POWER AND MEMORY EFFICIENT HASHING SCHEMES FOR SOME NETWORK APPLICATIONS

A Dissertation

by

HEEYEOL YU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2009

Major Subject: Computer Science

POWER AND MEMORY EFFICIENT HASHING SCHEMES FOR SOME NETWORK APPLICATIONS

A Dissertation

by

HEEYEOL YU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Rabi Mahapatra
Committee Members,	Duncan M. Walker
	Riccardo Bettati
	Gwan Choi
Head of Department,	Valerie E. Taylor

May 2009

Major Subject: Computer Science

ABSTRACT

Power and Memory Efficient Hashing Schemes
for Some Network Applications. (May 2009)
Heeyeol Yu, B.S., Korea Advanced Institute of Science and Technology;
M.S., University of California, Los Angeles
Chair of Advisory Committee: Dr. Rabi Mahapatra

Hash tables (HTs) are used to implement various lookup schemes and they need to be efficient in terms of speed, space utilization, and power consumptions. For IP lookup, the hashing schemes are attractive due to their deterministic $\mathcal{O}(1)$ lookup performance and low power consumptions, in contrast to the TCAM and Trie based approaches. As the size of IP lookup table grows exponentially, scalable lookup performance is highly desirable. For next generation high-speed routers, this is a vital requirement when IP lookup remains in the critical data path and demands a predictable throughput. However, recently proposed hash schemes, like a Bloomier filter HT and a Fast HT (FHT) suffer from a number of flaws, including setup failures, update overheads, duplicate keys, and pointer overheads. In this dissertation, four novel hashing schemes and their architectures are proposed to address the above concerns by using pipelined Bloom filters and a Fingerprint filter which are designed for a memory-efficient approximate match. For IP lookups, two new hash schemes such as a Hierarchically Indexed Hash Table (HIHT) and Fingerprint-based Hash Table (FPHT) are introduced to achieve a a perfect match is assured without pointer overhead. Further, two hash mechanisms are also proposed to provide memory and power efficient lookup for packet processing applications.

Among four proposed schemes, the HIHT and the FPHT schemes are evaluated

for their performance and compared with TCAM and Trie based IP lookup schemes. Various sizes of IP lookup tables are considered to demonstrate scalability in terms of speed, memory use, and power consumptions. While an FPHT uses less memory than an HIHT, an FPHT-based IP lookup scheme reduces power consumption by a factor of 51 and requires 1.8 times memory compared to TCAM-based and trie-based IP lookup schemes, respectively. In dissertation, a multi-tiered packet classifier has been proposed that saves at most 3.2 times power compared to the existing parallel packet classifier.

Intrinsic hashing schemes lack of high throughput, unlike partitioned Ternary Content Addressable Memory (TCAM)-based scheme that are capable of parallel lookups despite large power consumption. A hybrid CAM (HCAM) architecture has been introduced. Simulation results indicate HCAM to achieve the same throughput as contemporary schemes while it uses 2.8 times less memory and 3.6 times less power compared to the contemporary schemes.

To my family

ACKNOWLEDGMENTS

I would like to thank Dr. Mahapatra for his direction and support over the last 3 years. His faith in my abilities helped mould my transition from a graduate student into a researcher. I would also like to thank Drs. Walker, Bettati, and Choi for serving on my committee and being excellent teachers.

I want to thank my family who support me mentally and financially. In addition, my soccer club, the Korean Aggies Soccer Association (KASA), gave me wonderful joy during my study at Texas A&M University. In particular, I miss Bong Su Koh who always gave me a smile, Sanghyub Kang who was an ex-professional soccer player, Jaewoo Suh who always loves an over-night drink, Won Ju Sung who I just met for one semester, and Hyeongil Kwak who gave spiritual help. Furthermore, I want to give a life-lasting appreciation to Uichin Lee at University of California Los Angeles who helped me in many ways

NOMENCLATURE

HT	Hash Table
LHT	Legacy Hash Table
FHT	Fast Hash Table
TCAM	Ternary Content Addressable Memory
CTCAM	Cool TCAM
UTCAM	Ultra TCAM
STCAM	Selective TCAM
BTCAM	Beyond TCAM
ACSM	Approximate Concurrent State Machines
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
PC	Prefix Collapse
CPE	Controlled Prefix Expansion
TCP	Transport Control Protocol
IP	Internet Protocol
SIP	Source IP
DIP	Destination IP
BF	Bloom Filter
MBF	Multi-predicate Bloom Filter
FF	Fingerprint Filter
SBF	Segmented Bloom Filter
SL	Successful Lookup
UL	Unsuccessful Lookup

SS	Successful Search
US	Unsuccessful Search
BMF	Bloomier Filter
PPC	Parallel Packet Classifier
MPC	Multi-tiered Packet Classifier
2TPC	2-tiered Packet Classifier
3TPC	3-tiered Packet Classifier
MBHT	Multi-predicate Bloom filter Hash Table
HIHT	Hierarchically Indexed Hash Table
IT	Indexing Tree
HIT	Hierarchical Indexing Tree
FPHT	Fingerprint-based Hash Table
HCAM	Hybrid CAM
SMT	Segmented Multibit Trie

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

In packet processing, a router fast associates packets with a set of rules for packet forwarding or various network services. Provision of such a fast packet processing like IP lookup and packet classification becomes harder, as the demand for high-speed and large-scale routers continues to surge in networking. It has been reported that the traffic of the Internet is doubling every two years by Moore's law of data traffic [1] and the number of hosts is tripling every two years [2].

These rapidly increased traffic and host numbers lead to two major packet processing related problems for core routers. 1) Speed: a high-speed router needs to look up a rule table at the rate that satisfies its bandwidth requirement. For example, IP lookup at the rate of 160Gbps must process 500M lookups in a second, and this implies that a packet of minimum 40 bytes must be forwarded to a next hop in 2ns in the worst case. 2) Scalability: a fast packet processing must be made in searching an associated rule even with hundreds of thousands of rules. For instance, in packet classification domain the maximum number of rules is up to 2 to 104 due to the 104-bit length of a tuple of source, destination IPs, etc.

Since a fast packet processing is a in the router's critical data path, literature on packet processing has developed numerous fast lookup schemes using three major techniques, Ternary Content Addressable Memory (TCAM) [3–6], trie [7–10], and hashing [11–17]. Although a TCAM provides a deterministic and high-speed packet lookup [5, 6], due to its non-commodity nature and brute-force search method, its die area cost and power dissipation tend to become prohibitive for packets with a large

---

This dissertation follows the style of *IEEE Transactions on Networking*

number of rules and high line rates. Unlike TCAM, trie-based scheme uses a tree-like data structure to successively classify a packet a few bits at a time [7–10], but it inherently suffers from space to hold pointers from nodes to their children and the sequential memory accesses introduced by these pointers. In addition, an imbalanced memory access hinders the high IP lookup performance due to an irregular prefix distribution in a trie's tree structure. In contrast, the hash-based schemes neither perform brute-force lookups as in TCAM nor suffer from imbalanced memory access, so they can potentially receive an order-of-magnitude power and memory savings, respectively.

Traditionally, a hash table (HT) is popularly used for a fast search and this is due to its $\mathcal{O}(1)$ average memory access per lookup under reasonable assumptions. Recently, HTs are used in a wide variety of packet processing applications such as intrusion detection systems [18], packet classification [19, 20], TCP/IP system management [21], and IP lookup [12, 16].

In particular, the binary search on prefix lengths algorithm [22] has the best theoretical performance of any sequential algorithm for the longest prefix match in IP lookup by using HTs. In addition, packet classification applications utilize HTs [23, 24], so they first perform a lookup on a single header field and later leverage the results to narrow down the search to a smaller subset of packet classifiers. These use HTs with the expectation of $\mathcal{O}(1)$ memory access and encompass a more predictable worst-case lookup scheme.

However, as the table occupancy, or load, increases, collision occurs frequently, which in turn reduces the performance by increasing the cost of the primitive operations. Although there are two collision resolutions (i.e. open address and chaining), schemes in open address are not suitable for a fast lookup because of the worst case performance. In addition, a chaining suffers from pointers' overhead as it maintains

a set of linked lists.

While these solutions are designed to maintain a good average performance despite their high loads and increased collisions, their performance nevertheless meets the packet processing needs: high speed and scalability. To satisfy such needs, an on-chip Bloom filter (BF) is widely used because the BF of an m-bit vector can provide both the memory efficiency and the high throughput using an approximate membership testing. Packet processing applications using such a BF include IP lookup in [12], an intrusion detection system in [13], or packet classifications [19, 20]. Also, in trading off space, computation, and the impact of false positive lookup, an efficient lookup using a fingerprint filter (FF) or d-left hashing has been considered preferable in the literature on networking [21, 25, 26]. The reason is that even though a BF provides memory-efficient approximate lookup, an FF is found more efficient in power and memory usages for set representation than its counterpart.

However, such an approximate match is not suitable for all packet processing applications; the exception includes IP lookup, where its packets are required strictly to be forwarded to a next hop according to a prefix table. For instance, the recently-proposed IP lookup approaches [12, 16] have the following design flaws that are not suitable for a high-speed and large-scale router: 1) A Bloomier filter-based hash table (BFHT) [16] utilizes a Bloomier filter [27]. However, it inherits two disadvantages of a Blooimer filter: a setup failure in saving n keys and $\mathcal{O}(n \log n)$ setup complexity for n keys. 2) Authors in [12] propose a memory-efficient IP lookup by using BFs, each assigned to a set of the same-length prefixes. Although this scheme provides fast approximate matches in on-chip, the perfect prefix match is achieved in an off-chip HT due to the BFs' false positive match. Thus, the lookup time is bounded in a slow off-chip memory access.

A supportive scheme to [12]'s scheme is made to provide a fast off-chip HT

access in [15]. However, this scheme suffers from duplicate keys saved in off-chip memory, and the number of duplicates is depending on k which is the number of hash functions and it controls the lookup precision. Also, the *insert* and *delete* operations take approximately k times. Such a depending fact k of a large value is not suitable for performing fast lookups and key updates in high speed routers. Other Peacock and multilevel hash schemes for packet processing [28, 29] suffer from setup failures as well.

To address these flaws, such as key duplicates, complicated key updates, and setup failures, we propose scalable hash schemes for maintaining a fast packet processing by using BFs and an FF in pipeline. The four hash schemes in this dissertation are 1) a multi-tiered packet classifier (MPC), 2) a multi-predicate Bloom-filter HT (MBHT), 3) a hierarchically indexed HT (HIHT), and 4) a fingerprint-based HT (FPHT). The first two schemes are designed for general packet processing applications while the last 2 schemes are designed for IP lookup application due to a memory-efficient perfect match. These four schemes' overviews are the following:

A multi-tiered packet classifier (MPC) with n BFs provides a lookup distribution for higher power and throughput efficiencies, compared to a parallel packet classifier (PPC) of n parallel BFs [12–14, 17]. A PPC accesses n BFs for one lookup every cycle while an MPC accesses n BFs for several lookups every cycle with the same BFs' memory amount as that of a PPC. To build 2-tiered BFs, for an example of an MPC, the total PPC memory is split between a pre-stage of small-sized BFs with one read port and a post-stage of large-sized BFs with k-1 read ports. Then, a small-sized BF is logically connected to two large-sized BFs, so that a forest of binary trees is built [30].

