which have high capacity and can reduce total search time and energy consumption for search-intensive applications with huge data size.

We then propose four hybrid memristor-based storage structures, Hash-CAM, T-tree-CAM, TB⁺-tree, and TB⁺-tree-CAM, to solve the endurance problem. We use an analytical model to evaluate and compare the performance and lifetime of two software-implemented memory-based T-trees and these four hybrid storage structures. The results show that hybrid storage structures can utilize range search abilities, achieve better performance than memory-based T-trees, and improve lifetime from minutes to longer than 60 years. Furthermore, TB⁺-tree-CAM, a hybrid memristor-based storage structure combining T-tree, B⁺-tree and CAM, manages to balance between performance and lifetime and can outperform other storage structures when taking both performance and lifetime into consideration.
Contents

Abstract.............................................................................................................................................. iv

List of Tables ......................................................................................................................................... x

List of Figures ....................................................................................................................................... xii

Acknowledgements ................................................................................................................................. xv

1. Introduction ......................................................................................................................................... 1
   1.1 MemCAM/MemTCAM: Memristor-based Content Addressable Memory and Memristor-based Ternary Content Addressable Memory ................................................................. 3
   1.2 Hybrid CMOS-memristor-based Storage Structures ........................................................................ 5
   1.3 Thesis Outline ................................................................................................................................. 7

2. Background ......................................................................................................................................... 8
   2.1 Content Addressable Memory ......................................................................................................... 8
   2.2 Memristor, Implication Logic and Memristor Array ....................................................................... 9
      2.2.1 Memristor ................................................................................................................................ 9
      2.2.2 Implication Logic ....................................................................................................................... 11
   2.3 Memristor Array ............................................................................................................................. 13

3. MemCAM and MemTCAM: Memristor-based Content Addressable Memory and Ternary Content Addressable Memory ................................................................................................. 15
   3.1 MemCAM/MemTCAM Overview .................................................................................................. 15
   3.2 MemCAM/MemTCAM Cell Design ............................................................................................... 19
      3.2.1 MemCAM ............................................................................................................................... 19
         3.2.1.1 Comparison Process Supporting only Point Query ................................................................. 20
7.4 Processing-In-Memory and Distributed Memory Systems.................................. 105

8. Conclusions and Future Work ................................................................................. 107
  8.1 Conclusions ........................................................................................................ 107
  8.2 Future Work ......................................................................................................... 109

References ...................................................................................................................... 110

Biography ....................................................................................................................... 115
List of Tables

Table 1: States of Different Memristors and Voltages Applied Onto Different Memristors during an NAND Calculation ................................................................. 13

Table 2: Applied Voltages at Each Step during Comparison for MemCAM Supporting Point Query ........................................................................................................ 21

Table 3: States of Memristors at Each Step during Comparison for MemCAM Supporting Point Query ........................................................................................................ 22

Table 4: Values and Meanings of Cell Match Signals Based on Values of D and K........ 23

Table 5: Applied Voltages at Each Step during Comparison for MemCAM Supporting both Point and Range Queries ........................................................................ 23

Table 6: States of Memristors at Each Step during Comparison for MemCAM Supporting both Point and Range Queries ........................................................................ 24

Table 7: Relationship between $D_1D_0$, K and Match Signal ........................................ 25

Table 8: Applied Voltages at Each Step during Comparison for MemTCAM Supporting Point Query ........................................................................................................ 26

Table 9: States of Memristors at Each Step during Comparison for MemTCAM Supporting Point Query ........................................................................................................ 27

Table 10: Values and Meanings of Cell Match Signals Based on Values of $D_1$ and $D_0$ and SL ......................................................................................................................... 27

Table 11: Applied Voltages at Each Step during Comparison for MemTCAM Supporting both Point and Range Queries ........................................................................ 29

Table 12: States of Memristors at Each Step during Comparison for MemTCAM Supporting both Point and Range Queries ........................................................................ 29

Table 13: Applied Voltages at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting only Point Query ......................... 33

Table 14: States of Memristors at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting only Point Query ......................... 33
Table 15: Applied Voltages at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting both Point and Range Queries .... 35

Table 16: States of Memristors at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting both Point and Range Queries .... 35

Table 17: Power Consumption of a Memristor with Different Initial States and Applied Voltages ........................................................................................................................................................................ 38

Table 18: Parameter Values at 300K and 513K ........................................................................................................................................................................................................................................................................ 38

Table 19: Power Consumption (W) Based on Applied Voltage and Initial Memristor State ........................................................................................................................................................................................................................................................................ 41

Table 20: Search Time (ns) of MemCAM/MemTCAM with Different Query Support for 2^n-bit Key Word ........................................................................................................................................................................................................................................................................ 44

Table 21: Energy Consumption (fJ/bit/search) of MemCAM/MemTCAM with Different Query Support for 2^n-bit Key Word ........................................................................................................................................................................................................................................................................ 44

Table 22: Power Density (W/cm^2) of MemCAM/MemTCAM with Different Query Support for 64-bit and 1024-bit Key Word ........................................................................................................................................................................................................................................................................ 44

Table 23: Pros and Cons of Four Hybrid Storage Structures ........................................................................................................................................................................................................................................................................ 79

Table 24: Parameters in the Analytical Model ........................................................................................................................................................................................................................................................................ 84

Table 25: Number of Records Stored in Different Storage Structures ........................................................................................................................................................................................................................................................................ 90

Table 26: Theoretical Maximum Lifetime of Four Hybrid Data Structures (T_{access: memristor array read/write latency}) ........................................................................................................................................................................................................................................................................ 101
List of Figures

Figure 1: Design Space of Memristor-based Storage Structures ........................................ 5
Figure 2: Conceptual View of a Content Addressable Memory ........................................... 8
Figure 3: Four Fundamental Circuit Elements ..................................................................... 10
Figure 4: Memristor Diagram and Notation ....................................................................... 11
Figure 5: Memristor Implication Logic ............................................................................... 12
Figure 6: A 3-memristor Example for Implication Logic ..................................................... 13
Figure 7: Memory Organizations with MemCAM/MemTCAM ............................................ 16
Figure 8: MemCAM/MemTCAM Structure Overview ......................................................... 17
Figure 9: 6-memristor MemCAM Cell Design .................................................................... 20
Figure 10: MemCAM Comparison Process Supporting only Point Query ......................... 21
Figure 11: MemCAM Comparison Process Supporting both Point Query and Range Query ......................................................................................................................... 23
Figure 12: 7-memristor MemTCAM Cell Design .................................................................. 25
Figure 13: MemTCAM Comparison Process Supporting only Point Query ....................... 26
Figure 14: MemTCAM Comparison Process Supporting both Point Query and Range Query ......................................................................................................................... 28
Figure 15: 3 Bit x 3 Entry MemCAM/MemTCAM Cell Layout ............................................ 31
Figure 16: Match Signal Combination Example for MemCAM/MemTCAM Supporting only Point Query ......................................................................................................................... 32
Figure 17: Match Signal Combination Example for MemCAM/MemTCAM Supporting both Point and Range Queries ......................................................................................................................... 34
Figure 18: Memristor Spice Model ...................................................................................... 40
Figure 19: Memristor Spice Code......................................................................................... 40
Figure 20: Power at Room Temperature (300K) with 3V Applied Voltage............... 42
Figure 21: Power at 513K with Different Applied Voltages ........................................ 43
Figure 22: Hybrid Memristor-based CAM/TCAM-memory ............................................. 53
Figure 23: Hybrid CMOS-memristor based CAM/TCAM .............................................. 54
Figure 24: Hash-CAM.................................................................................................. 58
Figure 25: Hash-CAM Example .................................................................................... 60
Figure 26: Hash-CAM Example with a Non-order-preserving Hash Function ............ 61
Figure 27: A T-tree Node (T-node) ............................................................................... 62
Figure 28: T-tree-CAM............................................................................................... 63
Figure 29: T-tree-CAM Example ................................................................................ 64
Figure 30: TB-tree ....................................................................................................... 65
Figure 31: TB-tree Example ....................................................................................... 67
Figure 32: TB-tree-CAM ........................................................................................... 68
Figure 33: TB-tree-CAM Example ............................................................................ 69
Figure 34: Structure of Modified T-node ..................................................................... 74
Figure 35: A T-tree-CAM Partition Overflow Solution Example (LRC: Left Record Counter, RRC: Right Record Count) .............................................................................. 75
Figure 36: Structure of Modified B-tree Root ............................................................. 77
Figure 37: A TB-tree Partition Overflow Solution Example ....................................... 77
Figure 38: Search Time of Six Storage Structures with N = 10^9 ................................. 91
Figure 39: Search Time of Five Storage Structures with N = 10^10 .............................. 92
Figure 40: Search Time of Five Storage Structures with Different Number of Records ... 93

Figure 41: Search Time of Five Storage Structures with $T_{\text{access}} = 60\text{ns}$ and Different Cache Sizes ........................................................................................................................................ 95

Figure 42: Search Time of Five Storage Structures with $T_{\text{access}} = 120\text{ns}$ and Different Cache Sizes ........................................................................................................................................ 96

Figure 43: Search Time of Five Storage Structures with $T_{\text{access}} = 60\text{ns}$ and Different B'-tree Orders ........................................................................................................................................ 98

Figure 44: Search Time of Five Storage Structures with $T_{\text{access}} = 120\text{ns}$ and Different B'-tree Orders ........................................................................................................................................ 98

Figure 45: Search Time of Five Storage Structures with $T_{\text{access}} = 60\text{ns}$ and Different T-tree Orders ........................................................................................................................................ 100

Figure 46: Search Time of Five Storage Structures with $T_{\text{access}} = 120\text{ns}$ and Different T-tree Orders ........................................................................................................................................ 100
Acknowledgements

I would like to show my sincere gratitude to those who make this thesis possible, including my parents and sister, my advisor, my committee members, my colleagues, my friends and the staff of our department.

First of all I would like to thank my parents and my sister for their love and help. Without their support and understanding I would not have made my decision to pursue a PhD degree in the first place.

This thesis would not be possible without the guidance of my advisor, Professor Alvin R. Lebeck. Alvy has been providing continuous support through my PhD study. He has been a wise and excellent mentor who teaches me to explore wide research areas independently and provides advices and suggestions at the right time.

My thanks also goes to Professor Chris Dwyer. It is fortunate for me to join the Duke Self-Assembled Systems Group and get the chance to work with Chris and benefit from his broad knowledge, insights and ideas. I would also like to thank Professor Bruce Maggs, Professor Dan Sorin, and Professor Xiaowei Yang for taking their time to serve on my committee and provide helpful feedback.

During my PhD study, I have had inspiring and stimulating discussions with several colleagues and friends. They provided support and guidance my research and my life. I would like to thank everyone in the Duke Self-Assembled Systems Group,
especially Vince Mao, Muhammad Mottaghi, Vishwa Nellore, Jun Pang, Costantin Pistol, Viresh Thusu, and Siyang Wang. I would also like to thank Xin Guo, Jie Xiao, Bing Xie and Yi Zhang, who are supportive and valuable friends.

Last but not least, I would like to thank Diane Riggs and Marilyn Butler. Life would not been so easy and smooth in the department without you.
1. Introduction

Nowadays a huge amount of data is created everyday and the data size is growing very fast [1]. For example, Google has approximately 280 exabytes (1 exabyte = $10^6$ terabytes) of data indexed in 2009 with data size doubling every 16 months [2]. Data warehouses also have a significant data growth rate of 3 times every two years and the largest data warehouse has over 300 terabytes of data in 2008 [3]. As data size increases, search becomes more and more important in several applications such as network routing, DNA sequencing, and database querying, and search time becomes more and more critical to improve the performance of these search intensive applications.

There are several conventional approaches to accelerate search. Content addressable memory (CAM) and in-memory databases are two representative mechanisms adopted to reduce search time. However, both of them have disadvantages. CMOS-based CAM is not efficient in terms of cost or power due to additional circuitry to perform comparisons, detect matches and combine results [4]. An even more severe problem of CMOS-based CAM is its low capacity. Currently the capacity of CMOS-based CAM is only tens of megabits. CMOS scaling does not solve the low capacity problem. One reason is that based on the Moore’s Law, the growth rate of transistor density is twice every 18 months, which is lower than the data growth rate. Another reason is that, manufacturing is more and more difficult and expensive as CMOS feature size decreases. The cost of a new wafer foundry doubles from 2007 to 2010 and it can
cost more when we also consider the investment for new fabrication technologies since the current optical-based fabrication technology cannot support the resolution when CMOS features size becomes too small [5]. In a word, it is difficult to increase the capacity of CMOS-based CAM, which limits performance improvement of search intensive applications with large data size. In-memory databases have become more practical as memory capacity increases. However, capacity remains a problem since the memory capacity growth rate is lower than the data growth rate. Besides, the performance of in-memory databases is degraded as data size increases. In order to both achieve large capacity and improve performance, we seek to exploit emerging nanotechnologies.

Among several emerging nanotechnologies, memristors have become promising candidates to build storage structures with high capacity and good performance. The density of a memristor array can be 100 times higher compared with DRAM, which makes it possible to build storage structures with higher capacity with the same area. Memristors can also be used to perform computation, which makes it possible to perform parallel computing within the storage structure. However, memristors have much lower endurance than CMOS transistors and in-storage parallel computing further exacerbates the problem. In this thesis, we explore the design space of memristor-based storage structures to break the capacity limitation and improve performance and lifetime.
With the designs of different memristor-based storage structures we propose in this thesis, we provide configurability by using memristors as both memory latches and logic gates and by using both CMOS and memristor technologies. Designers can choose to configure a memristor array as CAM, memory or hybrid CAM-memory to trade among power, capacity and performance. Designers can also change the proportion of data stored in CMOS-based storage structures and memristor-based storage structures in order to trade between performance and lifetime.

We make three main contributions in this thesis. First, we combine compute and storage to build memristor-based storage structures. Second, we analyze the effect of temperature on power consumption and switching time of memristors, which is the first such analysis to our best knowledge. The analysis demonstrates the possibility to build memristor-based storage structures with even lower power consumption and shorter search time at temperatures higher than room temperature. Finally, we improve lifetime of memristor-based storage structures through hybrid CMOS-memristor-based storage structures.

1.1 MemCAM/MemTCAM: Memristor-based Content Addressable Memory and Memristor-based Ternary Content Addressable Memory

We first propose MemCAM and MemTCAM, which are configurable memristor-based CAM and TCAM designs. We use memristors, the fourth fundamental circuit element first predicted by Chua [6] and recently presented by HP Labs [7], as both
memory latches and logic gates. With the computation ability of memristors we can perform comparisons to decide whether an entry in MemCAM/MemTCAM is equal to, or greater than, or less than an input key, thus we can support both point query and range query, while CMOS-based CAM/TCAM can only support point query. Cell density of MemCAM/MemTCAM can be three orders of magnitude higher than CMOS-based CAM/TCAM and provides an opportunity to build CAM and TCAM with larger capacity and the same area compared with CMOS-based CAM/TCAM.

We use SPICE to model the memristor and analyze energy consumption and performance of MemCAM/MemTCAM. The results show that it is feasible to build a 1Gbit MemCAM/MemTCAM with 1cm x 1cm area. For 64-bit key word, the search time is approximate 75ns and the energy consumption is approximate 2fJ/bit/search for MemCAM/MemTCAM supporting only point query, and the search time is approximate 140ns and the energy consumption is approximate 5fJ/bit/search for MemCAM/MemTCAM supporting both point and range queries.

MemCAM/MemTCAM is a promising candidate to replace CMOS-based CAM/TCAM because of higher density, larger capacity and computation ability, which makes it possible to support range query. However, one major disadvantage of MemCAM/MemTCAM is that memristors have much lower endurance (at most $10^{10}$ write cycles so far [8]) than SRAM ($10^{16}$ write cycles [9]). In order to solve the endurance
problem, we propose a series of hybrid storage structures and briefly describe them in the next subsection.

1.2 Hybrid CMOS-memristor-based Storage Structures

The lifetime of MemCAM/MemTCAM is only a couple of minutes because of low endurance of memristors and it cannot be improved by wear-leveling since all MemCAM/MemTCAM cells are accessed simultaneously every cycle. In order to solve this problem we propose configurable hybrid storage structures using both CMOS and memristor technologies. These storage structures can be reconfigured to trade between performance and lifetime and to adapt to future memristors with improved endurance.

![Figure 1: Design Space of Memristor-based Storage Structures](image)

Figure 1: Design Space of Memristor-based Storage Structures

Figure 1 shows the design space of memristor-based storage structures. The lifetime of a memristor-based memory is the longest due to low write frequency and can be further improved by wear-leveling techniques because of non-uniform writes. However, the search time of a memristor-based memory is also the longest and increases as data size increases. MemCAM/MemTCAM has the shortest search time because all
data items can be searched simultaneously but also has the shortest lifetime due to high
write frequency. Wear-leveling techniques can not improve the lifetime of
MemCAM/MemTCAM because writes are already uniform. As long as endurance is still
a problem for memristors, hybrid storage structures are better choices because of
moderate write frequency and non-uniform writes, which makes it possible to apply
wear-leveling techniques to improve lifetime.

The main idea of our hybrid storage structures is to divide a logic tree structure
into two parts: the upper levels and the lower levels. We store the upper levels in
CMOS-based storage and the lower levels in memristor-based storage. The memristor-
based storage is divided into multiple partitions. When we perform a search operation,
we use the upper levels to direct search into one or two partitions. As a result, the
average write frequency is reduced and the lifetime is improved.

