BLOOMIER HASHING: CONFINING MEMORY BANDWIDTH AND SPACE IN NETWORK ROUTING TABLE LOOKUP AND MEMORY PAGE TABLE ACCESS

By

DAVID YI LIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2010

To my father and mother

ACKNOWLEDGMENTS

First, I thank my advisor, Dr Jih-Kwon Peir, for his continuous support throughout the whole Ph.D. program. I thank him for his insightful advices, support and mentoring. I also like to thank Dr. Shigang Chen for his advices and help for my research projects. I also extend my appreciation to my other committee members, Dr. Ye Xia, Dr. My T. Thai and Dr. Liuqing Yang for their valuable comments and support.

I thank my colleagues Zhuo Huang, Gang Liu, Jianming Cheng, Zhen Yang, Feiqi Su, and Xudong Shi for their help and friendships. I especially want to thank Zhuo Huang and Gang Liu for helping my research projects and learning the simulation environments.

Last, but not least, I thank my parents and my brother for their love, support and advices. None of these would be possible without them.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HAT	Hash anchor table
IPT	Inverted page table
LPM	Longest prefix match
TLB	Translation lookaside buffer
XOR	Exclusive or

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

BLOOMIER HASHING: CONFINING MEMORY BANDWIDTH AND SPACE IN NETWORK ROUTING TABLE LOOKUP AND MEMORY PAGE TABLE ACCESS

By

David Yi Lin

December 2010

Chair: Jih-Kwon Peir
Cochair: Shigang Chen
Major: Computer Engineering

In today's information age, efficient searching and retrieving of needed data are very important for many digital applications. Hashing is a common and fast method to store and lookup data. However, common hashing methods, such as random hashing, create collisions in hash buckets and can lead to long and unpredictable lookup latency. For this dissertation, we developed a new hashing method called Bloomier hashing. We present our hashing solutions through the applications of network routing table lookup and page table access for memory address translations.

Continuous advancement in network speed and internet traffic demands faster routing table lookup. It is difficult to scale the current TCAM-based or Trie-based solution for future-generation routing tables due to increasing demands on higher throughput, larger table size, and longer prefix length. We present a comprehensive solution for future routing table designs with three key contributions. First, we present a partitioned longest prefix matching (LPM) scheme by grouping prefixes according to their length distribution. Either TCAM or hash-based SRAM is selected in each prefix group to balance the space and bandwidth requirement for accomplishing the LPM

function. Second, we use the Bloomier hashing method to alleviate hashing collisions and to balance prefixes among the hashed buckets. Third, a constant lookup rate can be achieved by organizing the routing table as set-associative SRAM arrays along with proper prefix expansions to allow one memory access per table lookup. Performance evaluations demonstrate that the proposed routing table lookup scheme can maintain a constant lookup rate of 200 millions per second with less bandwidth and space requirement in comparisons with other existing hash-based approaches.

Computer architecture is moving into 64-bits virtual and physical addresses. Large address space causes problems for the traditional page table organizations. We introduce Incremental Bloomier hashing, a modified version of Bloomier hashing, to create a fast and space-efficient page table architecture. Simulation results show that our Incremental Bloomier hashing can decrease the average lookup time in the page table and therefore decrease the overall number of cycles spent on page table operations.

CHAPTER 1
INTRODUCTION

In the information age today, tremendous amounts of information are collected and processed to fulfill a wide array of needed daily functions. In this age, it has been characterized as the ability of individuals to transfer information freely, and to have instant access to knowledge that would have been difficult or impossible to find previously. Given the huge amount of data that must be stored and processed, it becomes increasingly difficult to efficient searching and retrieving the needed data. Therefore, it is essential to organize the huge data storage in a proper way so that information retrieval can be done quickly.

Let the collection of data elements be called a table, and each of the data elements called a record. Each record contains an identity called key, which can be used to differentiate between records. The simplest way to perform a search is by examining each of the record's key to the search key. The first record's key is examined and goes on until a record's key match the search key. If none of the record's key match the search key, then the search fails. This form of search is called a sequential search. The sequential search is time consuming, because many records must be examined sequentially. The worst case search time is the time that exams all records in the table. However, it is possible to organize the records in a way so that sequential search can be avoided. For example, records can be ordered by the values of their keys in the table. Searching is then can be done by comparing the value of the lookup key to a record's key, and decide how to continue the search based on the result of the comparison. A binary search is an example of this strategy and is very fast compare to sequential search. However, the average lookup time of this strategy still

depend on the total number of the records in the table. Hashing is another search strategy that has a fast average lookup time that is independent of the number of records in a table.

Hashing uses a data structure called the hash table. A hash table can be used to greatly reduce the number of records that need to be examined during a search. Hash table uses a hash function to transform each key into an index of a hashed bucket in a table. The record with the key is then store into the table according to the index. Searching in a hash table is done in two steps. First, the index of the search key is computed based on a hash function. Next, using the index, the records in the hashed buckets are examined to see if any record's key matches the search key. Searching can thus be reduced to only the records in the hashed bucket. Moreover, the average cost of searching is depending on the number of keys in a bucket, instead of the total number of keys in the table.

Ideally, the hash function should hash each key into a unique index, referred as a collision-free hashing. Hash collision occurs when one or more items are mapped into the same bucket. However, it is difficult and expensive to obtain a collision-free random hash functions. Moreover, when the numbers of hash buckets is smaller than the number of records, hash collisions always encounter. To hold multiple records in a bucket, chaining can usually be applied. Inside each hash bucket, a pointer is associated with each record. The pointer points to the next records in the hashed bucket to form a link list that contains all the records that hashed to the bucket. Each entry in the link contains the key value. During lookup, search may need to search all items in the link list. For efficient searching, a good hashing function can balance the

length of all the hashed link lists. However, good hashing functions such as perfect hash function [8, 50] which takes long time to computer the hash value for a key. It usually applies to a set of known keys. Therefore, a good hashing function can lengthen the lookup process [57].

One well-known approach of reducing the hashing collision is to use multiple hashing functions. Instead of a single bucket, multiple hashing functions hash a key to multiple buckets for placing a record. By placing each record into the least-loaded bucket among multiple hashed buckets, the length of the hashed buckets can be more balanced [10, 18]. Studies show that with the flexibility of placing a key into the smaller bucket using two hashing functions, balance of keys among buckets is indeed improved significantly [10, 18]. However, the downside of using multiple hashing is that the search must go through all the hashed buckets which increases the search complexity and lengthens the search time.

Furthermore, even with multiple hashing functions, unbalance among hashed buckets due to hash collisions is still unavoidable. Bloomier filter was introduced in [14] to completely eliminate conflicts and provide collision free hashing. Bloomier filter is an extension of the original bloom filter. Unlike bloom filter which only supports membership queries, the Bloomier filter can store arbitrary per-key function values for all the keys in the key set to achieve a collision-free hashing. Bloomier filter can be established as an intermediate index table. Multiple hashing functions are first applied to the intermediate index table. The encoded values stored in the hashed entries are fetched out and applied to certain mathematical function to produce a per-key index value. Such an index is then used to determine the hashed bucket in the hash table.

The unique advantage of having an intermediate index table is that the values stored in the index table can be set up in such a way to balance the record distribution among buckets. The original intent was to provide a collision-free per-key function. Such a collision-free mapping requires a hash table several times bigger than the number of records. However, instead of a collision-free hashing, we generalize and extend the Bloomier Filter approach by allowing collisions in the index table. We use the Bloomier index table for balancing the records distribution among the buckets.

There are many examples of information retrieval. Hashing techniques are commonly used among them. In the following subsections, we describe a few examples of information retrieval and our proposed solutions to the problems.

1.1 Bloomier Hashing and Its Application on Destination IP Address Lookup

The first example of information retrieval is in the internet domain. The internet is formed by a huge collection of the computer networks that are connected to each others and use a standard network protocol. Computers in the internet communicate to each others by sending packets. Packets are like binary mails. An internet packet contains two sections: header and data. The header of packet contains information for the network to deliver the data. The information includes source and destination addresses, error detection information and sequencing information. Packets go though the computer networks at the speed of light and there are many different paths for a packet to reach its destination. So, packets in the internet are like traffics in the real world and packet traffic controls is needed just as well. Routers are used for this purpose. The router is an important network device that is responsible for routing and forwarding the internet packets.

One of the main functions for a network router is to route packets it has received. A packet contains both destination and source IP address in the network data. Once the packet is received by the router, the router needs to determine the next hop or station to route the packet to base on the destination address. A lookup table called routing table is used to record the next hop. There are many entries in a routing table. Each entry in a routing table contains information including a destination IP address and next hop. The packet's next hop is determined by matching its destination IP address to an existing entry in the routing table. To save the routing table space, the destination addresses in a routing table are stored as prefixes. A prefix is a binary string which contains some wildcard bits at the end. Wildcard bits are basically "don't care" bits that can assume any values. An incoming packet's destination address may match more than one prefix in the routing table yet only one next hop can be selected. So, the longest-prefix match (LPM) is used in order to obtain the next hop. To perform the longest-prefix match, the router needs to match the incoming string to the longest prefix among all the prefixes in the routing table. Figure 1.1 shows an example of LPM in a routing table:

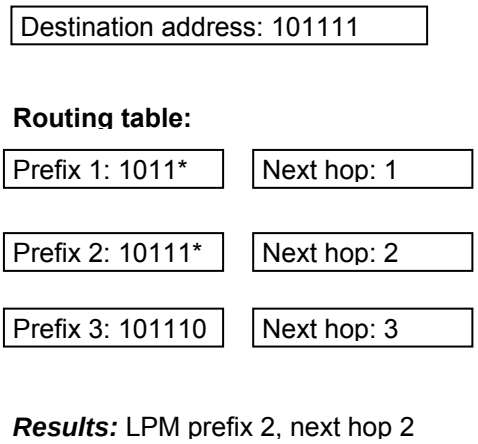


Figure 1.1. Longest prefix matching in a routing table

As seen in Figure 1.1, there are three prefixes in a routing table. For simplicity, let the length of an IP address be six bits, the router received a packet with a destination address 101111. The router performs LPM and decides the next hop. For IP address length of 6, prefix 1 has two wild card bits and prefix 2 has 1 wildcard bit at the end. Prefix 3 has no wild card bits. Wildcard bits can assume any values. For example, prefix 2 has the last bit as a wildcard bit and can match address 101111 and address 101110. Therefore, for the destination address 101111, the address matches both prefix 1 and prefix 2. However, since prefix 2 is longer than prefix 1 without the wildcard bits, the destination address match prefix 2, the longest prefix matched, and the next hop 2 is chosen.

In a backbone router, the number of prefix in the routing table can be up to 200-300 thousands [6]. LPM is a classic bottleneck in the backbone internet routers. With recent advances in optical network technology along with the new IPv6 routing table and a growing number of prefixes in internet backbone routers, it becomes increasingly difficult to provide an IP lookup scheme fast enough to handle the ever-increasing internet traffic. If the router is not up to speed, packet drops may occur.

There are three categories of LPM mechanisms including Ternary content addressable memory (TCAM) based schemes, tries based schemes and hash based schemes.

TCAM were the choice for lookup in small routing tables. However, TCAM is inefficient in silicon space with high power dissipation. TCAM compares the incoming address against all prefixes stored in a routing table. Therefore, TCAM is well suited for matching wild cards in a prefix. However, as the number of prefixes in the large routing

table grows continuously, it becomes prohibitively expensive and power hungry to use the TCAM solution. Caching a part of the routing table permits fast lookups for recent routed prefixes [1]. However, besides lack of locality, caching cannot provide a constant lookup rate and may cause package drops upon consecutive cache misses.

Tries-based search schemes are another approach for LPM. Tries are binary tree like data structures. To efficiently perform LPM, tries-based search algorithms match one or a few bits at a time with the prefixes in the routing table [2, 31, 35]. However, due to the multiple levels in tries, the lookup latency is directly proportional to the length of the prefix. For a long prefix, search need to go very deep in the tree, led to long search latency. The worst case lookup latency can directly affect the performance of the router. Moreover, each data node in the tries needs extra space to hold information such as child and parents nodes pointer. This led to large memory usage and forces the tries to be storage off-chip.

Many recent works consider building the routing table with conventional SRAM technology and using a hash-based approach to lookup the routing table [18, 65]. In this approach, a few fundamental issues need to be addressed.

- To accomplish the LPM, the router must match all possible lengths of the prefixes in the routing table. It incurs a significant bandwidth requirement and delays, especially for IPv6 where the prefix lengths can vary from 16 to 64 bits. This problem is exacerbated when the growing routing table can no longer fit into on-chip SRAM and must be fetched from off-chip.
- As the internet wire speed will soon exceed 100 Gbps, a router with a single network connection needs to forward more than 150 million packets per second [58]. The router must sustain the high constant lookup rate in order to avoid any dropping of incoming packages.
- One inherent difficulty in the hash-based approach stems from its collision in placing prefixes into the hashed buckets. When multiple prefixes are hashed to the same bucket, the search must cover the entire bucket. This problem is

exacerbated when hashing prefixes to the buckets are unbalanced so that the search delay must accommodate the longest bucket.

To reduce multiple searches for variable prefix lengths, the Control Prefix Expansion (CPE) [61] can be used. For a single prefix X prefix length L and wildcard length of W, control Prefix expansion replace the prefix X with a number of new prefix length of L+W, based on X's 2 to the power L values. CPE reduces the number of different prefix lengths to a small number with the cost of expanding the routing table. Meanwhile, Bloom Filter [19] has been considered to filter unnecessary accesses to the routing table for all possible prefix lengths. Although Bloom filter provides an efficient way to filter routing table lookups, it suffers a small percentage of false positive conditions. When this condition occurs, multiple routing table accesses can cause unpredictable delays in forwarding the incoming package.

Even with these issues, hash based approach still have some major advantages. During lookup, hashing method does not compare all prefixes in the table at the same time, so it used much less power than the brute force approach used by TCAM. Unlike tries, hash method lookup is O(1), regardless of the prefix length. Due to these advantages, hash-based approach seems to be the choice for LPM in future when IPV6 is commonly used.

We propose an IP lookup architecture that using a new hashing scheme called Bloomier hashing to greatly reduce the collisions in hash buckets. We also introduce partial prefix expansion. Partial prefix expansion allow search only on few different length while maintain a reasonable space overhead. By using these techniques, our destination IP lookup architecture can achieve much higher bandwidth than the other existing hashing based architecture.

1.2 Page Table Lookup Using Incremental Bloomier Hashing

Second example of information retrieval is in the computer virtual memory domain. Computer data used in computer programs are binary data that store in physical memory devices. Computer programmer likes to have a large and fast memory for their program. However, a memory device that has bigger capacity also has more latency. On the other hand, a fast memory device is more expansive and smaller. For example, a disk is a cheap memory device that has the most space but slowest access time. On the other hand, computer main memory is typical RAM (random access memory) device that is more expansive than the disk but has much faster access time. Since RAM is more expansive, the size of computer main memory is only a fraction of the disk space. Ideally, program data should be in the main memory all of time. However, due to the fact that some programs have large sizes, some program data need to store on disk when it is not in use. Keeping track of and managing the program data in the main memory and on disk at the same time are tiresome tasks. Virtual memory is used in today's operating systems to solve the problem. Virtual memory management unit combines both the disk and main physical memory into a single abstract memory unit. An application that uses virtual memory thinks that there is a large continuous memory to use. But in reality, some parts of the memory space may reside on the disk and some parts of the memory may be fragmented. Using virtual memory can make a programmer's job a lot easier when programming big applications.

When using virtual memory, applications request memory accesses by using virtual addresses. The memory allocations are done in a unit called a page. A page is a block of data resides in the data storage. Virtual address is used to address a page in the virtual memory. However, using virtual memory, some of the requested pages may

not be in the main physical memory. Therefore, the hardware/software needs to determine where the requested memory page is located. To do so, the hardware/software tries to translate the virtual address into physical address where the page is located in the physical main memory. Fail translation implies that the page is located on the disk. This translation process is called virtual to physical memory address translation. When the requested page is not located in the physical memory, a page fault occurs. The operating system must take over and move the page from disk into the physical memory.

To do a virtual memory address translation, a lookup table called page table is used. A page tables stores all the mapping between virtual addresses and physical addresses. Each page table entry contains the virtual address, the physical address, protection bits and status information [28]. To do virtual memory address translation, the hardware/software simply uses the virtual address to find a matching entry in the page table and return the content of the entry, if the virtual memory address has valid translation.

The address translation process is in the critical path of the computation and need to complete quickly. Since the page table can be fairly large, caching part of the page table can be useful. The translation lookaside buffer (TLB) is a very small buffer that caches some page table entries. Due to its size, TLB is very fast and located close to the processor. Many different TLB organizations [17, 38] was proposed and evaluated. Whenever a translation is needed, the TLB is first checked to see the translation exist on the TLB. However, due to the small size of the TLB, not all the valid

virtual address can be translated by the TLB. The pages table handles the translation when the TLB can not find a valid translation.

The straight forward way to implement the page table is to create a page table entry for each virtual address and organize these entries into a linear array. During a lookup, the virtual address is used to directly index the array and return the content of the entry. Obviously, such approach has huge space requirement. For example, if the virtual address is 64 bits, then the total number of entries in the page table is 2^{64} divided by the page size. If the page size is 8k, then the total number of entries in the page table is 2^{51}. Moreover, many of the entries do not contain physical address translation since the physical memory is much smaller than the virtual memory. Due to this, the table is quiet sparse and a lot of space are wasted. Moreover, the linear array is implemented in hardware using trees like data structures which has multiple levels [47]. For large virtual address, lookup needs to traverse many level deep down to tree with multiple memory references. Using reverse mapping can solve the problems of array approach. Instead of creating a page table entry for each virtual address, a page table entry is created for each physical page. The total number of entries in the page table is then equal to the total number of physical pages. Since each physical page is mapped to a virtual address, no entire in the page table is empty. However, a lookup can be difficult in the reverse mapping approach using virtual address. Typical hashing scheme can help with the lookup. One common approach is to use hashing by adding another intermediate table called hash anchor table. The virtual address is first hashed to the anchor table to retrieve the index to the inverted page table. Due to hash collisions, multiple virtual addresses can be hashed to the same entry in the anchor

table. Therefore a linked list must be built to link all the collided virtual addresses in the inverted page table. Consequently, the lookup time can still be unpredictable and long in the worst case. The average case lookup time in a page table can affect the overall system performance. We propose using Bloomier filter and our Bloomier hashing concept to the page table. By using our methods, we can reduce the search time and the additional memory accesses when accessing the page table to improve the overall system performance.

1.3 Dissertation Organization

The dissertation structure is as follow. In Chapter 2, we present our Bloomier hashing algorithm and apply this hashing method to create a high throughput LPM architecture. In Chapter 3, we first introduce the common page table organizations. Then, we present our Incremental Bloomier hashing method and apply this method to decrease the average lookup time in the hashed page table. Chapter 4 introduces other possible applications by using Bloomier hashing. Chapter 5 is a summary of the dissertation.

CHAPTER 2
BLOOMIER HASHING AND ITS APPLCATION ON DESTINATION IP ADDRESS LOOKUP

2.1 Introduction

Destination IP address lookup is an important function of the router. When an incoming packet arrives, the router need to find the correct output port for the packet. To do so, the router needs to perform a lookup on the routing table. Given an IP address, the routing table returns the next hop or output port that the packet can forward to. Because the address space for IP addresses is too big, routing tables do not store next hop information for every IP address. To save space, routing table stores next hop information for IP prefixes. An IP prefix summarizes a group of IP addresses. To perform a lookup on the routing table, a longest prefix match (LPM) algorithm is used. Longest prefix match (LPM) algorithms match the longest prefix in the routing table to the incoming IP address and return the next hop. There are three major approaches for the longest prefix match problem in destination IP address lookup. The three approaches are ternary content-addressable approaches, tries based approaches and hash based approaches. In Chapter 2, we present a new hash based approach for longest prefix matching problem by using a new hashing scheme called Bloomier hashing. Bloomier hashing can greatly improve the hash table performance by using a small intermediate table. We show the general architecture by using Bloomier hashing and present the results afterward.

2.2 Bloomier Hashing

The Bloomier Hashing is a generalized and extended solution from the original Bloomier Filter [14]. Bloomier filter is built based on the concept of multiple hashing. While Bloomier filters support retrieval of arbitrary per-key information and guarantee

collision-free hashing, the Bloomier hashing relaxes the constraint and balances hashing collisions for distributing prefixes among buckets. Like Bloomier filter, our Bloomier hashing requires an intermediate table. However, due to relaxing the collision-free constraint, the intermediate table in Bloomier hashing can be much smaller while achieving very good hashing performance. Let's first introduce some terminology for the Bloomier hashing. The collection of all prefixes is the prefix set. The prefix set is hashed into a set of hashed buckets. An intermediate hashing table called the Bloomier Index Table (BIT) is introduced where all prefixes in the prefix set have first been hashed. Proper values are stored in the BIT for determining the final hashed buckets for all prefixes. There are *k* hashing functions that are used to hash each prefix to *k* locations in the BIT where these *k* locations for each prefix are the prefix's hash neighborhood. If a prefix is hashed into a location that is not in any other prefix's hash neighborhood, the location is called a singleton. The collection of hashed prefixes in each location in the BIT is the prefix group of the respective location. The process of hashing and recording the prefix set to the BIT is referred to as the index table encoding. The setup of values in all entries of the BIT for placing and searching the prefixes based on the encoded index table is referred to as the index table setup.

We describe the complete encoding and setup algorithms of Bloomier hashing below then follow by a detail example:

Encoding of Bloomier Index Table:

Index table encoding is straightforward. Based on *k* randomized functions, all prefixes in the prefix set are hashed into the entries in the BIT. Due to collisions, each entry in BIT may have multiple prefixes hashed to it. Therefore, each prefix group has

the number of prefixes ranging from 0 up to the total number of prefixes in the prefix set. A counter is maintained for each prefix group for selecting the prefix group during the BIT setup. The Encoding algorithm returns back the index table.

Setup of Bloomier Index Table:

The setup method takes two parameters **S** and **T**. **S** is the sets of prefixes and **T** is the table return by the Encoding process described above. The setup algorithm involves two major functions as shown in Figure 2.1. The first function is called FindGroup(**S,T**) and the second function is called ProgramTable(∏, **T**). We describe each of the functions below.