Secondly, a multi-predicate Bloom-filtered HT (MBHT) with a set of multi-predicate BFs (MBFs) generates index addresses which have different base number

systems to a key table. The generated indexes are geared to *in parallel* access to a key table on-chip with simple switching circuitry, so that for a successful query at most one off-chip memory access is guaranteed for bandwidth requirement of a router. There are two benefits of an MBHT as regards to both on-chip and off-chip memory. For the on-chip memory, an MBF reduces the memory size in the base-2^x number system by x times compared to that of the base-2^1 number system with a binary predicate BF, where x is a positive integer larger than 1 [31].

Thirdly, a hierarchically indexed hash table (HIHT) is proposed and is used for approximate testing on keys' index paths in trees. Once the BFs of the last step of pipelining complete their index addresses to entries in a table, a *perfect* match is made by comparing the saved keys in the indexed entries with a given key, so that at the most one off-chip access is made to a known associated rule with the given key [32].

Finally, a **f**inger**p**rint-based HT (FPHT) generates indexes to a key table with the help of both memory-efficient BFs which are approximate membership testers and an FF which is the most memory-efficient set representation. Specifically, in an FPHT with no pointers, BFs play a role in key searching in a b-ary prefix tree, $b \in \{2, 4, 8, \cdots\}$, and an FF ensures a fewer number of false indexes to a key table in the worst lookup case [33].

In addition to hash-based approaches in packet processing, the TCAMs have become the de facto industrial standard solution for a high-speed IP lookup. More than 6 million TCAM devices were deployed worldwide in 2004 [34], and TCAMs are projected to be increasingly used as the next generation network search engines. Part of the TCAMs success is that they are developed with the abilities to store a "don't care" state for a prefix and to compare an input key against every TCAM entry, thereby enabling them to retain a single clock cycle lookup.

Table I. Lookup & update complexities.

schemes	
Trie	$\mathcal{O}(W)$ †
Hash	$\mathcal{O}(1)$
TCAM	1

† W: # of IP address bits

Table II. Hardware features of each scheme.

	TCAM	CAM	SRAM
clock†	266	333	400
Power ‡	≈15	≈1	≈0.1
Cell°	16	8	6

† MHz unit ‡ Watts unit
° # of transistors per bit

Despite TCAMs' popularity and simplicity, TCAMs have their own limitations with respect to IP lookup. 1) Throughput: parallel searches in all prefixes are made in one clock cycle for a single lookup, so that the throughput is simply 1 as shown in Table I. 2) Power: although a TCAM provides an one-cycle lookup, such an one-cycle lookup, which is made in parallel searches on all prefixes, requires at most 150 times more in power consumption than any SRAM-based scheme does. Table II shows such power consumption difference measured by CACTI [35]. Thus, reducing TCAM power usage is a paramount goal for a deterministic TCAM lookup.

A high TCAM throughput has been achieved through a partitioning technique [36, 37]. Its principle with pipelining depends on a parallel architecture that fulfills multiple lookups per clock cycle. Likewise, a SRAM-based parallel scheme [38] partitions a trie and maps subtries to pipelines using a solution of a *NP-complete* problem for a high throughput. However, these kinds of approaches suffer from a high power consumption and a complicated mapping algorithm complexity, respectively.

In dissertation we proposes a hybrid CAM (HCAM) IP lookup architecture for maintaining a high throughput and power efficiency. Our approach adopts prefix collapse and partitioning schemes with Bloom filters (BFs). A prefix collapse (PC) reduces the number of prefixes as opposed to the prefixes expansion. In such prefix collapse, the collapsed prefixes can be put in a deterministic lookup-capable CAM

to demonstrate further hardware efficiencies for power and the number of transistors per cell than a TCAM can as shown in Table II. A complete prefix match beyond the collapsed prefix match is made through a stride tree bitmap (STB) saved in SRAM. Also, the CAM for the same-length collapsed prefixes can be partitioned into CAM blocks to provide multiple lookups on the collapsed prefixes per clock cycle.

This dissertation has the following contributions of the four hash schemes.

- An MPC hashing scheme with n BFs is proposed in a multi-tiered configuration of BFs with the same memory capacity as that of a PPC.
- An MBHT scheme is proposed using a contiguous memory space in off-chip memory without using pointers to conduct a perfect match and a fast search.
- An HIHT scheme for fast and memory-efficient packet processing is introduced. It provides per-key information lookup to be used as an index to a key table in on-chip memory without pointer operation.
- An FPHT scheme provides indexes to a key table using BFs and an FF without incurring pointer operations.
- For each of the above schemes, new algorithms on *insert*, *query*, and `delete` operations are proposed, and they are as easy to implement as those of a BF or an LHT.
- In an MPC evaluation, it has been shown that the proposed MPC scheme has 4.2 and 2 times power and throughput efficiencies against a PPC, respectively.
- In comparison for scalability and speed, analyses on memory efficiency for an MBHT, an HIHT, and an FPHT are made and multi-fold times memory efficiency is achieved over other contemporary schemes.

- In IP lookup application of the proposed hash schemes, the HIHT and the FPHT, memory and power comparisons with TCAM-based and trie-based IP lookups are made. The proposed hash-based IP lookup schemes show at least 51 times power efficiency and 1.8 times memory efficiency, compared to TCAM- and tried-based schemes.

- In addition, the proposed HCAM-based IP lookup scheme achieves the same throughput as contemporary schemes while it uses 2.8 times less memory and 3.6 times less power compared to contemporary schemes

The rest of the paper is organized as follows. Sec. II presents several hash-based schemes for packet processing, such as an FHT, a BFHT, and the Peacock hashing. Sec. III discusses the basics of a BF and an FF in terms of their memory size and power consumption. Also, this section shows two applications of an HT to IP lookup and packet classification. Then, a detailed MPC build with n BFs for a packet classification is shown in Sec. IV. In the following Sec. V, the detail of an MBHT for a perfect match is discussed. In Sec. VI, the detail of an HIHT for a perfect match is explained. A detailed FPHT build in a binary prefix tree for a perfect match is illustrated in Sec. VII. As the last scheme, an HCAM is proposed for a high throughput and power saving in Sec. IX. In each of Secs. IV, V, VI, VII, and IX, the analysis on memory, power, or throughput efficiencies in comparison to other contemporary schemes is made. A conclusion and future work are presented in the following Sec. X.

CHAPTER II

RELATED WORKS IN PACKET PROCESSING

Packet processing has different objectives in each networking layer. For instance, in layer 2 a router needs to forward a packet to a corresponding port in a limited time with a large-scale routing table. In layer 3 a packet is classified into a flow for various purposes like firewall or qualify of service. In this chapter, related major research works on packet processings like IP lookup and packet classification are enumerated.

A. IP Lookup

Song *et al.* [15] claimed that for a perfect match an FHT with help of a BF improves the performance over an LHT by reducing the number of off-chip memory accesses needed for the most time-consuming lookups. This benefit is possible by combining hashed linked lists with k hash functions so that only the shortest linked list is used in the search. Although chaining in a linked list for resolving a collision is one solution, accessing a key in a linked list costs the same memory accesses as the number of keys in the linked list because of pointer operation. Beyond the generic limitation of linked list implementation, overlapping k linked lists in an FHT suffers from several others described here. First of all, due to merging k linked lists there is a chance that duplicate keys are saved in off-chip memory, depending on k. In that case, k is reversely proportional to collision rate, a need of very low collision rate for a high-speed router makes a number of copies of a key, proportional to k, in off-chip. Although searching for a key is expedited by choosing the shortest linked list, the *insert* and *delete* operations take at least k times memory accesses due to the k shared linked lists. These operations are not suitable for a dynamically changing set because any change in the set needs $2k$ times of off-chip memory access. Besides

time complexities of *insert* and *delete* operations, to obtain better performance over an LHT in terms of reduced collisions, an FHT needs a plethora of buckets used as pointers to off-chip memory and it holds a large wasted portion of buckets in on-chip memory. Also, perfect match is made in off-chip memory, so that every query needs at least one access to off-chip memory. Furthermore, due to the inherent drawback of a BF, the *delete* operation was designed by introducing a 4-bit counter in each bucket [39]. Yet, they did not consider the memory size of the counters, but just the number of buckets.

There is a fundamental limitation in a HT using a linked list: a sequential access to a key along the linked list. For example, to access key e located at the end of a linked list of t keys, t sequential accesses are necessary in t cycles, because memory address of key e is known after a previous key e' with a pointer to the next key e is obtained in the previous cycle. However, accessing a few entries with known indexes in a table can be processed in one cycle with a simple switching circuitry. To provide collision-free lookup with a key table, a BFHT [16] utilizes a Bloomier filter [27] capable of per-key information lookup. Per-key information by a Bloomier filer is considered as an index address of a key table given a packet, so that a BFHT performs *perfect* match to make a deterministic IP lookup with a key table. Although a BFHT contributes prefix collapsing as well, it also inherits two disadvantages of a Blooimer filter: first, there is a setup failure in saving per-key informations of n keys in a BFHT, so that another lookup mechanism is used for the failed keys in the setup. Thus the number of hash functions gets increased to reduce setup failure rate, leading to more memory need. Second, the setup complexity of n keys is $\mathcal{O}(n \log n)$, implying that a copy of a BFHT works for update of a new key in the rear of the BFHT for seamless lookups of other keys.

B. Packet Classification

The packet classification goal is to identify a flow characterized with a 5-tuple of source IP (SIP), destination IP (DIP), protocol, source port (SP), destination port (DP) and to forward the flow to a corresponding output port. Several types of packet classifiers like TCAM-based and SRAM-based ones are suggested [6, 20, 40–42]. In a hash-based approach, a packet classifier in [14] uses BFs in parallel, so that in a given packet lookup all BFs need to be checked to find the packet-associated flow and the packet is forwarded to a corresponding port where a BF returns 'yes'. However, in a high-speed lookup to a BF, the number of memory read ports in the BF is considerably large. Also, the number of BFs to be probed is as large as the number of a high-speed router's ports. Unlike the above schemes of the $\Theta(n)$ BF access complexity among n BFs, our MPC needs probabilistically less complexity than $\Theta(n)$ for a lookup

C. Other Packet Processing Applications

Besides BF applications for packet processing in the previous section, applications of other domains have utilized the benefit of BFs, such as dynamic BF for data management [17], wide-area web caching [39], content delivery across overlay networks [43], IP traceback [11], query routing in peer-to-peer networks [44]. Even in a wireless sensor networks where power saving is a paramount issue, a coordinated packet traceback mechanism in [45] is introduced with the concept of dimensions in hash algorithms in which a dimension can expand by the number of either hash functions, hash tables, or both.

A legacy BF does not support deletion operation because a bit-location in a bit-vector indexed by hash functions can be overlapped by more than one key. To avoid this problem, Fan *et al.* [39] introduced the idea of a counting BF in which

each entry in the BF is not a single bit, but rather a small counter in a couple of bits. Bonomi *et al.* [21] introduced Approximate Concurrent State Machines (ACSM). While similar in spirit to BFs, the scheme is based on a combination of hashing and fingerprints, using d-left hashing to obtain a *near*-perfect hash function in a dynamic setting. Although it is found that its data structure takes much less space than a comparable counting BF, the fundamental problem in their approach is that in an f-positive there is no way to verify a result given by a ACSM. In contrast, our three schemes (MBHT, HIHT, FPHT) provide a perfect match mechanism without a pointer. Cohen and Mattias [46] introduce Spectral Bloom Filter (SBF), an extension of the original BF to multi-sets, allowing the filtering of elements whose multiplicities are below a threshold given at query time. However, SBF does not support a function of relationship between a key and arbitrary per-key information.