We propose four hybrid storage structures: Hash-CAM, T-tree-CAM, TB'-tree
and TB'-tree-CAM and use an analytical model to evaluate the performance and
compare the results with conventional T-tree stored in DRAM and memristor-based
memory. The results show that Hash-CAM has the best performance. However, Hash-
CAM requires a uniform and order-preserving hash function, which may not be
generated efficiently. T-tree-CAM eliminates the requirement of a uniform and order-
preserving hash function but the lifetime is still short (approximately a year in the best
case). TB'-tree significantly improves lifetime but the improvement is at the sacrifice of
performance. TB*-tree-CAM, which is a combination of T-tree-CAM and TB*-tree, has advantages of both storage structures and thus has good performance (better than memory-based T-trees under most of the circumstances) and acceptable lifetime (longer than 60 years).

### 1.3 Thesis Outline

We organize the remainder of this thesis as following: Chapter 2 introduces background knowledge. Chapter 3 describes in detail both cell design and match signal combination of MemCAM and MemTCAM and the analysis of energy consumption and searching time. Chapter 4 introduces conventional lifetime improvement methods, wear-leveling and write avoidance, and analyze their effects on memristor-based storage structures. Chapter 5 proposes configurable hybrid CMOS-memristor-based storage structures and Chapter 6 evaluates the designs. Chapter 7 presents related work and Chapter 8 concludes and presents potential future research directions.
2. Background

2.1 Content Addressable Memory

Content addressable memory (CAM) is a type of associative memory that is used in high speed searching applications. Figure 2 shows the conceptual view of a content addressable memory containing $m$ words. Each of the $m$ words is compared with the input key word simultaneously and the matchlines are combined by an encoder to generate an output address. The output address is then used to access other storage structures, such as memory and disk.

![Conceptual View of a Content Addressable Memory](image)

CAM can be implemented using software or hardware. Software-based CAM can be implemented using binary trees or hash tables. However, memory access time is a bottleneck so software-based CAM is generally used for applications with large data size to trade search time for CAM capacity.

Today’s hardware-based CAM is mainly implemented based on SRAM. The main advantage of hardware-based CAM is constant one cycle search time, which is typically shorter than 10ns. However, besides storage space, additional circuitry is
required to perform comparisons, detect matches and combine results, so SRAM-based CAM is not efficient in terms of cost, capacity and power. Today’s largest commercialized CAM chip has only tens of megabits capacity [4], which limits the size of data that can be searched simultaneously.

CAM using CMOS technology also does not scale well because as CMOS technology scales beyond 32 nm, problems arise such as temperature going towards air cooling limit due to higher power consumption, higher integration cost due to new technology development, and lower yield and reliability due to more sensitive transistors [10, 11]. As a result, we seek to exploit emerging nanotechnologies in order to achieve higher CAM capacity while maintaining search time and energy consumption comparable to CMOS-based CAM.

2.2 Memristor, Implication Logic and Memristor Array

2.2.1 Memristor

The concept of memristor was first predicted by Chua in 1971 [6] as the fourth fundamental circuit element. There are four fundamental circuit variables: electric current $i$, voltage $v$, charge $q$ and magnetic flux $\phi$, and there is a relation between each pair of the variables, which results in six relations. The charge is the time integral of the current by definition, and the flux is the time integral of the voltage by Faraday’s law of induction (the induced electromotive force, the voltage generated by a battery or by the magnetic force, in any closed circuit is equal to the time rate of change of the magnetic
flux through the circuit). Three existing fundamental circuit elements define another three relations: the resistor \((dv=Rdi\) where \(R\) is the resistance), the capacitor \((dq=Cdv\) where \(C\) is the capacitance), and the inductor \((dq=Ldi\) where \(L\) is the inductance). Only one relation remains undefined and Chua predicted that there should be a forth fundamental circuit element defining this relation, which is the memristor \((dq=Mdq\) where \(M\) is the memristance) as shown in Figure 3.

![Figure 3: Four Fundamental Circuit Elements](image)

Recently a physical memristor model is presented by HP Labs [7]. A memristor is a non-volatile two-terminal nanoscale device whose state can be switched between ‘on’ (switch-closed) and ‘off’ (switch-open). Figure 4 shows the diagram of a memristor. In the physical memristor model from HP Labs, \(D\) is the thickness of a thin semiconductor film sandwiched between two metal contacts, and \(w\) is the thickness of the doped region of the film, which is also defined as the state variable of the memristor. The doped and undoped regions are regions with high dopant concentration and low dopant concentration respectively.
When a memristor is closed ($w \rightarrow D$), it has low resistance (defined as $R_{ON}$) and we consider it to represent logical value ‘1’; when a memristor is open ($w \rightarrow 0$), it has high resistance (defined as $R_{OFF}$) and we consider it to represent logical value ‘0’. The relationship between the memristance $M$ and the state variable $w$ of a memristor can be defined by the following equation:

**Equation 1:** \[ M(q) = R_{ON} \frac{w(t)}{D} + R_{OFF} \left(1 - \frac{w(t)}{D}\right). \]

We can apply different voltages onto a memristor to read or switch its state. We use $V_{COND}$ to represent a reading voltage which does not change the state of a memristor, $V_{SET}$ to represent a writing voltage which forces a memristor into its closed state, and $V_{CLEAR}$ to represent a writing voltage which forces a memristor into its open state. $V_{COND}$ and $V_{SET}$ are both negative voltages and the magnitude of $V_{SET}$ is larger than $V_{COND}$. $V_{CLEAR}$ is a positive voltage.

![Figure 4: Memristor Diagram and Notation](image)

### 2.2.2 Implication Logic

The natural logical operation to compute with memristors is material implication $p \rightarrow q$ [12], which changes $q$ to $\neg p \lor q$. Figure 5 shows two memristors $p$ and $q$ and a resistor $R_G$ used to perform implication logic. $R_G$ is a resistor with resistance chosen between the
‘on’ state resistance and the ‘off’ state resistance of the memristor. The voltage applied on memristor $p$ is $V_{\text{COND}}$ and the voltage applied on memristor $q$ is $V_{\text{SET}}$. If $q$ is the only memristor without $p$ and $R_G$, applying $V_{\text{SET}}$ onto it forces it into open state. However, with $p$ and $R_G$ and $V_{\text{COND}}$ applied onto $p$, $V_{\text{SET}}$ may change the state of $q$ depending on the initial states of both $p$ and $q$.

When we connect $p$ and $q$ with $R_G$ and apply $V_{\text{COND}}$ on $p$ and $V_{\text{SET}}$ on $q$ respectively, the final state of $q$ is decided by the initial states of both $p$ and $q$ as shown in Figure 5. If the initial state of $p$ is 1 (closed state with low resistance), the voltage pulse on $p$ ‘shorts out’ the voltage divider formed by $q$ and $R_G$ and both $p$ and $q$ are left unchanged. If the initial state of $q$ is 0 (open state with high resistance), there is little influence on the voltage divider so $q$ is set to 1 and $p$ is left unchanged.

![Figure 5: Memristor Implication Logic](image)

<table>
<thead>
<tr>
<th></th>
<th>$p$</th>
<th>$q$</th>
<th>$V_{\text{COND}}$</th>
<th>$V_{\text{SET}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 shows an example of three memristors, $p_1$, $p_2$, and $q$. We calculate $p_1 \text{ NAND } p_2$ through a series of implication logic operations and store the result in $q$. The calculation includes three steps. First we apply $V_{\text{CLEAR}}$ onto $q$ to force it into open state ($q = 0$). Then we apply $V_{\text{COND}}$ onto $p_1$ and $V_{\text{SET}}$ onto $q$ and $q$ becomes the complement of $p_1$ ($q$
Then we apply $V_{\text{COND}}$ onto $p_2$ and $V_{\text{SET}}$ onto $q$ and $q$ becomes the NAND of $p_1$ and $p_2$ ($q = \neg p_2 \lor \neg p_1 = \neg(p_2 \land p_1)$). Suppose the initial states of $p_1$ and $p_2$ are $X_1$ and $X_2$ respectively, Table 1 shows the voltages applied onto different memristors and the states of different memristors at different steps.

<table>
<thead>
<tr>
<th>Memristor States</th>
<th>Applied Voltages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$ $p_2$ $q$</td>
<td>$V_1$ $V_2$ $V$</td>
</tr>
<tr>
<td><strong>Step 1</strong> $X_1$ $X_2$ 0</td>
<td>0 0 $V_{\text{CLEAR}}$</td>
</tr>
<tr>
<td><strong>Step 2</strong> $X_1$ $X_2$ $\neg X_1$</td>
<td>$V_{\text{COND}}$ 0 $V_{\text{SET}}$</td>
</tr>
<tr>
<td><strong>Step 3</strong> $X_1$ $X_2$ $\neg(X_1 \land X_2)$</td>
<td>0 $V_{\text{COND}}$ $V_{\text{SET}}$</td>
</tr>
</tbody>
</table>

### 2.3 Memristor Array

A memristor array has ultra-high density (e.g., $10^{11}$ bits/cm$^2$ with a crossbar of approximately 17 nm half-pitch [13]) and is highly scalable with four-dimensional address topology [14] (e.g., the theoretical density is as high as $10^{14}$ bits/cm$^2$ with stacked multilayer crossbar arrays). Memristors also have short switching time (can be 1 ns [15] and low energy consumption (can be sub-pJ for a write [15]). A memristor-based
memory system has been proposed by HP Labs [16] and it shows that it is feasible to read from/write into a memristor array based on a nano-scale crossbar. The memristor array read/write latency is claimed to be within a factor of two compared with DRAM read/write latency [1]. A hybrid CMOS-memristor CAM has also been proposed to achieve larger capacity [9]. However, combining CMOS transistors with memristors reduces bit density and increases manufacturing difficulty. Instead of integrating CMOS transistors and memristors into the same array, we propose to build storage structures using only memristors to better utilize the high density feature of memristor arrays.

Using memristors as both memory latches and logic gate provides not only high density but also configurability. There are three main configurations for a memristor array. First, we can configure all the memristors as memory latches and the array works as memory. Second, we can configure CAM/TCAM cells using all the memristors and the array works as CAM/TCAM. In this case, we can also configure different number of entries with different key sizes. Third, we can configure CAM/TCAM cells using a proportion of the memristors and configure the remaining memristors as memory latches. In this case, we have a hybrid CAM/TCAM-memory. We can choose different configurations based on the requirements of different applications.
3. MemCAM and MemTCAM: Memristor-based Content Addressable Memory and Ternary Content Addressable Memory

Among emerging nano-scale RAM technologies, memristors have become strong candidates to replace CMOS-based memory because of high density [13] and high scalability [14]. Since HP Labs already demonstrate a memristor-based memory and the reading/writing process [16], we propose designs of a memristor-based CAM (MemCAM) and a memristor-based TCAM (MemTCAM) in this chapter. We first present an overview of MemCAM/MemTCAM and then explain it bottom-up. We focus on MemCAM/MemTCAM cell design and match signal combination. Peripheral circuitry required to write into and read from the memristor array is beyond the scope of this thesis.

3.1 MemCAM/MemTCAM Overview

Figure 7 shows potential memory organizations through which we can integrate MemCAM/MemTCAM with existing memory hierarchies. We can use MemCAM/MemTCAM to replace DRAM as shown in Figure 7(a) since the capacity of MemCAM/MemTCAM can be larger than DRAM. We can also add a CMOS-based CAM/TCAM as a buffer between a CMOS-based cache and MemCAM/MemTCAM to improve performance and reduce MemCAM/MemTCAM accesses as shown in Figure 7(b). Furthermore, we can access CMOS-based CAM/TCAM and MemCAM/MemTCAM simultaneously as shown in Figure 7(c). Since the search time of CMOS-based
CAM/TCAM is shorter than the search time of MemCAM/MemTCAM (which is shown later in this chapter), we can fetch the matched entries in CMOS-based CAM/TCAM and perform search operations in MemCAM/MemTCAM at the same time to further improve performance.

![Diagram](image.png)

(a) (b) (c)

**Figure 7: Memory Organizations with MemCAM/MemTCAM**

We use memristors as both memory latches and logic gates to build MemCAM/MemTCAM. MemCAM/MemTCAM is based on a nano-scale crossbar.

Multiple memristors in the same row are grouped together to form a MemCAM/MemTCAM cell as shown in Figure 8. Each dashed line rectangle represents a cell. Multiple cells in the same row are then grouped together to form an entry and the number of cells in an entry is equal to the number of bits in the input key. In Figure 8 there are three entries with three cells in each entry. In each cell, one memristor is used
to store the original data bit and another memristor is used to store the input key bit.

The remaining memristors are used to perform the comparison between the original bit and the input key bit and to combine cell match signals to generate entry match signals after the comparison.

![Figure 8: MemCAM/MemTCAM Structure Overview](image)

MemCAM/MemTCAM operations are different based on the types of queries. During a point query, which searches for the entries that are equal to an input key, we perform one search operation with the input key. During a range query, which searches for the entries that are within a given range, if we are searching for the entries smaller than a given upper bound, we perform one search operation with the upper bound as the input key; if we are searching for the entries larger than a given lower bound, we perform one search operation with the lower bound as the input key; if we are searching for the entries smaller than a given upper bound and larger than a given lower bound, we perform two search operations, one with the upper bound as the input key and the other with the lower bound as the input key, and then combine the results of the two search operations to generate the matches.
During one search operation, we first write the input key into every entry in MemCAM/MemTCAM. We then compare the input key with each entry simultaneously. The comparison process varies based on the types of queries. For a point query, we check whether an entry is a match to the input key. For a range query with a given lower bound as the input key, we check whether an entry is greater than the input key, and for a range query with a given upper bound as the input key, we check whether an entry is less than the input key.

After the comparison process, the match signal of a cell, which is one bit for point query and two bits for range query, is stored in a memristor within the cell. We then generate the match signal for each entry through match signal combination by combining the match signals of all the cells in each entry. For point query or range query with only one given bound (either upper bound or lower bound), the entry match signals can then be used to generate addresses to read the data related to the matched entries. We can either process the entry match signals within the memristor array using implication logic or read them out and process them using CMOS-based circuitry. The detailed process is out of the scope of this thesis. For range query with both upper and lower bounds, we have to read the entry match signals out into other storage structures and combine the entry match signals of the two search operations to generate the addresses for the data related to the matched entries. The detailed process is also out of the scope of this thesis.
3.2 MemCAM/MemTCAM Cell Design

Memristors are used as both memory latches and logic gates in MemCAM/MemTCAM cell design. Implication logic, which can be realized by combining a conventional resistor with two electrically connected memristors [12], is used as the basic operation to compute Boolean functions. In this section, we propose two different cell designs: MemCAM cell design and MemTCAM cell design. For each design, we propose two different comparison processes: one supporting only point query and the other supporting both point query and range query. Point query determines whether a given object is a member of a dataset and range query obtains all the objects whose attribute values exists within a range. Traditional CAM and TCAM only support point query because they can only check for equality. With the computation ability of memristors we are able to check whether an entry in MemCAM or MemTCAM is equal to, or greater than, or less than an input so we can support both point query and range query.

3.2.1 MemCAM

Figure 9 shows the design of a 6-memristor MemCAM cell. The horizontal wire (X) is connected to the load resistor \( R_c \) defined in the previous chapter. The vertical wires (Y1-Y6) are controlled by tri-state voltage drivers that have a high-impedance output state when undriven. Each square represents a memristor. D (representing Data) is the memristor used to store the data bit, and K (representing Key) is the memristor
used to store the input key bit. $M_1$ to $M_4$ are memristors used to perform comparison process and store match signals. We use 6 memristors to build a MemCAM cell because during the comparison of $D$ and $K$, we need two memristors to store the complements of $D$ and $K$, one memristor to store the cell match signal and one memristor to store the complement of the cell match signal for fast match signal combination which we will explain in detail later in this chapter.

![Figure 9: 6-memristor MemCAM Cell Design](image)

3.2.1.1 Comparison Process Supporting only Point Query

We first design a comparison process for MemCAM to support only point query, which includes 8 steps as shown in Figure 10. In the first 7 steps we generate the match signal for the MemCAM cell and in the last step we prepare for fast match signal combination. $M_1$ and $M_2$ are used to store $\neg D$ and $\neg K$. $D$ and $M_1$ are then used to compute the NAND of $D$ and $K$, and $M_1$ and $K$ are used to compute the NAND of $\neg D$ and $\neg K$. Finally, $K$, $M_1$ and $M_3$ are used to compute the XNOR of $D$ and $K$, which is the match signal for the MemCAM cell and is stored in $M_3$. $M_4$ is used to store the complement of $M_3$ and used for fast match signal combination. If we don't utilize fast match signal combination, we can skip the last step.
Figure 10: MemCAM Comparison Process Supporting only Point Query

Table 2 shows the voltages applied to the vertical control lines, \( Y_1 \)-\( Y_6 \), in a MemCAM cell at each step. Table 3 shows the states of \( K \) and \( M_1 \) through \( M_4 \) at each step. During the comparison process, the state of \( D \) is not changed so its state is not shown in Table 3.