Setup (**S**, **T**)
    ∏=FindGroup(**S, T**)
    ProgramTable(∏**,T**)
    Return **T**

Figure 2.1. Psuedocode for Bloomier index table setup

FindGroup(**S,T**) takes two parameter **S** and **T.** The function returns a stack ∏ which contains all the prefixes. The algorithm is shown in Figure 2.2. The algorithm is similar to the original Bloomier filter setup method but with some fundamental different. In the function, we first create the empty stack ∏**.** Next, while not all prefixes are on the stack, we try to find any prefixes that have singletons. If any prefixes with singletons exist, we push these prefixes and their singletons onto the stack ∏. We also erase these prefixes' locations in the table **T.** If none of the prefixes with exists**,** we do not redo the setup with different hashing functions like in the original Bloomier filter approach. Instead, we find the smallest prefix group, the prefix group with smallest counter values. After the group is found, we push all the prefixes which are in the group

and the group location onto the stack ∏. We also erase all these prefixes' locations in the table. The whole process is repeated until all prefixes are on the stack.

```
FindGroup (S,T)
    Create an empty stack ∏
    While Not all prefixes in S are on the stack
    Find all prefixes with singletons in table T
        If (prefix(s) with singleton(s) exist)
            Push the prefixe(s) and their singleton(s) to the stack ∏
            Erase all these prefixes' K locations in table T
        Else /*no key with singleton exist*/
            Find the smallest group /*prefix groups with smallest counter*/
            Push all prefixes in the group and the group location to stack ∏
            Erase all these prefixes' K locations in table T
    Return stack ∏
```

Figure 2.2. Psuedocode for FindGroup

To encode the proper values in the index table, the function ProgramTable (∏, **T**) is called. The algorithm is shown in Figure 2.3. The function takes two parameters ∏ and **T.** ∏ is the stack returned by FindGroup (**S,T**). The stack contains all the prefixes in **S. T** is the index table. The function does the following. While the stack is not empty, a prefix is popped from the stack. If the prefix is a prefix with a singleton, then a shortest bucket is chosen among all the hash buckets. The prefix is then placed into the bucket. The singleton location in the table is encoded with proper value with the function Encode(**p, b, s, T**). The function used the same Xor idea as the original Bloomier filter [14]. In the index table **T**, the function Encode (**p, b, s, T**) encodes the value **b** into the location **s** for prefix **p.** On the other hand, if the prefix that was popped form the stack is belong to a prefix group, then the rest of group **P** are popped from the stack as well. We then find **D** smallest buckets among all the hash buckets**.** We then

place one of the prefix in **P** into one of the **D** buckets by using the encode function**.** We try all combination by using each member in **P** and **D**. Using the function TotalSum (**P**) we then chose the assignment that gives the shortest total length of the affected buckets among all the hash buckets after encoding. The whole process repeats until all prefixes are popped from the stack.

```
ProgramTable (∏, T)
    While the Stack ∏ is not empty
        Pop a prefix p off the top the Stack ∏
        If the prefix p is a prefix with Singleton s
            Chose smallest bucket b in the hash table
            Encode (p, b, s, T)
        Else /*the prefix p is belong to a prefix group */
            Pop the rest of prefix group P and the group location l off the stack
            ∏
            Pick D smallest buckets in the hash table
            For each prefix t_i in the prefix group P
                For each bucket b_j in D
                    Encode (t_i, b_j, l, T)
                    Sum_i,j = TotalSum (P)
            /*Let t_s and b_s be the prefix and bucket that have the smallest sum
            value*/
            Encode (t_s,b_s,l,T)
```

Figure 2.3. Psuedocode for ProgramTable

Search and Update Prefixes:

After index table setup, searching prefixes are straightforward. The prefix is first hashed to multiple BIT locations and the contents of these locations are fetched out. The bucket ID is determined by exclusive-oring these values. Although an intermediate BIT access is necessary, the size of the BIT can be small in comparison with the

hashed buckets, and hence can be fitted into fast on-chip SRAM. We evaluate the performance impact of different BIT sizes later.

Example of Bloomier Hashing Encoding and Setup:

We now present a detail example of our Bloomier hashing algorithm using Figure 2.4.

Index table encoding is straightforward as illustrated. Each prefixes are hash into the table and a counter (not shown in Figure2.4) is used to keep track of how many prefixes are in one location. For example, location 1 in the Bloomier Index Table (BIT) has prefixes K0, K2, K3, and K4 hashed to it with the counter value of 4. Next all the prefixes groups are push onto the stack start with the smallest group. In the example, K0 and K1 in location 2 are selected first to be pushed onto the stack. All K0 and K1 are removed from BIT afterwards. Next, the remaining K4 and K5 in location 4 are selected and removed. Finally, K2 and K3 in location 1 are removed as marked by the circled sequence number.

All entries in the BIT are initialized with a randomized value ranging from 0 to the total number of buckets before the second step begins. In this example, K2 and K3 are the first group to be popped from the stack. In placing these prefixes to buckets, we consider a heuristic algorithm by placing one of the prefixes in the prefix group into one of the shortest bucket. In this example, we assume there are 4 buckets with the lengths of 4, 2, 1, and 2 when the simulation begins. Hence, bucket 2 with a single prefix is the target for placing either K2 or K3. For determining the hashed bucket from the intermediate BIT, we use the function encoding idea [14]. The ID of the bucket can be

calculated by an Exclusive-OR function of the values fetched from the hashed entries in the BIT. Since both K2 and K3 are hashed to the same two BIT entries 1 and 5, they

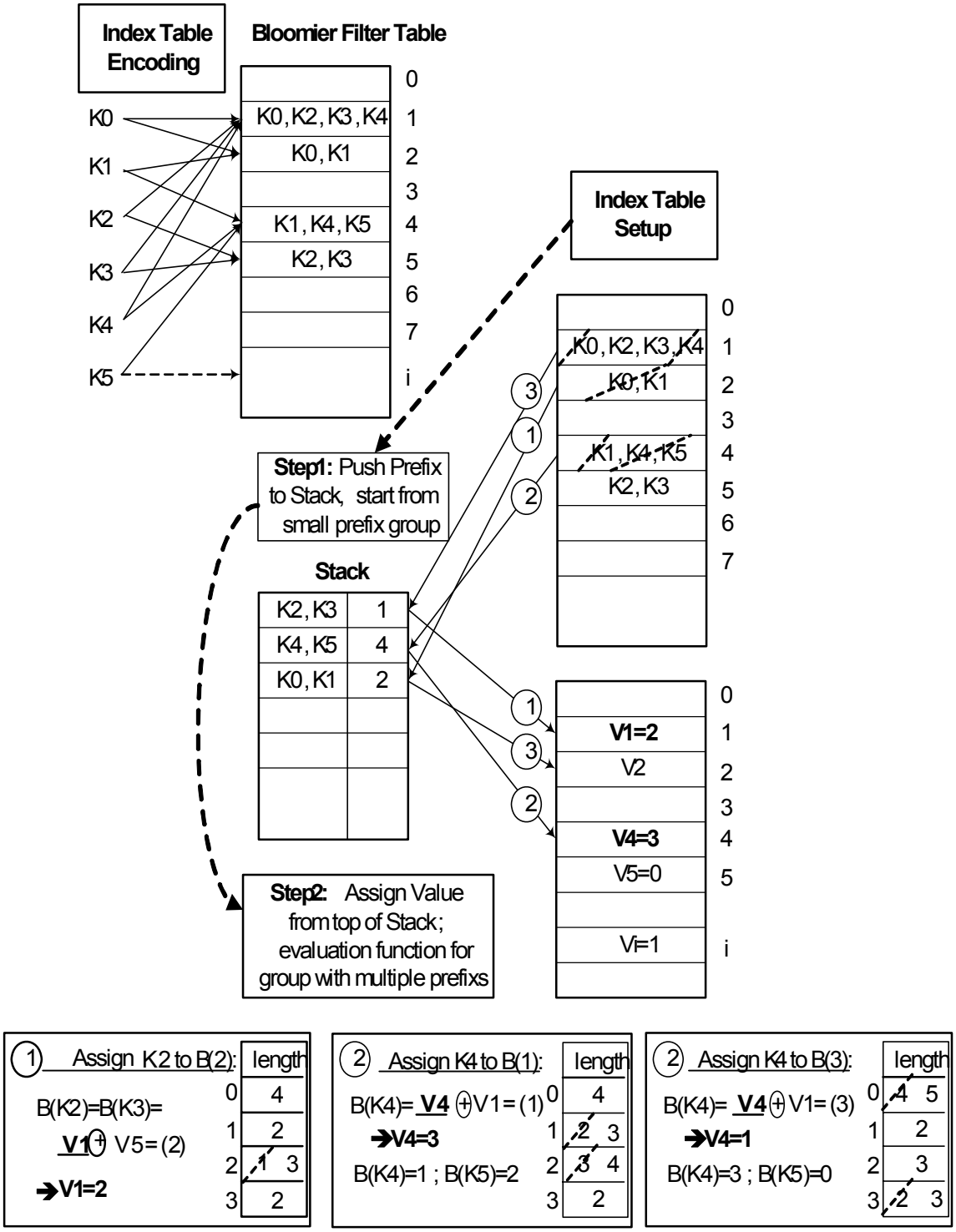


Figure 2.4. Bloomier Index Table encoding and setup example

must be placed to the same hashed bucket. The ID of the shortest bucket can be calculated as: $B(K2)=B(K3)=2=V1\oplus V5$, where *B(Ki)* represents the bucket ID of the prefix *Ki*, and *Vi* stands for the value stores in the *i*-th location of the BIT. In this example, we assume location 5 in the BIT (*V5*) contains an initial value of 0. The value in location 1 (*V1*) can be determined: $V1=V5\oplus 2=2$. The bucket lengths become 4, 2, 3, and 2 after the first placement.

Next, K4 and K5 are popped from the stack and placed into the buckets. Since buckets 1 and 3 are now the shortest, either K4 or K5 will be placed into them. In determining the value for *V4* in the BIT, we apply a simple heuristic function to see if either placing K4 or K5 in bucket 1 or bucket 3 can result in a shorter total length of the affected buckets. Note that the length of a bucket with *n* prefixes is calculated as the sum of the traversal length to each prefix in the bucket (*1+2+…+n*). The total length of the affected buckets can be determined from the sum of the individual buckets. In this example, K4 is first placed in bucket 1. Following the same procedure in determining the value of *V1*, *V4* is equal to 3. Hence, B(K4)=1; and B(K5)=2. The bucket lengths become 4, 3, 4, and 2 after the placement. Similarly, K4 can be placed in bucket 3. With the same calculation, *V4*=1. Given the initial value of *Vi*=1, B(K4)=3; and B(K5)=0. The bucket lengths become 5, 2, 3, and 3. Applying the simple heuristic function, the total length of the affected buckets 1 and 2 for the first placement is equal to (1+2+3)+(1+2+3+4) = 16; and the total length for the second placement is equal to (1+2+3+4+5)+(1+2+3) = 21. Therefore, the first placement is the choice. Next, K5 is placed into either bucket. Due to the associative property of the exclusive-or function, K5 placement shows identical results as the K4 placement. Therefore, *V4* can be set to

value 3. This process continues until all the prefix groups are popped from the stack and placed into the buckets.

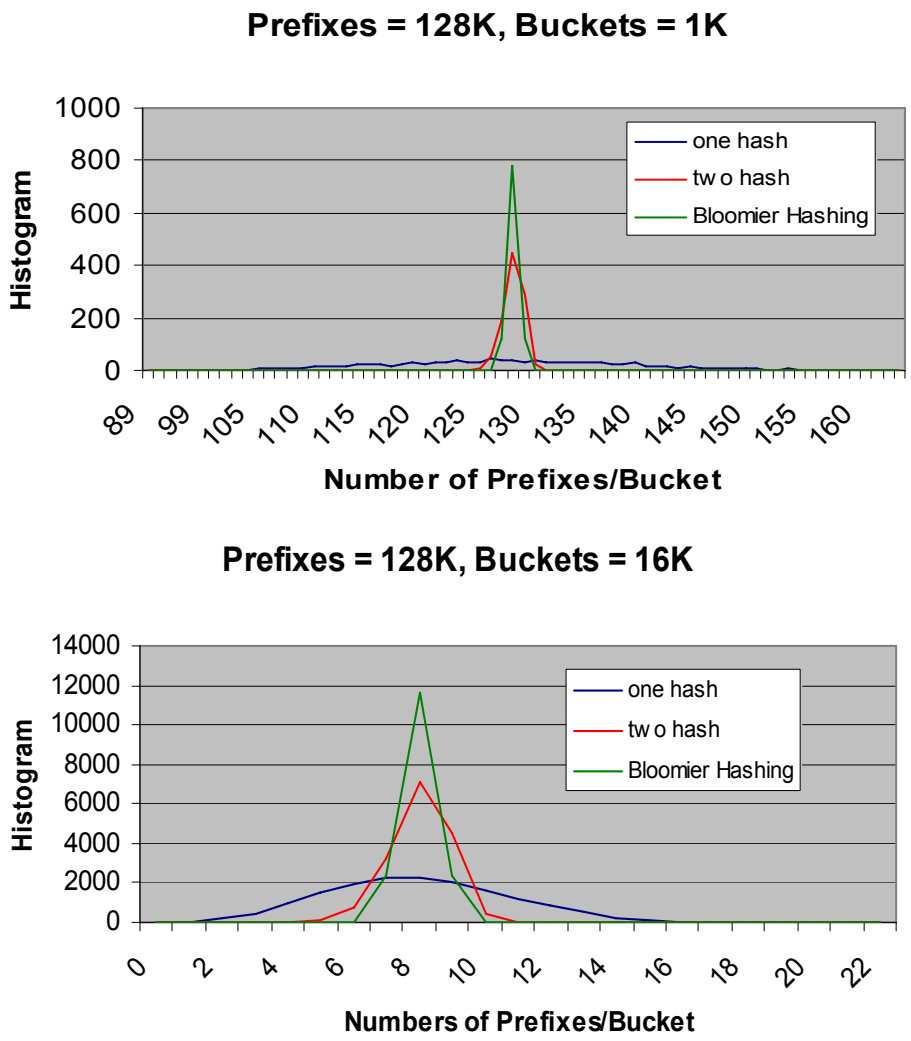


Figure 2.5. Distrbutions of prefixes into buckets with single hashing, two hashing and Bloomier hashing

In order to show the effectiveness of Bloomier hashing, we randomly generate 128K prefixes and hash these prefixes into 1K and 16K buckets using a single hashing function, two hashing functions and our Bloomier hashing method. With two hashing functions, each prefix is placed into the bucket with smaller number of prefixes. The results shown in Figure 2.5 clearly demonstrate the uneven distribution of prefixes to

buckets with a single randomized hashing function. With the flexibility of placing a prefix into the smaller bucket using two hashing functions, balance of prefixes among buckets is improved significantly. However, to perform a lookup with two hashing function, two buckets are searched. Using our Bloomier method with a small index table size of 16K entries, balance of prefixes among buckets is improved even further. Our Bloomier hashing method even outperform the two hashing method by a large margin.

2.3 IP Prefixes Expansion

Given the need to match the longest prefix, each routing table lookup must search for all possible prefix lengths. It incurs long delays for sequential searches or a huge bandwidth requirement for parallel searches. We study the IP prefix distributions and use controlled prefix expansion to decide the best compromise.

Studies on several routing tables in core internet routers have shown that the distribution of the prefixes according to their lengths is stable, but very uneven. In Figure 2.6, we plot the length distributions of five routing tables, as286, as1103, as4608, as4777 and as6447 [6, 54]. Similar to the report in [18], a majority of prefixes (>98%) have lengths between 16 and 24 bits with length 24 dominating about 54%. Prefixes longer than 24 bits are very few (<2%) and there is no prefix shorter than 8 bits. This uneven distribution provides an opportunity for partitioning the prefixes into groups and applying different LPM mechanisms in different groups. On-chip TCAM serves well when the number of prefixes is small with a wide range of lengths. On the other hand, a hash-based SRAM using proper prefix expansion can confine the space and bandwidth requirement when all prefix lengths are long, consecutive, and have small variations.

Based on the length distributions in Figure 2.6, the routing tables can be partitioned into three groups, length 8-18, length 19-24, and length 25-32 for the

following reasons. Length 25-32 represents less than 2% of the prefixes. Instead of including these eight different lengths in a hash-based table, it is inexpensive to use small on-chip TCAM to perform the LPM. Similarly, there are less than 1% of the prefixes with the lengths of 8-15, and hence they are also a good target for TCAM. The remaining 98% of the prefixes have lengths from 16 to 24. To avoid using expensive TCAM, these prefixes can be off-loaded to a hash-based SRAM array. However, there are still nine different prefix lengths and all need to participate in the longest match of this group.

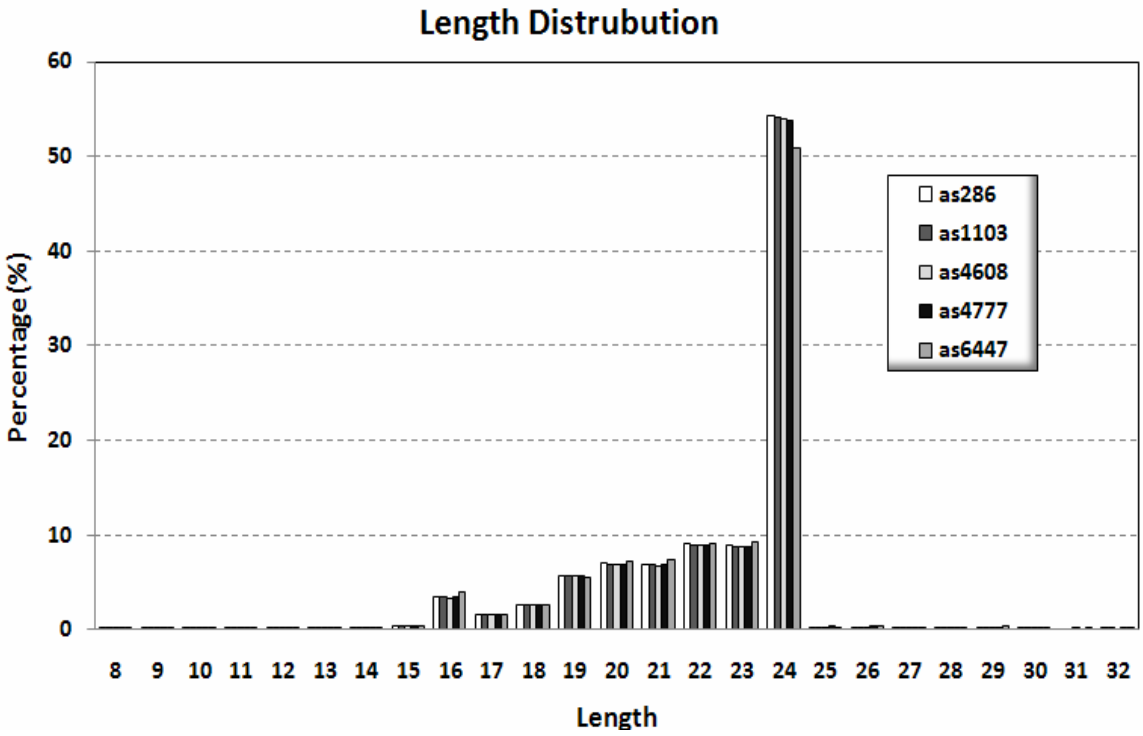


Figure 2.6. Distributions of prefixes based on prefix length from five routing tables

The amount of prefixes from length 16 to 24 is roughly in an increasing order. To further reduce the number of different prefix lengths in the SRAM table, hence reducing the LPM searches, more prefixes may be moved into the TCAM starting from the shortest length. The ratios of prefixes of varying the number of lengths are given in

Table 2.1. With six lengths from 19 to 24, the total amount of prefixes is over 90%. In other words, the remaining prefixes with 19 different lengths as a target for the TCAM solution are less than 10%. Further reducing the number of lengths in this group helps LPM searching, but it also boosts the need for larger TCAM. The final LPM is decided from the LPMs of the three groups.

Table 2.1. Percentage of prefixes in different length groups

	Length 16-24	Length 17-24	Length 18-24	Length 19-24	Length 20-24
AS286	99.2%	95.8%	94.3%	91.6%	85.9%
AS1103	99.0%	95.5%	94.0%	91.4%	85.8%
AS4608	98.1%	94.8%	93.3%	90.7%	85.1%
AS4777	98.3%	94.9%	93.4%	90.8%	85.2%
AS6447	97.2%	93.9%	92.5%	90.0%	84.3%

The off-chip memory organization plays a critical role in enabling one memory access per routing table lookup for prefixes with different lengths. Similar to [18, 33, 34], we consider two-dimensional set-associative SRAM arrays to support a constant lookup rate. The first dimension is the total number of hashed buckets while the second dimension is the maximum length among all buckets. To accomplish one memory access per lookup, each bucket is allocated in consecutive memory locations and can be fetched as a single block. Furthermore, to balance the routing table size and search bandwidth, we use a hybrid prefix expansion and bucket coalescing scheme to allocate prefixes with multiple lengths in the same bucket for accommodating the LPM in a single memory access.

Consider the prefix group of length 19-24 as an example. One straightforward solution for enabling one memory access per lookup is to expand all prefixes to the maximum length 24. To do so, all prefixes less than 24 are expanded to 24. To replace

the wild card bit in a prefix, the prefix is replaced by many prefixes to represent all the possible representation of the original prefix. For example, if a prefix need to replace 2 wild card bits, then the prefix is replace by four other prefixes. Given a majority of prefixes in this length group, such an expansion produces significant space overhead as shown in Figure 2.7. The other opposite solution is to keep all prefixes without any expansion. Multiple lengths of prefixes that are required for determining the LPM can be allocated in the same bucket based on the common 19 prefix bits for accommodating the shortest length. Although this bucket coalescing approach achieves one memory access per lookup, it suffers heavy bandwidth requirement since all prefixes must be hashed to buckets based on only 19 common bits. The reason is that by hashing only on the common bits, multiple different prefixes that have the same common bits always hash into the same hash bucket regardless how hashing is handled.