Unlike previous BF approaches for approximate membership testing, for the first time, Bloomier filter in [27] provides storage and retrieval of arbitrary per-key information. It guarantees perfect-hashing for a constant-time lookup in the worst case. However, a disadvantage lies in static support of membership. Also, there is setup failure probability of encoding all keys depending on k, the number of hash functions.

In an application of overlay networks, continuous reconfiguration of virtual topology by overlay management strives to establish paths with the most desirable end-to-end characteristics. The approximate reconciliation tree for overlay networks by Byers *et al.* [43] uses BFs on top of a tree structure to minimize the amount of data transmitted for verification.

D. Parallel IP Lookup Using TCAM or SRAM

Except a parallel SRAM scheme in [38], most parallel IP lookup engines for high throughput are TCAM-based due to benefit of employing parallel searches on TCAM prefixes [36, 37]. They partition the full routing table into several TCAM blocks and make parallel lookups on different blocks. This parallelism obtains power efficiency and throughput improvement.

Cool TCAM (CTCAM) was proposed in two separate schemes: bit-selection and trie-based schemes [47]. In the former, selected bits are used to index different TCAM blocks directly. The latter scheme splits the trie by carving subtries out of the full trie. However, the prefix distribution imbalance among the TCAM blocks can be noticeably high, resulting in low worst case performance.

Ultra TCAM (UTCAM) in [36] increases the throughput 4.0 times with a 25% TCAM entry redundancy. It uses distributed and parallel TCAM blocks aided by having an index logic to choose the destination TCAM block for a given packet. Likewise, Selective TCAM (STCAM) in [37] uses the multiple TCAM-block selectors with prefix TCAM caches. A collision among TCAM-block selection attributes a STCAM's need to resolve TCAM block contentions with arbiters, and these arbiters prevent from receiving a new lookup request. Thus, the STCAM throughput gain was reported to be at most 1.5 times even with multiple TCAM blocks without caches.

Unlike TCAM partitioning, beyond TCAM (BTCAM) scheme in [38, 48] is introduced for trie-partitioning using SRAMs where subtries were mapped to SRAM blocks with consideration of memory balance. However, such a mapping is proved to be *NP-complete*, so that remapping for prefix update during lookup operation is not feasible. Furthermore, leaf-pushing causes the increase number of trie nodes resulting in memory overhead.

Trie- and hash-based schemes shown in the above subsections are lack of high lookup performance. In contrast, a TCAM's lookup complexity is 1 and a TCAM has been considered as a natural choice of multi lookups due to its parallel searches through partitioning [36, 37]. The same characteristic is preserved in a CAM except the prefix match. After discussing an issue in prefix match by prefix collapse or expansion in Sec. C, a hybrid CAM (HCAM) scheme using CAM blocks is presented for high throughput in Sec. D.

CHAPTER III

BASICS ON HASH FOR PACKET PROCESSING

This chapter introduces the basics of a BF and an FF as well as their applications to packet processings, IP lookup and packet classification.

A. Basic Bloom Filter Theory

To understand the fundamental relationship among the number of buckets, m; the number of items, n; and the number of hash functions, k, the mathematics about a BF and a false positive, or f-positive are presented.

A legacy BF for representing set $S=\{e_0, e_1, ..., e_{n\text{-}1}\}$ of n elements is described by an array of m bits with each initially set to 0. A BF uses set H of k independent hash functions $h_0, h_1, ..., h_{k\text{-}1}$ with range $[0{:}m\text{-}1]$, implying that in hardware implementation a memory module for a BF needs k ports for memory read. For mathematical convenience, a natural assumption is made that these hash functions map each item in the universe to a random number uniform over the range. For each element $e_{j'} \in S$, the bits indexed by $h_{k'}(e_{j'})$ are set to 1 for $0 \le k' \le k\text{-}1$, $0 \le j' \le n\text{-}1$. To verify that item e' is in S, it is checked whether k bits in a BF indicated by $h_{k'}(e')$ are 1. If not, then clearly e' is not a member of S. Even if chosen bits indexed by $h_{k'}(y)$ have a value 1, there may be a probability called f-positive that item y is falsely believed to belong to set S due to the random gathering of k bits of value 1 set by independent items.

The above probability f of f-positive can be formulated in a straightforward way, given our assumption that hash functions are perfectly random. Among m bits, the chance of a bit being value 0 by one h_k is $1/m$. After all n elements of S are hashed k times into the BF, i.e. totaling $k \cdot n$ times, the probability that a specific bit is still 0 is asymptotically $p = (1\text{-}1/m)^{kn} \approx e^{-kn/m}$. Then, the probability of an f-positive by

randomly choosing k bits among m bits is

$$f \geq \{1-(1-1/m)^{kn}\}^k \approx (1-p)^k \geq (1/2)^{m \ln 2/n} \tag{3.1}$$

because k bits with probability of becoming 0, or p, could independently become more than 0 when a membership test is requested. This probability is bounded and the optimal k, the number of hash functions, that minimizes f is easily found $k = \ln 2 \dot{(}m/n)$ according to the results of Broder and Mitzenmacher [49]. After some algebraic manipulation, Broder and Mitzenmacher [49] claimed that the requirement of $f \leq \epsilon = 2^{-w}$ suggests

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44 n \log_2(1/\epsilon) = 1.44 n w, \tag{3.2}$$

where w is a precision in query operation. Furthermore, in an optimal configuration, k becomes w according to the following derivation:

$$k = \ln 2 \frac{m}{n_i} = \ln 2 \left(n_i \frac{\log_2(1/f)}{\ln 2} \right) / n_i = w. \tag{3.3}$$

Also, k needs to be at least 29 ($\approx \log_2 1/500M$) to be a scheme of a deterministic $\mathcal{O}(1)$ lookup processing 500M packets a second for a 160Gbps router.

Two important lemmas can be derived from Eq. (3.2), described as follows

LEMMA 1 (LINEAR PROPERTY) *Linear property between m and n exists in Eq. (3.2) because given f requires that variable n is linearly proportionate to variable m. Therefore, if n is reduced by half or decreased by constant α, the desired m for a given f is reduced by half or decreased by the constant of $\alpha \cdot 1.44 \log_2(1/\epsilon)$, respectively.*

Proof: Suppose function $F_m(n, f)$ of Eq. (3.2) has domain variables n and f. Once f is set to a constant ϵ as requirement, this function becomes a polynomial of variable n, $F'_m(n) = a \cdot n$, with degree one, where $a = 1.44 \cdot \log_2(1/\epsilon)$. Therefore,

$$F'_m(n/2) = a \cdot (n/2) = an \cdot 1/2 = F'_m(n)/2 \text{ and}$$
$$F'_m(n-\alpha) = a \cdot (n-\alpha) = an - a\alpha = F'_m(n) - a\alpha,$$

proving *Linear Property.*

Lemma 2 (Reverse Exponential Property) *The change of m has an exponential effect on f for a given n from Eq. (3.2). That is, if m is increased by constant α or multiplied x times, f is exponentially divided on base-2 by the power of constant α/c or powered by x times where $x>1$, constant $c=1.44n$.*

Proof: Suppose function $F_f(n,m)$ is derived from Eq. (3.2) and rearranged in $2^{-m/c}$, where $c=1.44.n$. Once n is set to a constant, this function becomes a exponential function of m, $F'_f(m)$. Therefore,

$$F'_f(m+\alpha) = 2^{-(m+\alpha)/c} = 2^{-m/c-\alpha/c} = F'_f(m)/2^{\alpha/c} \text{ and}$$
$$F'_f(xm) = 2^{-xm/c} = (2^{-m/c})^x = F'_f(m)^x,$$

proving *Reverse Exponential Property.* These *Linear* and *Reverse Exponential Properties* are used in introducing an MBF, so that an MBHT has the benefit of memory saving in on-chip memory by the *Linear Property*, and, thereinafter the saved memory is designed to decrease f exponentially by the *Reverse Exponential Property.*

We have linked the theoretical relationships between k, m, n for the required f-positive, ϵ, in a query. If a BF is to be used for IP lookup despite producing an approximate query result, a lookup precision w should be at least 29 ($\approx$ - $\log_2 1/500M$) for 160Gbps routers because a collision in 500M lookups in a second is not tolerable in bandwidth requirement satisfaction. Also, Eq. (3.3) suggests that a BF memory implementation for 160Gbps routers needs 29 read ports for the same number of hash functions, but this is not feasible in terms of cost and power concerns. To lessen these overheads, we adopt a segmented BF (SBF) [14] with memory banking. Using this scheme with commodity memory is more practical since IDT currently produces high-speed bank-switchable memory organized into a 64-bank memory array.

In an SBF, an m-bit vector is divided into k $m'(=m/k)$-bit subvectors, each put in an independent memory bank. k hash functions with the range [*0:m'-1*] are assigned to their corresponding subvectors, and an one-clock query in an SBF is based on k indexed values in k subvectors (or banks) together as in a legacy BF. Although a SBF's memory banking scheme removes the multiport overhead, the SBF's false positive probability, f', becomes the same BF's f as follows:

$$\begin{aligned} f &= \left(1\text{-}\left(1\text{-}1/m\right)^{kn}\right)^k = \left(1\text{-}\left(1\text{-}1/km'\right)^{kn}\right)^k \\ &= \left(1\text{-}\left(1\text{-}k/km'\text{+}o(1)\right)^n\right)^k \approx \left(1\text{-}\left(1\text{-}1/m'\right)^n\right)^k = f' \end{aligned} \tag{3.4}$$

where a small o function is negligible at a large m' value.

B. A Memory- and Power-Efficient Fingerprint Filter

Authors in [49, 50] claim that an FF is the most memory-efficient set representation scheme. In this section, beyond the theoretical FF benefit, it will be claimed that an FF is the power-efficient data structure in hardware implementation as well.

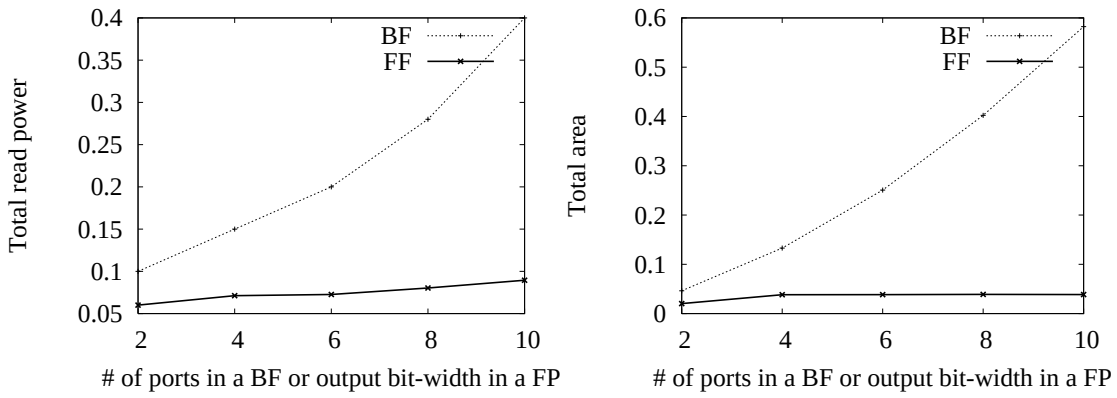
One method to determine the efficiency of a set representation scheme is to consider how many bits, m, are necessary for a set of n keys from a universe. An efficient scheme must not allow any false negative but can at most allow an f-positive of a fraction ϵ of the universe. As claimed in [49], the following inequality of m for a given required ϵ is made:

$$m \geq n \log_2(1/\epsilon) = nw = w + \cdots + w = \sum_{i=1}^{n} w. \tag{3.5}$$

Thus, an FF can be regarded as an array of n fingerprints (FPs) of $\log_2(1/\epsilon)$ bits for the approximate set representation. Since an FF does not have a constant time indexing mechanism like hash functions in a BF, knowing an index to a key's FP is other complicated search. However, once an index to the key's FP is known, the same

index can be used in a key table for a perfect match. Also, an FF needs 1.44 times less memory than a BF for required ϵ, based on Eq. (3.2) and Eq. (3.5).