### Table 2: Applied Voltages at Each Step during Comparison for MemCAM Supporting Point Query

<table>
<thead>
<tr>
<th>Step</th>
<th>( Y_1 )</th>
<th>( Y_2 )</th>
<th>( Y_3 )</th>
<th>( Y_4 )</th>
<th>( Y_5 )</th>
<th>( Y_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>0</td>
<td>( V_{\text{CLEAR}} )</td>
<td>( V_{\text{CLEAR}} )</td>
<td>( V_{\text{CLEAR}} )</td>
<td>( V_{\text{CLEAR}} )</td>
</tr>
<tr>
<td>Step 2</td>
<td>( V_{\text{COND}} )</td>
<td>0</td>
<td>( V_{\text{SET}} )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>0</td>
<td>( V_{\text{COND}} )</td>
<td>0</td>
<td>( V_{\text{SET}} )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>( V_{\text{COND}} )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( V_{\text{SET}} )</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>0</td>
<td>( V_{\text{SET}} )</td>
<td>( V_{\text{COND}} )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( V_{\text{COND}} )</td>
<td>( V_{\text{SET}} )</td>
<td>0</td>
</tr>
<tr>
<td>Step 7</td>
<td>0</td>
<td>( V_{\text{COND}} )</td>
<td>0</td>
<td>0</td>
<td>( V_{\text{SET}} )</td>
<td>0</td>
</tr>
<tr>
<td>Step 8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( V_{\text{COND}} )</td>
<td>( V_{\text{SET}} )</td>
</tr>
</tbody>
</table>
Table 3: States of Memristors at Each Step during Comparison for MemCAM Supporting Point Query

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
<th>M₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>K</td>
<td>¬D</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>¬D</td>
<td>¬K</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>¬D</td>
<td>¬D ∨ ¬K</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Step 5</td>
<td>¬D</td>
<td>¬D ∨ ¬K</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Step 6</td>
<td>¬D</td>
<td>¬D ∨ ¬K</td>
<td>D ∧ K</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Step 7</td>
<td>¬D</td>
<td>¬D ∨ ¬K</td>
<td>(D ∧ K) ∨ (¬D ∧ ¬K)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Step 8</td>
<td>¬D</td>
<td>¬D ∨ ¬K</td>
<td>(D ∧ K) ∨ (¬D ∧ ¬K)</td>
<td>D ⊕ K</td>
<td></td>
</tr>
</tbody>
</table>

3.2.1.2 Comparison Process Supporting both Point Query and Range Query

We then design a comparison process for MemCAM to not only support point query but also support range query, which includes 7 steps as shown in Figure 11. In order to support range query, we have to be able to check whether an entry is equal to, or greater than, or less than the input key. Thus we generate two bits based on D and K to represent the cell match signal, instead of one bit for only point query. Table 4 shows the values and meanings of cell match signals based on the values of D and K.

M₁ and M₂ are used to store ¬D and ¬K first. M₁ and M₂ are used to compute D ∨ ¬K and the result is stored in M₂. D and K are then used to compute ¬D ∨ K and the result is stored in K. Finally, M₃ and M₄ are used to store the complements of M₂ and K respectively, which represent the match signal for the MemCAM cell.
Figure 11: MemCAM Comparison Process
Supporting both Point Query and Range Query

Table 4: Values and Meanings of Cell Match Signals
Based on Values of \( D \) and \( K \)

<table>
<thead>
<tr>
<th>( D )</th>
<th>( K )</th>
<th>( M_1 = D \land K )</th>
<th>( M_4 = D \land \lnot K )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5 shows the voltages applied to the vertical control lines, \( Y_1 \)–\( Y_6 \), in a MemCAM cell at each step. Table 6 shows the states of \( K \) and \( M_i \) through \( M_4 \) at each step.
step. During the comparison process, the state of D is not changed so its state is not shown in Table 6.

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
<th>M₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>K</td>
<td>¬D</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>K</td>
<td>¬D</td>
<td>¬K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>K</td>
<td>¬D</td>
<td>D ∨ ¬K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>¬D ∨ K</td>
<td>¬D</td>
<td>D ∨ ¬K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>¬D ∨ K</td>
<td>¬D</td>
<td>D ∨ ¬K</td>
<td>¬D ∧ K</td>
<td>0</td>
</tr>
<tr>
<td>Step 7</td>
<td>¬D ∨ K</td>
<td>¬D</td>
<td>D ∨ ¬K</td>
<td>¬D ∧ K</td>
<td>D ∧ ¬K</td>
</tr>
</tbody>
</table>

3.2.2 MemTCAM

Figure 12 shows the design of a 7-memristor MemTCAM cell. The horizontal wire (X) is connected to the load resistor Rₒ defined in the previous chapter. The vertical wires (Y₁-Y₆) are controlled by tri-state voltage drivers that have a high-impedance output state when undriven. Each square represents a memristor. D₀ and ¬D₁ are two memristors used to store two bits representing the data bit, and K is the memristor used to store the input key bit. We store ¬D₁ instead of D₁ in order to save one step during the comparison process. M₁ to M₄ are memristors used to perform comparison process and store match signals. Table 7 shows the relationship between values of D₁D₀, K and the cell match signal. When D₁D₀ is 00, it means the data bit is 0. When D₁D₀ is 01, it means the data bit is 1. When D₁D₀ is 1X, it means the data bit is a ‘don’t care’ bit, which means that no matter what the input key bit K is, the stored data bit is always a match to K.
Figure 12: 7-memristor MemTCAM Cell Design

Table 7: Relationship between D1,D0, K and Match Signal

<table>
<thead>
<tr>
<th>D1</th>
<th>D0</th>
<th>K</th>
<th>Cell Match Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

3.2.2.1 Comparison Process Supporting only Point Query

We first design a comparison process for MemTCAM to support point query, which includes 9 steps as shown in Figure 13. In the first 8 steps we generate the match signal for the MemTCAM cell and in the last step we prepare for fast match signal combination. M1 and M2 are used to store ¬D0 and ¬K. D0 and M2 are then used to compute the NAND of D0 and K, and M1 and K are used to compute the NAND of ¬D0 and ¬K. K, M2 and M3 are then used to compute the NXOR of D0 and K. Finally, ¬D1 is combined with the value in M3 to generate the match signal for the MemTCAM cell, which is stored in M3. M4 is used to store the complement of M3 and used for fast match signal combination. If we don’t utilize fast match signal combination, we can skip the last step.
Figure 13: MemTCAM Comparison Process Supporting only Point Query

Table 8 shows the voltages applied to the vertical control lines, Y1-Y7, in a MemTCAM cell at each step. Table 9 shows the states of K and M1 through M4 at each step. During the comparison process, the states of D0 and ¬D1 are not changed so their states are not shown in Table 9.

Table 8: Applied Voltages at Each Step during Comparison for MemTCAM Supporting Point Query

<table>
<thead>
<tr>
<th>Step</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
<th>Y5</th>
<th>Y6</th>
<th>Y7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>0</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>V\text{COND}</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>V\text{COND}</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>0</td>
<td>V\text{SET}</td>
<td>V\text{COND}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>0</td>
<td>0</td>
<td>V\text{COND}</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 7</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>V\text{COND}</td>
</tr>
<tr>
<td>Step 9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{COND}</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 9: States of Memristors at Each Step during Comparison for MemTCAM Supporting Point Query

<table>
<thead>
<tr>
<th>Step</th>
<th>$K$</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$K$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$K$</td>
<td>$\neg D_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$K$</td>
<td>$\neg D_0$</td>
<td>$\neg K$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>$K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>$D_0 \lor K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>$D_0 \lor K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>$D_0 \land K$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>$D_0 \lor K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>$\neg(D_0 \oplus K)$</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>$D_0 \lor K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>$D_1 \lor \neg(D_0 \oplus K)$</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>$D_0 \lor K$</td>
<td>$\neg D_0$</td>
<td>$\neg D_0 \lor \neg K$</td>
<td>$D_1 \lor \neg(D_0 \oplus K)$</td>
<td>$\neg D_1 \land(D_0 \oplus K)$</td>
</tr>
</tbody>
</table>

3.2.2.2 Comparison Process Supporting both Point Query and Range Query

We then design a comparison process for MemTCAM to support both point query and range query, which includes 11 steps as shown in Figure 14. Similar to MemCAM supporting both point query and range query, we generate two bits based on $D_1 D_0$ and $K$ to represent the cell match signal, instead of one bit for only point query. Table 10 shows the values and meanings of cell match signals based on the values of $D_1 D_0$ and $K$.

Table 10: Values and Meanings of Cell Match Signals Based on Values of $D_1 D_0$ and SL

<table>
<thead>
<tr>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$K$</th>
<th>$M_3=D_1 \land \neg D_0 \land K$</th>
<th>$M_4=D_1 \land D_0 \land \neg K$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$D = K$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$D = K$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$D &gt; K$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$D &lt; K$</td>
</tr>
<tr>
<td>1</td>
<td>$X$</td>
<td>$X$</td>
<td>0</td>
<td>0</td>
<td>$D = K$</td>
</tr>
</tbody>
</table>
Figure 14: MemTCAM Comparison Process Supporting both Point Query and Range Query

$M_1$ and $M_2$ are used to store $\neg D_0$ and $\neg K$ first. $K$ and $M_2$ are then used to compute $D_0 \lor \neg K$ and the result is stored in $M_2$, and $D_0$ and $K$ are used to compute $\neg D_0 \lor K$ and the result is stored in $K$. $M_4$ is then used to store the value of $D_1$ and combined with the values of $M_2$ and $K$. Finally, $M_3$ and $M_4$ are cleared and used to store the match signal for the MemTCAM cell.

Table 11 shows the voltages applied to the vertical control lines, $Y_1$-$Y_7$, in a MemTCAM cell at each step. Table 12 shows the states of $K$ and $M_1$ through $M_4$ at each step. During the comparison process, the states of $D_0$ and $\neg D_1$ are not changed so their states are not shown in Table 12.
Table 11: Applied Voltages at Each Step during Comparison for MemTCAM Supporting both Point and Range Queries

<table>
<thead>
<tr>
<th>Step</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
<th>Y5</th>
<th>Y6</th>
<th>Y7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>0</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>V\text{CLEAR}</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>V\text{COND}</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>0</td>
<td>0</td>
<td>V\text{COND}</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>V\text{COND}</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>V\text{COND}</td>
</tr>
<tr>
<td>Step 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
</tr>
<tr>
<td>Step 8</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
</tr>
<tr>
<td>Step 9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{CLEAR}</td>
<td>0</td>
</tr>
<tr>
<td>Step 10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{COND}</td>
<td>V\text{SET}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 11</td>
<td>0</td>
<td>V\text{COND}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V\text{SET}</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 12: States of Memristors at Each Step during Comparison for MemTCAM Supporting both Point and Range Queries

<table>
<thead>
<tr>
<th>K</th>
<th>M_1</th>
<th>M_2</th>
<th>M_3</th>
<th>M_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>K</td>
<td>\neg D_0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>K</td>
<td>\neg D_0</td>
<td>\neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>K</td>
<td>\neg D_0</td>
<td>D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>\neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>\neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 7</td>
<td>\neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>\neg D_1 \lor D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 8</td>
<td>\neg D_1 \lor \neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>\neg D_1 \lor D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 9</td>
<td>\neg D_1 \lor \neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>\neg D_1 \lor D_0 \lor \neg K</td>
<td>0</td>
</tr>
<tr>
<td>Step 10</td>
<td>\neg D_1 \lor \neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>\neg D_1 \lor D_0 \lor \neg K</td>
<td>D_1 \land \neg D_0 \land K</td>
</tr>
<tr>
<td>Step 11</td>
<td>\neg D_1 \lor \neg D_0 \lor K</td>
<td>\neg D_0</td>
<td>\neg D_1 \lor D_0 \lor \neg K</td>
<td>D_1 \land \neg D_0 \land K</td>
</tr>
</tbody>
</table>
3.3 Match Signal Combination

After each MemCAM/MemTCAM cell finishes comparison and generates its cell match signal (CMS), we need to combine the match signals from all cells in an entry to generate the entry match signal (EMS). In this subsection we first propose a fast match signal combination process which requires little or no device variability and only supports point query. We then propose another match signal combination process to support both point and range queries and does not require little or no device variability.

3.3.1 Match Signal Combination with Little Device Variability

We first exploit the variable resistance characteristic of memristors to combine match signals for MemCAM/MemTCAM supporting only point query. From the previous subsection, we can see that the last step in the comparison processes supporting only point query inverts cell match signals. Figure 15 shows examples of where the inverted cell match signals are stored in MemCAM/MemTCAM. Figure 15(a) shows an example of a MemCAM with 3 entries and 3 bits in each entry and Figure 15(b) shows an example of a MemTCAM with 3 entries and 3 bits in each entry. The inverted cell match signals are stored in gray memristors.

After the comparison, reading is enabled at each gray memristor. At each row, one end of the horizontal wire is connected with the load resistor \( R_c \) and we measure the current at the other end. If there are cell match signals with value 1 (low resistance) in one row, the current should be larger than the current from a row where all the cell
match signals are 0 (high resistance), which indicates the corresponding entry is not a match. In order to be more accurate, we may add one extra row with 0s stored in all the gray memristors and use the corresponding current as a baseline.

![Image of 3 Bit x 3 Entry MemCAM and MemTCAM Cells](image)

(a) 3 Bit x 3 Entry MemCAM

(b) 3 Bit x 3 Entry MemTCAM

Figure 15: 3 Bit x 3 Entry MemCAM/MemTCAM Cell Layout

### 3.3.2 Match Signal Combination with Large Device Variability

Match signal combination based on the variable resistance characteristic of memristors is fast but requires little or no component parameter variability. If the variability is large, especially when the off state resistances of some memristors and the on state resistances of some other memristors are in the same order of magnitude, the previous match signal combination method does not work. In this case, we propose an alternative method and describe it in detail in this subsection.

We have:

\[
EMS_i = \text{Combination of } (CMS_{i,0}, CMS_{i,1}, CMS_{i,2}, \ldots, CMS_{i,n})
\]

in which \(EMS_i\) is the match signal of the \(i^{th}\) entry in MemCAM/MemTCAM, \(CMS_{i,j}\) is the
match signal of the $j^{th}$ cell in the $i^{th}$ entry and $n$ is the number of cells in an entry, which is also the number of bits in the key word.

### 3.3.2.1 MemCAM/MemTCAM Supporting only Point Query

We assume $n$ to be power of two here and use recursive doubling [17] to combine match signals for MemCAM/MemTCAM supporting only point query. For each Entry $i$, we first compute the AND of every CMS pair, $CMS_{i,2^j}$ and $CMS_{i,2^{j+1}}$ simultaneously and store the result in the memristor used to store $CMS_{i,2^j}$. We then compute the AND of every CMS pair $CMS_{i,4^j}$ and $CMS_{i,4^{j+2}}$ similarly. EMS is in the memristor used to store $CMS_{i,0}$ after $\log_2 n$ rounds.

![Match Signal Combination Example for MemCAM/MemTCAM Supporting only Point Query](image)

Each round of match signal combination includes 5 steps. Before match signal combination, each CMS is stored in $M_i$ in a cell. We use three memristors including two memristors already storing the CMSs to compute the AND of two CMSs from two cells.

Figure 16(a) shows an MemCAM example and Figure 16(b) shows an MemTCAM
example of two adjacent cells, Cell \(2j\) and Cell \(2j+1\) in Entry \(i\). Cell match signals are stored in gray memristors before combination. For brevity we use \(M_3\), \(M_4\) and CMS to represent \(M_{i,2j,3}\), \(M_{i,2j,4}\) and CMS\(_{i,2j}\), and \(M'\) and CMS' to represent \(M_{i,2j+1,4}\) and CMS\(_{i,2j+1}\). \(M_3\) is first cleared to state ‘0’ and then used to store \(\neg M_4\). \(M'\) and \(M_3\) are then used to compute the NAND of \(M_4\) and \(M'\) and the result is stored in \(M_3\). Finally, \(M_5\) is used to store \(\neg M_3\), which is the AND of the original \(M_4\) and \(M'\). The voltages applied to the vertical control lines at each step are shown in Table 13 and the states of memristors used for combination are shown in Table 14.

### Table 13: Applied Voltages at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting only Point Query

<table>
<thead>
<tr>
<th>Step</th>
<th>(Y_5)</th>
<th>(Y_6)</th>
<th>(Y'_6)</th>
<th>Other Y Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>(V_{\text{CLEAR}})</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>(V_{\text{SET}})</td>
<td>(V_{\text{COND}})</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>(V_{\text{SET}})</td>
<td>0</td>
<td>(V_{\text{COND}})</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>0</td>
<td>(V_{\text{CLEAR}})</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>(V_{\text{COND}})</td>
<td>(V_{\text{SET}})</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 14: States of Memristors at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting only Point Query

<table>
<thead>
<tr>
<th>Step</th>
<th>(M_3)</th>
<th>(M_4)</th>
<th>(M'_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>CMS</td>
<td>CMS'</td>
</tr>
<tr>
<td>Step 2</td>
<td>(\neg\text{CMS})</td>
<td>CMS</td>
<td>CMS'</td>
</tr>
<tr>
<td>Step 3</td>
<td>(\neg\text{CMS} \vee \neg\text{CMS}')</td>
<td>CMS</td>
<td>CMS'</td>
</tr>
<tr>
<td>Step 4</td>
<td>(\neg\text{CMS} \vee \neg\text{CMS}')</td>
<td>0</td>
<td>CMS'</td>
</tr>
<tr>
<td>Step 5</td>
<td>(\neg\text{CMS} \vee \neg\text{CMS}')</td>
<td>CMS (\wedge) CMS'</td>
<td>CMS'</td>
</tr>
</tbody>
</table>
3.3.2.2 MemCAM/MemTCAM Supporting both Point and Range Queries

Similar to the match signal combination process of MemCAM, for each Entry $i$, we first combine every CMS pair, CMS$_{i,2j}$ and CMS$_{i,2j+1}$ simultaneously and store the result in the memristor used to store CMS$_{i,2j+1}$. We then combine every CMS pair CMS$_{i,4j}$ and CMS$_{i,4j+2}$ similarly. EMS$_i$ is in the memristors used to store CMS$_{i,n-1}$ after $\log_2 n$ rounds. The difference is that each CMS has two bits to support both point and range queries, instead of one bit to support only point query, so the operation steps are different during each round of the match signal combination.