Now consider a general case where all prefixes have lengths from *m* to *n* with *n-m+1* different length. To accomplish one memory access per lookup, a fixed length *l*, $m \le l \le n$ can be chosen for balancing the space overhead from prefix expansion with the bandwidth requirement due to coalescing of different lengths into the same bucket. All prefixes with lengths less than *l* are expanded to *l* for determining the hashed bucket. By reducing the expansion from lengths *n* to *l*, the expansion ratio can be reduced up to a factor of $2^{(n-l)}$. However, in using the common *l* bits for hashing, a maximum of $2^{(n-l)}$ prefixes may now be allocated in the same bucket. We refer to this collision as the

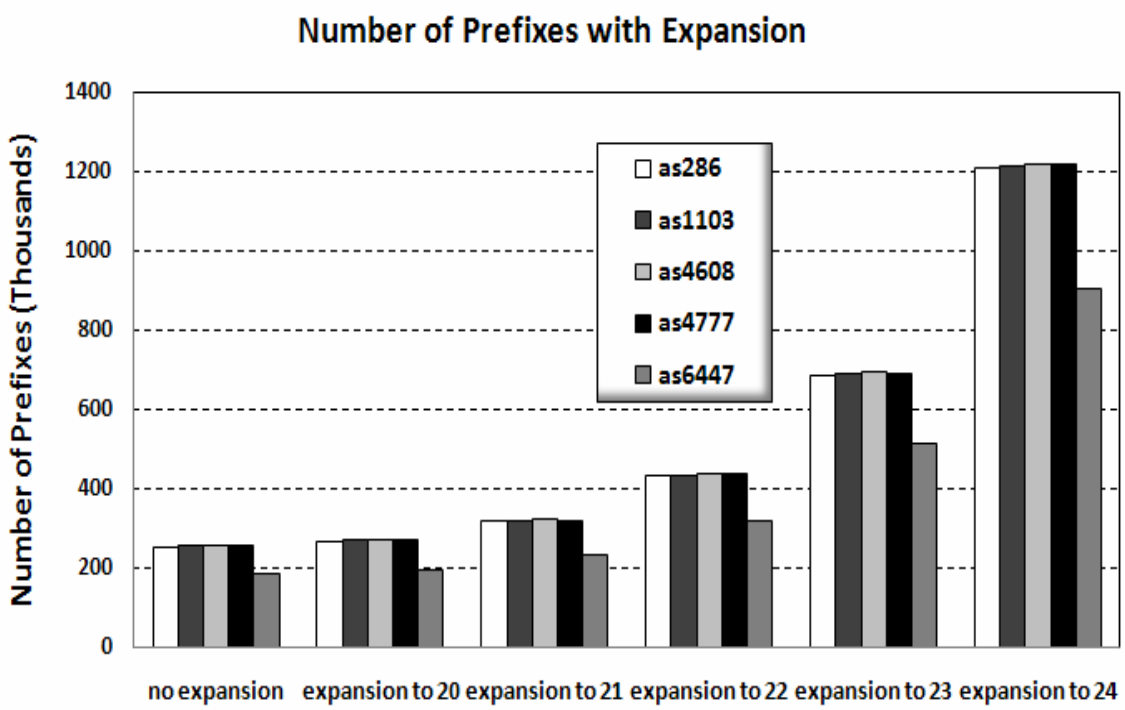


Figure 2.7. The number of prefixes with expansions to various lengths

lower bound of the maximum number of prefixes in a bucket. For example, with about 70% of space overhead, we can expand length 19-21 to 22 to be used for hashing. With this expansion, however, the lower bound for the maximum bucket length becomes 4. Similarly, in order to reduce the lower bound to 2, we must expand prefixes 19-22 to 23 with about 2.7 times of the prefixes. Note that besides the collision due to limited hashing bits, the maximum length in a bucket is determined by the overall hashing collisions since prefixes with different common bits may still be hashed into the same bucket. Detailed evaluations of the space/bandwidth tradeoff are presented in Section 2.6.

2.4 Architecture and Implementation of the BH-RT

The block diagram of the basic architecture for a BH-RT based router is illustrated in Figure 2.8. There are a number of key highlights.

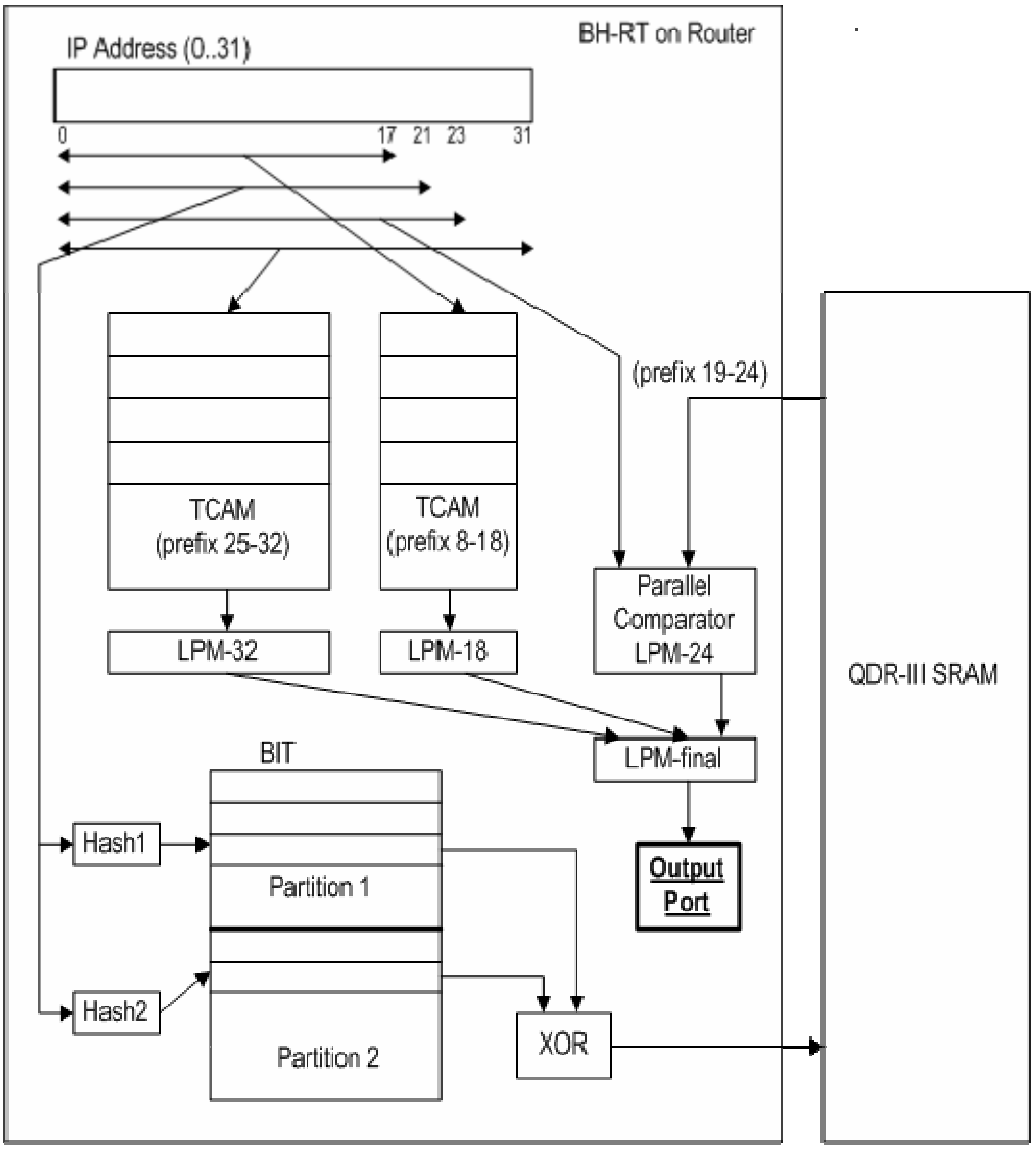


Figure 2.8. The basic architecture for BH routing table lookup

- **Partitioned two-level longest prefix matching (LPM):** Given an uneven distribution of various prefix lengths from several IPv4 routing tables, the prefixes are grouped into three classes, length 8-18, length 19-24, and length 25-32 in this example design. Group 19-24 has over 90% of the prefixes which is saved in hash-based SRAMs, likely off-loaded from the router chip. The remaining two groups with less than 10% of the prefixes reside in on-chip TCAMs for performing the LPM function.

- **Bloomier Hashing (BH) with on-chip Bloomier Index Table (BIT):** As described in Section 2.2, the BH scheme is used to balance the prefixes among the hashed buckets for Group 19-24, located off-chip. A hybrid prefix

expansion/coalescing scheme described in highlight 4 below, confines prefixes for each lookup in one hashed bucket. For fast accesses, the BIT is partitioned into equal regions, each associated with a hashing function.

- **Off-chip set-associative ODR-III SRAM:** To maintain a constant lookup rate, the off-chip memory is organized as a two-dimensional set-associative array. The first dimension is the set, which is the total number of hashed buckets. The second dimension is the set-associativity, which is the maximum length among all the buckets. Each bucket is fitted in a block of fixed size in consecutive memory locations and can be fetched as a single unit. High-bandwidth 500+ MHz QDR-III SRAM is used which supports 72-bit reads/writes per cycle [52].

- **Hybrid prefix expansion/coalescing scheme**: To achieve one memory access per LMP lookup, prefixes of different lengths that are required for the LMP, must be allocated in the same hashed bucket. A proper length l, i.e. 22 with prefix bits 0 to 21 in the illustrated design, is chosen as the indices to determine the hashed bucket. All prefixes with length less than l must be expanded to l to provide the needed common hashing bits for avoiding multiple memory fetches.

Unbalanced hash collisions increase both the fetch bandwidth and the memory space requirement. As illustrated in Figure 2.8, an intermediate Bloomier Index Table (BIT) is constructed on-chip for the new BH-RT scheme. Multiple hashing functions using the common prefix bits determine the locations in the BIT. An exclusive-or function of the contents from the hashed BIT locations provides the bucket address in the memory.

A simple hashing function based on direct-mapping is considered. The lower-order k bits from the common prefix bits are selected, where 2^k is the size of the BIT. In case the number of common prefix bits is less than k, each prefix is hashed to a single location in BIT using all the available hashing bits. For multiple hashing functions, lower bits are rotate with upper bits to obtain a new hash value. Furthermore, to avoid conflicts in fetching multiple contents out of the BIT, the BIT is partitioned equally according to the number of hashing functions. Our evaluations show that there is little impact on hashing collisions using a unified or a partitioned BIT.

2.5 Performance Evaluation Methodology

Five routing tables from internet backbone routers [6, 54] are selected to carry out the performance evaluations and comparisons. As286 (KPN Internet Backbone), as1103 (SURFnet, The Netherlands), as4608 (Asia Pacific Network Information Center, Pty. Ltd.) and as4777 (Asia Pacific Network Information Centre) are downloaded from [54], using the tables dumped at 7:59am July 1st, 2009. The fifth table as6447 (Oregon-IX Oregon Exchange) was downloaded from [6] in August, 2006. The numbers of prefixes in these five tables are 277K, 279K, 283K, 281K, and 212K respectively.

We compare five hashing mechanisms in routing table lookups including simple direct-mapped hashing (Direct-mapped), randomized hashing (Randomized), hashing with Extended Bloom Filter (Extended), multiple hashing (2Hash), and the new Bloomier hashing (Bloomier). For fast hashing without complicated hardware, Direct-mapped decodes lower-order prefix bits for the hashed bucket. To alleviate collisions, randomized selected the hashing bits by exclusive-oring the lower-order prefix bits with the adjacent higher-order bits. Extended follows the scheme described in [57]. Each prefix is hashed into multiple buckets based on multiple hashing functions and store one copy into the shortest bucket. A shared counter in each bucket along with proper links across buckets allows the search only through the shortest bucket. 2Hash stores each prefix into the shortest bucket based on multiple hashing functions. Multiple buckets must be searched, which incurs not only additional bandwidth, but unexpected delays due to fetching non-adjacent memory blocks. The benefit diminishes in using more than two hashing functions to balance the buckets. Hence, we only consider two simple hashing functions, direct-mapped and rotation of high/low prefix bits. The proposed Bloomier also uses the same two hashing functions to hash to the BIT. The size

(number of entries) of the BIT impacts the balance of the hashed buckets. In our evaluation, we consider the BIT with 8K to 64K entries, denoted as Bloomier-nK, where nK is the respective number of BIT entries.

The number of hashed buckets greatly impacts the number of prefixes in a bucket. Obviously, the larger the number of hashed buckets, the smaller the number of prefixes in each bucket and hence the less the bandwidth is required for each lookup. Nevertheless, increasing the number of buckets negatively increases the total memory size to hold the routing table with the same number of prefixes. In comparing different hash-based routing table designs, we vary the number of hashed buckets from 16K to 2M to evaluate the tradeoff between the maximum bucket length and the overall memory size.

2.6 Performance Results

Figure 2.9 shows the lookup bandwidth and memory space comparisons of the five hash-based lookup schemes collected from simulating the five routing tables. For Bloomier, we include the results of two BIT sizes with 16K and 64K entries. We consider the prefix group of lengths 19-24 with prefix expansions to both lengths 22 and 23 bits for hashing the buckets. After expansions to a length of 22, the numbers of prefixes in this group are 433K, 435K, 438K, 436K, and 318K for the respective routing tables, which are about 70% increases from the original prefixes. When expanding to a length of 23, the numbers of prefixes become 687K, 689K, 693K, 692K, and 508K with about 170% increases. We use these prefix numbers as the basis to calculate the memory expansion ratio for comparing the efficiency of the memory space requirement. Memory expansion ratio is equal to the product of number of buckets and maximum prefixes in a bucket divided by the total number of prefixes.

Note that while the CPE expands the routing table for reducing the number of different lengths, the memory expansion ratio calculates the memory expansion due to allocations of each bucket in a fixed size memory block for achieving a constant lookup rate. We choose the expansion ratio, instead of the size for a uniform comparison across routing tables with variable number of prefixes. The maximum number of prefixes in a bucket determines the bandwidth requirement for each lookup and the memory expansion ratio represents the memory size requirement. Given that the number of hashed buckets is incremented by a power of 2, the expansion ratio of the horizontal axis is plotted on a logarithmic scale. In Figure 2.9, we confine the maximum bucket size and the memory expansion ratio to 60 and 64 respectively.

Several observations are to be made from Figure 2.9. First, as expected, Bloomier shows superior performance when compare to other approaches. It requires the lowest memory size expansion to obtain small prefixes in a bucket for both expansion sizes. This is mainly because of Bloomier's ability to balance the prefixes among hashed buckets through the BIT. Extended and Randomized show little improvement from Direct-mapped. Extended does not perform well when the number of prefixes is much bigger than the number of hashed buckets. In this case, each bucket counter in keeping the length information is incremented multiple times whenever a prefix is hashed to it. Extended does not perform well when the number of prefixes is much bigger than the number of hashed buckets. In this case, each bucket counter in keeping the length information is incremented multiple times whenever a prefix is hashed to it. 2Hash suffers from twice the bandwidth requirement due to the need to

search two buckets. In addition, unlike Bloomier which places each prefix into the shortest bucket, 2Hash must make a choice even if both hashed buckets are long.

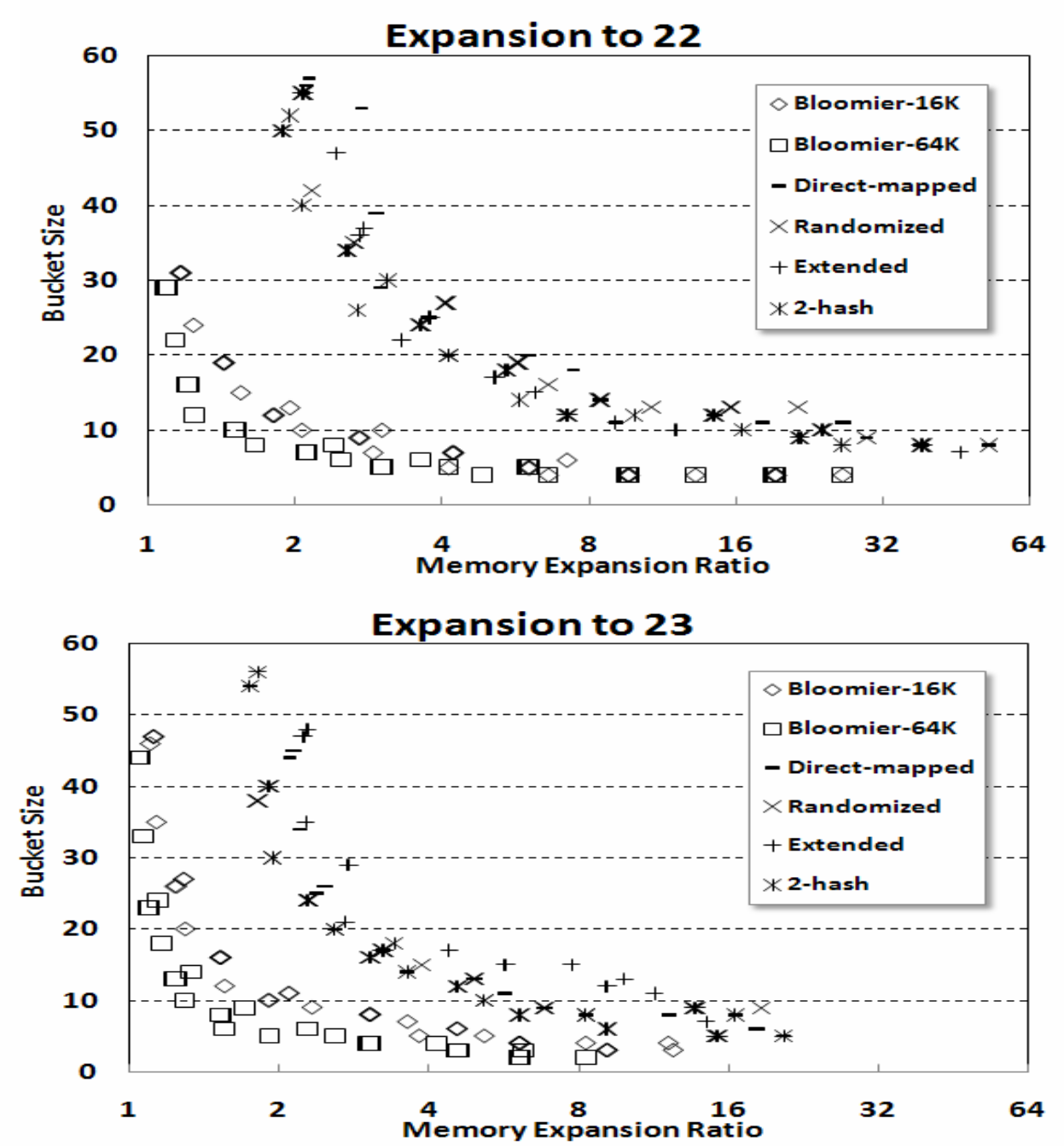


Figure 2.9. Bandwidth and memory space requirement for the five hash-based schemes

Note that Cuckoo [18] or Peacock [39] hashing helps balancing the buckets with multiple hashing, but they require to relocate the prefixes in memory to avoid overflow, and hence are not included.

Second, larger BITs indeed help, especially with small memory expansion. However, to achieve minimum bandwidth requirement with larger expansions, the BIT size makes rather minor differences. A careful sensitivity study on the impact of the BIT size is discussed later.

Third, the bandwidth and memory space requirement of the five hashing schemes are similar between the two partial expansion lengths 22 and 23. However, the lower bounds of the maximum prefixes in a bucket are 4 and 2 respectively for these two expansions. As shown in Figure 2.9, these lower bounds can be obtained with about 6 times of the memory expansion using Bloomier. But, such bounds are unreachable by any other hashing schemes even when the expansion ratio grows to 64. Further discussions in achieving the lower bound is given in the following sensitivity study.

Recall that a combination of partial prefix expansions and coalescing of different lengths of prefixes into the same bucket permits each LPM lookup in one memory access. In Figure 2.10, we show the memory space and bucket size tradeoffs of the prefix group 19-24 with partial expansions to lengths 21, 22, 23, and 24. Note that in Figure 2.10, the memory size is calculated as the product of the number of hashed buckets and the maximum prefixes in a bucket. The Bloomier hashing with a 64K BIT is simulated. The results are obtained from an average of the first four tables. The fifth table is not considered due to its smaller size.

We can make the following observations. As shown in Figure 2.7, the number of prefixes increases with the respective expansion lengths by about 25%, 70%, 170%, and 400%. On the other hand, the lower bound of the maximum prefixes in a bucket decreases with the expansion lengths from 8, 4, 2, to 1. Given enough hashed buckets, such lower bounds can be reached. For example, with 128K, 1M, and 2M hashed buckets, the lower bounds of 8, 4, and 2 prefixes in a bucket are achieved for the expansion lengths 21, 22, and 23. Nevertheless, total memory space must consider both the number of buckets and the maximum prefixes in a bucket. Therefore, the expansion length of 23 with 2M buckets and 2 prefixes in a bucket has the same memory size requirement as the expansion length of 22 with 1M buckets and 4 prefixes in a bucket. Overall, the expansion length of 23 shows the best tradeoff. With a SRAM array capable of holding slightly over 4M prefixes, this expansion can achieve the lower bound of 2 prefixes per lookup. For a lower bound of 4 prefixes per lookup, the expansion length of 23 only needs a SRAM to hold 2M prefixes, instead of holding 4M prefixes for the expansion length of 22. The expansion length of 21 suffers from its large lower bound of 8 unless the bandwidth requirement is not a problem. Although the expansion length of 24 has the minimum lower bound, it requires the number of buckets beyond 2M and incurs huge memory overhead to achieve the lower bound.

Consider a specific design using 500+ MHz QDR-III SRAMs which support 72-bit read/write operations per cycle. A burst read of 2 or 3 cycles can fetch 144 or 216 bits respectively. Since each prefix in this group has 24 bits along with 16 bits for the output port and a few bits for distinguishing the prefix length due to partial prefix expansions, a bust read can fetch 3 or 5 prefixes in 2 or 3 cycles. Considering that each lookup only

requires 2 or 4 prefixes, a constant lookup rate of 250M/sec or 167M/sec is achievable with additional rooms for bigger routing tables and/or longer prefixes.