In addition to the theoretical benefit, in terms of memory architecture using an FF SRAM requires a simpler memory read port design than an SBF SRAM does, so that area and power benefits in memory architecture are gained. Suppose there are two SRAMs for an SBF and an FF and the required false positive ϵ is 2^{-29} for 160Gbps. The SBF requires 29 read ports to query a key as a result of its simultaneous accesses while the FF needs only one read port to an FP of 29 bits. That is, an SBF SRAM with n keys is designed as a $w \times 1.44n \times 1$-bit memory array with w read ports of 1-bit output width while an FF SRAM is made of an $1 \times n \times w$-bit memory array with one read port of w-bit output width.



(a) Power (W) (b) Area (mm^2) for cost

Fig. 1. Comparison of power and area for a BF and an FF through CACTI.

Fig. 1 shows the power and area comparisons for an 11520-bit SBF and an 8000-bit FF in different ports or output bit widths. $.09\mu m$ process technology in CACTI 4.2 [35] is used. In Fig. 1 a) and b), gaps between the SBF and the FF are significant as w get larger. or example, an FF memory consumes 9.5 times less power, compared to an SBF while the former needs 76 times less area. Thus, for a

perfect match query, utilizing few-port SBFs in a binary search for a key's fingerprint in an FF and accessing a key table through the indexed fingerprint is a memory- and power-efficient hash scheme, and this scheme is introduced in the following section.

C. IP Lookup Using Hashing

Hash function maps a value in domain to a specific value in range uniformly. Thus, hash-based schemes, like a BFHT and an FHT, do not address the issue of supporting wildcard bits in prefixes. In this section, we present two kinds of schemes to support prefix match in a hash-based IP lookup: Controlled Prefix Expansion (CPE) and Prefix Collapse (PC).

1. Controlled Prefix Expansion

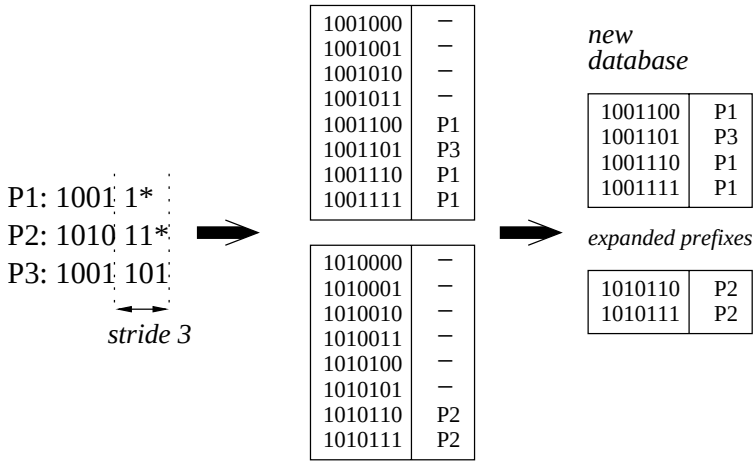


Fig. 2. Prefix conversion of a CPE with 3 prefixes in stride 3.

A CPE in [7, 16] is to transform a set of prefixes by combining prefix expansion and prefix capture to reduce any set of arbitrary length prefixes into an expanded set of prefixes in optimized sequence of length. With dynamic programming, it was applied to tries where the worst-case IP lookup time is $\mathcal{O}(W)$, where W is the length

of IP address. For a hash-based scheme, CPE was used in [16] to support wildcards in prefixes.

Fig. 2 shows a CPE mechanism with prefix database of 3 prefixes as a running example. In expanding bits and wildcard of 3 prefixes, prefix 1001101 is overlapped with prefix 10011*, so that the total number of expanded prefixes is 6. The 6 expanded prefixes in a new database are keyed to a hash function in hash-based IP lookup schemes. Although a CPE removes wildcards in prefixes for the hashing mechanism, the number of expanded prefixes along with the same number of next hops can become 2 times larger compared to the original prefix set. In general, the expansion is made in multi-fold and by simulation work on BGP tables, AS65000 and AS6447 [51, 52]. We found that the number of expanded prefixes increases as the stride size gets larger and that the number is about 5 times larger in stride 5. The reason is that a given prefix of stride l can be expanded to 2^l prefixes if there is no overlapping with other prefix, unlike prefix 10011* and 1001101.

2. Prefix Collapse

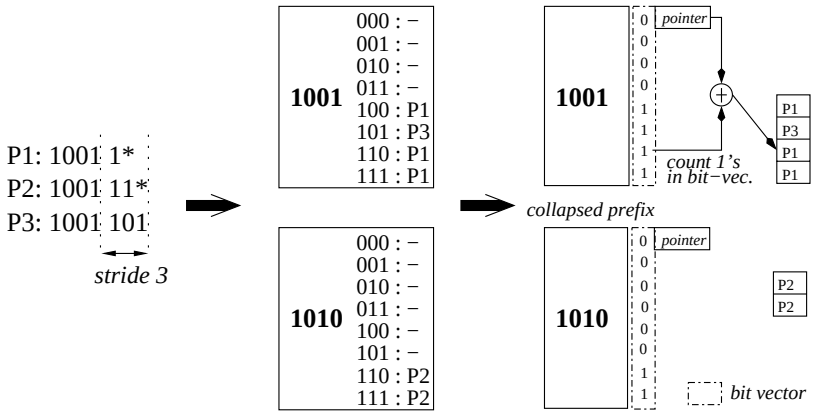


Fig. 3. Prefix conversion of PC with the same 3 prefixes of Fig. 2.

Unlike inflating the number of expanded prefixes and the next-hop informations

in a CPE, a PC converts a prefix of length x into a single prefix of shorter length x-l by replacing its l least significant bits with a wildcard [16]. The truncated prefix of length x-l is collapsed with others of the same x-l bits, so that the number of collapsed prefixes is reduced. Fig. 3 shows the prefix collapse mechanism with the same set of prefixes as in Fig. 2 for a CPE. Although the first conversion expands wildcards of the prefixes in stride 3 like a CPE, the second conversion adopts a bit vector indicating the relative index to a next-hop table. In addition, after expansion of the wildcard, the first and the third prefixes are same among 3 truncated prefixes of length 4. Thus, the final collapsed prefixes are prefix 1001 and 1010 with bit vectors (00001111) and (00000011). Compared to the example in Fig. 2, the number of collapsed prefixes is reduced, while the number of next-hops maintains the same as that for a CPE but yet it still increased 2 times than the original set.

3. IPv6 IP Lookup

The addressing architecture for IPv6 is detailed in RFC 3513. In terms of the number of prefix lengths in forwarding tables, the important address type is the global unicast address which many be aggregated. RFC 3513 states that IPv6 unicast addresses may be aggregated with arbitrary prefix lengths like IPv4 address under classless inter-domain routing. While this provides extensive flexibility, it is not foreseen that this flexibility necessarily results in an explosion of unique prefix lengths. The global unicast address format has three fields: a global routing prefix, a subnet ID, and an interface ID. All global unicast addresses, other than those that begin with 000, must have a 64-bit interface ID in the Modifed EUI-64 format. These identifiers may be of global or local scope; however, we are only interested in the structure they impose on routing databases. In such cases, the global routing prefix and subnet ID fields must consume a total of 64 bits. If these policies are followed, it could be anticipated that

IPv6 routing tables will not contain a significant difference from the current IPv4 tables except a prefix length distribution. Thus, hash-based IP lookup schemes can play a major role in saving memory and power for IPv6 as for IPv4, compared to TCAM- and trie-based IP lookup schemes.

D. Packet Classification Using Hashing

The issue of how to reduce the number of used BFs in processing a packet with n BFs is a paramount power concern in any packet processing [12, 14, 53] as well as network application including wireless sensor network [45]. However, in this section we formalize and restrict the issue to packet classification domain.

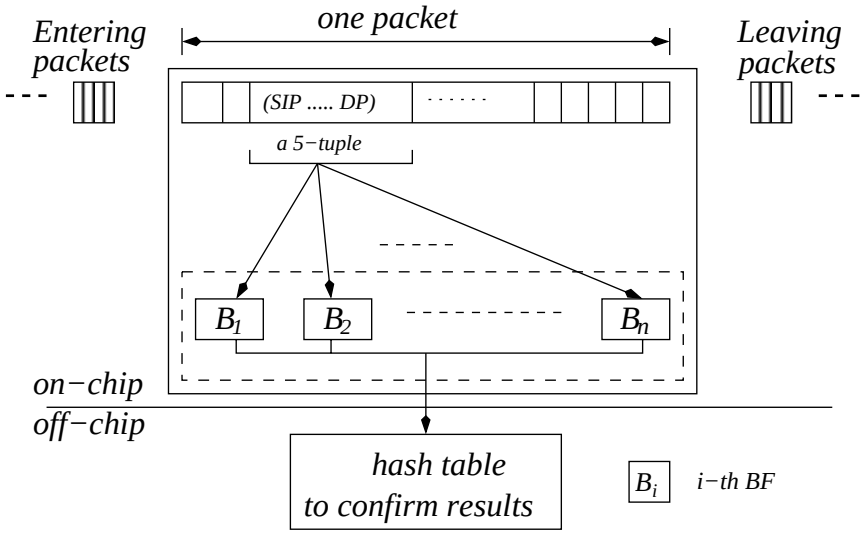


Fig. 4. Parallel packet classifier engine of n BFs in a given packet.

A parallel lookup with n BFs is a common configuration in packet classification [14] as shown in Fig. 4 where a 5-tuple of SIP, DIP, protocol, SP, and DP is extracted from a packet and a lookup of the 5-tuple is made among n BFs. Fast on-chip packet processing with n BFs is beneficial because it reduces the number of off-chip hash probes [12, 22]. Due to f-positives from the BFs, all positives are required to

be confirmed by a hash table of recorded flows. Due to QoS and security concern, providing a perfect match is necessary in packet classification. Thus, there is BFs' access contention to the hash table. BFs can be fabricated in on-chip due to memory efficiency while the hash table is located in off-chip due to its large size as in other schemes [12, 13, 20]. Thus, the packet lookup throughput is bounded to the processing time in the off-chip hash table.

The worst case throughput can be calculated in the following way: given a lookup of a minimum 40-byte packet, there are two kinds of lookups, an unsuccessful lookup (UL) in which a key is relentlessly searched although it does not exist in BFs, and a successful but time-consuming lookup (SL) in which a key is to be searched in BFs. Let t_s and t_u denote processing times in an off-chip hash table (HT) for an SL and a UL, respectively. Then, the packet lookup throughput in n BFs is calculated as

$$T = \frac{40 \cdot 8}{p_s\{t_s + t_u \cdot (n\text{-}1)f\} + (1\text{-}p_s)\{t_u \cdot nf\}} bits/sec., \tag{3.6}$$

where p_s is an SL rate and the nf and $(n\text{-}1)f$ terms explain the expected numbers of f-positives which is based on the binomial distribution of identical and independent BFs in an SL and a UL, respectively.