![Match Signal Combination Example for MemCAM/MemTCAM Supporting both Point and Range Queries](image)

**Figure 17**: Match Signal Combination Example for MemCAM/MemTCAM Supporting both Point and Range Queries
Table 15: Applied Voltages at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting both Point and Range Queries

<table>
<thead>
<tr>
<th>Step</th>
<th>Y₃</th>
<th>Y₆</th>
<th>Y₅'</th>
<th>Y₆'</th>
<th>Y₂</th>
<th>Y₄</th>
<th>Other Y Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V_CLEAR</td>
<td>V_CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V_SET</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>0</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V_SET</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>0</td>
<td>V_SET</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 5</td>
<td>V_SET</td>
<td>0</td>
<td>0</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 6</td>
<td>0</td>
<td>0</td>
<td>V_CLEAR</td>
<td>V_CLEAR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 7</td>
<td>0</td>
<td>V_SET</td>
<td>0</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 8</td>
<td>0</td>
<td>0</td>
<td>V_SET</td>
<td>0</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 9</td>
<td>0</td>
<td>V_COND</td>
<td>V_SET</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 10</td>
<td>V_COND</td>
<td>0</td>
<td>0</td>
<td>V_SET</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 16: States of Memristors at Each Step during One Round of Match Signal Combination for MemCAM/MemTCAM Supporting both Point and Range Queries

<table>
<thead>
<tr>
<th>Step</th>
<th>M₃</th>
<th>M₄</th>
<th>M'₃</th>
<th>M'₄</th>
<th>M₂</th>
<th>M'₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>CMSᵢ</td>
<td>CMS₀</td>
<td>CMS'₁</td>
<td>CMS'₀</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>CMSᵢ</td>
<td>CMS₀</td>
<td>CMS'₁</td>
<td>CMS'₀</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 3</td>
<td>CMSᵢ</td>
<td>CMS₀</td>
<td>CMS'₁</td>
<td>CMS'₀</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 4</td>
<td>CMSᵢ</td>
<td>CMSᵢ ∨ CMS₀ ∨ CMS₁</td>
<td>CMS'₁</td>
<td>CMS'₀</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 5</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>CMS'₁</td>
<td>CMS'₀</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 6</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>0</td>
<td>0</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 7</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>CMS₁</td>
<td>0</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 8</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>CMS₁</td>
<td>CMS₀</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 9</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>CMSᵢ ∨ (CMS₀ ∧ CMS₁)</td>
<td>0</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
<tr>
<td>Step 10</td>
<td>CMSᵢ ∨ CMS₀</td>
<td>CMSᵢ ∨ CMS₁</td>
<td>CMSᵢ ∨ (CMS₀ ∧ CMS₁)</td>
<td>CMS₀ ∨ (~CMS₀ ∧ CMS₀)</td>
<td>~CMSᵢ</td>
<td>~CMS₀</td>
</tr>
</tbody>
</table>
Each round of match signal combination includes ten steps. We use six memristors including four memristors already storing the CMSs to combine two CMSs from two cells. Figure 17(a) shows an MemCAM example and Figure 17(b) shows an MemTCAM example of two adjacent cells, Cell 2j and Cell 2j+1 in Entry i. Cell match signals are stored in gray memristors before combination. For brevity we use M3, M4, M2 and CMS to represent M_{i,2j,3}, M_{i,2j,4}, M_{i,2j,2} and CMS_{i,2j} and M’3, M’4, M’2 and CMS’ to represent M_{i,2j+1,3}, M_{i,2j+1,4}, M_{i,2j+1,2} and CMS_{i,2j+1}. The voltages applied to the vertical control lines at each step are shown in Table 15 and the states of memristors used for combination are shown in Table 16.

### 3.4 Evaluation

We propose MemCAM/MemTCAM to achieve higher CAM/TCAM capacity while maintaining search time and power consumption comparable to CMOS-based CAM. To analyze search time and power consumption of MemCAM/MemTCAM, we use a simplified SPICE model proposed by Mahvash and Parker [18] to measure switching time and power consumption of a single memristor. We then analyze the results and discuss alternatives to reduce search time and energy consumption of MemCAM/MemTCAM.

#### 3.4.1 Methodology

A memristor array can have ultra-high density. We can achieve $10^{11}$ memristors/cm² if we build the memristor array on 17-nm-wide nanowires [13]. Our
evaluation is based on a conservative design, a memristor array built on 50-nm-wide nanowires [15] with 50nm x 50nm x 10nm memristors and memristor density of the array is $10^{10}$ memristors/cm$^2$.

We model a memristor as a variable resistor whose resistance (memristance) can be described by the following equation [7]:

**Equation 2:** $M(q) = R_{ON} \frac{w(t)}{D} + R_{OFF} \left(1 - \frac{w(t)}{D}\right). \quad (2)$

in which $M(q)$ is the memristance of a memristor, $R_{ON}$ and $R_{OFF}$ are the resistances of a memristor in the closed and open states, respectively, $w(t)$ is the width of the doped region, and $D$ is the total width of the TiO$_2$ film used to build the memristor. $w(t)$ is given by:

**Equation 3:** $w(t) = \frac{\mu_w R_{ON}}{D} q(t). \quad (3)$

in which $\mu_w$ is the dopant mobility. When $R_{ON} \ll R_{OFF}$, (2) can be simplified to:

**Equation 4:** $M(q) = R_{OFF} \left(1 - \frac{\mu_w R_{ON}}{D^2} q(t)\right). \quad (4)$

The power consumption of each memristor at each step depends on the states of the memristor before and after the step and the voltage across the memristor during the step. We use $P_{xYZ}$ to represent the average power consumption of a memristor at one step. $x$ can be either $w$ (writing voltage $V_{SET}$ or $V_{CLEAR}$ applied) or $r$ (reading voltage $V_{COND}$ applied). $Y$ and $Z$ are the states of the memristor before and after the step. Table 17 shows the average power consumption of a memristor with different initial states and
different applied voltages. When Y is the same as Z, \(P_{x,Y,Y}\) can be estimated based on the applied voltage, resistance at Y state and \(R_c\). When Y and Z are different, we use a simplified SPICE model proposed by Mahvash and Parker [18], to measure switching time and power consumption. Since the magnitude of \(V_{\text{CLEAR}}\) is smaller than \(V_{\text{SET}}\) [15], we measure only the power consumption of switching a memristor from ‘off’ (logic value 0) to ‘on’ (logic value 1), which is considered as the upper bound of the power consumption of switching a memristor from ‘on’ to ‘off’. We use the left column of values of the parameters [15] shown in Table 18 for simulations at room temperature (300K). We define \(R=r\Omega\) where \(r\) can be different for different manufacturing methods and materials.

| Table 17: Power Consumption of a Memristor with Different Initial States and Applied Voltages |
|-------------------------------------------------|-----------------|-----------------|-----------------|
| \(V_{\text{SET}} = 3V\)                        | \(V_{\text{CLEAR}} = 2V\) | \(V_{\text{COND}} = 1V\) |
| OFF \(P_{\text{w,OFF,OFF}}\) or \(P_{\text{w,OFF,ON}}\) | \(P_{\text{w,OFF}}\)                      | \(P_{\text{OFF,OFF}}\)            |
| ON \(P_{\text{w,ON,ON}}\)                      | \(P_{\text{w,ON,OFF}}\)                   | \(P_{\text{ON,ON}}\)             |

| Table 18: Parameter Values at 300K and 513K  |
|-------------------------------------------------|-----------------|-----------------|-----------------|
| Parameter                                      | Value at 300K   | Value at 513K   |
| \(R_{\text{OFF}}\)                             | 160\(R\)       | 16\(R\)        |
| \(R_{\text{ON}}\)                              | \(R\)          | 0.1\(R\)       |
| \(\mu_c\)                                      | \(10^{-10} \text{ cm}^2 \text{s}^{-1} \text{V}^{-1}\) | \(10^{-3} \text{ cm}^2 \text{s}^{-1} \text{V}^{-1}\) |

Figure 18 shows the SPICE model we use and Figure 19 shows the corresponding code lines. \(V_r\) is a dependent voltage source that generates the voltage across the
memristor based on the sensed current times the desired resistance, which is $M(q)$ defined by Equation 2. The current $I$ used to calculate $V_c$ is measured at an independent voltage source $V_{\text{sense}}$, which is 0.0 volts. A capacitor $C_{\text{sense}}$ with capacitance $C$ is used to sense the charge $q = V_c / C$ in the circuit, where $V_c$ is the voltage across $C_{\text{sense}}$. We then have:

**Equation 5:** $V_r = I \times M(q) - V_c = I \times R_{\text{OFF}} \left(1 - \frac{\mu_v R_{\text{ON}}}{D^2} \times \frac{V_c}{C}\right) - V_c$.  

We also use SPICE to measure how the power consumption of a single memristor changes when temperature increases in order to demonstrate the possibility to reduce power consumption at higher temperatures. The values of $R_{\text{ON}}$, $R_{\text{OFF}}$ and $\mu_v$ change as temperature increases. While the decrease of resistance increases power consumption, the increase of drift mobility makes it easier to switch a memristor and can lead to lower power consumption. Previous studies show that oxygen vacancy drift mobility can increase by about seven orders of magnitude [19] and resistance can decrease by about one order of magnitude [20] as temperature increases from room temperature (300K) to 513K. Based on this previous work, the right column of values shown in Table 18 is used for simulations at 513K.
3.4.2 Results and Analysis

Figure 20 shows the power and time of a memristor switched from ‘off’ to ‘on’ at room temperature. We can see that the maximum power is approximately $(0.6/r)$ W and the average power $(P_{w,OFF,ON})$ is approximately $(0.1/r)$ W. We calculate an upper bound of the average power based on the simulation results. When $V_{CLEAR}$ is applied, we use $P_{w,OFF,ON}$ instead of $P_{w,ON,OFF}$ since the power is higher to switch a memristor from ‘off’ to ‘on’ and the power consumption is $(0.1/r)$ W for a memristor. When $V_{SET}$ is applied, if the initial state of the memristor is ‘off’, we use $P_{w,OFF,ON}$ and the power consumption is $(0.1/r)$ W for a memristor; otherwise, we use $P_{w,ON,ON}$ since it is higher than $P_{w,OFF,ON}$ and the power consumption is $(9/r)$ W for a memristor. When $V_{COND}$ is applied, we use the
average of $P_{ON,ON}$ and $P_{OFF,OFF}$ and the power consumption is $(0.5/r)$ W for a memristor.

Table 19 shows power consumption of a single memristor based on the applied voltage and its initial state and $r$ can be as high as $10^8$ [16].

Table 19: Power Consumption (W) Based on Applied Voltage and Initial Memristor State

<table>
<thead>
<tr>
<th></th>
<th>$V_{SET}$</th>
<th>$V_{CLEAR}$</th>
<th>$V_{COND}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>9/$r$</td>
<td>0.1/$r$</td>
<td>0.5/$r$</td>
</tr>
<tr>
<td>OFF</td>
<td>0.1/$r$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Power consumption decreases as temperature increases. The main reason is that $\mu v$ increases and makes it easier to switch a memristor. We can either apply a lower voltage, or have a shorter switching time. When $\mu v$ increases by one order of magnitude, we can reduce the applied voltage by one order of magnitude and obtain the same switching time as at room temperature, or we can use the same applied voltage and reduce the switching time by one order of magnitude. The simulation results actually show that at a certain temperature, the product of the applied voltage and the switching time is constant. Figure 21 shows two extreme cases of the power and time when a memristor is switched from ‘off’ to ‘on’ at 513K with different applied voltages. Figure 21(a) shows the same applied voltage as at room temperature but a much shorter switching time, and Figure 21(b) shows the same switching time but a much lower applied voltage.
Figure 20: Power at Room Temperature (300K) with 3V Applied Voltage

From Figure 20 and Figure 21, we can see that when we reduce the switching time by one order of magnitude, we can reduce the power density by one order of magnitude, and when we reduce the applied voltage by one order of magnitude, we can reduce the power density by two orders of magnitude. Considering the fact that power density increases by one order of magnitude due to the decrease of resistances, we can reduce the applied voltage by seven orders of magnitude and maintain the same switching time to achieve up to thirteen orders of magnitude reduction in power density at 513K.

However, due to the activation energy of oxygen vacancy, a threshold voltage limits the reduction of the applied voltage. The applied voltage must be higher than the threshold voltage in order to switch a memristor. Further study on activation energy
and threshold voltage is required to decide the reduction of the applied voltage we can obtain.

(a) Applied voltage = 3V

(b) Applied voltage = 300 nV

Figure 21: Power at 513K with Different Applied Voltages
Table 20: Search Time (ns) of MemCAM/MemTCAM with Different Query Support for $2^n$-bit Key Word

<table>
<thead>
<tr>
<th></th>
<th>MemCAM</th>
<th>MemTCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support only point query</td>
<td>$14+10n$</td>
<td>$16+10n$</td>
</tr>
<tr>
<td>Support both point and range queries</td>
<td>$16+20n$</td>
<td>$22+20n$</td>
</tr>
</tbody>
</table>

Table 21: Energy Consumption (fJ/bit/search) of MemCAM/MemTCAM with Different Query Support for $2^n$-bit Key Word

<table>
<thead>
<tr>
<th></th>
<th>MemCAM</th>
<th>MemTCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support only point query</td>
<td>$0.61+0.22n$</td>
<td>$0.8+0.22n$</td>
</tr>
<tr>
<td>Support both point and range queries</td>
<td>$0.44+0.82n$</td>
<td>$0.83+0.82n$</td>
</tr>
</tbody>
</table>

Table 22: Power Density (W/cm$^2$) of MemCAM/MemTCAM with Different Query Support for 64-bit and 1024-bit Key Word

(a) 64-bit Key Word

<table>
<thead>
<tr>
<th></th>
<th>MemCAM</th>
<th>MemTCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support only point query</td>
<td>26</td>
<td>28</td>
</tr>
<tr>
<td>Support both point and range queries</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

(b) 1024-bit Key Word

<table>
<thead>
<tr>
<th></th>
<th>MemCAM</th>
<th>MemTCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support only point query</td>
<td>24.6</td>
<td>25.9</td>
</tr>
<tr>
<td>Support both point and range queries</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

Besides activation energy and threshold voltage, we also have to consider the effect of RC delay, which limits the reduction of step time (time required to perform a step of operation). Step time depends on both switching time and RC delay, which can overlap because switching starts as soon as the voltage across a memristor goes beyond
the threshold voltage. Based on the RC delay of 35 nm Cu-Low κ technology (250 ps for a 1 mm line [21]), methods such as repeater insertion are required to obtain a < 200ps RC delay for a 1-cm-long 50-nm-wide line at both room temperature and 513K. As a result, we can obtain a step time of < 2 ns at room temperature and 1 ns at 513K. The utilization of emerging nanotechnologies such as CNT [21] can further reduce RC delay but it may be more difficult to build a crossbar.

Now we have the step time (2 ns) and the power consumption of a memristor based on the applied voltage and its initial state (Table 19), we calculate the search time, energy consumption, and power density for MemCAM/MemTCAM with different comparison processes described in the previous subsection. Table 20 and Table 21 show the search time and energy consumption of MemCAM/MemTCAM with different query supports for 2^n-bit key word and continuous search operations. Table 22 shows the power density of MemCAM/MemTCAM with different query supports for 64-bit and 1024-bit key words and continuous search operations. We can see that the power density decreases slightly as the size of key word increases for MemCAM/MemTCAM supporting only point query but remains almost the same for MemCAM/MemTCAM supporting both point and range queries. The reason is that the average power consumption during the comparison dominates for MemCAM/MemTCAM supporting only point query, while the average power consumption during the match signal
combination dominates for MemCAM/MemTCAM supporting both point and range queries.

Besides memristors, wires also consume power. During one experiment from HP Labs, wires consume up to 80% of the power [15]. However, in their experiment, the number of memristors and the number of wires are similar while in our 1cm x 1cm MemCAM there are $10^{10}$ memristors but only $2 \times 10^5$ wires. As a result, the wire power percentage in MemCAM should be lower. Furthermore, wire power density can be reduced by methods that could dramatically reduce the wire resistance and capacitance [15] since interconnect power is proportional to the wire capacitance [21, 22]. We conservatively assume that wires consume 50% of the total power, which leads to a total power density of approximate 55W/cm$^2$ for MemCAM/MemTCAM supporting only point query and 80W/cm$^2$ for MemCAM/MemTCAM supporting both point and range queries.