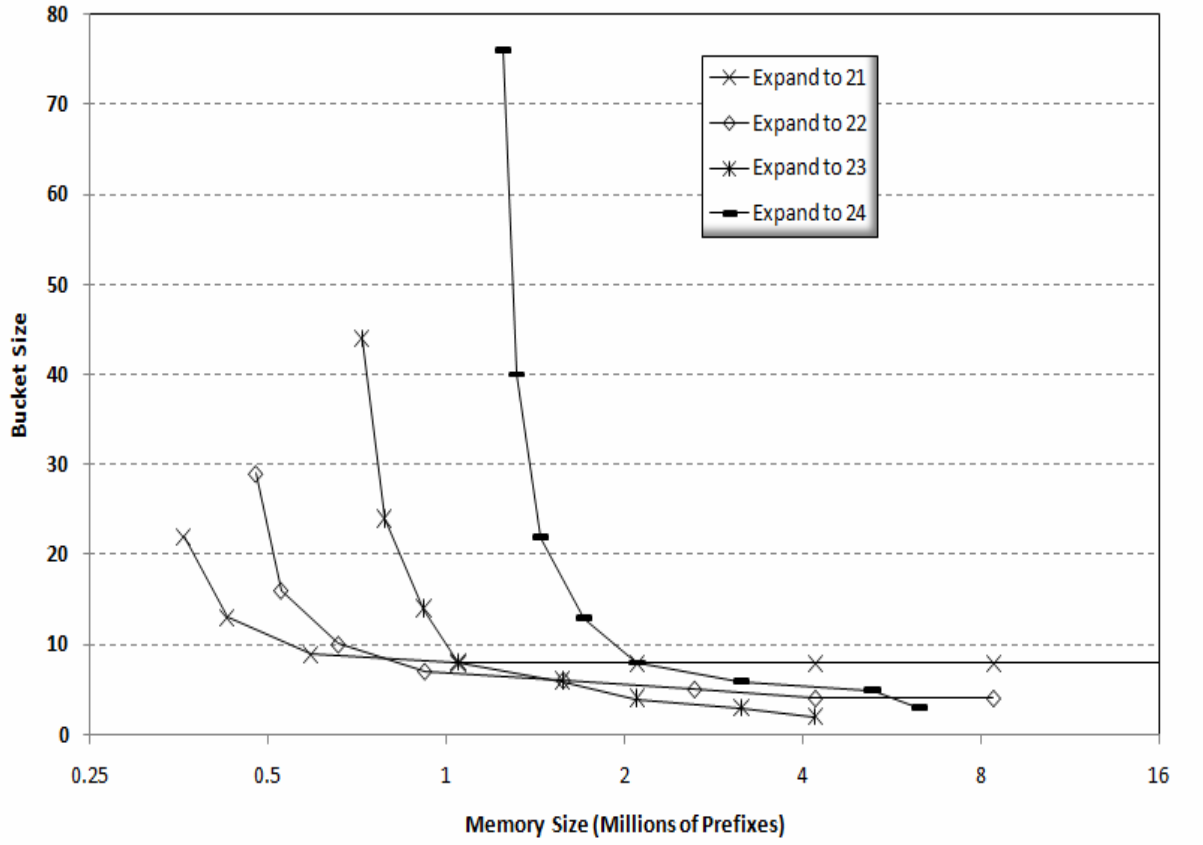


Figure 2.10. Sensitivity on the prefix expansion for different hashing bits

The results of a sensitivity study on the BIT size are given in Figure 2.11. We simulate 4 BIT Sizes with 8K, 16K, 32K, and 64K entries. We only show the expansion length of 22 in Figure 2.11 since the expansion length of 23 has similar behavior. Bigger BIT indeed helps balancing the buckets especially with smaller memory sizes. Note that since the memory size requirement goes with the total number of hashed buckets, the difference between large and small BIT is more evident when the number of buckets is small. However, to achieve the lower bound of prefixes in a bucket, the number of buckets must be sufficiently large. Therefore, the impact of the BIT size is not as significant. For example, except for 8K entries of BIT, the other 3 bigger BIT sizes can

all achieve the lower bound of 4 prefixes in a bucket with 1M hashed buckets. Considering a BIT with 16K entries and each entry saves 22 bits for hashing to 4M buckets, the total on-chip BIT size is only 44KB, which is smaller than a typical L1 cache and can be fitted into on-chip SRAMs.

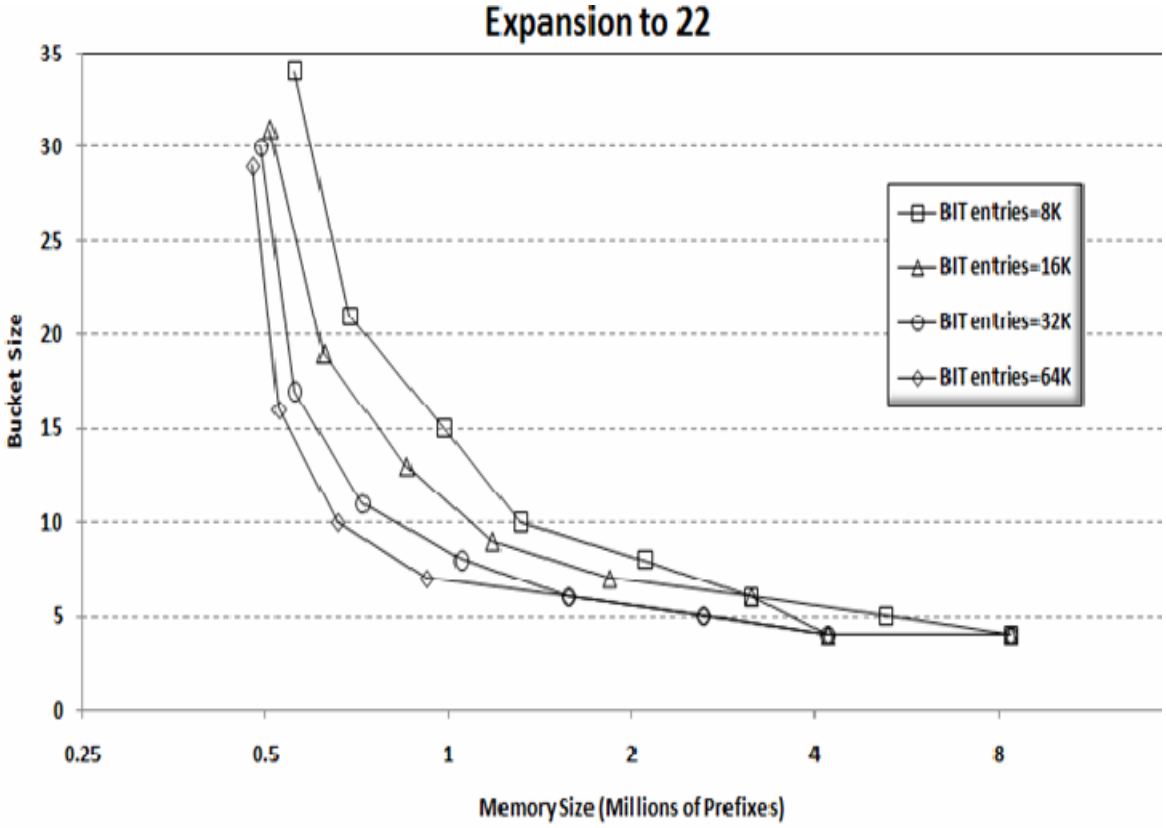


Figure 2.11. Sensitivity on the BIT size

2.7 Related Works

There are three categories of LPM approaches including Ternary Content Addressable Memories (TCAM) [46, 2, 49], trie-based searches [21, 35, 31], and hash-based approaches [10, 25, 48, 57, 26, 18, 39]. TCAMs are custom devices that can search all the prefixes stored in memory simultaneously. They incur low delays, but require expensive tables and comparators and generate high power dissipation. Trie-based approaches use a tree-like structure to store the prefixes with the matched output

ports. They consume low amount of power and less storage space, but incur long lookup latencies involving multiple memory operations that make it difficult to handle new IPv6 routing tables. Moreover, tries-based approaches also required large space overhead for storing pointers. The pointers are store in nodes and used to traverse up and down in the tries.

Hashed-based approaches store and retrieve prefixes in hash tables. It is power-efficient and is capable of handling large number of prefixes. However, the hash-based approach encounters two fundamental issues, hash collisions and inefficiency in handling the LPM function. Sophistic hashing functions [27] can reduce the collision by using expensive hardware with long delays. On the other hand, multiple hashing functions allocate prefixes into the smallest hashed bucket for balancing the prefixes among all hashed buckets [9, 3, 10]. Cuckoo [18] and Peacock [39] multiple-hashing schemes further improve the balance with relocations of prefixes from long buckets. The downside of having multiple choices is that the searches must cover multiple buckets to find the one that contain the correct information. Extended Bloomier Filter [57] places each prefix into multiple buckets and uses a counter to count the number of prefixes in each bucket. Searches are only needed from the shortest bucket. To reduce the space overhead and the length of the buckets, duplicated prefixes are removed. For efficient searches, proper links with shared counters must be established. However, the number of buckets needs to be large to achieve good lookup rate. Hence, the number of entries in the table that store the array of counters can be quite huge.

Handling LPM is difficult in hash-based routing table lookup. To reduce multiple searches for variable prefix lengths, Control Prefix Expansion (CPE) [61] and its

variances [18, 58] reduces the number of different prefix lengths to a small number with high space overhead. Organizing the routing table in a set-associative memory and using common hash bits to allocate different lengths prefixes into the same bucket reduces the number of memory operations to perform the LPM function [33, 34, 18]. However, by coalescing buckets with multiple prefix lengths, it creates significant hashing collisions and hence increases the bandwidth requirement.

Bloom Filter was considered to filter unnecessary IP lookups of variable prefix lengths [19]. Multiple Bloom filters are established, one for each prefix length to filter the need in accessing the routing table for the respective length. Due to uneven distribution of the prefix lengths, further improvement by redistribution of the hashing functions and Bloom filter tables can achieve balanced and conflict-free Bloom table accesses for multiple lengths of prefixes [58]. In these filtering approaches, however, the false-positive condition may cause unpredictable delays. Given the fact that LMP requires searching all possible prefix lengths, none of the above solutions can guarantee a constant lookup rate.

For IP lookups, Chisel [26] introduces the Bloomier hashing idea which is originated from the Bloomier Filter [14]. In their approach, it requires a huge intermediate index table as well as the number of hashed buckets to achieve a conflict-free hashing. Inspired by the original Bloomier filter paper, our Bloomier hashing approach uses the intermediate index table to balance the hashed buckets stored in regular SRAMs. By accepting conflicts, the intermediate index table and the number of hashed buckets can be much smaller. With balanced buckets, both the bandwidth and the space requirement for IP lookups can be reduced.

CHAPTER 3
PAGE TABLE LOOKUP USING INCREMENTAL BLOOMIER HASHING

3.1 Introduction

Today's operating systems use virtual memory. Virtual memory gives an application the impression that it has a large contiguous memory to work with, but in fact, the actual physical memory may be much smaller and fragmented. Programs use virtual memory each time they request memory access via a virtual address. A virtual address needs to translate to a physical address before the data can be fetched. This process is called memory address translation. A page table stores all the mapping between virtual addresses and physical addresses. If the requested memory page is currently located in the physical main memory, then looking up the page table via a virtual address automatically translates to the memory page's physical address. A page table can be implemented in different ways. We describe the conventional ways to implement a page table and their shortcomings. Next, we describe how to implement a page table by using both the Bloomier filter and the Incremental Bloomier hashing approaches. We discuss each implementation's space and time requirements by assuming virtual addresses are 64 bits, physical addresses are 64bits and page size is 4KB. Evaluation results are given after.

3.2 Conventional Page Table Organizations

There are three conventional implementations for page tables: the forward mapping table, the inverted page table and the hashed page table. A forward mapping table utilizes a straightforward method which uses the virtual address to index a series of tables until a translation is found. On the other hand, an inverted page table uses reverse mapping. Indexing an inverted page table with a physical address returns the

entry containing the virtual address that is currently mapped to the physical address. Hashed page table is based on the inverted page table. We describe each of the organizations next.

3.2.1 Forward Mapping Page Table

A forward-mapping or multi-level page table is a collection of tables put in a hierarchical order. Figure 3.1 shows an example of this type of table. In this example, there are three levels of tables. Parts of the bits in the virtual address are used to index each of the levels. The last level contains the leaf pages. The intermediate tables contain pointer. For each valid virtual address, there is an entry on a leaf page that holds the translation [60]. Each leaf page entry also contains extra information such as status and protection bits [28]. The Intel x86 machines are currently using this kind of page table [30].

Time Analysis: For each page table lookup, each level in a table needs one memory access. The total time for a page table lookup in the forward-mapped page table is therefore directly related to the total number of levels. Consider a system that is using a 64 bits virtual address and has a 4KB page size. For the leaf pages, each entry needs to store 8 bytes of physical address and other information. Each entry in the intermediate tables needs to store 8 bytes for the pointer. So for a 4KB page size, the total entries in an intermediate table are 512 entries and 9 bits in the virtual address can be used to index a table. Therefore, a virtual address which excludes the offset is 52 bits and there are 6 levels of tables. Each page table lookup needs to access the memory 6 times.

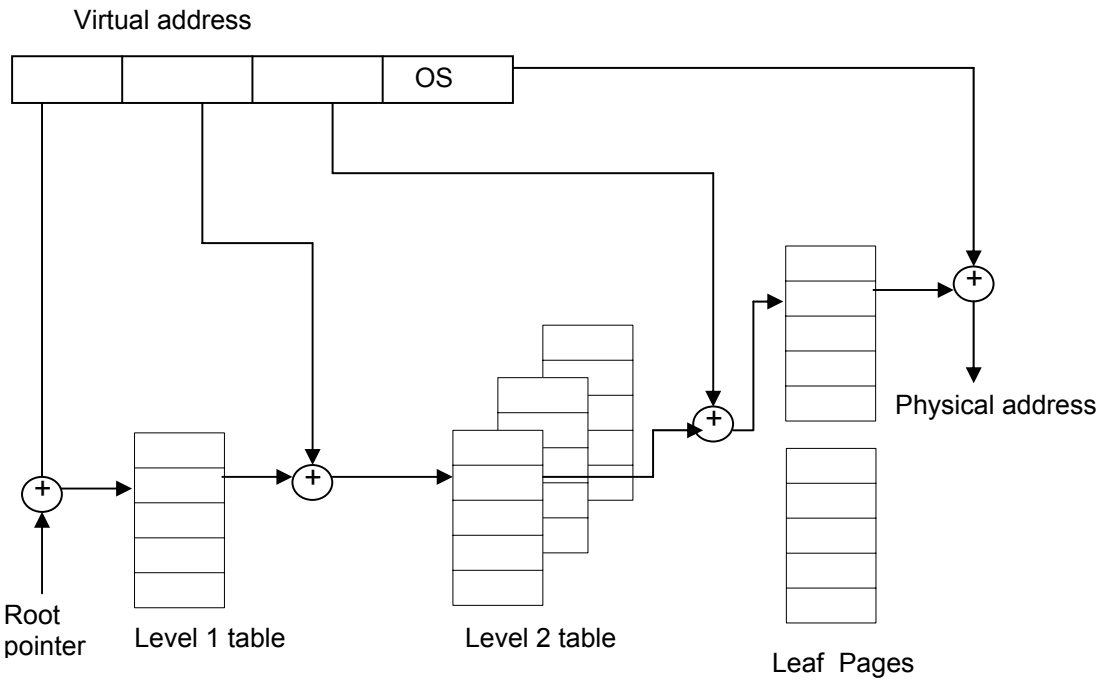


Figure 3.1. Forward-mapped page table

Space Analysis: If every virtual page contains an entry in the page table, then there are 2^{52} entries in the leaf pages. If each entry in the leaf page is 8 bytes, then 2^{55} bytes, 32 petabytes are needed. However, it is unlikely a process uses all 64 bits of address space. To access the leaf pages, five entries in the intermediate tables are accessed. Assuming 8 bytes for each entry in the intermediate tables, then the total space overhead for a leaf page is 40 bytes.

Based on the analysis by using a 64 bit virtual address, the forward-mapped page table requires big space overhead. Moreover, a memory translation requires many memory accesses to reach the leaf page. [5] explored the idea of using cache to accelerate the memory address translation process. However, the method requires dedicated page table cache entry, which can decrease the already limited cache

memory. Therefore, more compact tables that require fewer memory accesses for lookup can increase the performance of the 64bits system.

3.2.2 Inverted Page Table

Another common page table organization is called the Inverted page table (IPT). Many large address space machine used IPTs [41, 11, 29]. When comparing to the forward-mapped page table, an inverted page table is a much more compact data structure. The IPT records all virtual to physical page mappings for those virtual pages that currently reside in memory. Each entry in the IPT contains the virtual address that is currently mapped to the physical address. Since a physical address can use as an index to an IPT entry, the number of entries in the IPT is equal to the numbers of physical pages.

By using virtual addresses as keys, searching is difficult in the inverted page table. Using a brute force method, each entry in the table is examined until a match is found. The worst search time is therefore equal to the total numbers of physical pages. Instead of a brute force approach, one can use hashing as an efficient search strategy. A hash anchor table (HAT) is created alongside with the IPT for this purpose. Each entry in the HAT contains the page frame number that is used to index the IPT. The organizational structure is shown in Figure 3.2.

To insert an entry into the IPT, a part of the virtual page address is randomized by XORing with higher-order bits to produce a hash value. This value is then used to index the HAT. The physical page number is then put into the entry in the HAT. To handle hashing conflicts in a bucket in the HAT, a link is built by using pointer spaces in the IPT. Each pointer in the IPT is also a physical page number that indexes the next member in the linked list.

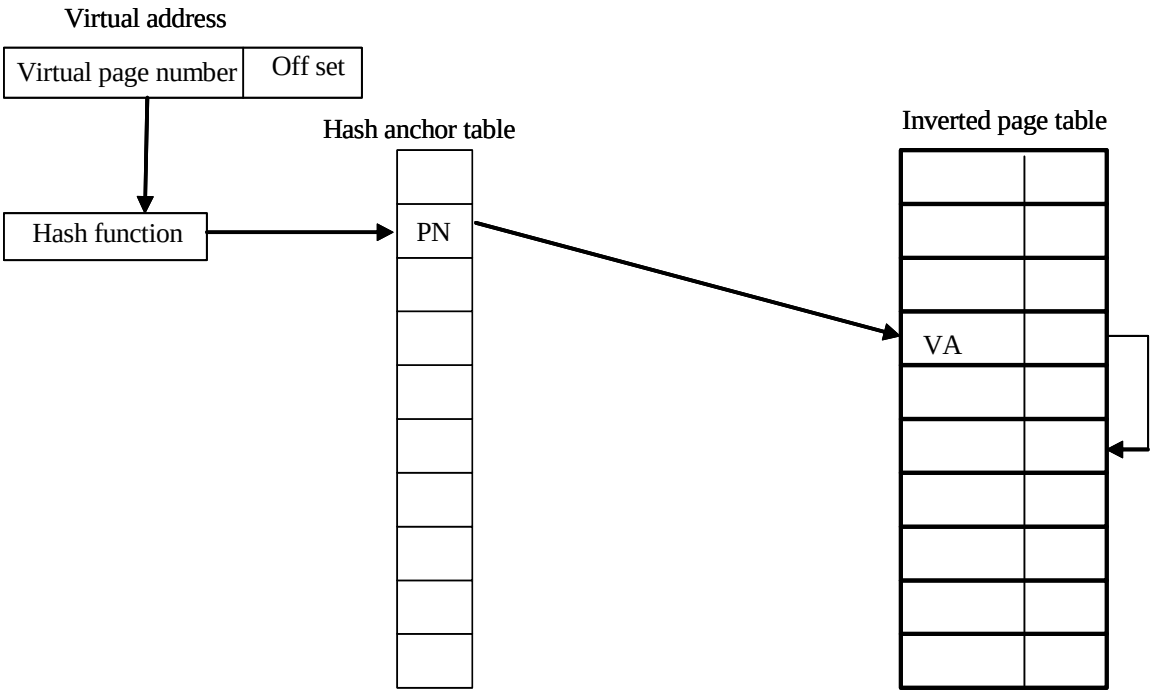


Figure 3.2. Inverted page table with hash anchor table

Time Analysis: During a search, the virtual address is hashed into an entry in the anchor table. Then, the physical address in the entry is used to index another entry in the IPT. If the virtual address matches the entry in the IPT, then the translation is completed. In this case, there are two memory references. However, if there are collisions in the anchor table entry, then the linked list in the IPT is traversed. Therefore, there may be more memory references to other elements in the linked list.

Space Analysis: Each entry in the IPT contains the virtual address tag (8 bytes), a pointer to the next element in the link (8 bytes) and other information (4 bytes). The total number of entries in the IPT is equal to the total number of physical pages. So if the size of the entry in the IPT is 20 bytes and there are 2^{19} physical pages, then the size of the IPT is 12 megabytes. Each entry in the anchor table only needs to store the physical page number, since the index of the IPT can compute from the physical page

number. Therefore, each entry in the anchor table can use up to 8 bytes. The total number of entries in the anchor table is not fixed but should be greater or equal to the number of physical pages. Large anchor table sizes can be used to reduce the average length of the linked lists in the IPT.

For the IPT with anchor table organization, small anchor table or poor hashing methods can create long linked lists in the IPT and affect the overall performance. On the other hand, large anchor tables can create significant space overhead. Hashed page table is introduced to eliminate the anchor table.

3.2.3 Hashed Page Table

The hashed page table is introduced by [28]. The hashed page is built based on the inverted page concept. The hashed page table combines the inverted page table and the hash anchor table into one data structure. Due to the removal of the hash anchor table, fewer memory accesses are needed for page table accesses. However, the page table entry in the hashed page table must now contain both the physical and virtual addresses, since the physical address can no longer be computed from the page table's index. Figure 3.3 shows the hashed page table.

When a TLB (translation lookaside buffer) miss occurs, the virtual page number is hashed into a hashed page table's index. The lookup virtual address is then compared to the virtual address in the page table entry. If the addresses match, then the translation is completed. If the addresses do not match, then the rest of the entry in the chain is used for comparison until either one matches the lookup address or the end of the chain is reached. Page fault occurs when end of the chain is reached without a match.

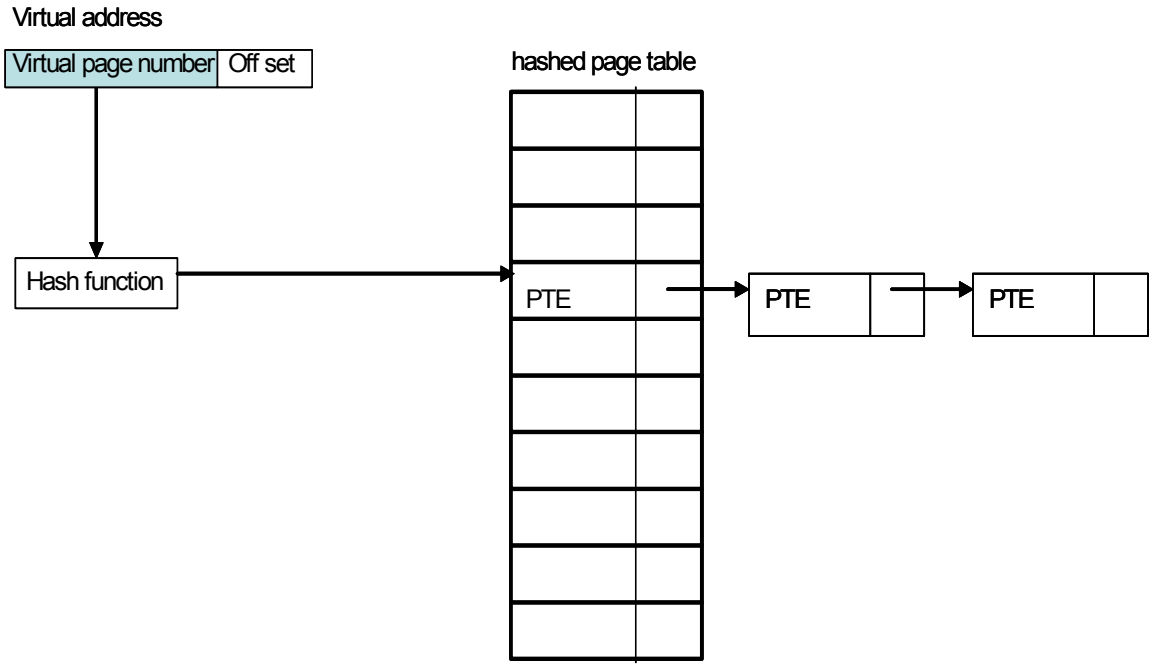


Figure 3.3. Hashed page table

Time Analysis: During a search, the virtual address is hashed into an entry in the hashed table. If the virtual address matches the entry in the hashed page table, the translation is completed. In this case, there is just one memory reference. However, if there are collisions in the table, then there may be more memory references to other elements in the linked list.