Based on Eq. (3.6), Fig. 5 shows the throughput where HT's processing time in an SL, t_s, is 1.001 times of 2ns in a modern T-RAM [54] and t_u is set to 0.5 times of 2ns. In the worst case of p_s=1, the lookup throughput with BFs of k=10 read ports can barely keep up with 160Gbps while BFs of k=15 read ports can meet the bandwidth. Thus, a large number of read ports in a BF memory are required for a high throughput, and avoiding irrelevant BFs of such a large number of ports for a lookup is preferable. In the following section, such an avoidance is made by a PPC which distributes lookups through small-sized BFs of a few ports, so that a subset of the lookups are processed in large-sized BFs in one clock cycle for a higher power

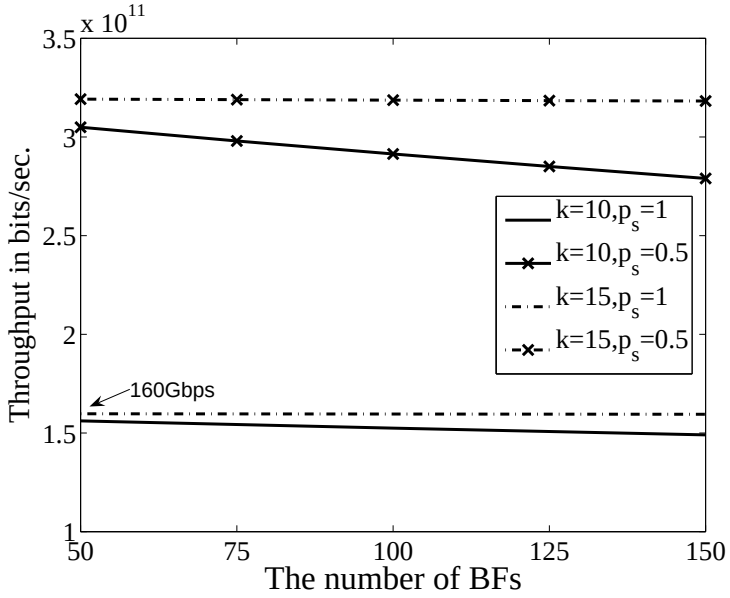


Fig. 5. Throughput comparison in a different number of BFs, p_s, and k.

and throughput efficiencies.

CHAPTER IV

A MULTI-TIERED PACKET CLASSIFIER WITH N BFS

This chapter introduces how to build an MPC and implement `insert`, `query`, and `delete` operations in an MPC for better lookup performance.

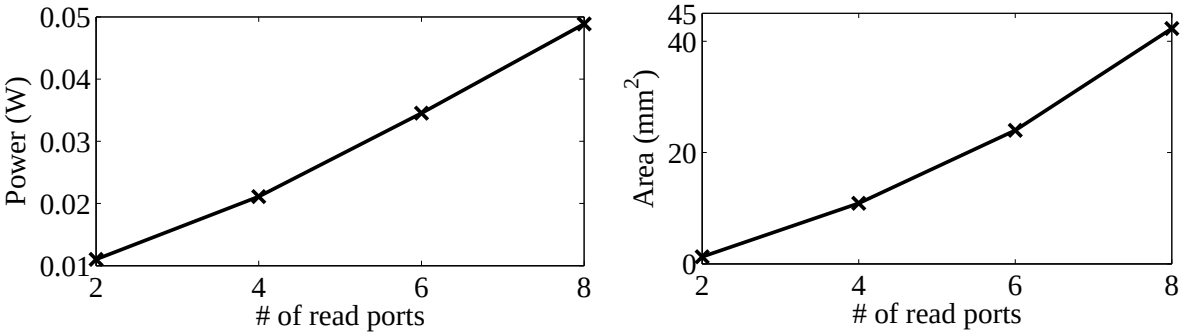


Fig. 6. Power and area in multi memory read ports for 64K×1-bit memory.

Each hash function corresponds to one random lookup in an m-bit BF. Thus, a BF having k hash functions for high throughput needs the exact same k of memory read ports in an m-bit memory module. Although state-of-the-art VLSI technology can fabricate memory with multiple ports, supporting more than 10 ports is tremendously hard as noted in a concise summary of the recent embedded memory technologies [55]. Fig. 6 shows such a difficulty in terms of the power and area costs measured by CACTI [35], according to the number of read ports in a single memory module. The conclusion from the figure is that the power and area costs is superlinear to the number of read ports. Thus, a BF is considered as a high computation element due to the large value of k for the high-speed router, and thereby reconfiguring such BFs for a power- and throughput-efficient lookup is necessary.

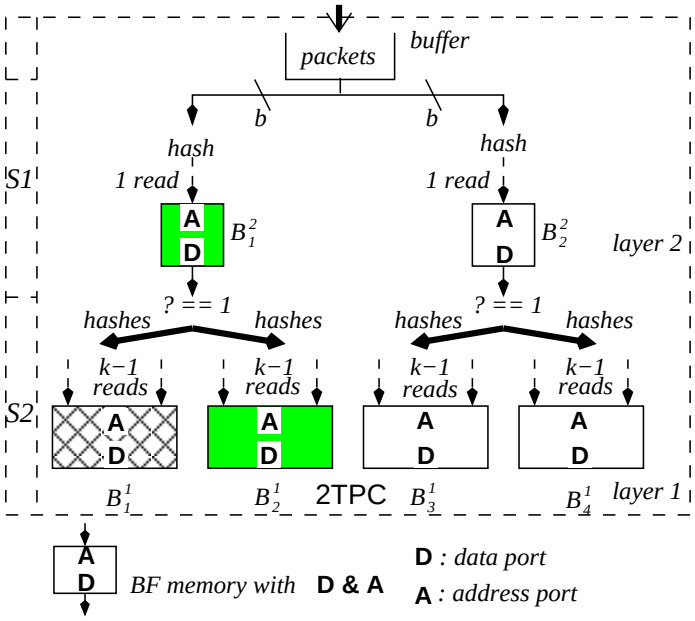


Fig. 7. Pipeline memory architecture of a 2TPC in a forest. S1 and S2 are pipeline stages. B_j^i means the j-th BF at layer i. n=4. k=w due to Eq. (3.3). w_2=1, w_1=k-1. b is a buffer size.

A. Building a Multi-tiered Packet Classifier

In this section, we derive mathematical proof that an MPC uses the same memory size as that of a PPC while the detailed insertion and query are mentioned in Secs. B and C. Fig. 7 shows a configuration example of an MPC, a 2-tiered PC (2TPC) on top of 4 BFs, in place of a PPC used in a dashed box of Fig. 4. Also, Fig. 8 shows a 3-tiered PC (3TPC) on top of 8 BFs. Given desired f-positive f=2^{-w}, the total PPC memory in bits with n BFs is $n{\cdot}m$, where m is a BF's memory based on Eq. (3.2). However, with *linear property* between m and n_i and an additive operation on memory size m_t, we can reconfigure BFs in a (r+1)-tiered way, r>0, while the same memory size, m_M, for an MPC is used as follows:

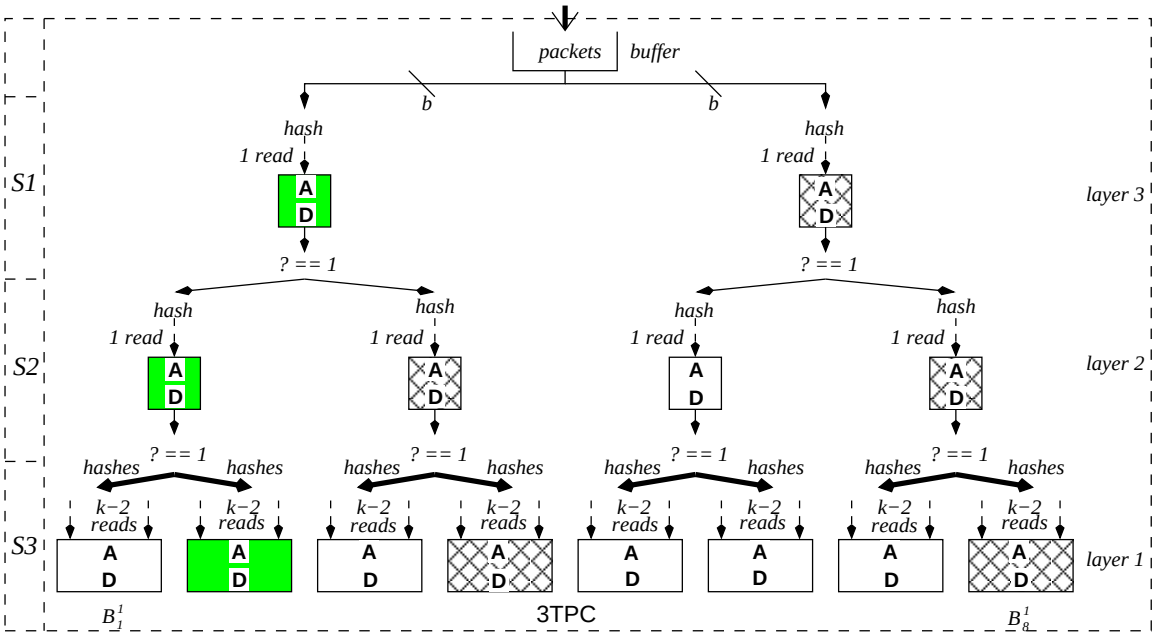
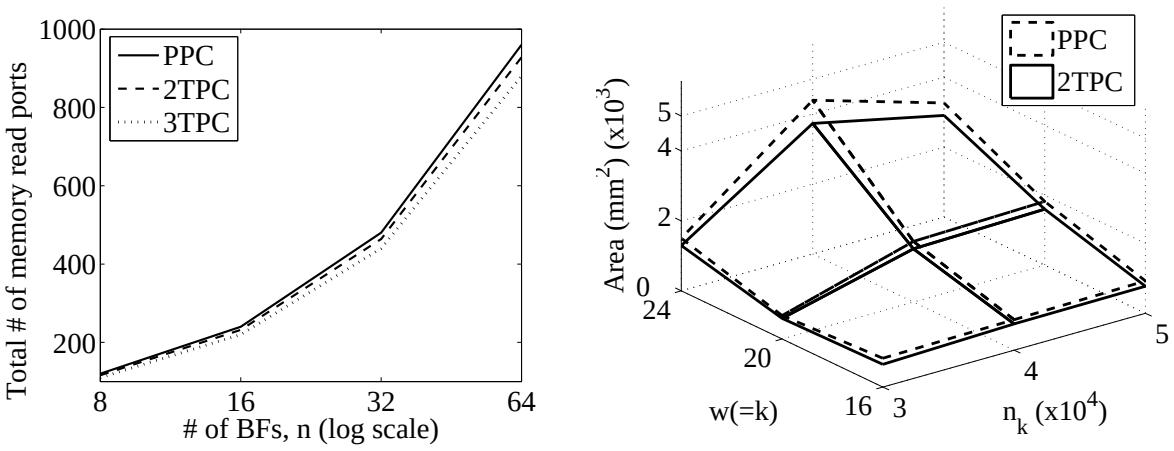


Fig. 8. Memory architecture of a 3TPC in a forest and *in pipeline*. B_j^i means the j-th BF at layer i. n=8. k=w due to Eq. (3.3).

$$
\begin{aligned}
n \times m &= n \times \{1.44 \cdot n_i \cdot \log_2(1/f)\} \\
&= n \times \{1.44 \cdot n_i \cdot w\} = n \times \{1.44 \cdot n_i \cdot (w - r + r)\} \\
&= n \cdot 1.44 \cdot n_i \cdot (w - r) + \sum_{t=1}^{r} \{n \cdot 1.44 \cdot n_i \cdot 1\} \\
&= \sum_{i=1}^{n} (1.44 \cdot n_i \cdot (w\text{-}r)) + \sum_{t=1}^{r} \sum_{i=1}^{n/2^t} (1.44 \cdot (2^t n_i) \cdot 1) \qquad (4.1) \\
&= m_1 + \sum_{t=1}^{r} m_{t+1} = m_M,
\end{aligned}
$$

where m_t is the total memory of BFs on layer t, r+1 is the number of tiers, $2^t n_i$ is the number of keys in B_i^t, and the lookup precisions of a BF on layer 1 and t, w_1 and w_t, are w-r and 1, respectively. Based on Eq. (3.2), the f-positives of BFs on layer 1 and 2 in a 3TPC are expected to be $2^{-(w-2)}$ and 2^{-1}, respectively, and the second term, $\sum_{t=1}^{r} \sum_{i=1}^{n/2^t} (1.44 \cdot (2^t n_i) \cdot 1)$, in Eq. (4.1) is the sum of small-sized BFs from



(a) The read port number (b) The area cost (w_2=1)

Fig. 9. (a) The total number of read ports in different number of BFs. w_3=w_2=1, w_1=13 for a 3TPC. w_2=1, w_1=14 for a 2TPC. f=2^{-15}. (b) 2TPC and PPC area costs with n=8 in .13μm process technology.

layer 2 to layer r+1. Also, a BF from layer 1 covers n_i elements, and a BF from layer 2 covers $2n_i$ keys. Generally, B_i^j covers all keys from $B_{2i}^{j\text{-}1}$ and $B_{2i+1}^{j\text{-}1}$, $1 \le i \le n/2$, $1 < j \le r$ in an MPC.