The power density can be reduced by utilizing the configurability of MemCAM/MemTCAM based on the requirement of different applications. We can configure the memristor array into a hybrid memory-CAM/TCAM. We can then use the CAM/TCAM part to store the key bits and use the memory part to store the data bits. The reduction of power density depends on the ratio of the number of key bits and the number of data bits in an entry. For example, if there are 128 key bits and 896 data bits in
every 1024-bit entry, we can reduce the overall power density to approximate 1/8 of the CAM/TCAM power density.

From the simulation results and analysis above, we can conclude that it is feasible to build a 1Gbit MemCAM/MemTCAM with 1cm x 1cm area while maintaining acceptable search time and energy consumption. For example, for 64-bit key word, the search time is approximate 75ns and the energy consumption is approximate 2fJ/bit/search for MemCAM/MemTCAM supporting only point query, and the search time is approximate 140ns and the energy consumption is approximate 5fJ/bit/search for MemCAM/MemTCAM supporting both point and range queries. At higher temperatures, we can obtain shorter search time and can further reduce power density by reducing applied voltage or switching time with emerging technologies to reduce the threshold voltage and RC delay.

We can see that the search time and energy consumption of MemCAM/MemTCAM are worse than CMOS-based CAM/TCAM. However, the performance and energy gaps can be reduced and probably be eliminated with higher capacity. When the data of an application are too large to fit into CMOS-based CAM/TCAM, we may have to partition the data and load one partition into CMOS-based CAM/TCAM each time and perform the search. The energy and time to load data into CAM/TCAM can be saved when MemCAM/MemTCAM is used. MemCAM/MemTCAM also does not have static power consumption because it is
nonvolatile. As a result, although the energy consumption and search time of MemCAM/MemTCAM are both higher than CMOS-based CAM/TCAM, we may still save total energy consumed and reduce execution time for applications whose data are too large to fit into CMOS-based CAM/TCAM.

In a word, the evaluation and discussion above show that MemCAM/MemTCAM has higher capacity than CMOS-based CAM/TCAM and has a strong potential to reduce total search time and energy consumption for applications with a huge amount of data. However, memristors have much lower endurance compared with SRAM/DRAM, which limits the benefits we can obtain from MemCAM/MemTCAM. We identify the endurance problem and explore the design space of memristor-based storage structures for solutions from the next chapter.
4. Conventional Lifetime Improvement Methods

MemCAM/MemTCAM is a promising candidate to replace CMOS-based CAM/TCAM because of capacity improvement, search time reduction and energy saving. However, a major disadvantage of MemCAM/MemTCAM is that memristors have much lower endurance (at most $10^{10}$ write cycles so far [8]) than DRAM/SRAM ($10^{16}$ write cycles [9]). The comparison and match signal combination processes described in Chapter 3 both require writes into the memristors. As a result, the lifetime of MemCAM/MemTCAM is only a couple of minutes when we have continuous searches, and we have to solve the endurance problem in order to benefit from the advantages of memristors.

Besides memristor-based storage structures, there are other memory storages with endurance problems, such as Flash memory ($10^5$ writes [9]) and Phase-Change Memory (PCM) ($10^9$ writes [23]). Several technologies have been proposed to improve lifetime of Flash memory and PCM [23-26]. The technologies can be divided into two main categories: wear-leveling [24, 25] and write avoidance [23, 26]. In order to solve the endurance problem of memristor-based storage structures, we first exploit these two categories of conventional lifetime improvement approaches, wear-leveling and write avoidance.
4.1 Wear-leveling

Wear-leveling techniques make writes uniform to improve lifetime of storage systems built using emerging memory technologies such as Flash memory and PCM. Without wear-leveling, Flash memory and PCM can fail very fast [25] due to the wear-out of a few frequently written lines. We can avoid this case by remapping: writes to lines that are written more frequently are remapped to lines that are written less frequently.

Most wear-leveling mechanisms are implemented in hardware. For example, we can have a lookup table implemented in a memory controller [24] placed between OS and the memory system. The lookup table is used to map logical block addresses to physical block addresses. We can then change the map and move blocks around in memory in order to evenly distribute writes. We can invalidate a physical block and allocate a new physical block and link it with the logical block when OS requests to write a block back into memory; or we can track write counts and perform the invalidation and allocation only when the number of writes to a certain physical block reaches a predefined threshold; or we can periodically swap two or two groups of blocks.

Table-based wear-leveling techniques can achieve almost 100% of the theoretical maximum lifetime. However, access latency is increased because of mapping table lookup and extra storage space is required to store the tables. Memristor-based memory
can have very large capacity, which leads to large storage space overhead. As a result, table-based wear-leveling techniques are not suitable for memristor-based memory systems.

People from IBM have proposed Start-Gap, a hardware-implemented line-level wear-leveling technique without table lookup overhead and requiring only two registers and an extra line for each region within which writes are evenly distributed [25]. Start-Gap can obtain average lifetime of more than 97% of the ideal algorithm. The properties of Start-Gap, including significant lifetime improvement, small storage overhead, fine granularity, and the idea of partitioning the memory into multiple regions and applying wear-leveling onto each region, makes it a good candidate technique to apply to memristor-based systems to solve the endurance problem.

Unfortunately the lifetime of MemCAM/MemTCAM cannot be improved by just wear-leveling techniques because all the cells are accessed simultaneously every cycle. All MemCAM/MemTCAM cells have the same write frequency and will wear out at almost the same time. As a result, we turn to another conventional lifetime improvement approach, write avoidance, to see whether it can efficiently improve lifetime of MemCAM/MemTCAM.

4.2 Write Avoidance

Besides evenly distributing writes, we can also reduce the average write frequency to increase the overall lifetime of memory systems. Typical write avoidance
approaches include new cache replacement policies such as preferring clean entries when choosing a victim [23], methods that only writes modified bits back into memory [23], and new software designs for existing logical data structures and algorithms to reduce the number of writes back to memory [26]. Some of the write avoidance mechanisms are hardware implemented (e.g., the new cache replacement policy) and some are software implemented (e.g., new software designs). The software-implemented mechanisms are transparent to user level application programmers but some are not transparent to system level application programmers (e.g., new software designs). We reduce the average write frequency to MemCAM/MemTCAM through hybrid designs in this section and estimate the efficiency and overhead of these write avoidance approaches.

4.2.1 Hybrid Memristor-based CAM/TCAM-memory

We can reduce the average write frequency by utilizing the configurability of a memristor array. One advantage of memristor-based storage structures is that a memristor array can function as CAM/TCAM or memory based on the voltages applied on to the memristors. We can divide a memristor array into multiple partitions with each partition having the same capacity and configure one partition as CAM/TCAM and other partitions as memory. We can then ‘rotate’ the CAM/TCAM partition within the memristor array as shown in Figure 22.
The improvement of lifetime by using the hybrid memristor-based CAM/TCAM-memory design is approximately proportional to the number of partitions. However, this design requires a large memristor array to obtain acceptable lifetime of a small MemCAM/MemTCAM. For example, if we have continuous searches, to achieve a one-month lifetime for 1MB of MemCAM/MemTCAM requires a 36GB memristor array even if there are no writes to the memory partitions. With the improvement of memristor endurance in the future, this design may become more efficient, but currently the high storage overhead makes it not practical.

4.2.2 Hybrid CMOS-memristor-based CAM/TCAM

We can also reduce write frequency by designing a hierarchical storage structure. We can use a CMOS-based CAM/TCAM as a buffer of MemCAM/MemTCAM as shown in Figure 23. We store hot data (data searched more frequently) in CMOS CAM/TCAM buffer and store cold data in MemCAM/MemTCAM. The search frequency of MemCAM/MemTCAM is reduced and so is the write frequency.

The improvement of lifetime by using the hybrid CMOS-memristor based CAM/TCAM design is dependent on the capacities of both CAMs/TCAMs and the access frequencies of both hot and cold data. Hot data has to be accessed $4 \times 10^5$ times
more frequently than cold data in order to achieve a one-year lifetime, which is not true for a lot of applications and limits the application area of this design.

![Figure 23: Hybrid CMOS-memristor based CAM/TCAM](image)

### 4.3 Discussion

MemCAM/MemTCAM has high write frequency because every cell performs computation every cycle. High write frequency leads to short lifetime and the already evenly distributed writes makes it impossible to apply conventional wear-leveling techniques to improve lifetime. Thus we exploit two write-avoidance designs in this chapter to reduce the average write frequency of MemCAM/MemTCAM. The hybrid memristor-based CAM/TCAM-memory design limits the access to only one partition of a memristor array at a time, and the hybrid CMOS-memristor-based CAM/TCAM design reduces the access frequency to the memristor array. However, neither of the two designs can solve the endurance problem efficiently.

Although the two write-avoidance designs we propose in this chapter are both inefficient to improve lifetime of MemCAM/MemTCAM, they provide a starting point:
if we can achieve a lower write frequency for each memristor (such as the hybrid CMOS-
memristor-based CAM/TCAM), we can certainly improve lifetime; even if we can only
achieve a lower write frequency for some memristor (such as the hybrid memristor-
based CAM/TCAM-memory), we make it possible to apply wear-leveling techniques
since the writes become nonuniform.

Each of the two designs is not efficient but the combination of them may be
better. Also, based on Chapter 3 we can see that reducing average computation per cycle
can reduce write frequency. Thus a hybrid storage structure as a combination of the two
design, which is between memory and CAM/TCAM (with more computation than
memory and less computation than CAM/TCAM), can probably both utilize the
computation ability of memristors and increase lifetime. Thus we propose to combine
the two designs to design hybrid memristor-based storage structures and describe the
structures in the next chapter.
5. Hybrid Memristor-based Storage Structures

5.1 Overview

From the previous chapter we can see that partitioning a memristor-based storage structure (hybrid memristor-based CAM/TCAM-memory) or adding a CMOS-based buffer (hybrid CMOS-memristor-based CAM/TCAM) alone cannot efficiently reduce write frequency. Thus we combine the two methods and propose a series of configurable hybrid storage structures to utilize the computation ability of memristors and ensure that wear-leveling techniques can be used to improve lifetime.

We start with a logical tree structure and divide it into two parts, the upper levels (the levels near the root) and the lower levels (the levels near the leaves). We then implement the two parts with different data structures and technologies. The upper levels can be implemented as a hash table or a T-tree and are stored in a CMOS-based storage structure (e.g., cache), and the lower levels can be implemented as a CAM or a B’-tree and are stored in a memristor-based storage structure. The main idea is to direct search through the upper-level implementation so only one part of the memristor-based storage structure is accessed per search.

We decide the implementations of the two parts of the logical tree based on whether we can efficiently generate a hash function which is both uniform and order-preserving. A hash function is uniform if it maps the expected input as evenly as possible over its output range, and a hash function $F$ is order-preserving if for inputs $k_1$ and $k_2$...
and \( k_2, k_1 < k_2 \) implies \( F(k_1) < F(k_2) \). The properties of hash functions, together with the implementations, decide the functionality of the storage structure – whether it can only support point query, or both point query and range query.

When we can efficiently generate a hash function which is both uniform and order-preserving, we implement the upper levels as a hash table and the lower levels as a CAM (Hash-CAM). When we can efficiently generate a hash function which is only uniform but not order-preserving, we can still implement the logical tree as Hash–CAM but can only perform point query, which means that the comparison process can only decide that whether an entry is equal to the input key or not. If we also want to perform range query, in which we want to know whether an entry is greater than, or less than, or equal to the input key, we have to implement the upper levels as a data structure with sorted data instead of a hash table. We choose to implement the upper levels as a T-tree in this case (T-tree-CAM) because a T-tree is a balanced binary search tree with the advantages of both an AVL tree and a B\(^{+}\)-tree. Furthermore, based on T-tree-CAM, we propose TB\(^{+}\)-tree and TB\(^{+}\)-tree-CAM to provide more configurability so we can further improve lifetime and trade between lifetime and performance.

We assume a memory-mapped interface for our hybrid storage structures. We use regular memory instructions to address the memristor-based storage structures and any general purpose register can send or receive data to or from the memory-based storage structures. The memory-mapped interface reduces hardware complexity
compared with port-mapped interface and is practical because the memory address space is large enough with 32-bit and 64-bit processors.

In the remainder of this chapter, we propose four hybrid memristor-based storage structures: Hash-CAM, T-tree-CAM, TB+-tree and TB+-tree-CAM. We show how data are stored in each hybrid storage structure and how searches are performed using an example of a table containing 40 entries (40 integers from 0 to 39). We also provide solutions to the partition overflow problem for each hybrid storage structure and discuss configurability to trade between lifetime and performance.

5.2 Hash-CAM

![Figure 24: Hash-CAM](image)

A Hash-CAM is a hybrid hash table and CAM storage structure used to implement a logical tree. The hash table is used to implement the $i$th level of the tree with one node stored in one hash table entry. The CAM is divided into multiple partitions and one partition is linked with one hash table entry as shown in Figure 24.
The hash table is used to store keys to direct search into one part of the CAM and the CAM is used to store all the records.

For point query, the input key goes through the hash function and the search is directed to one CAM partition. The corresponding CAM partition is searched with the process described in Chapter 3 and the matched results are read out based on entry match signals. For range query, the two input bound keys go through the hash function and the search is directed to two CAM partitions (bound CAM partitions). The two CAM partitions perform comparisons and output records with keys within the given range and any records in the partitions between the two bound CAM partitions.

From the search process we can see that at most two CAM partitions perform computations per search. As a result, the improvement of lifetime is proportional to the number of CAM partitions (which is also the number of hash table entries) when wear-leveling techniques are applied. For example, we need approximately a million partitions in order to achieve a 3-year lifetime for MemCAM supporting both point and range queries when we have non-stop searches with 64-bit key words. Since the required number of partitions is large, applications with huge number of entries in a table, such as DNA sequencing with hundreds of millions of sequence reads [27], can obtain more lifetime improvement with Hash-CAM. We can also obtain significant lifetime improvement with Hash-CAM for applications whose table size is not so large if we can configure a part of the memristor array as memory and ‘rotate’ the CAM part.
Certainly it requires storage overhead similar to the hybrid memristor-based CAM/TCAM-memory design in Chapter 4, but the overhead is much lower than the previous design and we can store the actual data related to keys stored in MemCAM/MemTCAM. For example, to achieve a one-month lifetime for 1MB of MemCAM/MemTCAM requires a 36MB memristor array instead of 36GB when we have 1,000 partitions, which is practical for most large data warehouses with tables containing hundreds of thousands of entries [28]. The number of partitions required will be even smaller if we also consider the time required to access other storage structures (e.g., DRAM, memristor-based memory, or even disks) to read out the actual data related to the matched entries because the total time for every search is the sum of search time (to search for the matches) and data transferring time (to transfer the matches) instead of only search time.

![Diagram of Hash-CAM Example](image)

**Figure 25: Hash-CAM Example**

Figure 25 shows how a table containing 40 integers (from 0 to 39) is stored in Hash-CAM. The hash function we use here is $F(k) = (k/10) \mod 4$ where $k$ is the input key. There are four entries in the CMOS-based hash table and four partitions in the memristor-based CAM. Each hash table entry has a pointer pointing to a CAM partition. When we receive an input key $k=23$, we use the hash function to calculate the index
(which is 2 in this case) of the pointer pointing to the CAM partition with matches and we then search the corresponding CAM partition. When we receive a given range (15, 32), we first search for 15 which ends up in the second CAM partition. We then search for 32 which ends up in the fourth CAM partition. Finally we read every entry greater than 15 in the second CAM partition, every entry less than 32 in the fourth partition, and all the entries in the CAM partitions between the second and the fourth, which is the third in this case.

![Figure 26: Hash-CAM Example with a Non-order-preserving Hash Function](image)

If we can only efficiently generate hash functions that are only uniform but not order-preserving, Hash-CAM can only support point query but not range query because records within a range may be distributed among all CAM partitions. For example, if the hash function we can generate efficiently for the example in Figure 25 is $F'(k) = k \mod 4$, the table is stored in the memristor-based CAM as shown in Figure 26. We can see that the entries are still uniformly distributed with 10 entries in each CAM partition. However, we cannot support range query because once we find the upper(lower) bound, the entries less(greater) than it can be in any of the CAM partitions. In this case, in order to support range query, we replace the hash table with a T-tree to implement the upper levels of the logical tree.
5.3 T-tree-CAM

A T-tree-CAM is a hybrid T-tree and CAM storage structure used to implement a logical tree and support both point and range queries when we cannot efficiently generate a hash function which is both uniform and order-preserving. A T-tree is a data structure evolving from AVL trees and B-trees and mainly used in main-memory databases [29]. Figure 27 shows a T-tree node (T-node). It has binary search nature similar to an AVL tree, and good update and storage characteristics similar to a B-tree. It also has better packaging density than an AVL tree and requires fewer comparisons than a B’-tree for each node to decide which subtree has the matches.

![Figure 27: A T-tree Node (T-node)](image)

We implement the upper levels of the logical tree with a T-tree to preserve the orders to support range query. The lower levels are implemented with a CAM. The CAM is divided into multiple partitions and one partition is linked with one node as a child in the lowest level of the T-tree as shown in Figure 28. Both point query and range
query in a T-tree-CAM are similar to a Hash-CAM. The only difference is that the input keys go through a T-tree instead of a hash function.