Space Analysis: Each entry in the hashed page table contains the virtual address tag, a pointer to the next element in the link, the physical page number and other information. An entry in the hashed page table should contain the virtual address (8 bytes), the physical address (8byte), the next pointer (8 bytes) and other information (4bytes). So if the size of an entry in the page table is 28 bytes and there are 2^{19} physical pages, then the size of the hashed page table is 2^{19} * 28 bytes, or 14 megabytes. The average length of the chain is depending on the size of the hashed

page table. Moreover, different collision resolution methods also play important parts on the average lookup time. Collisions can be chained within the table itself or chained into an overflow table.

3.3 Using Collision Free Bloomier Filter for Inverted Page Table

Inverted page tables with the hash anchor approach use two level data structures to achieve fast lookups while maintaining a compact size. However, due to the limitation of a single hash function, the lookup time can be unpredictable and long in the worst case. Replacing the HAT with a Bloomier filter can solve these problems. Bloomier filters offer collision free hashing [14]. By replacing the hash anchor table with a Bloomier filter table, a constant lookup time on the IPT can be achieved. Moreover, space overhead becomes smaller. The modified architecture is shown below.

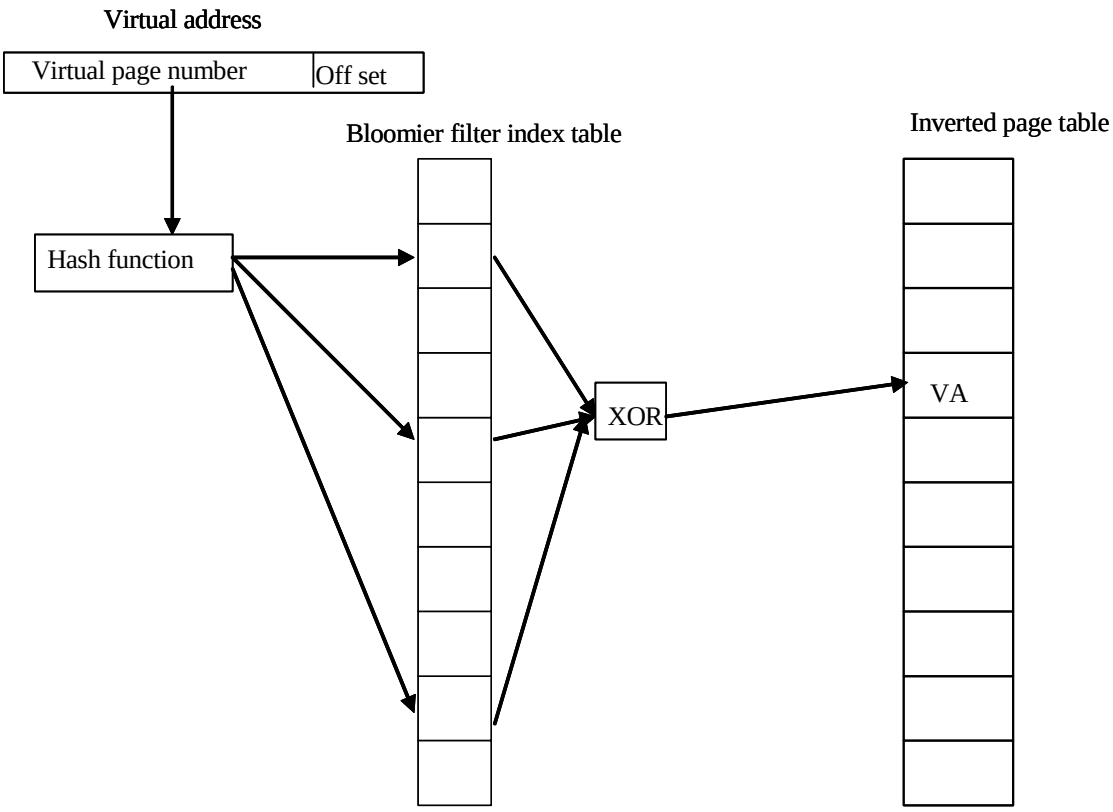


Figure 3.4. Inverted page table with Bloomier filter index table

As seen in Figure 3.4, we replace the HAT with the Bloomier filter index table. The Bloomier filter index table works the same way as the Bloomier filter. The Bloomier filter index table can perform the setup by using all the virtual addresses that are currently mapped to the physical pages as keys. The Bloomier filter table stores the physical addresses, or the indexes to the IPT, for these virtual addresses.

Time Analysis: During a lookup, the virtual address is hashed to a few locations in the index table. The contents of these locations are fetched and XORed into a page address number or index to the IPT. An entry in the IPT is then fetched out using the index. If the virtual address in the IPT entry matches the lookup address, then the translation is completed. Otherwise, the OS initiates a page fault handling. Since Bloomier filters offer collision free hashing, fetching the entry in the IPT only needs one memory access. However, performing lookups on the index table requires multiple memory accesses. But since the accesses are independent, they can be done in parallel in multiple banks memory devices.

Space Analysis: The Bloomier filter table approach guarantees collision free hashing, so no pointer space is needed in the IPT. Therefore, the Bloomier filter index table is the only space overhead. Each entry in the IPT only needs to contain the virtual address tag (8 bytes) and other information (4 bytes). So if the size of the entry in the IPT is 12 bytes and there are 2^{19} physical pages, then the size of the IPT is $12*2^{19}$ bytes, or about 6 megabytes. The size of each entry in the index table needs to be the same size as the physical page address. So, each entry in the index table is up to 8 bytes. The total number of entries in the index table needs to be large enough so that the setup can perform successfully. We show the space requirement for the index table a little later.

When a page fault occurs, the page table deletes the replaced page and inserts the new page. The replacement process may force the Bloomier index table to perform the time consuming setup again. Moreover, page faulty handling also takes a significant amount of time. However, in most applications, page faults happen rarely. Therefore, re-setup is rarely needed. Moreover, some kind of victim cache approach can also mitigate the problem. Next, we show the space requirement for a Bloomier filter index table by considering different parameters.

The Bloomier filter provides collision free hashing. Bloomier filters need to perform a successful setup before being used. The setup process is time consuming and can fail when no singleton is found [14]. When the previous setup fails, setup can be redone with different hash functions. The setup is more likely to succeed if there is enough allocated space for the Bloomier filter. Not allocating enough space can cause the setup to fail and redo many times [14].

The Bloomier Filter has three parameters: M (size of the filter or number of entries in the filter table), N (number of keys) and K (number of hashing functions). Before a Bloomier filter of size M can be used, the filter must be able to setup using K hashing functions for N keys. In Figure 3.5, we try to setup the Bloomier filter by using different combination of parameters. We set the number of keys to be 1000 and 2000.

For each combination of the parameters, 100 runs were performed using randomly generated keys. The setup fail percentage is obtained by the number of times setup failed divided by the total number of setups performed. For each new run, we generated new hash values for each key. There are several observations that can be made from Figure 3.5. First, as expected, larger filter size increases the likelihood that

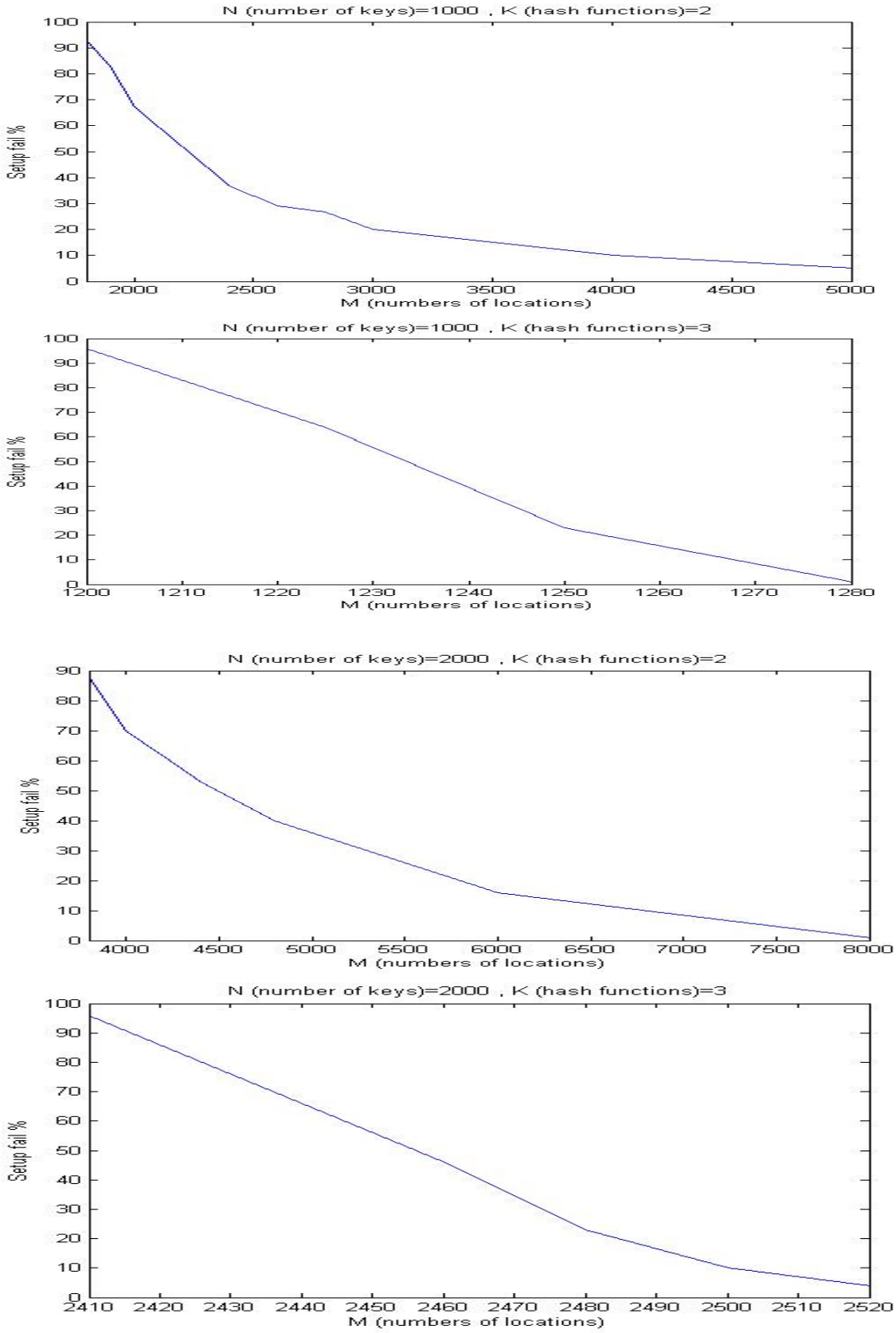


Figure 3.5. Bloomier filter setup failure percentage for different parameters

the setup can perform successfully. Second, more hash functions can reduce the space needed to complete the setup. For example, when N is equal to 1000, M is greater than 2250 and with two hash functions, we can obtain a success rate of more than 50%. On the other hand, if we use three hash functions, we can obtain the same setup success rate when M is greater than 1235 and N stays the same. Moreover, by using more hashing functions, the setup failure probability drops into a much smaller range than using less hashing functions. For example, when using N equal to 1000, M close to1800 and two hashing functions, the setup failure probability is greater than 90%. To get a setup failure probability less than 10%, M needs to be close to 5000 and other parameters stay the same. If we using the same N value with three hashing functions, then the setup failure probability is greater than 90% when M is around 1200. To get a setup failure probability less than 10%, M only needs to be close to 1280.

From the results, we can see that the Bloomier filter size does indeed affect the setup success rate. However, by using more hashing functions, less space is required for the filter. We can see that by using three hashing functions and setting the size of the filter to be larger than 1.3 times the size of the key set, the setup can be performed successfully most of the time.

Back to the IPT comparison, the hash anchor table IPT approach requires extra space overhead in the IPT. These spaces are used for the pointers in the linked list. Hash anchor table (HAT) by itself requires extra space. On the other hand, the Bloomier filter table approach guarantees collision free hashing. Therefore, no pointer space is needed in the IPT and the Bloomier index table is the only space overhead. Using this information, the Table 3.1 compares the space and the average linked list

length between the two approaches. In Table 3.1, the value N is the number of physical pages.

Table 3.1 shows that if the hash anchor table (HAT) has the number of entries equal to N, then the average linked list length in the page table is 1.5. For HAT size of 2N, the average linked list length is 1.25 [36]. Moreover, IPT with anchor table needs extra spaces to store pointers. Therefore, there is another N space which is contributing to space overhead. The total space overhead is shown in the last column in the table. On the other hand, by using Bloomier filter indexing tables which contains 1.4 N to 2N entries and three hashing functions, we can setup the Bloomier filter with an IPT easily. We can use simple hashing such as rotating or XORing lower bits with higher bits in a virtual address. For better hashing coverage, we can also use the hardware hashing function that was described in [53]. Most importantly, the lookup time in the page table for the Bloomier filter approach is constant one lookup. This can be very important for some real time application or machines [67, 68]. The Bloomier filter approach can still have problems when applying to this application. Section 3.4 introduces the Incremental Bloomier hashing which tries to solve the problems.

Table 3.1. Space overhead and average linked list length comparisons between Bloomier Filter and hash anchored inverted page table

Scheme	Index or anchor table size	Average link list length	Space overhead in IPT	Total space overhead
HAT	1N	1.5	1N	2N
HAT	2N	1.25	1N	3N
Bloomier Filter	1.4N to 2N	Constant 1	none	1.4N to 2N

3.4 Using Incremental Bloomier Hashing for Hashed Page Table

Although the Bloomier filter inverted page table approach can eliminate collisions in the IPT, the Bloomier filter index table itself may still be too large. Moreover, updating is not easy. Therefore, we can relax the non-collision constraint and use the Bloomier hashing described in Chapter 2 instead. However, like the Bloomier filter, the normal Bloomier hashing previously described also needs to perform setup before it is used. For the network routing table application, all the prefixes are known before setting up the routing table. Therefore, performing an index table setup is straightforward. For the page table applications, we use virtual addresses as keys. Unlike the routing table application (where all the keys are known before the routing table is used), virtual addresses are highly dynamic and unknown before they are used. Therefore, performing index table setup during program execution may not be feasible. We introduce the new hashing method called incremental Bloomier hashing—which requires no table setup. The pseudo codes are described below. The terminology is the same as described in Section 2.2.

```
PageTableAccess
        CheckPageTable (V)
        If (Translation does not exist)
                InsertPageEntry(P,V)
    END
```

Figure 3.6. Psuedocode for page table access

Figure 3.6 shows the Psuedocode for page table accesses. Page table accesses go through two levels tables like in the inverted page table. The first level is the index table, which is initialed with random value for each entry. The second level is the page

table, which contains translations for all the physical memory pages and is empty in the beginning. During the access, the function CheckPageTable(**V**) is first called. This function uses the virtual page address **V** as the input and performs a lookup on the page table. CheckPageTable(**V**) returns the correct translation if **V** matches a page table entry in the page table. Otherwise, CheckPageTable(**V**) returns translation not found error and a page fault is occurred. In this case, the OS handles the page fault. After the page fault handling, the OS returns with a pair of addresses **P** and **V,** which need to be inserted into the page table. To do so, the function InsertPageEntry(**P,V**) is called. We discuss the two important functions below.

```
CheckPageTable (V)
        Hash V into two location l1 and l2 in the index table
        Compute the page table index ID by XORing l1 and l2
        Go to the bucket using the index ID and traverse the linked list
                For each entry, compare the Virtual page tag Vt to V
                If Tag Vt match V
                        Return the translation
        If End of linked list is reached
                Return Not found
End
```

Figure 3.7. Psuedocode for check page table

Figure 3.7 shows the psudocode for CheckPageTable (**V**). CheckPageTable (**V**) takes one parameter **V.** The function returns the translated physical page address for **V** if **V** matches a page table entry. Otherwise, a translation not found error is returned. The function first hashs **V** into two locations l_1 and l_2 in the index table. The contents of these two locations are XORed into an index **ID** in the page table just like in the normal Bloomier hashing decoding process described in Section 2.2. Using the index, the

linked list started at the index **ID** is traversed. If **V** matches any of the page table entry in the linked list, then the translation is found. On the other hand, if the end of the linked list is reached, then **V** is not currently in the main memory and the not found error is returned. In this case, the function InsertPageEntry(**P,V**) is called. We describe the details for the function below.

```
InsertPageEntry(P,V)
    If Physical page address already exists, delete the old entry
    Hash V into two location l1 and l2
    Compute the page table index D by XORing l1 and l2
    If the indexed bucket in the page table is empty
        Insert the translation P V into the bucket
        Marked  l1 and l2 as used
        Return
    If the indexed bucket in the page table is not empty
        Check if l1 or l2 are occupied by other keys
        If one of l1 or l2 are not occupied
            Choose an empty bucket B in the page table
            Encode the index of B into l1 or l2 whichever is not occupied
            Marked both l1 and l2 as occupied
        Else
            Insert into the linked list start at D
            Return
    End
```

Figure 3.8. Psuedocode for insert page table entry

Figure 3.8 shows the psuedocode for InsertPageEntry. This function is called when a page fault is encountered, and a new translation is needed to insert into the page table. The function accepts two arguments **P** and **V. P** is the physical address and V is the corresponding virtual address. The function checks to see if **P** already exists in

an entry in the page table. If so, the old entry that contains **P** is deleted, since it contains an old translation. The virtual address **V** is hashed into two location l_1 and l_2 in the index table. The content of these two locations are XORed into an index **ID** in the page table just like the normal Bloomier hashing decode process described in Section 2.2. The bucket at location **ID** is checked to see if it is empty. If the bucket is empty, then we simply put the translation in that bucket. If the bucket is not empty, then we try to place the new translation in an empty bucket instead. To do so, we check to see if l_1 or l_2 are occupied by other keys. If l_1 and l_2 are both occupied, then we have no choice but to put the translation in the non empty bucket. We then use some form of collision resolution methods to handle the overflow. If l_1 or l_2 is not occupied, then we choose an empty bucket **B** in the page table and encode the index of **B** into the unoccupied location in the index table. The encoding process is the same which is described in Section 2.2. After encoding is done, both l_1 and l_2 are marked as occupied.

The incremental Bloomier hashing requires no index table setup. Moreover, lookup is straightforward. Figure 3.9 uses randomly generated numbers to show average linked list length comparisons between the incremental Bloomier hashing scheme and normal hashing. The numbers are generated by using the standard C++ pseudo-rand integral number generator. We chose to compare the average linked list length because the average linked list length is directly related to the average lookup time in a page table. We consider the average lookup time be one of the most important measurements in page table performance.

Figure 3.9 compares the average linked list lengths of the normal hashing scheme and our incremental Bloomier hashing scheme at three different index table sizes. The

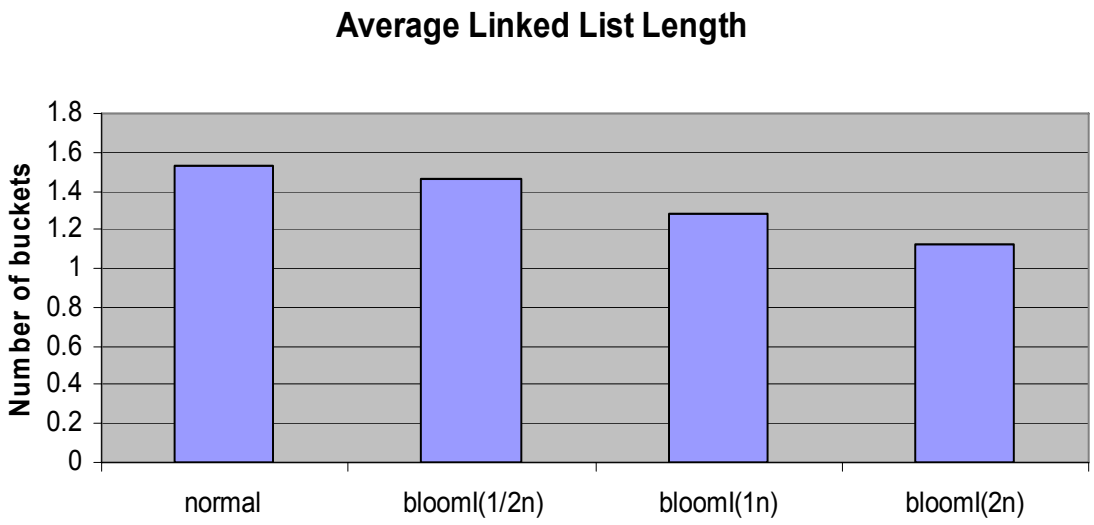


Figure 3.9. Average linked list length comparisons between normal hashing and Incremental Bloomier hashing with randomly generated numbers

size of the hash table is 64K entries. We use different hashing methods to hash 64K randomly generated numbers into the table. For the incremental Bloomier hashing scheme, the three index table sizes are as follows: half the size of the page table, the same size as the page table and double the size of the page table. Figure 3.9 shows that our scheme can balance the hash buckets and reduce the average linked list length. A shorter average linked list length means a shorter average search time and better hash table performance. From Figure 3.9, we can also see that— just like the normal Bloomier hashing— increasing the index table size indeed helps balance the buckets. If we use an index table size of 32K entries, we can reduce the average search time by a small amount. But if we use an index table size of 128K, we can reduce the average search time by a large amount.

After the discussion on the incremental Bloomier hashing algorithms, the modified architecture is shown below.