In this multi-tiered and pipelined configuration with b=1, power in accessing memory (or probing BFs) can be saved. For example, B_2^1 has a key and there is a lookup for the key. By preprocessing the lookup in stage S1 with B_1^2 and B_2^2, if B_2^2 returns 'no' in the lookup there is no need to probe B_3^1 and B_4^1. Thus, a power used to probe them can be saved.

In addition to the power concern, simply setting b to more than 1 does not achieve a higher throughput efficiency. Although Eq. (4.1)'s derivation shows that an MPC has the same memory size as a PPC, processing a lookup in small-sized BFs of one read port does not provide a higher throughput in large-sized BFs on a lower layer. For instance, even if b in Fig. 7 with w_2=1 is set to 2, a one-read-port BF on layer 2 cannot process 2 lookups in one cycle. Thus, the number of read ports in the small-

sized BF needs to be the same as b. In general, the number needs to be $b \cdot w_2$ for a throughput-efficient MPC. As suggested in [12], using mini-BFs with few read ports is the solution without degrading lookup accuracy. However, even if a BF is broken into several mini-BFs, the total number of read ports in the mini-BFs is the same as that of a PPC. Thus, breaking a BF into mini-BFs only gives the possibility of fabricating BFs for packet processing, not the benefit of high throughput. However, a proposed MPC has two benefits of few number of read ports and an area cost which can lead to fabricate small-sized BFs of multi read ports for a high throughput without area overhead.

Figs. 9(a) and 9(b) show such two benefits: the smaller number of fabricated read ports and the smaller area for a 2TPC. Fig. 9(a) shows the required numbers of read ports in fabricating a different number of BFs for a PPC, a 2TPC, and a 3TPC, respectively. In fabricating, a 2TPC and a 3TPC use 4% and 10% less number of read ports than a PPC in all cases. Fig. 9(b) shows 2TPC and PPC area costs in a different number of w and n_i, and in each case the area costs of using 4 mini-BFs for a BF are measured by using CACTI model [35].

Now, we show how to fabricate multi-ports in a small-sized BF without hardware overhead. There is a noticeable gap between dotted and solid meshes in Fig. 9(b), and the reason is that fabricating multi-ports in a small-sized memory does not need area as much as in a large-sized memory. Due to page limit, we did not plot the area costs for 2 through 5 read ports in a small-sized BF memory on layer 2. However, there is a small area increase for the multi-port memory, compared to a PPC's area. Thus, it is clear that the buffer size b can amount to 5 at the most. Also, utilizing dual reads on falling and rising edges in a clock [56] can double the memory read capacity and a lookup throughput (i.e. double data rate scheme does in DRAM and AMD Athlon64). Thus, the buffer size becomes twice and the maximum b is 10 without

memory overhead in an MPC.

B. Insert Operation in an MPC

`Insert` operation of a key in a BF on layer 1 is as simple as the key's insertion in a legacy BF. Similarly, on layer j, if a key to hash is assigned to B_i^j, the key is given to $B_{\lfloor i/2 \rfloor}^{j+1}$ for `insert` operation, $1<j\leq s$. The detailed procedure is shown in **Procedure** `insert` which does k_j times memory write on layer j. Therefore, the memory write complexity of one key insertion is $\sum_{t=1}^{s} k_t = w = k_P$ which is the same as a PPC, where k_P is based on Eq. (3.3). Also, note that the first vertically lined **for** can be in pipeline because BF memories on a layer are independent ones from other layers. Thus, in every cycle one key insertion is performed on the condition that B_i^1 on layer 1, $1\leq i\leq n$, supports multiports.

Procedure `insert`

Input: Key e and index i for a BF on layer 1
Output: Encoded 2TPC for key e

```
1 for layer j = 1 to s do
2 |   for t = 0 to k_j − 1 do                    // h_t is t-th hash func.
4 |     B^j_{⌊i/2^{j-1}⌋}[h_t(e)]=1;             // B is Mem. on SRAM for BF
5 |   end
6 end
```

C. Query Operation in an MPC

Unlike `insert` operation where only the involved BFs are accessed, `query` operation needs to access all BFs to find which BFs return 'yes'. Because except one involved BF the rest of irrelevant BFs give f-positives leading to packet misclassification, the irrelevant BFs in an MPC are not considered for probing, so that the BF access complexity in processing a lookup with n BFs is far less than n. To provide such a

complexity, we split the memory of a PPC into small-sized BFs and large-sized BFs in multi-tiers, and they are connected in binary trees. Then, accesses to large-sized BFs are made only if their parents of small-sized BFs return 'yes' (or value 1 in D) as shown in Fig. 7. Also, BFs in multi-tiers can be in pipeline so that there is no performance degradation. Before the detail procedure, let us introduce definitions of a true path and a false path entangled in an MPC.

Definition 1 (True Path)

In ***query*** *operation among a forest shown in Fig. 7, a true path, t-path, occurs. It is composed of shadowed BFs from a root of a tree to return 'yes'. These were involved in the previous* ***insert*** *operation for a key. The length of a t-path is 2 in case of 2TPCs.*

For example, if a key is assigned to set B_2 in PBFs, the BFs on a t-path for 2TPCs are B_1^2, B_2^1 as shown in Fig. 7. From the above definition, in `query` operation all BFs on the t-path should return 'yes' for a given key as a legacy BF returns 'yes' because each BF has the key as a member.

Unlike a t-path, a false path is made from a group of BFs giving f-positives so that packet misclassification occurs. The detailed definition of a false path is as follows:

Definition 2 (False Path)

In ***query*** *along consecutive layers, a group of BFs giving f-positives makes a false path, f-path. The series of BFs can be from either the off-branch BFs from a t-path or a root of a tree to the bottom of the tree as shown in the checked boxes of Fig. 7 and 8.*

The f-positives by the BFs, neither stemming from a branch of a t-path nor being a complete path in a tree among a forest, can not contribute an f-path by the definition. Also, the number of f-paths means the number of packet misclassifications. An important fact from the above definition is that the probability of misclassification for an f-path contributing one packet misclassification is cumulatively calculated in product of each f-positive on the f-path.

1. False classification in a successful lookup

We divide a lookup in two ways: 1) a successful lookup and 2) an unsuccessful lookup. In network application, given a packet a router needs to determine the destination based on a flow table about classification information. If there is a flow in the table, we call the lookup an SL. Now, we show the misclassification probability in an SL.

By a recursive definition, the probability $P_a(i)$ that root a in a binary tree has i packet misclassifications is the product of the following three: the probability of an f-positive in root a of the binary tree, the probability that a left subtree has i-j packet misclassifications, and the probability that a right subtree has j packet misclassifications as the following:

$$P_a(i) = \sum_{j=0}^{i} f_a \times P_l(i-j) \times P_r(j), \tag{4.2}$$

where f_a is the probability of an f-positive from BF a, and as a base case, $P_{B_1^1}(1)=f_{B_1^1}$. Finally, the dominant probability, $\mathcal{P}_s(1)$ that a single packet misclassification occurs across a forest is the following:

$$\mathcal{P}_s(1) = \sum_{j=1}^{r-1} P_{B_t^j}(1) + \sum_{i=2}^{n/2^{r-1}} P_{B_i^r}(1), \tag{4.3}$$

where r is the number of tiers, the first term is the summation of Eq. (4.3)'s probabilities of BFs attached on a t-path and the second term is the summation of

probabilities of the remaining trees among the forest.

2. False classification in an unsuccessful lookup

Since all packets are not under specific flows based on a flow table, a UL is important as much as an SL. Unlike an SL, in a UL there is no t-path. This means that what a BF returns, if any, is an f-positive. The dominant probability, $\mathcal{P}_u(1)$ that a single packet misclassification happens in a UL is

$$\mathcal{P}_u(1) = \sum_{i=1}^{n/2^{r-1}} P_{B_i^r}(1). \tag{4.4}$$

Procedure `query` shows the details of query operation on an MPC. The code in

Procedure `query`

Input: Forest F of binary trees for an MPC and key e
Output: Set S for a true path and a group of false paths
1 **for** *tree* $T \in$ *in forest* F **do**
2 | $S = S \cup$ `query_BT`(T, e) ;
3 **end**
4 **return** S;

the vertical line of **Procedure** `query` can be implemented *in parallel*. Also, it calls subroutine `query_BT` which is working *recursively* and *in pipeline* on each layer in a binary tree to check a BF for the key e as a legacy BF does. Also, pipelining on layers in a binary tree makes it sure that the `query` complexity is $\Theta(1)$ as a PPC' complexity is.

Based on Eqs. (4.3) and (4.4), the expected packet misclassification considering SL and UL rates is

$$p_s \sum_{i=1}^{n-1} i \cdot \mathcal{P}_s(i) + (1-p_s) \sum_{i=1}^{n} i \cdot \mathcal{P}_u(i) = p_s \cdot E_s + (1-p_s) E_u, \tag{4.5}$$

where p_s is an SL rate, and E_s and E_u are the average packet misclassifications for

an SL and a UL, respectively.