Figure 28: T-tree-CAM

The improvement of lifetime is decided by the number of CAM partitions (which is twice of the number of nodes at the lowest level of the T-tree) when wear-leveling techniques are applied, which also decides the number of upper levels. We need approximately 30,000 partitions in order to achieve a 3-year lifetime when we have non-stop searches. The number of partitions required is smaller than Hash-CAM because the search time is longer than Hash-CAM but still requires a large table size (millions of entries) to achieve large lifetime improvement. Similar to Hash-CAM, we can combine T-tree-CAM with the hybrid memristor-based CAM/TCAM-memory design with much lower storage overhead to achieve significant lifetime improvement for applications whose tables are not too large. Also the number of partitions required will be smaller if we consider the time required to read out the matched entries.

Figure 29 shows how a table containing 40 integers (from 0 to 39) is stored in T-tree-CAM. The order of the T-tree, T, is 4 in the example, which means that there are at
most 4 entries in a T-node. When we receive an input key \( k=23 \), we start from the root of
the T-tree. The input key is greater than the maximum entry, 21, in the root, so the
search is directed to the right child and then into the third CAM partition. When we
receive a given range \((15, 32)\), we first search for 15 which ends up in the second CAM
partition. We then search for 32 which ends up in the right child of the root. Finally we
read every entry greater than 15 in the second CAM partition, traverse the T-tree in-
order (in the order of left child, root, right child) from the second CAM partition (as the
right child of the left child of the root) to the T-node with 32 in it, and read every entry
on the way (including entries in the root, the third CAM partition, and the right child of
the root except entry 32 in this case).

Figure 29: T-tree-CAM Example

T-tree-CAM does not require a uniform and order-preserving hash function to
improve lifetime of memristor-based storage. However, the lifetime improvement is
limited by the capacity of CMOS-based storage. In order to solve this problem, we
propose another hybrid storage structure – a TB^{'}-tree.
5.4 TB⁺-tree

A TB⁺-tree is a combination of T-tree and B⁺-tree. The upper levels of a logical tree are implemented by a T-tree and stored in CMOS-based storage and the lower levels are implemented by a forest of B⁺-trees and stored in memristor-based storage. Each B⁺ tree is linked with a node at the lowest level of the T-tree as a child as shown in Figure 30. For point query, we go through one path in one B⁺-tree to the leaf. For range query, we go through two paths in one or two B⁺-trees to the leaves. Only a part of at most two B⁺-trees, not two complete B⁺-trees, performs computations per search so we can obtain lower average write frequency (thus longer lifetime) compared with T-tree-CAM. We can also achieve more configurability based on changing the order of B⁺-tree.

![Figure 30: TB⁺-tree](image)

We implement lower levels using B⁺-tree instead of T-tree because B⁺-tree is shallower than T-tree, which reduces the average time required to perform search/delete/insert operations. B⁺-tree is not efficient for traditional main-memory databases because binary search is required to search within a sorted node and linear search is required to search within an unsorted node [26], which significantly increases...
search time. However, the intra node search time can be saved by memristor-based storage. We can perform comparisons between the input key and all the keys stored in a B'-tree node simultaneously by utilizing the computation ability of memristors. As a result, we can fully utilize the benefits of unsorted nodes to reduce write frequency.

The lifetime improvement of a TB'-tree is decided by the number of upper levels and the number of lower levels. We can apply B'-tree-level wear-leveling by swapping B'-trees frequently accessed with B'-trees infrequently accessed. We can also apply node-level wear-leveling within a B'-tree by swapping frequently accessed nodes with infrequently accessed nodes since within a B'-tree, only some nodes perform computations per search. As a result, TB'-tree can have longer lifetime than T-tree-CAM. However, TB'-tree also has longer search time than T-tree-CAM because searches have to go through a B'-tree to the leaves.

Figure 31 shows how a table containing 40 integers (from 0 to 39) is stored in TB'-tree. The order of the T-tree, T, is 4 in the example, which means that there are at most 4 entries in a T-node. The order of the B'-tree, B, is 3 in the example, which means that each node in the B'-tree has at least two children and at most three children. When we receive an input key k=23, we start from the root of the T-tree. The input key is greater than the maximum entry, 21, in the root, so the search is directed to the right child and then into the third B'-tree. We then start from the root of the third B'-tree and go down to the leaf level to get the match. When we receive a given range (15, 32), we first search
for 15 which ends up in the second B*-tree. We then search for 32 which ends up in the right child of the T-tree root. Finally we read every entry greater than 15 in the B*-tree node containing 15 (none in this case), traverse the B*-tree in-order (in the order of left child, root, right child) from the B*-tree node containing 15 to the T-node with 32 in it, and read every entry in a B*-tree leaf node or a T-node on the way.

![Figure 31: TB*-tree Example](image)

Both T-tree-CAM and TB*-tree have advantages and disadvantages. T-tree-CAM has shorter search time but limited lifetime. TB*-tree have longer lifetime but also longer search time. In order to balance performance and lifetime, we propose another configurable hybrid storage structure, a TB*-tree-CAM in the next subsection.

### 5.5 TB*-tree-CAM

A TB*-tree-CAM is a combination of T-tree-CAM and TB*-tree as shown in Figure 32. In TB*-tree-CAM, we group leaf nodes of one subtree in one B*-tree and align them physically so we can perform CAM search operations described in Chapter 3. The root of one subtree can be any node in a B+-tree. If the root of a subtree is the root of the B*-tree, TB*-tree-CAM becomes T-tree-CAM. If the root of a subtree is one leaf node, TB*-
tree-CAM become TB\(^{-}\)-tree. If the root of a subtree is an internal node, TB\(^{-}\)-tree-CAM becomes a storage structure between T-tree-CAM and TB\(^{-}\)-tree with moderate search time and lifetime.

\[ \text{Figure 32: TB}\(^{-}\)-tree-CAM} \]

We use an example in Figure 33 to show the cases when we use different types of nodes in a B\(^{-}\)-tree as the root of the subtree. The order of the T-tree, T, is 4 in the example, which means that there are at most 4 entries in a T-node. The order of the B\(^{-}\)-tree, B, is 3 in the example, which means that each node in the B\(^{-}\)-tree has at least two children and at most three children. Squares in gray at the leaf level represent entries in a leaf node which is a right child of its parent and squares in white at the leaf level represent entries in a leaf node which is a left child of its parent. Squares grouped together at the leaf level represent entries in the same CAM partition. Links in black in the B\(^{-}\)-trees represent the links we follow when performing search operations.
are four roots as the subtree roots and store the leaves of each subtree in a CAM partition. There are four B'-tree so there are four CAM partitions. Each CAM partition is linked with a T-

Figure 33: TB'-tree-CAM Example

Figure 33(a) shows the structure of the TB'-tree-CAM when we use the B'-tree roots as the subtree roots and store the leaves of each subtree in a CAM partition. There are four B'-tree so there are four CAM partitions. Each CAM partition is linked with a T-
node at the last level of the T-tree directly and the structure is the same as a T-tree-CAM.

When we receive an input key $k=23$, we start from the root of the T-tree. The input key is greater than the maximum entry, 21, in the root, so the search is directed to the right child and then into the third $B^\prime$-tree. Then, instead of starting from the root of the third $B^\prime$-tree and go down to the leaf level to get the match, we directly search the third CAM partition linked with the root of the third $B^\prime$-tree, which is the same as we do with T-tree-CAM. The range search process is also the same as T-tree-CAM.

Figure 33(b) shows the structure of the TB$^\prime$-tree-CAM when we use the $B^\prime$-tree leaves as the subtree roots. There is only one node in each subtree so we actually store all the entries in each leaf in a CAM partition. There are sixteen leaves so there are sixteen CAM partitions. Each CAM partition is linked with a $B^\prime$-tree node at the second last level of a $B^\prime$-tree. The structure is actually the same as a TB$^\prime$-tree and the processes for both point and range queries are the same as a TB$^\prime$-tree.

Figure 33(c) shows the structure of the TB$^\prime$-tree-CAM when we use the nodes at the second level of the $B^\prime$-trees as the subtree roots and store the leaves of each subtree in a CAM partition. There are eight such internal nodes so there are eight CAM partitions. Each CAM partition is linked with a $B^\prime$-tree root as a child. When we receive an input key $k=23$, we start from the root of the T-tree. The input key is greater than the maximum entry, 21, in the root, so the search is directed to the right child and then into the third $B^\prime$-tree. We then start from the root of the third $B^\prime$-tree and then, instead of
going down to the leaf level to get the match, we go directly into the fifth CAM partition, which is linked to the root of the third B+-tree as the left child and search it for the matches. When we receive a given range (15, 32), we first search for 15 which ends up in the fourth CAM partition. We then search for 32 which ends up in the right child of the T-tree root. Finally we read every entry greater than 15 in the fourth CAM partition containing 15, traverse the B+-tree in-order (in the order of left child, root, right child) from the fourth CAM partition (which can be considered as the right child of the root of the second B+-tree) containing 15 to the T-node with 32 in it, and read every entry in a CAM partition or a T-node on the way.

From Figure 33 we can see that the number of subtrees increases when we use B+-tree nodes at lower levels as the roots of the subtrees. The number of CAM partition increases and leads to larger lifetime improvement as the number of subtrees increases. However, we have to go through more levels in the B+-trees during searches to have more subtrees, which increases search time. In a word, longer search time leads to longer lifetime and we can trade between performance and lifetime by using B+-tree nodes at different levels as the roots of subtrees in TB+-tree-CAM.

5.6 Solutions to Partition Overflow

We propose hybrid memristor-based storage structures to improve lifetime while we are still able to utilize the computation ability of memristors to accelerate search for applications with large data sizes. The main idea is to divide the memristor-based
storage structure into multiple partitions and direct search through the CMOS-based storage structure into one or two partitions so only one part of the memristor-based storage structure is accessed per search. The lifetime improvement depends on that we can distribute data uniformly among partitions of the memristor-based storage structures. As a result, we have to solve a problem called as partition overflow, which means that a CAM partition is full and we cannot store more records into it. Partition overflow is not a problem for databases with few or no updates. However, for databases with heavy updates, if we consider the complete CAM to be full when there is a partition overflow, it can cause a huge waste of space because there can still be non-full or even empty partitions.

We describe how to solve the partition overflow problem for each of the four hybrid memristor-based storage structures in this section. The main idea of our solution is to allocate a new partition and move a part of the data in the overflowed partition into the new partition. We know the capacity of a partition when we configure the memristor-based storage structure (given the capacity of the memristor-based storage structure and the number of partitions) and we can calculate the maximum number of records that fit into a partition based on the data size of a record. We maintain the number of records actually stored in each partition and check whether a partition is full when there is an insert operation.
5.6.1 Hash-CAM

Hash-CAM does not have to deal with partition overflow because the hash function is uniform and the utilization rates of all the partitions are similar. When a partition overflow happens, other partitions are either full or near-full so we can consider the complete CAM to be full without causing too much waste of space. However, for the other three storage structures we have to deal with the partition overflow problem.

5.6.2 T-tree-CAM

Besides a maximum record counter, we also maintain a record counter for each CAM partition in T-tree-CAM. The record counters are modified every time we insert/delete a record into/from the corresponding CAM partitions and are stored in linked T-nodes as shown in Figure 34. When we need to insert a record into a partition whose record counter already reaches the maximum, we allocate a new partition, split the overflowed partition into two and use the maximum of the left half and the minimum of the right half to form a new T-node.

Figure 35 shows an example explaining how to deal with partition overflow in T-tree-CAM. The order of the T-tree, $T$, is 4 in the example, which means that there are at most 4 entries in a T-node. Each CAM partition can store at most 10 entries. Figure 35(a) shows a T-node with two children pointers pointing to two CAM partitions and the gray CAM partition is full. When we try to insert 33 into the gray CAM partition, we first
allocate a new CAM partition. If the allocation fails, we consider the complete CAM to be full. If the allocation succeeds, we sort the records in the gray partition and move the half of them into the new partition. We then use the maximum of the left half and the minimum of the right half to form a new T-node. If there is no space in the CMOS-based storage structure to store the new T-node, we also consider the complete CAM to be full. Otherwise, the original T-tree re-balancing algorithm can be used if the T-tree becomes unbalanced after the new T-node is added. The T-tree-CAM after we allocate the new partition and split the data is shown in Figure 35(b). We then insert 33 into the T-tree-CAM as shown in Figure 35(c).

![Structure of Modified T-node](image)

**Figure 34: Structure of Modified T-node**

The problem of the proposed solution is that it causes large time overhead especially when the capacity of a CAM partition is high because we need to sort all the records in the overflowed CAM partition. As a result, only applications with few or no updates can benefit from T-tree-CAM.
5.6.3 TB⁺-tree

There are no CAM partitions in TB⁺-tree. However, in order to reduce search time, every B⁺-tree should have similar depth, which means that entries should be distributed uniformly among B⁺-trees. Thus each B⁺-tree is similar to a CAM partition and we have to solve the partition overflow problem for TB⁺-tree. The basic idea of the
solution is similar to T-tree-CAM. We maintain a record count for each B*-tree, allocate a new partition and move a part of the data in the overflowed partition into the new partition, and add a T-node formed by the maximum of the left half and the minimum of the right half. The main difference is that we do not need to sort all the records in the overflowed partition since B*-tree is an ordered structure.

We modify the structure of B*-tree roots so we can deal with the partition overflow problem more efficiently. Figure 36 shows the structure of a modified B*-tree root. For each B*-tree root with n keys, there are n+1 subtrees. We maintain a record counter for the left n subtrees, the record counter of the right most subtree can be calculated by subtracting the left n record counter from the record counter of the partition. We also keep the B*-tree roots sorted to make partition splitting more efficient.

Figure 37 shows an example of TB*-tree partition overflow solution. When there is a partition overflow, we first allocate a new partition. If allocation fails, we consider the complete memristor-based storage structure to be full. If allocation succeeds, we split the B*-tree root at Keyi where \( \sum_{j=1}^{i} RC_j - RC_{\text{partition}} \) is minimum. We use Keyi and Keyi+1 to form a new T-node similar to the T-tree-CAM solution. If there is no space in the CMOS-based storage structure to store the new T-node, we also consider the complete CAM to be full. We then use the maximum in Subtreei to replace Keyi, and the minimum in Subtreei+1 to replace Keyi+1. A special case is when there is only one key in a B*-tree root.
In this case, we delete the root and use its two children as the new roots for the overflowed partition and the new partition.

![Diagram of Modified B*-tree Root](image)

**Figure 36: Structure of Modified B*-tree Root**

(a)

![Diagram of TB*-tree Partition Overflow Solution Example](image)

(b)

**Figure 37: A TB*-tree Partition Overflow Solution Example**

The time overhead of TB*-tree partition overflow solution is smaller than the T-tree-CAM solution. Since we do not need to sort all the records in an overflowed
partition, the time required is $O(n)$, where $n$ is the maximum number of records in a partition, while for T-tree-CAM partition overflow solution, the time overhead is $O(n \log n)$.

### 5.6.4 TB$^+$-tree-CAM

The lower levels in TB$^+$-tree-CAM are stored in the same way as in TB$^+$-tree but only searched in different ways. As a result, the solution to the partition overflow problem is the same for both TB$^+$-tree and TB$^+$-tree-CAM.

### 5.7 Discussion

We propose four hybrid storage structures in this chapter. All the designs are based on a logical tree divided into two parts, the upper levels and the lower levels. The main idea is to partition the lower levels and for every search/insert/delete operations, direct access to one or two of the partitions through the upper levels. Since at most two partitions are accessed per operation, the write frequency is reduced for the same number of operations, which leads to the improvement of lifetime. The improvement of lifetime is proportional to the number of partitions. We can decrease the number of partitions by decreasing the number of upper levels (an extreme case is MemCAM, in which the lower levels are implemented with a CAM and the number of partitions is 1) or increase the number of partitions by increasing the number of upper levels. Users/Designers can choose different numbers of partitions to trade between
performance and lifetime based on the requirements of different applications or when the endurance of memristors is improved by future research.

Table 23: Pros and Cons of Four Hybrid Storage Structures

<table>
<thead>
<tr>
<th>Storage Structures</th>
<th>Search Time</th>
<th>Lifetime</th>
<th>Require hash function?</th>
<th>Partition overflow problem exists?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash-CAM</td>
<td>Shortest</td>
<td>Limited</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>T-tree-CAM</td>
<td>Short</td>
<td>Limited</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TB⁻-tree</td>
<td>Longest</td>
<td>Longest</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TB⁻-tree-CAM</td>
<td>Tunable, between T-tree-CAM and TB⁻-tree</td>
<td>Tunable, between T-tree-CAM and TB⁻-tree</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 23 summaries the pros and cons of the four hybrid storage structures. We can see that Hash-CAM has the shortest search time. However, it requires a uniform and order-preserving hash function, which is not always possible to be generated efficiently. Also, there have to be a huge number of entries in a table for Hash-CAM to achieve significant lifetime improvement. As a result, only applications with huge tables can benefit the most from Hash-CAM. T-tree-CAM eliminates the hash function requirement but its lifetime improvement is still limited. TB⁻-tree does not have hash function requirement or lifetime improvement limitation but the search time is too long. TB⁻-tree-CAM combines the advantages of T-tree-CAM and TB⁻-tree and provide the configurability to trade between search time and lifetime. T-tree-CAM, TB⁻-tree and TB⁻-tree-CAM all have partition overflow problem which is caused by updates. Due to the
time overhead of the solutions to the partition overflow problem, only applications with few or new updates can achieve the most benefits from these three storage structures.