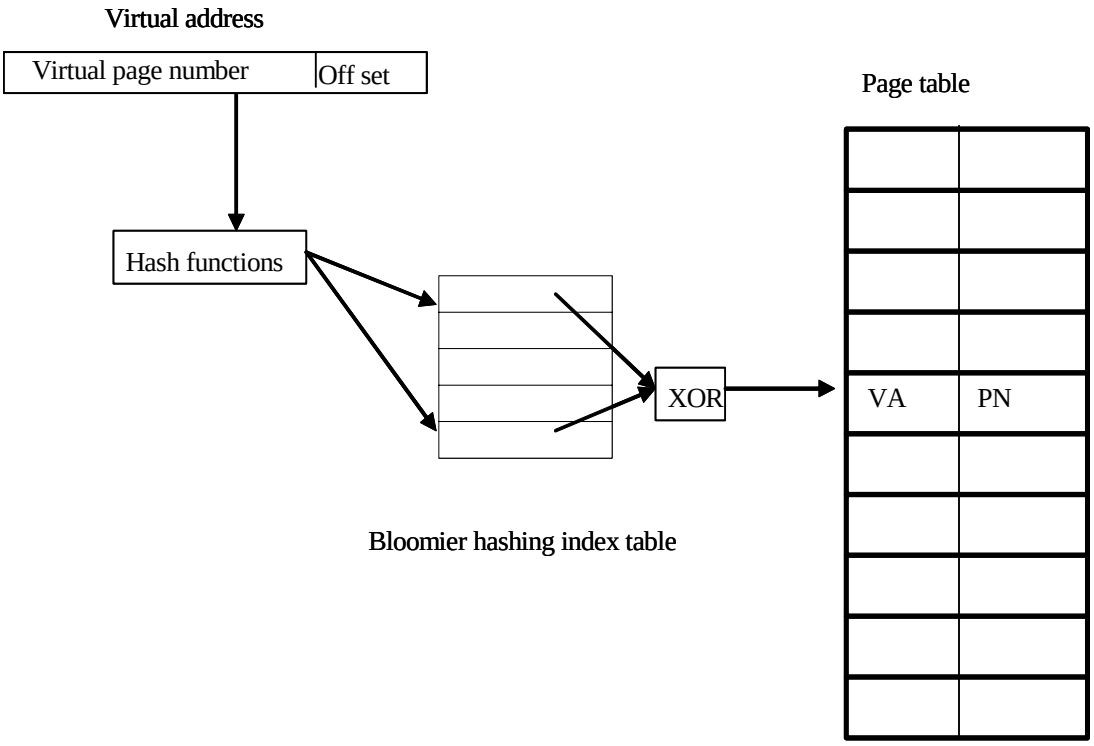


Figure 3.10. Hashed page table with incremental Bloomier hashing index table

As seen in Figure 3.10, instead of using the inverted page table, we use a general hashed page table. The hashed table is similar to the one described in [28]. Each entry in the hash table contains both the virtual address and physical page number since the physical page number can no longer be computed from the entry's index. Comparing this to our previous Bloomier Filter table approach, the Incremental Bloomier hashing index table approach uses fewer hashing functions in the index table. We can also update the index table a lot easier. The lookup operation is very similar to the previous Bloomier filter approach. By using the virtual address and the hashing functions, we fetch out two locations in the Bloomier Hashing index table. The contents of the two locations are XORed to produce an index. By using this index, we fetch an entry in the page table and compare the virtual address tag with the lookup address.

Unlike the Bloomier filter approach which guarantees collision free hashing, collisions in the page table are once again possible when using our incremental Bloomier hashing method. When two or more items are hashed into the same bucket, collisions are occurred. There are many possible ways to handle the collisions in the page table. Coalesced hashing, separate chaining and the set associated approach are the common and well known methods for handling collision [36]. Coalesced hashing does not use extra space overhead. On the other hand, both separate chaining and the set associated approach use extra space overhead to achieve shorter linked list length. Each of the collision resolution methods has its advantages and disadvantages. We examine each method below.

3.4.1 Coalesced Hashing Approach

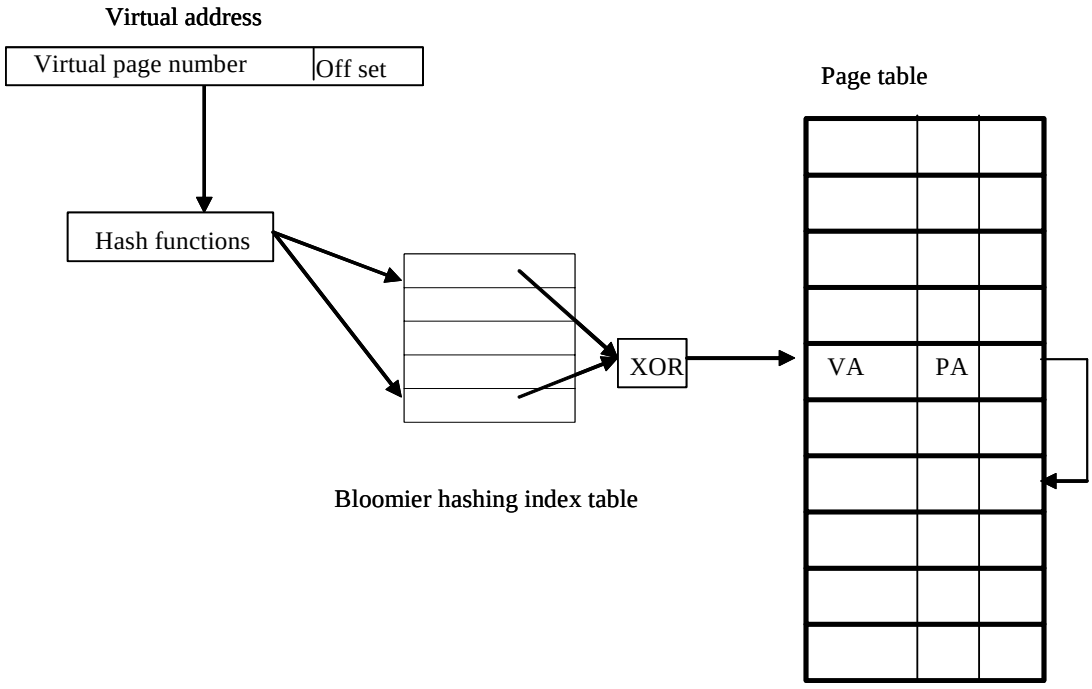


Figure 3.11. Hashed page table with incremental Bloomier hashing index table (Coalesced)

Figure 3.11 shows a page table organization that uses a modified version of coalesced hashing [36]. Coalesced hashing uses ideas from two other common hashing schemes, separate chaining and open addressing hashing [36]. In coalesced hashing, whenever collisions occur in a bucket, the first empty bucket in the table is used to hold the overflow item and a link to the index of the overflow bucket is put into the collision bucket. Since our incremental Bloomier hashing scheme does not guarantee collision free hashing, we use the coalesced hashing idea to handle collisions. If a translation is hashed into a bucket in the page table that is already occupied by another translation, we then randomly choose an empty bucket in the page table and put the overflow item in the bucket. We call this the overflow bucket. We then link the overflow bucket to the collision bucket using a pointer. However, since the collision bucket's pointer field may already be occupied by a previous overflowed item, we may need to traverse the linked list to find an empty pointer field and put the pointer in there.

Time Analysis: First, we access two entries in the index table in parallel. Then, like the IPT with anchor table approach, we search the page table by traversing the linked list until the virtual address tag in a page table entry matches the lookup address. If the end of the linked list is reached, then a page fault is occurred. Therefore, at least one memory reference on the page table is needed. The average length of the linked list depends on the size of the index table. We show the performance difference for different index table sizes in Section 3.6.

Space Analysis: Each entry in the hashed page table contains the virtual address tag, a pointer to the next element in the link, the physical page number and other information. Coalesced hashing has the advantage that no bucket in the hash table is

wasted. Because of this, the coalesced hash table is a very compact structure like the inverted page table. By setting the size of the page table equal to the number of physical pages, all entries in the coalesced page table are used. An entry in the page table should contain the virtual address (8 bytes), the physical address (8 bytes), the next pointer (8 bytes) and other information (4 bytes). So if the size of an entry in the page table is 28 bytes and there are 2^{19} physical pages, then the size of the hashed page table is 2^{19} * 28 bytes, or 14 megabytes. Like IPT with Bloomier filter index table approach, each entry in the index table is up to 8 bytes, since the number of buckets in the page table is still equal to the number of physical pages. However, by using hashed page table, we can create index table and page table for each process. For example, an entry in the page table should contain the virtual address, the physical address, other information and a pointer. The size of the entry should be around 24 bytes. If a process uses 256 megabytes of memory and the page size is 4 kilobytes, then there are 64 thousands or 2^{16} entries in a per process page table. The size of the index table entry is depended on the number of entries in the hashed page table. So, the size of an index table entry will only be around 2 bytes, which is only 1/12 the size of a page table entry. Therefore, both the index table and the hashed page table can become smaller if they are pre-process tables. Once again, the index table size has a direct effect on the average length and worst case length of the linked lists. However, the index table size is only a fraction of the page table's size. We show some simulation results with different index table sizes in Section 3.6.

Coalesced hashing can achieve a very compact page table with no wasted space. However, the average case and worst case linked list length can become long due to additions of items to the already present linked list. An example is shown below:

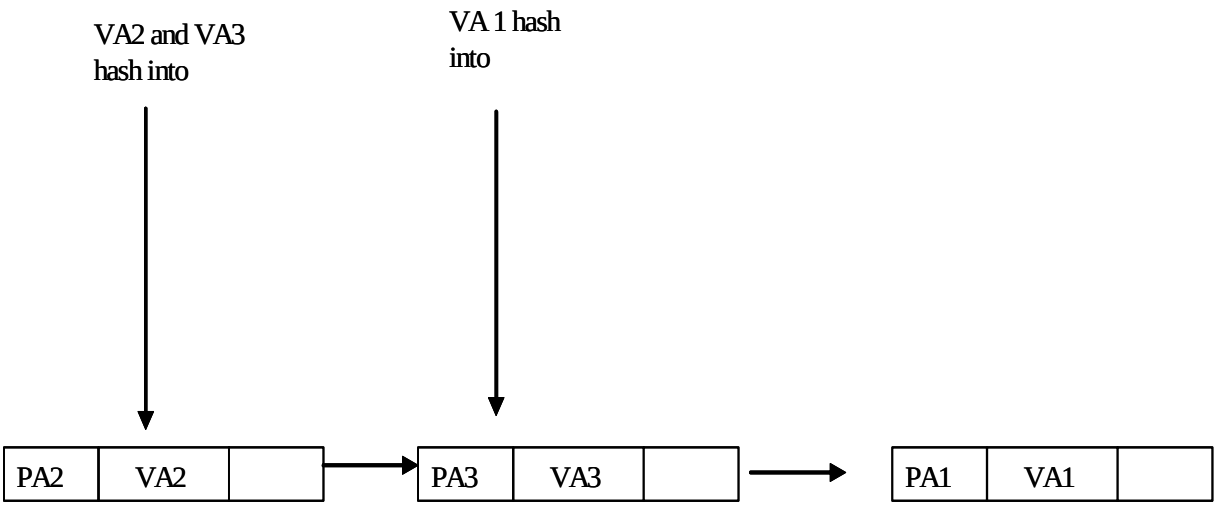


Figure 3.12. Example of shared linked list in coalesced hashing

In this example, the keys or virtual addresses VA2 and VA3 are hashed into the same bucket. Due to the collision, VA3 is then put into an empty bucket 2 with its physical address PA3. The linked list starting at bucket 1 has length of two at this point. Sometime later, VA1 is hashed into Bucket 2. But since Bucket 2 is already occupied by VA3, VA1 is then placed into Bucket 3. The linked list starting at bucket 2 has two items at this point. However, if we traverse the linked list starting at bucket 1, then there are three items. So, using coalesced hashing can increase both the average and worst linked list length. This problem can be solved by using extra overflow areas.

The separate chaining approach and the set associative approach are two collision resolution approaches that use extra overflow area to handle collisions. The separate chaining approach stores overflow items by allocating new table entries in memory space and chaining them together. Each entry in a link is chained together by

using pointers. On the other hand, the set associative approach allocates extra spaces to handle the overflow in advance.

3.4.2 Separate Chaining Approach

Figure 3.13 shows the Bloomier hashing separate chaining approach. In this approach, each bucket contains a linked list for overflow items. Each linked list can grow by allocating a new page table entry in memory and chaining it to the linked list.

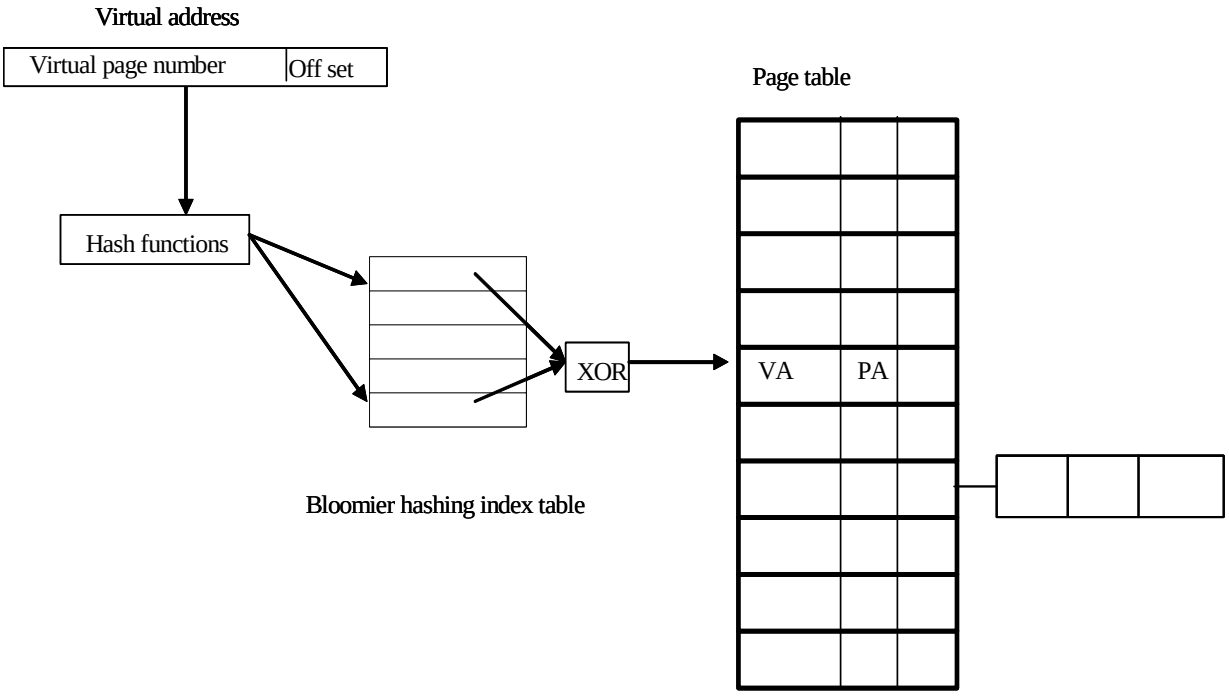


Figure 3.13. Hashed page table with incremental Bloomier hashing index table (Separate Chaining)

Time Analysis: First, we access two entries in the index table in parallel. Then, like the IPT with anchor table approach, searching in this approach is performed by traversing the linked list associated with each bucket until a match is found. If we reach the end of the list, then a page fault is occurred. The average length of the linked list is dependent

on the size of the index table. We show the performance difference for different index table sizes in Section 3.6.

Space Analysis: The separate chaining approach is space efficient. A new page table entry is allocated on an as-needed basis when collision occurs. Therefore, the size of the page table is not fixed. The chains can grow or shrink at run time. An entry in the page table should contain the virtual address (8 bytes), the physical address (8 bytes), the next pointer (8 bytes) and other information (4 bytes). So if the size of an entry in the page table is 28 bytes and there are 2^{19} entries in the table, then the size of the hashed page table is at least 2^{19} * 28 bytes, or 14 megabytes. Like the IPT with Bloomier filter index table approach, each entry in the index table is up to 8 bytes, since the number of buckets in the page table is still equal to the number of physical pages. However, by using hashed page table, we can create index table and page table for each process. Therefore, both the index table and the hashed page table can become smaller if they are pre-process tables. Once again, the index table size has a direct effect on the average length and worst case length of the linked lists.

This separate chaining approach has poor cache performance because the next member of the linked list is likely not in the same physical page as the previous member of the list. This forces another page to be fetched from the memory and can cause additional page faults. However, if we know the worst case linked list length, we can allocate extra space in advance for all the collisions. We can use the set associative approach like our Bloomier hashing design for IP lookup in the Chapter 2.

Figure 3.14 shows the incremental Bloomier hashing set associative approach. In this approach, all the items that are hashed to the same bucket are put into a set. Each

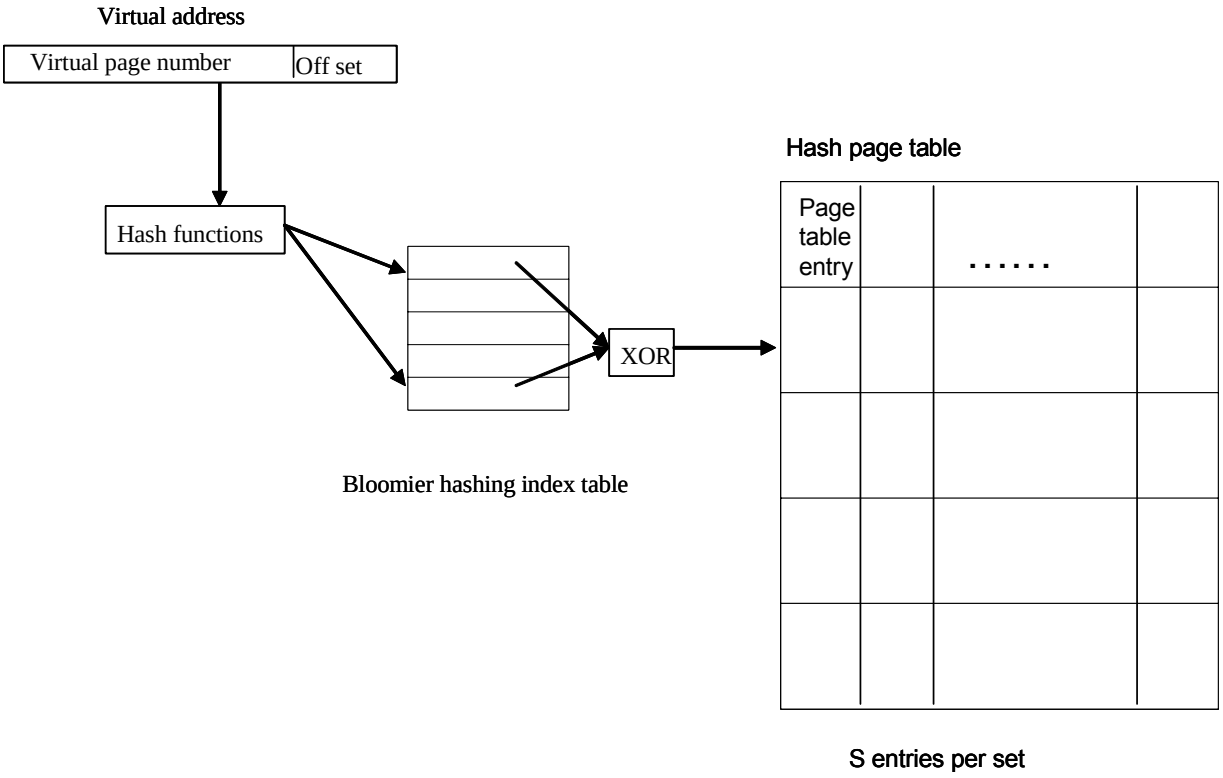


Figure 3.14. Hashed page table with incremental Bloomier hashing index table (Sets associated)

bucket in the page table is a set of page table entries and the number of entries per set is fixed. Since the page size is fairly large compared to the size of the page entry, the number of entries per set can also be large. The number of entries per set is needs to be sufficiently large to handle the worst case collisions. In an update, unlike the separate chaining linked list approach where we can always allocate the extra nodes, if the number of collisions in a bucket is bigger than the max entries per set, then the page table may need to increase the size for all the set. This can cause problems because all the previous page table entries may need to allocate to different physical pages which can cause long delay.

Time Analysis: Once again, we access two entries in the index table in parallel. However, unlike the separate chaining approach, all elements in the same set are in the same page. Therefore, only one memory access is needed to fetch out the entire set. Lookup is then done by comparing each entry in the set to the lookup address.

Space Analysis: Since the page size is fairly large compared to the size of the page entry, the number of entries per set can also be large. The number of entries per set needs to be sufficiently large to handle the worst case collisions. Since the number of entries per set is fixed, some of the entries in some sets may not be used at all. An entry in the page table should contain the virtual address (8 bytes), the physical address (8 bytes), and other information (4 bytes). So if the size of an entry in the page table is 20 bytes and there are 2^{19} entries in the table, then the size of the hashed page table is 2^{19} * 20 bytes, or 12 megabyte for one entry per set. Assuming the number of entries per set is four, the total size of the page table is then 2^{19} * 20 * 4 megabytes, or 48 megabytes. Like the IPT with Bloomier filter index table approach, each entry in the index table is up to 8 bytes, since the number of buckets in the page table is still equal to the number of physical pages. However, by using hashed page table, we can create index table and page table for each process. Therefore, both the index table and the hashed page table can become smaller if they are pre-process tables. The index table size has a direct effect on the worst case length of the linked lists. The worst case length determines the numbers of entries per set.

Overall, the set associative approach is much less space efficient than the separate chaining approach but can have better lookup performance.

3.5 Performance Evaluation Methodology:

Many previous studies [32, 45, 7] on memory behaviors used the Simics full system simulator. For the experiment, we also ran a Virtutech Simics 3.0 simulator [44] on an x86 target machine. The target machine used Linux operating system. We chose eight workloads in the SPEC CPU 2000 and SPEC CPU 2006 benchmarks. These eight workloads are Swim, Applu, Gcc06, Apsi, Wupwise, Milc, Lbm and CactusADM. Table 3.2 shows the footprints for these workloads. The footprints are measured in a period of 10 billion instructions for each workload.

Table 3.2. Footprints for eight SPEC 2000/2006 workloads

	Swim	Applu	Gcc06	Apsi	Wupwise	Milc	Lbm	CactusADM
Footprints (in MBs)	178	175	127	125	178	430	404	412

For the workloads Swim, Applu, Gcc06, Apsi and Wupwise, we ran them using a target machine with 128 Megabytes of memory. Since the workloads Milc, Lbm and CactusADM had footprints much greater than 128 Megabytes, we ran them using a target machine with 256 Megabytes of memory. Comparing to the real machines these days, the simulated machines contain small amount of memory. However, the simulated machines only run one process which is the workload most of the time while the real machines run multiple processes at the same time. If we use large memory size, then the results won't able to show the performance different between different page table schemes. The page size was 4k for all machines.