There is a minuscule classification performance degradation in using an MPC. Fig. 10 shows the average packet misclassification of a PPC and a 2TPC based on Eq. (4.5) with a rate of successful lookup p_s. There are three important considerations: 1)

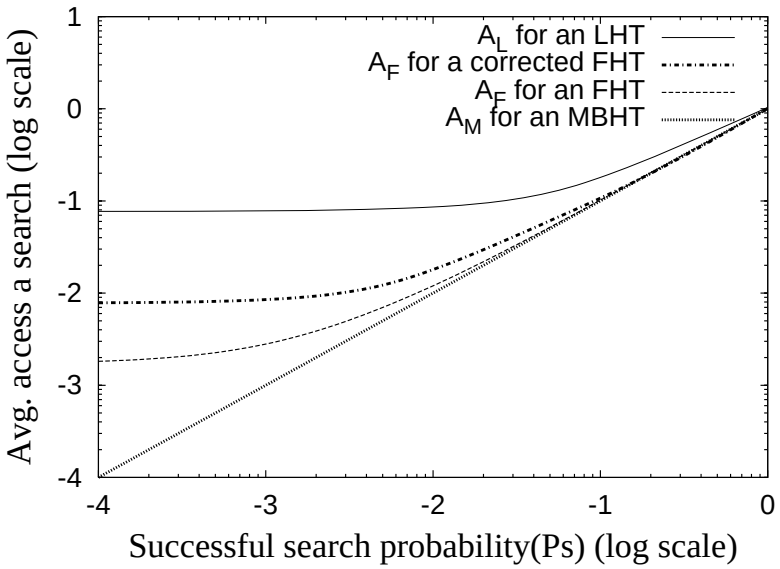


Fig. 10. The average packet misclassification for a PPC-n and a 3TPC-n in a different SL rate. f=2^{-w}=2^{-30}, w_1=28, w_2=w_3=1. $n \in \{32, 64, 128\}$.

Given desired f-positive, f, as long as the n is larger, the value of the average packet misclassification is getting larger due to bigger binomial coefficient value $B(f, n)$. 2) Given the same memory size, the probabilities of PPC-n and 2TPC-n for a UL are the same while in a dominant rate of an SL, there is a minuscule difference, 2E-9, between them. 3) The difference gets smaller as long as the n is larger. In conclusion, as long as the number of BFs, n, and the rate p_s are larger, the difference of packet misclassifications between a PPC and a 2TPC is negligible. The one-packet misclassifications of Eqs. (4.3) and (4.4) show the same phenomenon shown in Fig. 10.

D. Delete Operation in an MPC

`Delete` operation is not as easy as `insert` because a basic BF in [12–14, 17] does not support deletion of a key which was encoded in the BF. If a counting BF [39] or a low power counting BF (L-CBF) [57] is adopted, `delete` operation can be as easy as the basic BF. Line 4 in `Insert` procedure, $B^j_{\lfloor i/2^{j-1} \rfloor}[h_t(e)]$=1, shows bit setting for the basic BF. However, if `delete` operation is provided the line needs to be changed to $B^j_{\lfloor i/2^{j-1} \rfloor}[h_t(e)]$++ as a counting BF is used at line 4 for `delete` procedure.

Procedure `delete`

Input: Key e and index i for a BF on layer 1
Output: Deleted 2TPC for key e

```
1 for layer j = 1 to s do
2 |   for t = 0 to k_j − 1 do                    // h_t is t-th hash func.
4 |     B^j_{⌊i/2^{j-1}⌋}[h_t(e)] − −;           // B is Mem. on SRAM for BF
5 |   end
6 end
```

E. Simulation Result for an MPC

CACTI [35] models SRAM architecture in terms of area, access time, and power. With the help of CACTI model, we measured throughputs and powers of PPC and MPC with IP traces which are from NLANR PMA and Internet Traffic Research Group [58]. We assume that a PPC needs one cycle to process a packet lookup to n parallel BFs, and in an MPC a small-sized BF with multiports can process a group of lookups in one cycle while a large-sized BF with multiports processes a lookup in high precision. The used IP traces are PUR, SDA, FRG, and PSC which have 19.4K, 29.5K, 39.7K, and 37.9K flows as rules, respectively. The simulation used 193.3K, 292.2K, 337K, and 314.3K packets in flow identification with different number of router ports, each having the same number of flows equally.

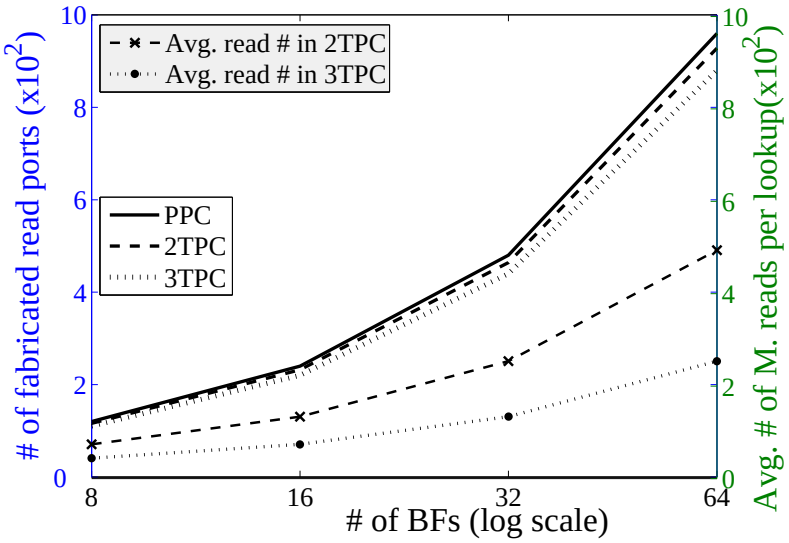
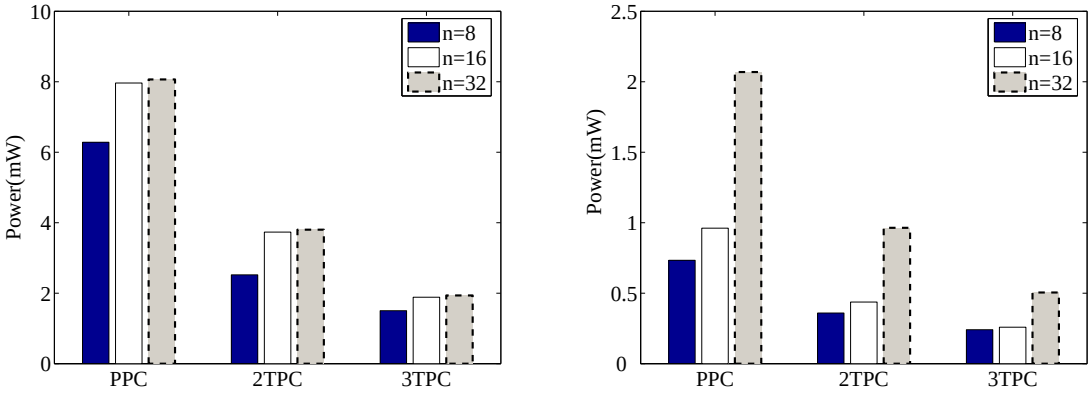


Fig. 11. The number of read ports and average number of memory reads in different number of BFs. w_3=w_2=1, w_1=13 for a 3TPC. w_2=1, w_1=14 for a 2TPC. f=2^{-15}.

1. Experiment for Power

For power estimation, each pipeline stage is designed to process a single lookup, contrast to a multi-lookup capability in a throughput experiment of the following section. For theoretical comparison, we calculate the average number of memory reads per lookup in MPCs based on Eq. (4.5). As suggested in [12], using mini-BFs with few read ports is the solution without degrading lookup accuracy. However, even if a BF is broken into several mini-BFs, the total number of read ports in the mini-BFs is the same as the number of the original BF. Thus, breaking a BF into mini-BFs gives only the possibility of fabricating BFs for high throughput in packet processing, but it does not benefit reducing power and area costs. However, the propose MPCs offer benefits such as fewer number of read ports and reduced power during lookup operation.

Fig. 11 shows such two benefits: the smaller number of fabricated read ports and



(a) AMP (b) PSC

Fig. 12. Power consumption by two traces in PPCs, 2TPCs, and 3TPCs. Also, $n \in \{8, 16, 32\}$.

the smaller number of memory reads for a lookup in 2TPCs and 3TPCs. Suppose 15 ports are required in a BF's fabrication in PPCs. The first three solid lines show the required number of read ports in fabrication of different number of BFs for PPCs, 2TPCs, and 3TPCs, respectively. The other two marked lines are the number of operational memory reads for a given lookup. In fabricating, 2TPCs and 3TPCs use 4% and 10% fewer number of read ports than PPCs. In addition, for a given packet lookup, the average number of operational memory reads in 64 BFs is rapidly reduced to 1.9 and 3.8 times memory reads for 2TPCs and 3TPCs, respectively, compared to PPCs. Thus, we are certain that during a lookup in MPCs less power is consumed in a real packet classification.

Table III. Power value by CACTI in PPC(31Kx1, 20 ports), 2TPC(29Kx1, 19 ports), and 3TPC(14Kx1,18 ports).

	A BF power(W)	**A small-sized BF power(W)**
PPC	0.120	N/A
2TPC	0.110	0.002
3TPC	0.097	0.008

Table III shows the typical power value used in CACTI in the case of AMP trace. Based on these values, we measure the power for other trace PSC as shown in Fig. 12. Fig. 12 shows the average power of four traces by 10 runs in different configurations (PPCs, 2TPCs, and 3TPCs). We set w=20 for a PPC, and the lookup precisions of a large-sized BF in layer 1 are set to 19 and 18 for 2TPCs and 3TPCs, respectively. The power efficiency ratios of 3TPCs against PPCs in AMP and PSC are at most 4.2, 4.1, 3.7, 3.2, respectively. Also, the power efficiency ratios of 3TPCs against 2TPCs in AMP and PSC are 1.9, 1.9, 1.7, and 1,5, respectively. From these results, it is clear that an MPC is more power efficient that a PPC, and as the number of multi-tiers becomes larger, the power efficiency becomes better.

2. Experiment for Throughput

The throughput is defined as the number of packets over the number of simulation cycles to process the whole IP traces, and we assume that each small- or large-sized BF takes one clock cycle to process a lookup. Fig. 13 shows the average throughput ratios of four traces by 10 runs in a 2PC architecture where each small-sized BF on layer 2 has a b-sized buffer to process b packets in the buffer in one cycle. Once they process packets in the their buffers, the results are forwarded to large-sized BFs on layer 1. A BF on layer 1 works on a partially processed packet only if a parent BF of the BF returns 'yes' to the packet. Thus, if a BF on layer 2 returns 'no' for a packet, the children BFs of large size can process other following packets, leading to a higher throughput. In each subfigure, in all different numbers of BFs the larger is the buffer size, the higher throughput ratio is, proving that our MPC gives a higher throughput performance than a PPC. At most 2.0 times throughput was observed in PSC trace. Although we simulated a case of, at most, 64 BFs, our MPC shows higher throughput than those in Fig. 13 if a larger number of BFs and buffer size b is used.