In the next chapter we evaluate the four hybrid storage structures proposed in this chapter and compare the results with two software-implemented memory-based T-trees without in-memory computations.
6. Evaluation

We propose four hybrid memristor-based storage structures, Hash-CAM, T-tree-CAM, TB+tree and TB+-tree-CAM to improve lifetime of memristor-based storage structures in the previous chapter. In this chapter, we evaluate these four hybrid storage structures and compare the results with two conventional storage structures: a software-implemented CMOS-based T-tree and a software-implemented memristor-based T-tree.

One major advantage of MemCAM/MemTCAM and hybrid storage structures is higher density which leads to higher capacity with the same area compared with CMOS-based CAM/TCAM and DRAM respectively. With the assumption of the same area, the capacity of MemCAM/MemTCAM and hybrid storage structures can be at the terabyte level based on the densities of DRAM and memristor arrays, and the capacity of DRAM in state-of-the-art servers. Such a large data size precludes evaluations with conventional cycle-accurate simulators so we turn to analytical models for evaluations.

We develop an analytical model to evaluate and compare the average record search time of six storage structures, a CMOS-based T-tree, a memristor-based T-tree, a hybrid Hash-CAM, a hybrid T-tree-CAM, a hybrid TB+-tree and a hybrid TB+-tree-CAM. All the six storage structures have the upper levels stored in a CMOS-based cache. The CMOS-based T-tree has the lower levels stored in DRAM. The memristor-based T-tree has the lower levels stored in a memristor-based memory. The four hybrid storage
structures have the lower levels stored in a memristor array with combined compute and storage.

The results and analysis show that TB'-tree-CAM has the best balance between performance and lifetime among the four hybrid memristor-based storage structures. The maximum theoretical lifetime of TB'-tree-CAM is more than 60 years, which is much longer than the lifetime of Hash-CAM (shorter than 6 years) and T-tree-CAM (around a year). Although TB'-tree-CAM has a shorter lifetime than TB'-tree (longer than 80 years), it has much higher performance (the search time of TB'-tree-CAM is only half of the search time of TB'-tree). As long as the data size is large (larger than the capacity of DRAM) and the memristor array read/write latency is moderate (longer than 40ns), TB'-tree-CAM is the best storage structure to choose.

In the first subsection of this chapter we describe the analytical model, including different parameters and their meanings, equations to calculate the average record search time for all six storage structures (two software-implemented T-trees and four hybrid hardware-implemented storage structures). We then present the results and analysis in the second subsection, and discuss and summarize the results in the last subsection.

6.1 Analytical Model

Table 24 shows the parameters we use to develop our analytical model. We define record nodes as nodes containing pointers to the records. For the T-tree, all the
nodes are record nodes. For the TB\(^{-}\)-tree, all the nodes at upper levels and all the leaves are record nodes. Given the number of records \(N_R\), the order of the T-tree \(T\) (which means that there are at most \(T\) records in a T-node), the order of the B\(^{-}\)-tree \(B\) (which means that each internal node in the B\(^{-}\)-tree has at least \(\lceil B/2 \rceil\) children and at most \(B\) children), and the number of upper levels \(Level_U\), with the assumption of 100% node utilization (which means that every node has the most number of records) the total number of record nodes in the T-tree and the TB\(^{-}\)-tree can be calculated by the following equations:

Equation 6: \(N_T = \frac{N_R}{T}\)

Equation 7: \(N_{TB} = 2^{Level_U} - 1 + \frac{N_R - (2^{Level_U} - 1) \times T}{B}\)

The total number of record nodes in the T-tree \((N_T)\) is calculated by dividing the number of record \((N_R)\) by the order of the T-tree \((T)\). The total number of record nodes in the TB\(^{-}\)-tree \((N_{TB})\) is the sum of the number of record nodes at the upper levels and the lower levels, and the number of record nodes at the lower levels is calculated by dividing the number of record nodes at the lower levels by the order of the B\(^{-}\)-tree \((B)\).

We can then calculate the number of lower levels in the T-tree \((Level_LT)\) and the TB\(^{-}\)-tree \((Level_{LTB})\). The number of lower levels in the T-tree is the difference between the total number of levels in the T-tree and the given number of upper levels. We assume that all the B\(^{-}\)-trees in the TB\(^{-}\)-tree have the same depth and the number of lower levels
in the TB'-tree is equal to that depth, which is the logarithm of the number of record
nodes in a B⁺-tree to based B. Equation 8 and Equation 9 are the equations to calculate

\[ \text{Level}_{LT} \] and \[ \text{Level}_{LTB} \] respectively.

**Equation 8:** \[ \text{Level}_{LT} = \left\lfloor \log_2 \frac{N_R}{T} \right\rfloor - \text{Level}_U \]

**Equation 9:** \[ \text{Level}_{LTB} = \left\lfloor \log_B \left( \frac{N_R - \left(2^{\text{Level}_U} - 1\right) \times T}{2^{\text{Level}_U}} \right) \right\rfloor \]

### Table 24: Parameters in the Analytical Model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{AvgTime}_T )</td>
<td>Average record search time of the T-tree</td>
</tr>
<tr>
<td>( \text{AvgTime}_{TB} )</td>
<td>Average record search time of the TB'-tree</td>
</tr>
<tr>
<td>( \text{Level}_U )</td>
<td>Number of upper levels</td>
</tr>
<tr>
<td>( \text{Level}_{LT} )</td>
<td>Number of lower levels in the T-tree</td>
</tr>
<tr>
<td>( \text{Level}_{LTB} )</td>
<td>Number of lower levels in the TB'-tree</td>
</tr>
<tr>
<td>( \text{Level}_{LTB}' )</td>
<td>Number of lower levels in the TB'-tree-CAM</td>
</tr>
<tr>
<td>( \text{NodeTime}_U )</td>
<td>Time to search a node at upper levels</td>
</tr>
<tr>
<td>( \text{NodeTime}_{LT} )</td>
<td>Time to search a node at lower levels in the T-tree</td>
</tr>
<tr>
<td>( \text{NodeTime}_{LTB} )</td>
<td>Time to search a node at lower levels in the TB'-tree</td>
</tr>
<tr>
<td>( N_T )</td>
<td>Total number of nodes storing records in the T-tree</td>
</tr>
<tr>
<td>( N_{TB} )</td>
<td>Total number of nodes storing records in the TB'-tree</td>
</tr>
<tr>
<td>( N_R )</td>
<td>Total number of records</td>
</tr>
<tr>
<td>( T )</td>
<td>Order of the T-tree</td>
</tr>
<tr>
<td>( B )</td>
<td>Order of the B⁺-tree</td>
</tr>
</tbody>
</table>

When we search for a record, we first search for the record node containing the
record and then search within the record node to find the record. For a record node at an
upper level, we just need to go through some upper levels from the root to reach it. For a
record node at a lower level, we have to go through all the upper levels first and then
search it in the lower levels. The search process in the lower levels varies based on different implementations: for CMOS-based T-tree and memristor-based T-tree we search the corresponding subtree with the search process similar to the search process in the upper levels; for Hash-CAM and T-tree-CAM, we search the corresponding CAM partition with the comparison and match signal combination described in Chapter 3; for TB'-tree, we go through all the B'-tree levels to the leaf level; for TB'-tree-CAM, we go through a per-defined number of B'-tree levels and search all the leaves in the corresponding subtree with the comparison and match signal combination described in Chapter 3.

The time required to reach a node at the $i^{th}$ upper level is $i \times \text{NodeTime}_U$ where $i \in Z$ and $i \in [1, \text{Level}_U]$. The time required to reach a node at the $i^{th}$ lower level in the T-tree is $\text{Level}_U \times \text{NodeTime}_U + i \times \text{NodeTime}_{LT}$ where $i \in Z$ and $i \in [1, \text{Level}_{LT}]$. The time required to reach a node at the $i^{th}$ lower level in the TB'-tree is

$$\text{Level}_U \times \text{NodeTime}_U + \text{Level}_{LTB} \times \text{NodeTime}_{LTB}$$

We assume a random uniform distribution of keys to search and define the average record search time as the average time required to reach the corresponding record node. We also assume that each record node has the same number of records and calculate the average record search time by dividing the sum of the time required to reach each record node by the number of record nodes.
Equation 10:

\[
\text{AvgTime}_T = \sum_{i=1}^{\text{Level}_U} (\text{NodeTime}_U \times i) \\
+ \sum_{i=1}^{\text{Level}_{LT}} (\text{NodeTime}_U \times \text{Level}_U + \text{NodeTime}_{LT} \times i) \\
+ (\text{NodeTime}_U \times \text{Level}_U + \text{NodeTime}_{LT} \times \text{Level}_{LT}) \times (N_T - 2^{\text{Level}_U + \text{Level}_{LT}-1} + 1) / N_T
\]

Equation 11:

\[
\text{AvgTime}_{TB} = \sum_{i=1}^{\text{Level}_U} (\text{NodeTime}_U \times i) \\
+ \text{NodeTime}_U \times \text{Level}_U \times \left( N_{TB} - 2^{\text{Level}_U} + 1 \right) \\
+ \text{NodeTime}_{LTB} \times \text{Level}_{LTB} \times \left( N_{TB} - 2^{\text{Level}_U} + 1 \right) / N_{TB}
\]

Equation 10 and Equation 11 show how we calculate the average record search time for a T-tree and a TB*-tree respectively. The first item in both equations is the sum of the time required to reach each record node at an upper level. In Equation 10 the second item is the sum of the time required to reach each record node at a lower level except the lowest level, and the third item is the sum of the time required to reach each record node at the lowest level. In Equation 11 the second item is the sum of time required to go through the upper levels to reach each record node at the lowest level (in the B*-tree, all the record nodes are at the lowest level), and the third item is the sum of time required to go through the B*-tree to reach each record node at the lowest level. After calculating the summations we have Equation 12 and Equation 13.
Equation 12:

\[
\text{AvgTime}_T = \{\text{NodeTime}_U \times ([\text{Level}_U - 1] \times 2^{\text{Level}_U} + 1) + \text{NodeTime}_U \times \text{Level}_U \times (N_T - 2^{\text{Level}_U} + 1) + \text{NodeTime}_{LT} \times ([\text{Level}_{LT} - 2] \times 2^{(\text{Level}_U + \text{Level}_{LT} - 1)} + 2^{\text{Level}_U}) + \text{NodeTime}_{LT} \times \text{Level}_{LT} \times \left[N_T - \left(2^{(\text{Level}_U + \text{Level}_{LT} - 1)} - 1\right)\right] / N_T
\]

Equation 13:

\[
\text{AvgTime}_{TB} = \{\text{NodeTime}_U \times ([\text{Level}_U - 1] \times 2^{\text{Level}_U} + 1) + \text{NodeTime}_U \times \text{Level}_U \times (N_{TB} - 2^{\text{Level}_U} + 1) + \text{NodeTime}_{LTB} \times \text{Level}_{LTB} \times (N_{TB} - 2^{\text{Level}_U} + 1)\right) / N_{TB}
\]

Equation 14:

\[
\text{AvgTime}_{HC} = \text{HashTime} + \text{MemCAMSearchTime}
\]

Equation 15:

\[
\text{AvgTime}_{TC} = \text{NodeTime}_U \times \text{Level}_U + \text{MemCAMSearchTime}
\]

Equation 16:

\[
\text{AvgTime}_{TBC} = \text{NodeTime}_U \times \text{Level}_U + \text{NodeTime}_{LTB} \times \text{Level}_{LTB} + \text{MemCAMSearchTime}
\]

We calculate the average record search time of a CMOS-based T-tree and a memristor-based T-tree by changing the value of \(\text{NodeTime}_{LT}\) in Equation 12. We use Equation 13 to calculate the average record search time of a hybrid CMOS-memristor TB*-tree. We also calculate the average record search time of Hash-CAM, T-tree-CAM, and TB*-tree-CAM using Equation 14 to Equation 16 based on MemCAM search time described in Chapter 3. The average record search time of Hash-CAM is the sum of the
time to access the hash table and the time to search a CAM partition. The average record search time of a T-tree-CAM is the sum of the time to go through all upper levels in the T-tree and the time to search a CAM partition. The average record search time of TB*-tree-CAM is the sum of the time to go through all upper levels in the T-tree, the time to go through a predefined number of levels in a B*-tree, and the time to perform CAM-like search operation in a subtree. We do not consider the records in upper levels when we calculate the average record search time for these three hybrid storage structures because most of the records are in lower levels.

6.2 Results and Analysis

We assume that a T-node can fit into one cache block and cache/memory access time dominates the search time for the T-tree. Thus the time required to search a node at an upper level ($NodeTime_U$) is the same as cache access time and the time required to search a node at a lower level in the T-tree ($NodeTime_LT$) is the same as DRAM access time. We use the data from performance analysis of Intel’s latest processors [30] for $NodeTime_U$ (16ns) and $NodeTime_LT$ (60ns). Our evaluation is based MemCAM search operations supporting both point and range queries. It requires one write into the memristor array (to write the key), two reads from the memristor array (one to read the comparison results, and one to read the address of the next node), and a search operation (including comparison and match signal combination) within the memristor array to search a node at a lower level in the TB*-tree. The memristor read/write latency
varies from 10ns [31, 32] to 120ns [1]. We use 8-byte keys and 8-byte pointers and thus the search time is 136ns (16 + 20*log_2 64). As a result, NodeTime_{LM} (3*memristor read/write latency + search time) varies from 166ns to 496ns. We also use the internal B’-tree nodes at the second level in B’-trees as the roots of subtrees in TB’-tree-CAM. So Level_{LTB} is 1 which means that in order to reach a record in a record node at a lower level, we have to go through one level in the B’-tree after we go through all the levels in the T-tree.

Our evaluation is based on 32MB cache [33] and 128GB DRAM [33]. Based on the densities of DRAM (15Gbit/cm^2 [34]) and a memristor array (1Tbit/cm^2 [14]), we assume the same area size and evaluate an 8TB memristor based memory. We need 6 memristors to build a MemCAM cell so we also evaluate a 1TB MemCAM. We choose T=10 (which means that we can store at most 10 records in a T-node) and store 17 levels of T-tree in cache. We choose B=80 (which means that each internal B’-tree node has at least 40 children and at most 80 children) to balance between the depth and node utilization rate of the B’-tree. Table 25 shows the number of records we can store in different storage structures. We can see that the memristor-based T-tree can store the most records since it has the highest capacity. The number of records that can be stored in each hybrid storage structure (ranging from 2.8x10^{10} to 6.9x10^{10}) is between the memristor-based T-tree (3.4x10^{11}) and CMOS-based T-tree (5.4x10^9).
Table 25: Number of Records Stored in Different Storage Structures

<table>
<thead>
<tr>
<th>Storage structure</th>
<th>Number of records</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOS based T-tree</td>
<td>$5.4 \times 10^9$</td>
</tr>
<tr>
<td>Memristor based T-tree</td>
<td>$3.4 \times 10^{11}$</td>
</tr>
<tr>
<td>Hash-CAM</td>
<td>$6.9 \times 10^{10}$</td>
</tr>
<tr>
<td>T-tree-CAM</td>
<td></td>
</tr>
<tr>
<td>TB’-tree</td>
<td>$2.8 \times 10^{10}$</td>
</tr>
<tr>
<td>TB’-tree-CAM</td>
<td></td>
</tr>
</tbody>
</table>

6.2.1 Impact of Memristor Array Read/Write Latency

We first choose the number of records, $N_r$, as $10^9$, which means that all the records can be stored in any of the six storage structures, and compare the performance. Figure 38 shows the average record search time as we increase memristor array read/write latency from 10ns to 120ns. We can see that only Hash-CAM (from 200ns to 300ns) and T-tree-CAM (from 400ns to 600ns) perform better than software-implemented memory-based T-trees. The performance of TB’-tree-CAM (from 600ns to 1150ns) is between the performance of CMOS-based T-tree (800ns) and memristor-based T-tree (from 350ns to 1300ns) and the performance of TB’-tree (from 800ns to 1800ns) is worse than both memory-based T-trees.

The performance of TB’-tree is worse than the T-trees because in order to search a node at a lower level, we need one write into the memristor array and two reads from the memristor array instead of only one read from either DRAM or memristor-based memory. TB’-tree-CAM reduces the performance gap between TB’-tree and T-trees by reducing the number of lower levels we go through. The short search time of T-trees is
also the result of the assumption that a T-node can fit into a cache block so we can read a T-node out of memory (either DRAM or memristor-based memory) with only one read. If the data size of a node is larger than the capacity of a cache block, the average record search time is at least doubled for both T-trees but remains the same for the four hybrid storage structures. In this case, Hash-CAM, T-tree-CAM and TB*-tree-CAM all perform better than both T-trees and the performance gap between TB*-tree and T-trees is also reduced.

Figure 38: Search Time of Six Storage Structures with \( N_R = 10^9 \)

We then choose the number of records, \( N_R \), as \( 10^{10} \), which means that the capacity of DRAM is no longer high enough to store all the records. Figure 39 shows the average record search time as we increase memristor array read/write latency from 10ns to
120ns. We do not show the performance of DRAM in Figure 39 since we have to access disk in this case and the average record search time increases to milliseconds. We can see from Figure 39 that Hash-CAM (from 200ns to 300ns) and T-tree-CAM (from 400ns to 600ns) still perform better than memristor-based T-tree (from 400ns to 1700ns). TB-tree (from 800ns to 1800ns) still performs worse than memristor-based T-tree but the performance gap decreases as memristor array read/write latency increases. TB-tree-CAM (from 600ns to 1150ns) outperforms memristor based-T-tree as long as memristor array read/write latency is longer than 40ns.