For each machine, we also included a fully associated data TLB and an instruction TLB. Both TLBs had 32 entries. We chose to use small TLBs because we like to better

test the page table performance. Large TLBs can decease the effectiveness of a page table's performence. For each memory reference, Simics output both a virtual address and its physical address. Using the virtual address, the simulator first performed a lookup on the TLB to see if the TLB contained the translation. If the TLB did not contain the translation, then the simulator performed a page table lookup. If the page table lookup succeeded, then the translation is inserted into the TLB. But if the page table lookup failed, then a page fault occurred. After the operating system handled the page fault, the virtual address and the corresponding physical address were inserted into both the page table and the TLB.

The simulation ended when eight million TLB misses occurred for each workload. Section 3.6 compares our schemes to other schemes. The other schemes include the normal inverted page table (IPT) with the anchor table and the hashed page table. For faster hashing without complicated hardware, one hash function decodes lower-order virtual address bits for the hash bucket. Since our scheme requires more than one hashing function, we use the rotation method, which rotates the lower order bits with higher order bits to obtain the second hash value. For our incremental Bloomier hashing scheme, we include results with different index table sizes. The size of the page tables is equal to the number of physical pages in the machines.

3.6 Performance Results

One of the most important measurements of page table performance is the average access time for page table hits. We show the average access time for the page table hits below. For each workload, the average is calculated over a period of eight million page table accesses. Of these eight million accesses, over 98% are hits. The table below shows the results for two collision resolution methods. Using our

Bloomier hashing scheme, results for three different index table sizes are also shown. We first show the results for the separate chaining hashed page table. Then, we show the results for the coalesced hashed page table.

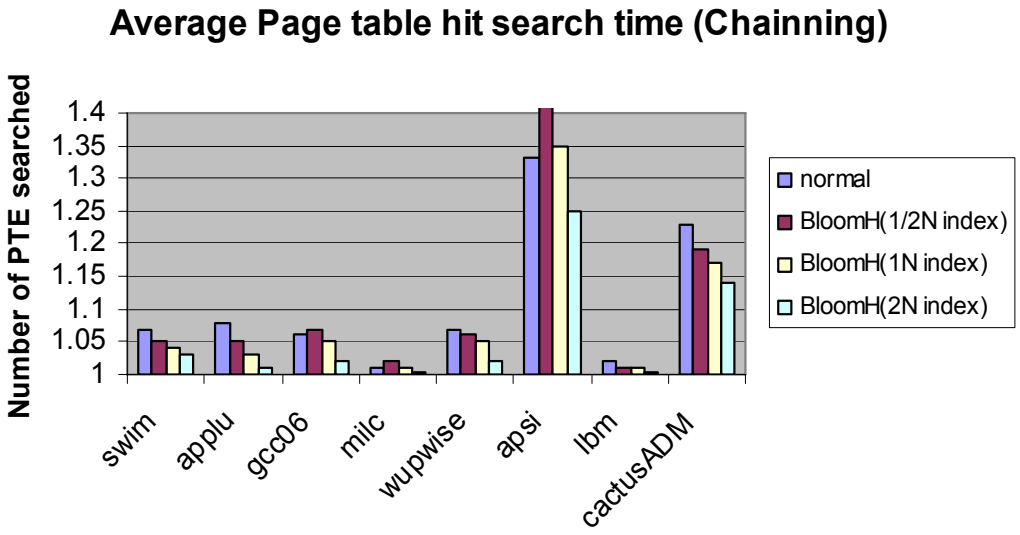


Figure 3.15. Average page table hit search time comparison by using separate chaining hashed page table

Figure 3.15 compares the average search time for different methods by using separate chaining hashed page table. The time measurement is calculated in terms of the number of page table entries searched before the translation is found. N is the size of the page table. For the workloads that run on a 128MB machine and 4K page size, the size of the page table is 32K entries. For the workloads that run on a 256MB machine and 4K page size, the size of the page table is 64K entries. Comparisons are made between the hashed page table approach and our incremental Bloomier hashing approach (BloomH) at three different index table sizes. For our incremental Bloomier hashing approach, there are 3 sizes for index table: half the size of page table (1/2N), the same size as the page table (1N) and twice the size of the page table (2N). From

Figure 3.15, we can see that our incremental Bloomier hashing scheme does indeed decrease the average lookup time for the page table. When using the index table that is the same size as the page table, all but one workload has a smaller average access time as compared to the normal hashing method. If we double the size of the index table, the improvement is even greater. When using a smaller index table size (1/2N), there is still improvements for five out of the eight workloads.

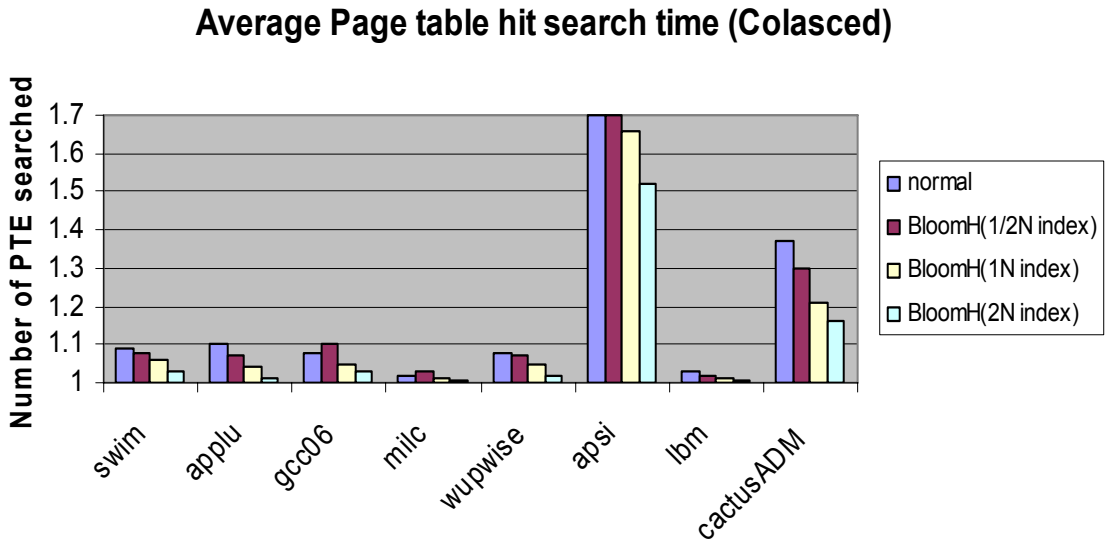


Figure 3.16. Average page table hit search time comparison by using coalesced hashed page table

Figure 3.16 compares the average search time for different method by using the coalesced hashed page table. When using the coalesced page table, collisions are chained within the page table itself. Due to this fact, the page table using coalesced method for collision resolutions can have long linked lists. Figure 3.16 indeed shows that the average lookup times are worse than the separate chaining hashed page table for most cases. Nevertheless, when comparing to the normal hashing scheme, our incremental Bloomier hashing performs well. When using an index table which has the

same size as the page table, all workloads have smaller average search time as compared to the normal hashing method. If we double the size of the index table, the improvement is even greater. Moreover, the average access time is now about the same as in the separated chaining method that was shown in the Figure 3.15. When using a smaller index table size (1/2N), there is still improvements for five out of the eight workloads. Using our incremental Bloomier hashing method, the worst case search time is also reduced as shown in Figure 3.17 below.

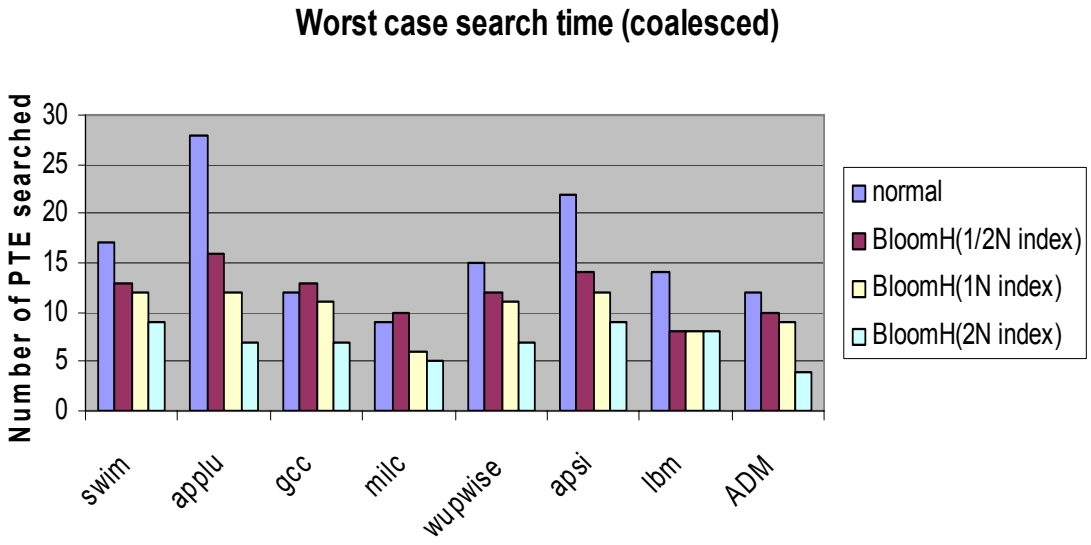


Figure 3.17. Worst case search time comparison for coalesced hashed page table

Figure 3.17 shows the worst case search time for different methods by using the coalesced hashed page table. Our incremental Bloomier hashing method can decrease the worst case search time. When using the index tables of the same or bigger size than the page tables, all workloads have a smaller worst case search time with the incremental Bloomier hashing method than with the normal hashing method. For workloads such as Applu and Apsi, the difference is even greater. Our hashing method worst case search time is half the time using a normal hashing method.

During the 40 million TLB misses, the numbers of instructions executed are also gathered for each workload and are shown below as the number of TLB misses per kilo instructions for each workload:

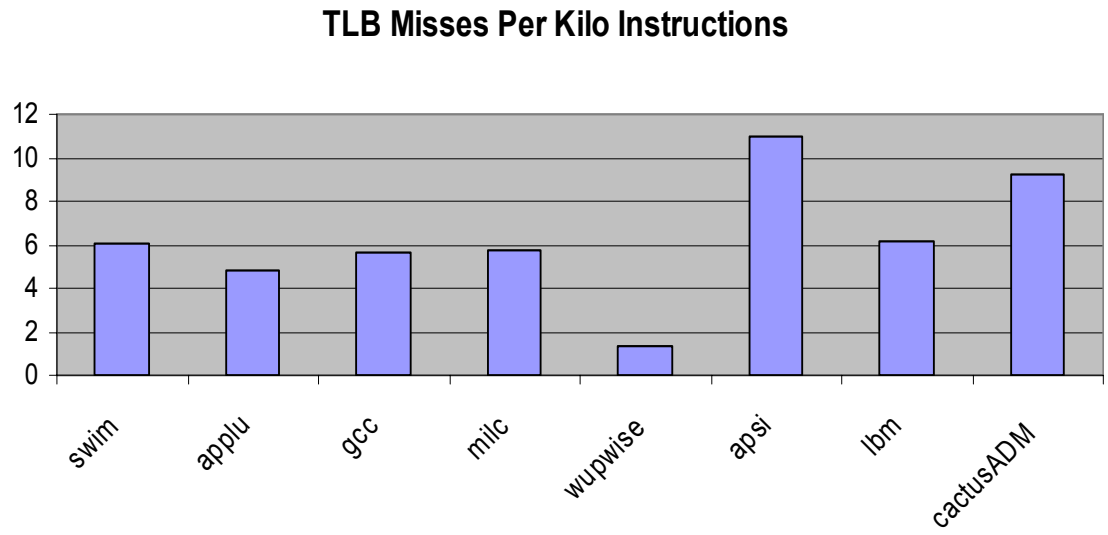


Figure 3.18. TLB misses per kilo instructions for different workloads

For each TLB miss, a page table search needs to be performed. The gathered data show that over 98% of page table searches result in hits. By using this knowledge about the TLB misses rate, we can compare the amount of time spent on page table searches in terms of CPU cycles by assuming the following. Due to the size, the page table likely resides in off-chip memory. Since the index table is only a fraction of the page table, it can be kept in cache. Generally, memory is slow and cache is much faster. Therefore, accessing an entry in page table in memory should take 200 cycles. Cache is must faster. Therefore, accessing the index table in cache should only take about 5 cycles. For the normal hashing scheme, the average cycles spend on page table search is the average number of page table entries searched per lookup times the number of cycles for accessing a page table entry. For the Incremental Bloomier

hashing as well as IPT with anchor table, there are two levels of date structures for lookup. Each lookup always access the intermediate table once. Therefore, the average cycles spend on page table search is the sums of the cycles spent on index table access and the average cycles spent on the page table. Since each page table lookup always access the index table once, the number of cycles spends on index table access is 5 cycles. The average number of the cycles spend on page table search is also the average number of page table entries searched per lookup times the number of cycles for accessing a page table entry. By using these formulas, we show the results in Figure 3.19 below.

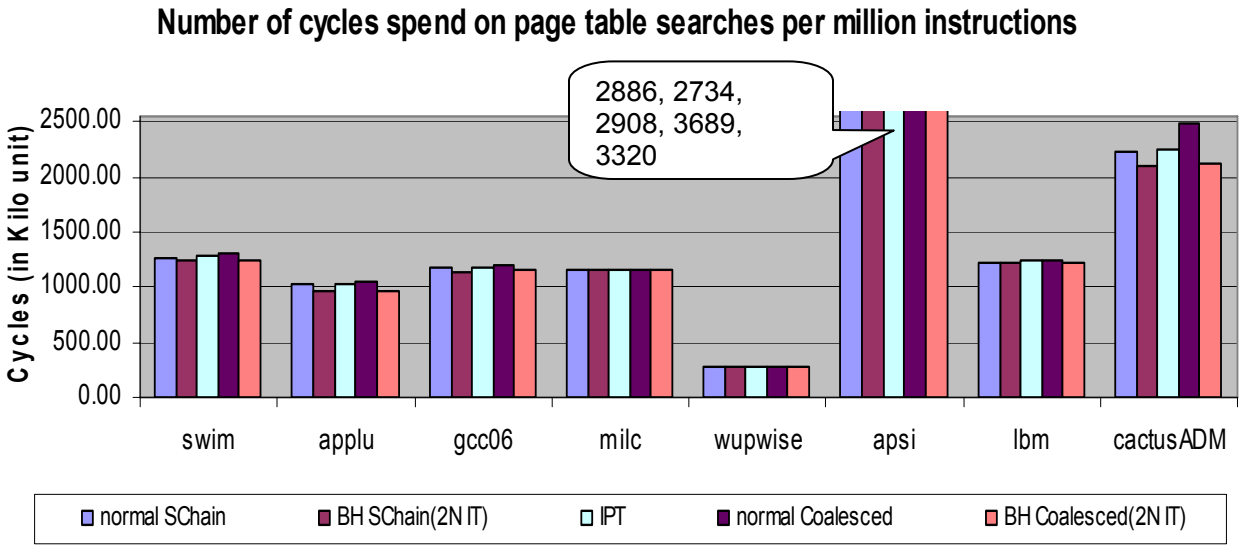


Figure 3.19. Average number of cycles spent per kilo instructions for different schemes

Figure 3.19 compares the average numbers of cycles spent on page table searches per kilo instructions for different schemes. The leftmost two bars are results for using separate chaining hashed page table. The rightmost three bars are results for using coalesced hashing page table. We include the results for the inverted page table (IPT) with the anchor table as well. The inverted page table with the anchor table is a

form of coalesced hashing. The inverted page table with the anchor table puts the collisions in the anchor table and handles the overflow by chaining together entries within the page table. Due to this, the inverted page table with the anchor table can achieve the same average look up time as the normal hashed page table with the separate chaining collision resolution method—without using overflow space. However, due to the need for accessing the anchor table, the inverted page table with the anchor table has extra time overhead. For our incremental Bloomier hashing approach (BH), we use index table size same as the page table size. Sensitivity study on the index table size will show later.

From figure 3.19, when using separate chaining hashed page table and comparing to normal hashing method, we can see that by using our hashing method, all workloads but two have positive improvements. The workload Milc and Wupwise show little to no improvements. On the other hand, by using coalesced hashing collision resolution method, the average number of cycles spent on page table search is higher than using separate chaining collision resolution method for most schemes. This is expected, since coalesced page table does not have extra overflow space for handling collisions. Therefore, all entries in the coalesced page table are used. On the other hand, separate chaining page table can have unused buckets. By using coalesced hashing page table and comparing to normal hashing method, we can see that our hashing method shows improvements for most of the workloads. The inverted page table with anchor table size of 1N performs the same as the normal hashing scheme except for the workload Apsi and CactusADM. Another interesting observation is that by using our incremental Bloomier hashing method, we achieve the around the same performance regardless

which collision resolution method is used. Next, we show the sensitivity study on the index table sizes for our schemes.

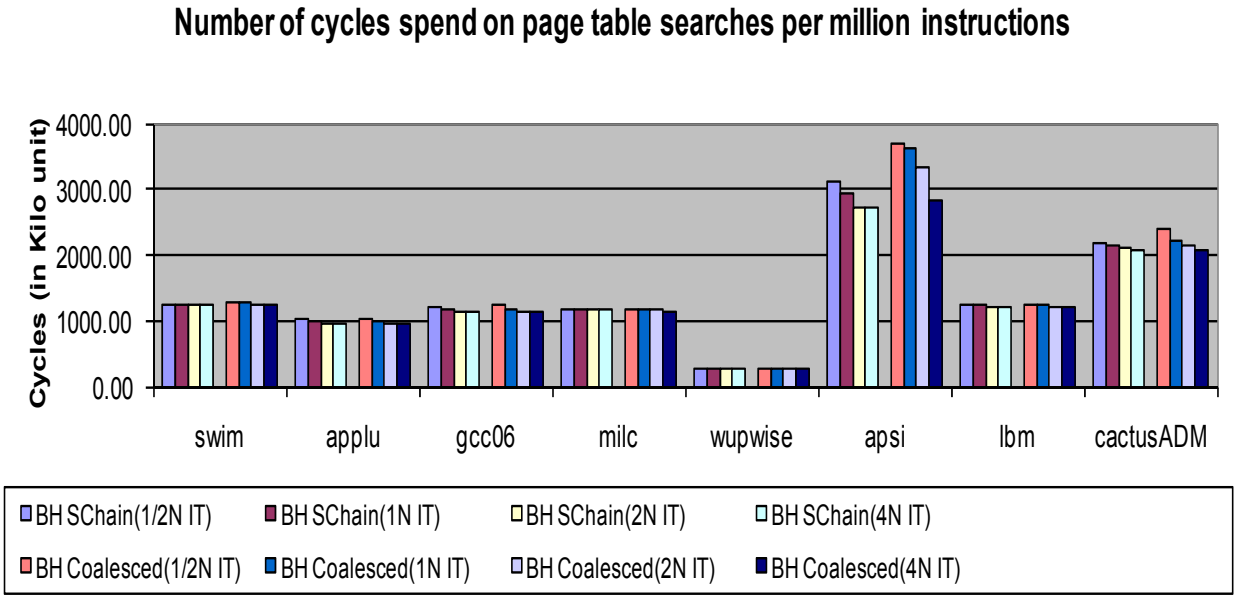
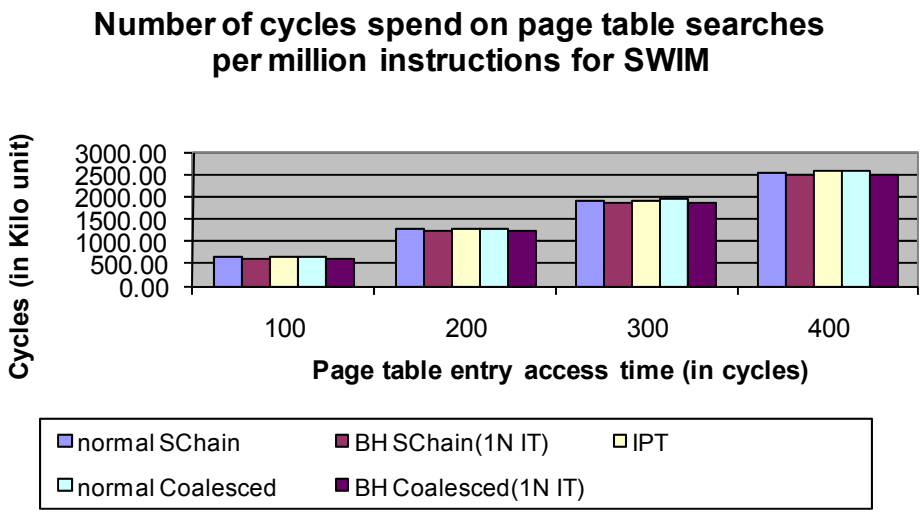
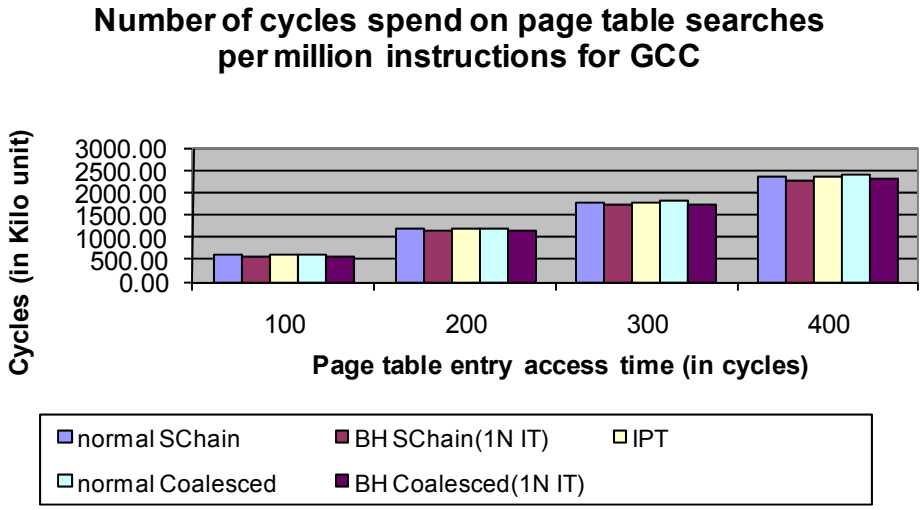


Figure 3.20. Sensitivity on incremental Bloomier hashing index table size

The results of a sensitivity study on the incremental Bloomier hashing index table size are given in Figure 3.20. We simulate four different incremental Bloomier hashing index table sizes. The four sizes are half the size of the page table (1/2N IT), the same size of the page table (1N IT), twice the size as the page table (2N IT) and four times the size of the page table (4N IT). We also show the results for both the separate chaining and coalesced hashing collision resolution methods. From Figure 3.20, we can see that bigger incremental Bloomier hashing index table indeed help decrease average cycles spent on page table search. Regardless of which collision resolution method is used, our incremental Bloomier hashing methods perform the best when using an index table size of 4N. This is most evident by looking the results for the workload Apsi and CactusADM. On the other hand, the different between using index





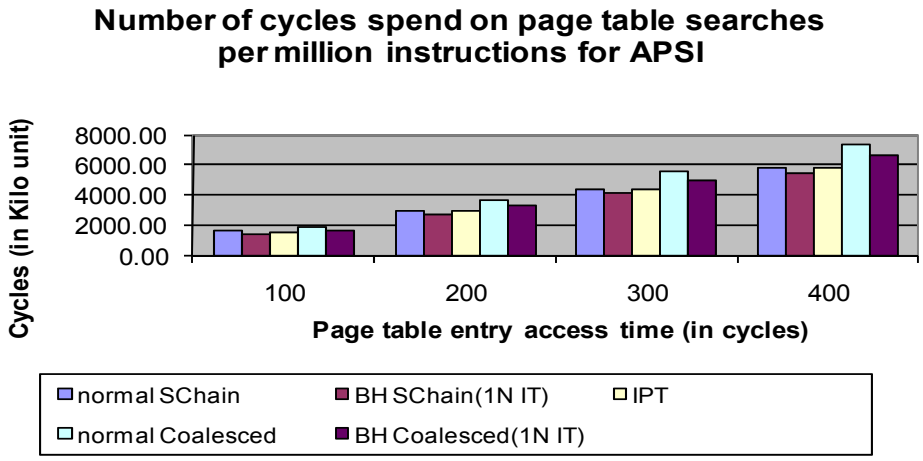


Figure 3.21. Sensitivity on page table entries access latency

table size of 1/2 N and 1 N is quite small for most of the workloads. Next, we show the sensitivity study on page table entries access latency.