![Figure 39: Search Time of Five Storage Structures with $N_{R}=10^{10}$](image)

In general, TB-tree and TB-tree-CAM perform better when the number of records is $10^{10}$ than when the number of records is $10^{9}$. The reason is that when there are
10^9 records, the node utilization rate of B^+-trees is low, which means that we can not benefit from the shallowness of B^+-trees. Again, the short search time of T-trees is the result of the assumption that a node can fit into a cache block so we can read a node out of memory with only one read. As node size increases, all hybrid storage structures perform better than memristor-based T-tree.

6.2.2 Impact of Data Size

![Figure 40: Search Time of Five Storage Structures with Different Number of Records](image)

We then change the number of records from 10^9 to 10^{20} to see how data size affects performance of five storage structures. We do not evaluate the software-implemented CMOS-based T-tree because as soon as the number of records goes beyond 10^{10}, DRAM is no longer large enough to store all the records and the average record
search time increases to millisecond level since we have to access the disk. From Figure 38 and Figure 39 we can see that memristor-based T-tree has better performance when memristor array read/write latency is lower, so we choose memristor array read/write latency to be 10ns to have the best possible performance of memristor-based T-tree.

The results are shown in Figure 40. We can see that the search time of memristor-based T-tree (from 350ns to 700ns) and TB*-tree (from 800ns to 1650ns) increase as the number of records increases. The search time of Hash-CAM (200ns), T-tree-CAM (400ns), and TB*-tree-CAM (600ns) remain almost the same. The reason is that more records with the same number of upper levels lead to more lower levels in memristor-based T-tree and TB*-tree and the search time is increased, while with the same number of upper levels we only have larger CAM partitions in Hash-CAM, T-tree-CAM and TB*-tree-CAM but the time to go through the upper levels and to search a CAM partition remains the same. As a result, the performance gap between memristor-based T-tree and TB*-tree-CAM decreases as the number of records increases and when the number of records goes beyond $10^{16}$, TB*-tree-CAM outperforms memristor-based T-tree.

6.2.3 Impact of Cache Size

We then change the cache size and evaluate the performance of five storage structures for $10^9$ records. When the record size remains the same, we can store more records as cache size increases so the number of upper levels increases. Figure 41 shows
the results when we assume a 60ns memristor array read/write latency and Figure 42 shows the results when we assume a 120ns memristor array read/write latency.

Figure 41: Search Time of Five Storage Structures with $T_{access} = 60$ns and Different Cache Sizes

When the memristor array read/write latency is 60ns, we can see from Figure 41 that TB⁻'tree (from 1500ns to 1600ns) performs worse than T-tree (from 1500ns to 1250ns) as cache size increases. The reason is that the time required to search a node at a lower level is much longer in TB⁻'tree than in T-tree and the number of records is not large enough for the shallowness of B⁻'-trees to overcome the disadvantage of longer node search time. However, Hash-CAM (200ns), T-tree-CAM (from 400ns to 500ns) and TB⁻'tree-CAM (from 800ns to 900ns) all perform better than T-tree as the number of upper levels stored in cache increases from 15 to 20. The reason is that in these three
storage structures, we do not need to go through multiple lower levels in order to reach the record node.

![Graph showing search time of five storage structures with T_access = 120ns and different cache sizes.](image)

**Figure 42: Search Time of Five Storage Structures with T_access = 120ns and Different Cache Sizes**

When the memristor array read/write latency is 120ns, we can see from Figure 42 that all four hybrid storage structures perform better than T-tree when the number of upper levels is smaller than 19. When the number of upper levels is equal to or larger than 19, TB*-tree performs worse than T-tree but all the other three hybrid storage structures still perform better than T-tree. In a word, as cache size increases, the performance of T-tree is improved and the performance of all hybrid storage structures slightly decreases. The reason is that for T-tree, larger cache means more records in the cache which can be accessed much faster than in memory. However, for hybrid storage
structures, larger cache means more upper levels and more time to go through upper levels to reach the memristor-based storage. Since the majority of records are in the memristor-based storage, the average search time is increased. However, the average search time only increases slightly because the cache access latency is short. For example, the time required to go through all the upper levels only increases by 90ns as the number of upper levels increases from 15 to 20.

6.2.4 Impact of B+-tree Order

We then change the order of B+-tree and evaluate the performance of five storage structures for 10^9 records. The order of B+-tree (B) decides the number of children (ranging from \( \lceil B/2 \rceil \) to \( B \)) an internal node can have. The results are shown in Figure 43 for 60ns memristor array read/write latency and Figure 44 for 120ns memristor array read/write latency.

We can see that Hash-CAM (260ns and 400ns), T-tree-CAM (540ns and 700ns) and TB+-tree-CAM (850ns and 1150ns) always perform better than T-tree (1400ns and 2500ns) and their performances remain the same when the order of B+-tree changes. The performance of TB+-tree (1550ns and 2250ns) is worse than T-tree when memristor array read/write latency is 60ns but is better than T-tree when memristor array read/latency is 120ns and the order of B+-tree is larger than 60.
Figure 43: Search Time of Five Storage Structures with $T_{\text{access}} = 60\text{ns}$ and Different B'-tree Orders

Figure 44: Search Time of Five Storage Structures with $T_{\text{access}} = 120\text{ns}$ and Different B'-tree Orders
As the order of B⁺-tree increases, the performance of TB⁺-tree is improved or remains the same. This is because for the same number of records, when we increase the order of B⁺-tree, there can be two effects: either the height of the B⁺-tree is decreased, or the node utilization rate of the B⁺-tree is reduced. If the height of the B⁺-tree is decreased, the performance of TB⁺-tree is improved; if the node utilization rate of the B⁺-tree is decreased, the performance of TB⁺-tree remains the same.

6.2.5 Impact of T-tree Order

Finally, we change the order of T-tree and evaluate the performance of five data structures for 10⁶ records. The order of T-tree (T) decides the number of records in a T-node (at most T). The results are shown in Figure 45 for 60ns memristor array read/write latency and Figure 46 for 120ns memristor array read/write latency.

We can see that Hash-CAM (260ns and 400ns), T-tree-CAM (540ns and 700ns) and TB⁺-tree-CAM (850ns and 1150ns) all perform better than T-tree (from 1400ns to 1200ns, and from 2500ns to 2100ns) and their performances remain the same as the order of T-tree increases. TB⁺-tree (1150ns and 2250ns) only performs better when the memristor array read/write latency is 120ns and the order of T-tree is smaller than 40. As the order of T-tree increases, the performance of T-tree is improved because the height of T-tree decreases. However, the performances of all hybrid storage structures remain almost the same because the number of upper levels does not change and the majority of records are at lower levels stored in memristor-based storage.
Figure 45: Search Time of Five Storage Structures with $T_{\text{access}} = 60\text{ns}$ and Different T-tree Orders

Figure 46: Search Time of Five Storage Structures with $T_{\text{access}} = 120\text{ns}$ and Different T-tree Orders
6.2.6 Lifetime Improvement

Besides evaluating the performances (search time) of the four hybrid CMOS-memristor-based storage structures, we also calculate their theoretical maximum lifetime with the assumption of continuous search operations, which is shown in Table 26. Generally, lifetime increases as search time increases, since increased search time results in reduced write frequency. The exception here is T-tree-CAM, whose lifetime is limited by the capacity of cache. However, we can see that even in the worst case we can achieve a one-year lifetime. If we take into account the time required to access other storage structure (such as memory and disk) to read out the matched records, the lifetime will be even longer.

**Table 26: Theoretical Maximum Lifetime of Four Hybrid Data Structures**

(T<sub>access</sub>: memristor array read/write latency)

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Lifetime (years)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T&lt;sub&gt;access&lt;/sub&gt;=10ns</td>
</tr>
<tr>
<td>Hash-CAM</td>
<td>2.4</td>
</tr>
<tr>
<td>T-tree-CAM</td>
<td>0.8</td>
</tr>
<tr>
<td>TB'-tree</td>
<td>88.6</td>
</tr>
<tr>
<td>TB'-tree-CAM</td>
<td>68.2</td>
</tr>
</tbody>
</table>

6.3 Discussions

The results and analysis in the previous section show that with the same area, hybrid storage structures and a memristor-based T-tree can store much more records than DRAM. As a result, the performance of hybrid storage structures and a memristor-based T-tree is much higher than DRAM when the data size is larger than the capacity of
DRAM but still smaller than the capacity of hybrid storage structures and memristor memory since we have to access disks for a CMOS-based T-tree, which significantly increases search time.

For memristor-based storage structures, including memristor-based memory and hybrid CMOS-memristor-based storage structures, we can see that if we can reduce memristor read/write latency to below 40ns and the number of records is smaller than $10^{16}$, software-implemented memristor-based T-tree is the best storage structure for search applications. Otherwise, TB'-tree-CAM performs better through combined compute and storage and has an acceptable lifetime which is longer than 60 years.
7. Related Work

7.1 Applications of CAM

There are several network applications that can benefit from constant search time of CAM, such as network switches, IP filters, and ATM (Asynchronous Transfer Mode) switches [35]. In a network switch, CAM compares the destination address from an incoming data packet with the addresses stored in it and the comparison result decides where the packet should be routed. In an IP filter, a security feature which provides firewall services, CAM compares an incoming packet with the IP filter rules and the comparison result decides whether the packet should be permitted or not. In an ATM switch, CAM acts like an address translator, which change the VPI/VCI (Virtual Path Identifier/Virtual Channel path Identifier) into the value used for the next segment of connection for an ATM cell.

Besides network applications, there have been proposals to utilize associative memory in databases to alleviate memory bottlenecks and accelerate performance from last century [36, 37] to recent years [38, 39]. However, CAM assisted database management is currently limited by CAM capacity. Today’s databases can have terabytes of data [40]. The fact that current CMOS-based CAM only has megabytes capacity and does not scale well makes it important to exploit emerging technologies to build larger CAMs.
7.2 Existing Memristor-based Storage Structures

A memristor-based crossbar memory system has been demonstrated by HP Labs [16]. Strategies and peripheral circuitry have been designed to write into and read from a memristor array. The memristor memory has much higher density and access time comparable to CMOS RAMs, and demonstrates that it is feasible to apply specific voltages onto selected memristors. Our MemCAM/MemTCAM and hybrid storage structures rely on the mechanisms proposed in the memristor-based memory system to write into and read from a memristor array.

A hybrid CMOS-memristor CAM (MCAM) has also been proposed to increase the capacity of CAMs [9]. MCAM uses both CMOS transistors and memristors to build CAM cells to reduce silicon area and power dissipation. However, the combination of CMOS transistors with memristors into the same array reduces bit density and increases manufacturing difficulty. Besides, the endurance problem of memristors remains unsolved in MCAM design. We propose to build CAM/TCAM using only memristors to better utilize the high density feature of memristor arrays, and we propose hybrid storage structures to solve the endurance problem.

7.3 Conventional Lifetime Improvement Approaches

Besides memristors, there are other emerging memory technologies with endurance problems and PCM (Phase Changing Memory) is a typical case. There have been several technologies to improve the lifetime of PCM [23, 25]. For example, instead
of widely used cache replacement policy such as LRU (Least Recently Used), we can choose to evict clean cache blocks so we do not need to write them back to PCM. We can also reduce the number of write by only writing the changed bits back into memory. These approaches are orthogonal to our hybrid storage structures and can be apply to the hybrid storage structures to further improve lifetime.

Recent work from Intel and Microsoft proposes new B+-tree algorithms for PCM to improve performance and reduce the number of writes [26]. They design unsorted node organizations to reduce the number of writes incurred during insert and delete operations. Their results show that the unsorted-leaf scheme (in which only the leaves are unsorted) performs better than the unsorted scheme (in which all the nodes are unsorted). The reason is that it requires linear time to search within a node in PCM. However, with the computation ability of memristors, we can perform simultaneous comparisons and reduce search time within a node, which makes the unsorted scheme a better choice for hybrid compute and storage technologies such as memristors.

7.4 Processing-In-Memory and Distributed Memory Systems

Extensive research have been down on processing-in-memory (PIM) to improve performance by combining processing units and memory [41-44]. In Terasys work stations [42] a standard 4-bit memory is augmented with a single-bit ALU controlling each column of memory. In DAAM (Dynamic Associative Access Memory) [43] a large number of small processing elements are put in a DRAM’s sense amps. DIVA [41]
incorporate multiple PIM chips to a conventional microprocessor. In the architecture of Smart Memories [44], we have multiple processing tiles which can be configured based on the requirements of applications. The main idea of PIM is to combine compute and storage, which is the same as our memristor-based data structures.

Recent research [45] shows that we can allow memory to be “disaggregated” to expand and share memory across servers. We can also provide large storage to datacenters through RAMCloud [46], an approach aggregating the main memories of thousands of commodity servers and storing data entirely in DRAM. Both proposals aim at expanding memory capacity and enabling memory sharing among servers. With our hybrid storage structures, we can build non-volatile disaggregated memory system or RAMCloud, which reduce the need for backing up data stored in memory. We can also perform in-memory computations to reduce the amount of data to be transferred among servers, which reduces the bandwidth requirement and data transferring time.
8. Conclusions and Future Work

8.1 Conclusions

It has become more and more critical to reduce search time as the data sizes of several applications grow exponentially nowadays. Representative conventional approaches to reduce search time, such as CAM and in-memory databases, are no longer efficient because of the data explosion: CMOS-based CAM has low capacity which cannot be increased through CMOS scaling, and in-memory databases have performance degradation as data size increases. As a result, we have to exploit emerging nanotechnologies to accelerate search.

Among emerging nanotechnologies, memristors have become promising candidates to build storage structures because of high capacity, short switching time and low power consumption. We exploit memristors to accelerate search through combined compute and storage. We first propose MemCAM/MemTCAM, a configurable memristor-based CAM/TCAM design, in which we use memristors as both memory latches and logic gates. Computation ability of memristors makes it possible to perform range search using MemCAM/MemTCAM. High density of memristors provides an opportunity to build MemCAM/MemTCAM with large capacity and small area. We use SPICE to model the memristor and analyze power and performance at different temperatures. The results show that it is feasible to build a 1Gbit MemCAM with 1cm x 1cm area. For 64-bit key word, the search time is approximate 75ns and the energy
consumption is approximate 2fJ/bit/search for MemCAM/MemTCAM supporting only point query, and the search time is approximate 140ns and the energy consumption is approximate 5fJ/bit/search for MemCAM/MemTCAM supporting both point and range queries.

Both MemCAM and MemTCAM have high capacity and are able to reduce total search time and energy consumption for search-intensive applications with huge data size. However, the benefit we can obtain from these storage structures is limited by low endurance of memristors. In order to utilize the computation ability of memristors and deal with the endurance problem, we then propose a series of configurable hybrid storage structures using both CMOS and memristor technologies to solve the endurance problem.

We propose four hybrid CMOS-memristor based storage structures: Hash-CAM, T-tree-CAM, TB'-tree, and TB'-tree-CAM. We use an analytical model to evaluate and compare the performance and lifetime of two software-implemented memory-based T-trees and these four hybrid storage structures. The results show that hybrid storage structures can utilize range search abilities and improve lifetime from minutes to more than 60 years. Furthermore, TB'-tree-CAM, a hybrid CMOS-memristor storage structure combining T-tree, B'-tree and CAM, manages to balance between performance and lifetime and can outperform other storage structures when taking both performance and lifetime into consideration.
There is research currently going on to improve the endurance of memristors in HP Labs. If the memristor endurance can be significantly improved in the future (e.g., comparable to SRAM/DRAM), MemCAM/MemTCAM is a power-efficient candidate to accelerate search. Otherwise, hybrid CMOS-memristor-based storage structures, especially TB*-tree-CAM, are promising designs to reduce search time with an acceptable lifetime.

### 8.2 Future Work

Because of the limitation of large data size we use only an analytical model for evaluation. We also focus on search operations and assume a uniform distribution of keys to search. In the future, we can utilize approaches such as profiling, trace-based simulation and statistical models for evaluation to obtain more accurate results. We can also evaluate insert/delete operations to discover a threshold update rate which decides whether an application can benefit from our hybrid storage structures.

In this thesis we focus on the design of MemCAM/MemTCAM and hybrid storage structures. We currently assume a memory-mapped interface. One possible future research direction is to explore other interfaces and management aspects such as virtualization. We also focus on comparisons since our goal is to accelerate search-intensive applications. With possible endurance improvement, we may be able to perform more complicated computations in a memristor array and have more benefits with such in-memory computations.
References


Biography

Yang Liu was born on January 23rd, 1982 in Jiangsu Province, China. She received her Bachelor of Science in Computer Science and Technology from Peking University in Beijing, China in 2003 and Master of Science in Computer Science from Peking University in Beijing, China in 2006. She also received another Master of Science in Computer Science from Duke University in 2010. Her research interests include architectures for emerging technologies, Network-on-Chip (NoC), Processing-in-Memory (PIM), and multicore systems. Her Ph.D dissertation explores the design space of memristor-based storage structures to accelerate search through combined storage and compute.

Publications:


115