The results of a sensitivity study on the page table latency are given in Figure 3.21. In Figure 3.19 and Figure 3.20, we used 200 cycles latency for accessing a page table entry. In Figure 3.21, we show the results for 100 cycles latency, 200 cycles latency, 300 cycles latency and 400 cycles page table entry access latency by using workload Gcc, Swim and Apsi. The results are shown for the different hashing schemes and the two different collisions resolution methods. From Figure 3.21, we can see that coalesced hashing resolution method generally performs worse than the separate chaining method resolution method, just as we have seen before. We can see that our incremental Bloomier hashing scheme shows improvements for latency values of 100 and 200 cycles. For the latency values of 300 and 400 cycles, our incremental Bloomier hashing scheme performs even better.

3.7 Related work

To perform the Bloomier filter setup, the setup algorithm tries to find a singleton for each key. Setup can fail if no singleton can be found for any one of keys. If the setup fails, then all the keys are needed to be rehashed [14]. Chisel [26] proposes another idea. They suggest that if the setup fails, then a few problematic keys can be removed and put into a spill over TCAM. The setup process then resumes. However, by using their method, a lookup needs to be performed in parallel between the filter and the spill over TCAM. Another approach is the linear equations method [13]. This method decides what value to encode for each location in the filter by solving sets of linear equations. This method requires less space for the filter, but longer setup time.

Two common page table organizations, forward-mapped and inverted page tables, are described in details in Section 3.2. The forward-mapped page table is very flexible since entries can be duplicated per process [28]. However, the forward-mapped table suffers from multiple memory accesses per lookup. Moreover, most of the tables are quite sparse. On the other hand, the inverted page table is a compact data structure. Instead of storing all valid virtual address mappings, the IPT stores only the virtual pages that currently map to the physical pages. Physical addresses can be computed directly from the indexes in the IPT. When using hashing, searching the IPT is fairly fast and requires few memory accesses. However, address aliasing becomes a problem since only one physical to virtual address mapping can exist in the table at a time.

The hashed page table [28] is another page table organization with fast lookup times. The hashed page table is based on the inverted page table. The difference is that the hashed page table does not use the hash anchor table. The virtual address is directly hashed to a bucket in the hashed page table. Due to this, each entry in the table needs to contain both the physical address and virtual address since the physical address can no longer be computed from the index in the hashed page table. Moreover, hashed page tables also require a collision resolution table to handle the collisions in the page table. Hashed page tables support address aliasing. The authors show that the hashed page table can reduce the memory references during lookup at the expense of requiring more space for table entries.

Guarded page table [42] combines the advantages of multiple-levels page table and hashed page table. However, the guarded page table's performance gains are achieved by specific assembler level optimization. Likewise, in [56], the authors try to

combine both IPT and forward-mapped page table into a page table in order to obtain the benefits in both approaches. However, the hybrid page table also introduces extra hardware cost and extra complications during lookup.

The Cluster page table [63] is based on the design of the hashed page table. Each entry in the table now stores the translation information for a block of several consecutive pages. The number of pages in an entry can vary depending on how sparse the address space is. This number is called the subblocking factor. The authors show that cluster page tables can have more efficient page table operations. They also show that the average linked list length in the cluster page table is shorter than the average linked list length in the hashed page table. Shorter linked lists mean the access time can improve. However, for programs that do not show spatial locality, cluster page tables are not as effective.

CHAPTER 4
USING BLOOMIER HASHING TECHNIQUES FOR OTHER POSSIBLE APPLICATIONS

Hash table is a basic information storage and retrieval method. Our Bloomier hashing techniques can generally replace the normal hash table for any digital applications and systems. Therefore, Bloomier hashing is very general and can be applied in other applications involving information storage and retrieval. Applications such as intrusion detection based on virus signature searches, key word matching in the Google™ search engine, maintaining connection record or per-flow states in network processing and packet classification in network are possible candidates that can benefit from using the Bloomier hashing ideas. We introduce each of the possible applications below.

Intrusion detection based on virus signature searches: An intrusion detection system, such as Snort [59], performs Deep Packet Inspection for each packet. The content of each packet is compared against a signature database. A string matching approach can be applied for the matching process [37]. Hash-based approaches [15] can also be used for fast lookup. Therefore, we can apply our hashing methods to Deep Packet Inspection.

For example, in [15], an input pattern to generate an index value. Then the algorithm uses the index value to fetch the virus patterns that are stored in the memory. Since several different virus patterns may share the same hash values, the input pattern may need to compare multiple virus patterns. We can use our Bloomier hashing ideas to better balance the virus patterns among the buckets and therefore increase the overall performance.

Key word matching in the Google™ search engine: Bigtable [24] is a high performance database system which was built on the Google™ file system [12, 24]. Many Google™ services, such as Google Maps™, Google™ search engine and Orkut™, use Bigtable [24]. Bigtable has multiple dimensions. The table's row and column names are arbitrary strings. Each table cell also contains a time stamp [12]. Therefore, cells may share the same content but have different time stamps, and Bigtable can use the time stamps for versioning purposes [12]. The Google™ search engine uses a large number of machines to serve huge amounts of users at the same time [24]. Because of this, Bigtable is split by rows and becomes tablets [12]. Each Google™ machine then stores a large number of tablets and thus can independently serve users [12]. However, a disproportionate load-balance may cause some machines to become less busy than others. Therefore, our Bloomier hashing can be used as a load-balancing method to more equally distribute workloads among the machines.

Maintaining connection record in network processing: Many network systems need to maintain connection records. To do so, these systems use hash tables. For example, intrusion detection systems, such as Snort [59] and Bro [51], maintain records in hash tables for all the TCP connections. Each connection record is created by hashing the 5 – tuples in the TCP header. Each record contains information on the connection state. The record is also updated every time a new packet arrives in that connection. Network monitoring systems, such as Netflow [16] and Adaptive NetFlow [22], maintain connection records in off-chip DRAM. Other hardware implementations [20, 55] also store the records in DRAM. Therefore, by using our Bloomier hashing

techniques, we can balance the number of records stored per bucket and thus increase system performance.

Packet classification in network: Packet classification is one of the most important functions in a router. When given an IP packet, the router must classify the packet based on a number of fields in the packet's header. Many packet-classification methods first examine a single field in the header and then narrow down the search to a smaller subset of classification rules [40, 4, 43, 23]. Since a hash table can perform a field lookup, we can apply our Bloomier hashing idea and thus increase performance quality.

For example, [62] introduces packet classification using Tuple Space Search. A tuple is a combination of header field length. The search algorithm groups the classification rules into sets of tuples. The tuples are then ordered by their lengths and a hash table stores each group in the memory. When a new packet arrives, the algorithm hashes the header fields and performs matches on all the hash tables. Therefore, we can apply the Bloomier hashing idea to improve the hash table's performance.

Exact flow matching is also a form of packet classification. The exact flow matching lookup operation performs exact matches on five header fields in a packet's header. In [64], the authors propose using hash tables for lookup. A small, on-chip hash table contains valid bits. Furthermore, a big hash table stores all filters in off-chip memory. In their lookup algorithm, when a new packet arrives, lower-order bits—which consist of source and destination addresses in the packet's header—produce a hash index. By using the index, the on-chip table is checked to see if the valid bit is set. If

the system verifies the bit's validity, then the system again uses the index to check the off-chip hash table. The hash tables use separate chaining for collision resolution. Therefore, balancing the hash buckets by using Bloomier hashing can improve the overall system performance.

CHAPTER 5
DISSERTATION SUMMARY

Information retrieval is an important task for many systems in today's digital information world. Massive amounts of information is stored first and retrieved later. Proper ways to store and retrieve information are critical for the system's performance. Hashing is a classic technique that has fast lookup time. However, typical hashing still suffers from unpredictable and worst case lookup time. We introduce the concepts of Bloomier hashing and incremental Bloomier hashing based on the previous work of the Bloomier filter [14]. By using a small intermediate table, we greatly reduce the collisions in a hash table. Hence, we greatly reduce the average time and worst case time for retrieving an item in the table. We showed two examples of information retrieval problems and propose our hashing techniques to solve the problems.

The first example is the IP lookup problem in routers. IP routing tables are growing bigger and bigger every year. Because of this, a fast way for lookup in a routing table is needed. We present a comprehensive hash-based LMP solution for future back-bone internet routers. A small portion (<10%) of the prefixes are implemented in TCAM to help in space and bandwidth efficiency during LPM lookups of the majority of prefixes from regular SRAMs. The SRAMs holding the prefixes are organized as two-dimensional arrays to allow a constant fetch rate. Multiple lengths of prefixes that are required for LPM lookup are allocated in the same memory block using partial prefix expansion and bucket coalescing to enable one memory access per look up. The Bloomier hashing through an intermediate indexing mechanism balances the hashed buckets and achieves the best space and bandwidth tradeoffs for the LPM lookup function.

The second example is the page table lookup in virtual memory. The page table is an important part of the virtual memory system. Programs address memory by using virtual addresses. Virtual addresses need to translate into a physical memory address before the data can be fetched out. The address translation needs to be done as fast as possible when the processor is waiting for the data. The inverted page table is a compact data structure. Combined with a hashing anchor table, inverted page also has a fast lookup time but still suffers the short comings of the traditional hashing technique. We apply our incremental Bloomier hashing techniques and the Bloomier filter technique to the page tables. Results show that we can reduce the number of page table accesses and space reduction is also possible.

Bloomier hashing is very general and can be applied in other applications involving information storage and retrieval. Applications such as intrusion detection based on virus signature searches, maintaining connection record or per-flow states in network processing and key word matching in the Google™ search engine are possible candidates that can benefit from using the Bloomier hashing idea.

LIST OF REFERENCES

[1] M. J. Akhbarizadeh and M. Nourani, “Efficient prefix cache for network processors,” *In 12th Annual IEEE symposium on High Performance Interconnects*, Aug 2004.

[2] M. J. Akhbarizadeh M. Nourani. D. S. Vijayasarathi and P. T. Balsara, “Pcam: A ternary cam optimized for longest prefix matching tasks,” *ICCD 04,* 2004.

[3] Y.Azar, A broder, and E. Upfal, ”Balanced allocations,” *In Proceedings of 26th ACM Symposium on the Theory of Computing,* 1994.

[4] F. Baboescu and G. Varghese, “Scalable packet classification,” *In ACM Sigcomm, San Diego, CA,* August 2001.

[5] T. W. Barr, A L. Cox and S. Rixner, “Translation Caching: Skip, Don’t Walk (the Page Table),” *Proceedings of the 37th annual international symposium on Computer architecture,* 2010.

[6] BGP Routing Table Analysis Report. *http://bgp.potaroo.net/,* 2009.

[7] A. Bhattacharjee and M. Martonosi, “Characterizing the TLB Behavior of Emerging ParallelWorkloads on Chip Multiprocessors,” *18th International Conference on Parallel Architectures and Compilation Techniques,* 2009.

[8] F C. Botelho, Y Kohayakawa, N Ziviani, "A Practical Minimal Perfect Hashing Method," *Experimental and Efficient Algorithms, 4th InternationalWorkshop,* 2005.

[9] A. Broder and A. Karlin, “Multilevel adaptive hashing,” *In Proceedings of 1st ACM-SIAM Symposium on Discrete Algorithm,* 1990.

[10] A. Z. Broder and M.Mitzenmacher, “Using Multiple Hash fuctions to Improve IP Lookups,” *In INFOCOM 01*, 2001.

[11] A. Chang and M. F. Mergen, “801 Storage: Architecture and Programming,” *ACM Transactions on Computer Systems, Vol 6, No 1, pp 28-50.* February 1988.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, *OSDI,* 2006.

[13] D. Charles and K. Chellapilla, “Bloomier Filters: A Second Look,” D. Halperin and K. Mehlhorn (Eds.): *ESA 2008, LNCS 5193, pp. 259–270*, 2008.

[14] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables,” *In The Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms,* 2004.

[15] Y. H. Cho and W. H. Mangione-Smith, “A Pattern Matching Coprocessor for Network Security,” *DAC*, 2005.

[16] Cisco netflow. http://www.cisco.com/warp/public/732/Tech/netflow.

[17] D. Clark and J. Emer, “Performance of the VAX- 11/780 translation buffer Simulation and measurement,” *ACM Transactions on Computer Systems, 3(1):31-62*, February 1985.

[18] S. Demetriades, M. Hanna, S. Cho, and R. Melhem, “An Efficient Hardware-based Multi-hash Scheme for High Speed IP Lookup,” *In Proceedings of the Annual IEEE Symposium on High-Performance Interconnects (HOTI),* August 2008.

[19] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, “Longest Prefix Matching using Bloom Filters,” *In ACM SIGCOMM-03,* 2003.

[20] S. Dharmapurikar and V. Paxson, “Robust TCP stream reassembly in the presence of adversaries,” *In USENIX Security Symposium*, August 2005.

[21] W. Eatherton, G. Varghese, and Z. Dittia, “Tree Bitmap: hardware/software IP Lookups with Incremental Updates,” *ACM SIGCOMM Computer Communication Review,* 2004.

[22] C. Estan, K. Keys, D. Moore, and G. Varghese, “Building a better NetFlow,” *In ACM Sigcomm,* 2004.

[23] A. Feldmann and S. Muthukrishnan, “Tradeoffs for packet classification,” *In Proceedings of IEEE INFOCOM,* 2000.

[24] S. Ghemawat, H. Gobioff And S. T. Leung, “The Google file system,” *In Proc. of the 19th ACM SOSP,* Dec 2003.

[25] M. Hanna, S. Demetriades, S. Cho, and R. Melhem “CHAP: Enabling Efficient Hardware-Based Multiple Hash Schemes for IP Lookup,” *IFIP International Federation for Information Processing* 2009.

[26] J. Hasan, S. Cadambi, V. Jakkula, and S.T.Chakradhar, “Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture, ” *In ISCA 06,* 2006.

[27] HDL Design House. HCR MD5: MD5 crypto core family, December, 2002.

[28] J. Huck and J. Hays, “Architectural Support for Translation Table Management in Large Address Space Machines,” *In ISCA 93,* 1993.

[29] IBM, “IBM System/38 technical developments,” Order no. G580-0237, IBM, Atlanta, GA., 1978.

[30] B. Jacob and T. Mudge, “A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations,*” In ASPLOS 98*, 1998.

[31] W. Jiang, Q. Wang and V. Prasanna, “Beyond TCAMS: An SRAM-based Parallel Multi-Pipeline Architecture for Terabit IP Lookup,” *In IEEE INFOCOM-08,* 2008.

[32] G.B. Kandiraju and A. Sivasubramaniam, “Characterizing the dTLB Behavior of SPEC CPU2000 Benchmarks,” ACM SIGMETRICS Perform. *Eval. Rev., Vol. 30, No. 1, pages 129–139,* 2002.

[33] S. Kaxiras and G. Keramidas, “IPStash: A Power Efficient Memory Architecture for IP lookup,” *In Proc. of MICRO-36,* November 2003.

[34] S. Kaxiras and G. Keramidas, “IPStash: A set-associative memory approach for efficient ip-lookup,” *pp. 992–1001, IEEE Infocom,* 2005.

[35] K. S. Kim and S. Sahni, “Efficient construction of pipelined multibit-tire router-tables,” *IEEE Trans. Computers, 56(1):32-43*,2007.

[36] Donald E. Knutb. “The Art of Computer Programming - Volume 3: Sorting and Searching*,” Addison Wesley,* 1973.

[37] L. Tan,T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” *ISCA*, 2005.

[38] G. Taylor, P. Davies and M. Farmwald, “The TLB slice a low-cost high-speed address translation mechanism,” *In The 17th Annual International Symposium on Computer Architecture.* May 1990.

[39] S. Kumar, J. Turner, and P. Crowley, “Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms,” *In IEEE INFOCOM-08,* 2008.

[40] T. V. Lakshman and D. Stiliadis. “High-speed policy-based packet forwarding using efficient multi-dimensional range matching,” *In ACM Sigcomm,* September 1998.

[41] R. B. Lee, “Precision Architecture,” *Computer,* January 1989.

[42] J. Liedtke and K. Elphinstone, “Guarded page tables on Mips R4600 or an exercise in architecture-dependent micro optimization,” *SIGOPS Operating Systems Review*, 1996.

[43] J. Lunteren and T. Engbersen, “Fast and Scalable Packet Classification,” *IEEE Journal on Selected Areas in Communications,* 21, May 2003.

[44] P. S. Magnusson et al, “Simics: A Full System Simulation Platform,” *IEEE Computer,* Feb. 2002.

[45] P. Magnusson and B.Werner, “Efficient Memory Simulation in SimICS,” *Proceedings of the 28th Annual Simulation Symposium,* 1995.

[46] H. Miyatake, M. Tanaka and Y. Mori, “A design for high-speed low-power cmos fully parallel content-addressable memory macros,” *IEEE Journal of Solid-State Circuits, 36(6):956-968,* 2001.

[47] D Nagle, R Uhlig, T Stanley, S Sechrest, T Mudge and R Brown, “Design Tradeoffs for Software- Managed TLBs,” *ACM Trans. on Computer Systems, 12(3):175–205,* August 1994.

[48] X. Nie, D.Wilson, J. Cornet, G.Damm and Y. Zhao, “IP Address Lookup Using a Dynamic Hash Function,” *In CCECE/CCGEI*, 2005.

[49] H. Noda et al. “A Cost-Efficient High-Performance Dynamic TCAM with Pipelined Hierarchical Searching and Shift Redundancy Architecture,” *IEEE J. Solid-State Circuits, 40(1): 245–253,* Jan. 2005.

[50] R. Pagh, “Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions,” *BRICS Report Series*, 1999.

[51] V. Paxson, “Bro: A system for detecting network intruders in real time,” *Computer Networks,*1999.

[52] M. Pearson, “QDRTM–III: Next Generation SRAM for Networking,” *http://www.qdrconsortium.org/presentation/QDR-III-SRAM.pdf,*2009.

[53] M.V. Ramakrishna, E. Fu, and E. Bahcekapili, “Efficient Hardware Hashing Functions for High Performance Computers,” *IEEE Transactions ON Computers, VOL. 46, NO. 12,* Dec 1997.

[54] Routing Information Service, *http://www.ripe.net/ris/,* 2009.

[55] D. V. Schuehler, J. Moscola, and J. W. Lockwood, “Architecture for a hardware-based TCP/IP content scanning system,” *In IEEE Symposium on High Performance Interconnects (HotI),* August 2003.

[56] I. J. Shyu and S.P. Shieh, “Virtual Address Translation for Wide-Address Architectures,” *ACM SIGOPS Operating Systems Review,* 1995.

[57] H. Song. S. Dharmapurikar, J. Turner and J.W. Lockwood, “Fast hash table lookup using extended bloom filter: an aid to network processing,” *In SIGCOMM 05,* 2005.

[58] H. Song, F. Hao, M. Kodialam, T.V. Lakshman, “IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards,” *In INFOCOM 09,* 2009.

[59] Snort - The Open Source Network Intrusion Detection System. http://www.snort.org.

[60] SPARC International Inc, “The SPARC Architecture 199 Manual, Version 8,” 1991.

[61] V. Srinivasan and G. Varghese, “Fast address lookups using controlled prefix expansion,” *ACM Transactions on Computer Systems, 17(1): 1-40,* 1999.

[62] V. Srinivasan, S Suri, and G Varghese, “Packet classification using tuple space search,” *In SIGCOMM, pages 135–146,* 1999.

[63] M. Talluri, M. D. Hill and Y. A. Kalidi, “A New Page Table for 64-bit Address Spaces,” *In SIGOPS 95,* 1995.

[64] D. Taylor, A. Chandra, Y. Chen, S. Dharmapurikar, J. Lockwood, W. Tang, and J. Turner, “System-on-chip packet processor for an experimental network services platform,” *In Proceedings of IEEE Globecom,* 2003.

[65] J. T. M. Waldvogel, G. Varghese and B. Plattner, “Scalable High Speed IP Routing Lookups,” *In ACM SIGCOMM-97,* 1997.

[66] X. Zhou and P. Petrov, “The Interval Page Table: Virtual Memory Support in Real-Time and Memory-Constrained Embedded Systems,” *Proceedings of the 20th annual conference on Integrated circuits and systems design*, 2007.

[67] X. Zhou and P. Petrov, “Direct Address Translation for Virtual Memory in Energy-Efficient Embedded Systems,” *ACM Transactions on Embedded Computing Systems (TECS),* 2008.

[68] X. Zhou and P. Petrov, “Arithmetic-based address translation for energy-efficient virtual memory support in low-power, real-time embedded systems,” *Proceedings of the 18th annual symposium on Integrated circuits and system design*, 2005.

BIOGRAPHICAL SKETCH

David Yi Lin was born in the Guangdong province, China. He moved to United States of America with his parents during the year 1993. He received his B.S. degree in computer engineering and M.S. degree in computer engineering from University of Florida in 2002 and in 2004 respectively. In 2005, he entered the Ph.D. program in Computer Engineering at University of Florida. In 2010, he received his Ph.D. degree in Computer Engineering. His research interests include network router design and computer architecture.