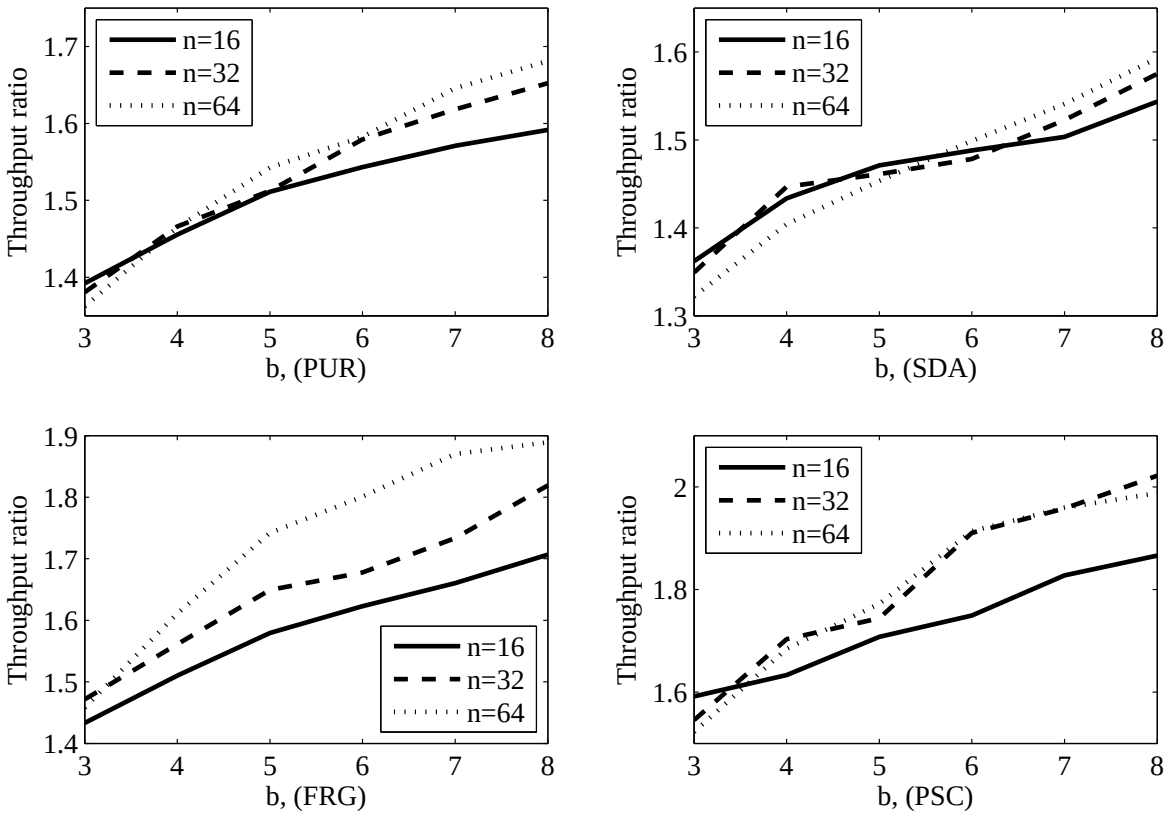


Fig. 13. Throughput ratios of a 2TPC against a PPC with four traces in different number of buffer size b and n BFs. w_1=28, w_2=2.

CHAPTER V

MULTI-PREDICATE BLOOM-FILTERED HASH TABLE

In this chapter, we propose a novel hash architecture that uses a set of BFs *in parallel* for a perfect match. BFs used in our hash mechanism are designed to support a multi-predicate rather than a simple membership tester, i.e. binary-predicate, of a legacy BF. Our scheme using multi-predicate BFs reduces the memory size in base-2^x number system by x times compared to that of base-2^1 number system with a binary predicate BF, where x is a positive integer larger than 1.

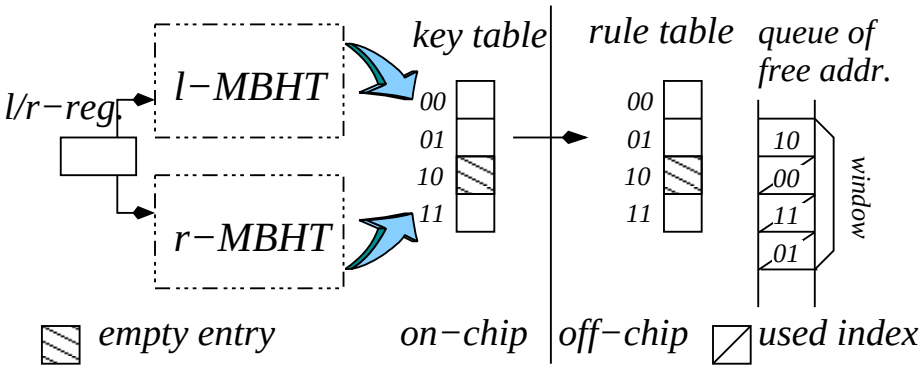


Fig. 14. Macro view of an MBHT in on/off-chip memory of base-2. n=2^2.

Fig. 14 shows the macro view of our architecture with two MBHTs, l-$MBHT$ and r-$MBHT$, and a key table residing in on-chip memory while there are a rule table of n=2^2 entries and a queue of free addresses in off-chip memory. One of MBHTs is involved in `insert` operation depending on l/r-register. This register is to be switched l or r whenever n `insert`s are made on one MBHT, so that once a window of the queue is used up the peer MBHT is cleaned up for future `insert`. Through this rotation, without counting BFs costing 4 times memory, dual MBHTs can provide seamlessly `insert` and `delete` operations for incremental updates of rules. In contrast, both of MBHTs are involved in `query` and `delete` operations because it is not known where

a wanted key is located.

A. Index Address to a Key Table in Base-b

Unlike a legacy BF [12–14], a new hashing architecture is proposed, and it is capable of indexing a key table of on-chip memory in the base-b number system for perfect match, and accessing to a rule table in off-chip by the index address of the matched entry for a given key. Although the BF is returning 'yes' approximately, a MBF is capable of telling arbitrary per-key information associated with a given key when a membership test is met.

Assume there are n keys to hash in key and rule tables where the keys are saved in contiguous and flat memory space. In perfect hash like an LHT and an FHT [15], the buckets are an array of pointers to linked lists in off-chip. These pointers are in the form of a binary address and the total number of buckets, n_b, is determined in relation to collision rate. Thus, a HT in an LHT and an FHT is an array of n_b pointers of length $\log_2 n$. However, on-chip memory for the array partitioned to MBFs and MBFs are grouped column-wise The n keys are saved in an arbitrary order at index A^b of the two tables, where b is the base-b number system in a positive integer. Given n keys and the base-b number system, there are $r = \log_b n$ digits in an index for a key in a key table, and each of r digits in the index $A^b = a_0 a_1 \cdots a_{r\text{-}1}$ is expressed in $\log_2 b$ bits, i.e. $a_i \in \{0, \cdots, b\text{-}1\}$. Denote $a\text{-}BF$ a multi-predicate BF embedded in on-chip memory module, implying that if a membership test is met value a in $\log_2 b$ bits for the base-b number system is considered for a part of an index of a given query. Provided that the address space is based on a number system of base-b, partitioning the address space with a set of $a\text{-}BF$s, $a \in \{0, \cdots, b\text{-}1\}$, is made so that each a_i of base-b in A^b is to be covered by $a_i\text{-}BF^i$, $0 \le i \le r\text{-}1$. After the digits are partitioned

column-wise by the set of MBFs, a_i-BF^i in them is involved for a_i in an `insert` operation described in Sec. C, and the relevant a-BF from each column is to imply value a in the `query` operation explained in Sec. C.

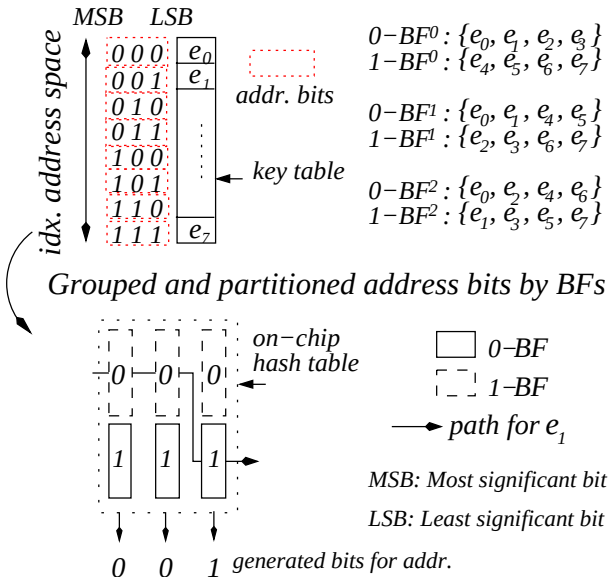


Fig. 15. Partitioning of 8 elements in base-2 with 0-BFs and 1-BFs.

Fig. 15 shows an example of the base-2 number system with 2^3 keys and three pairs of 0-BFs and 1-BFs. The indexes in the upper figure are drawn in a rectangle of 0- and 1-bits based on the base-2 number system, where each column has the same number of 0/1 digits as shown to the right table. Below, 8 keys are regrouped in every column according to their bits in the column, so that each of 0-BF^vs and 1-BF^vs, $v \in \{0, 1, 2\}$, has its own set for `insert` as shown in the right side. For instance, suppose e_1=e_{001_2} is to be saved at address 001_2 of off-chip memory. In an MBHT, 0-BF^0, 0-BF^1, and 1-BF^2 from column 0, 1, and 2, respectively, are involved in saving e_1 as shown in the figure.

The base-2 number system used in Fig. 15 can be expanded into an arbitrary number system for the benefit of memory efficiency, as shown in Fig. 16. All address

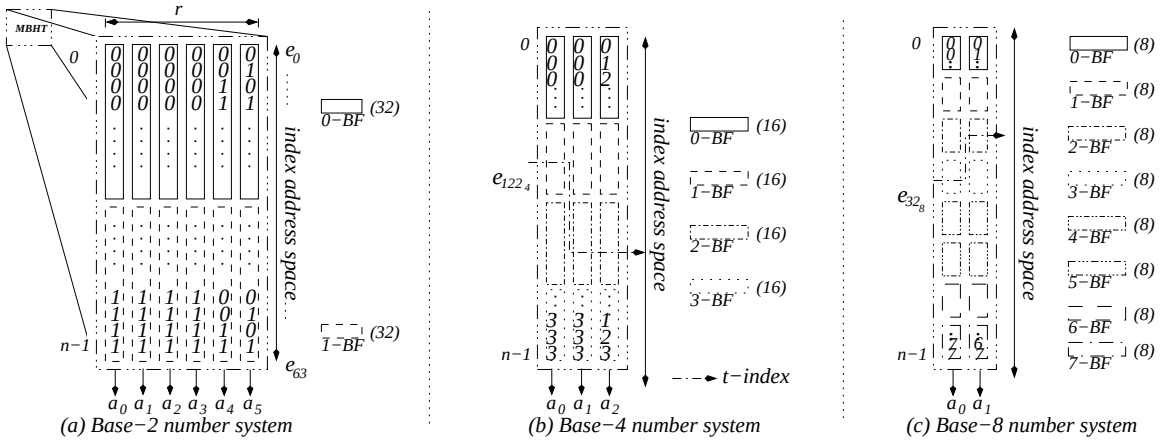


Fig. 16. Conversion of the base-2 number system to base-4 and base-8 for 64 elements. $n = 2^6$. By (X), X means the number of the same digits in a BF.

spaces in subfigures are partitioned column-wise and grouped by MBFs. The address space for 2^6 elements in the base-2 number system in Fig. 16 (a) is transformed to other address spaces of base-4 and base-8 number systems in Fig. 16 (b) and (c), respectively, resulting in the fewer columns in each address space. However, this transformation does not affect addressing an index to a key table. For example, suppose item e_{011010_2} for base-2 is located at 011010_2. This can be at 122_4 and 32_8 of base-2^2 and base-2^3, respectively, as shown in Fig. 16 (b) and (c).

Even if the address space in one column is partitioned by b MBFs in base-b system, they can be accessed with the same memory address by stacking MBFs in the following way for hardware implementation: On-chip memory for b MBFs is an array of words of b bits so that given an address to on-chip memory indicated by a hash function b MBFs are involved. Therefore, if all k words indicated by k hash functions for a-BF have bits of value 1 in offset of a in the words, a-BF has a correct membership test in `query` for a given key and the offset a is used for a part of address A^b. In this way the number of on-chip memory modules is r and they work *in parallel* for `insert`.

B. Memory Efficiency with a Larger Base-b

With an invariant about addressing systems shown in Fig. 16, given the base-b_1 number system and requirement of f, *Linear Property* of Lemma 1 regarding variables m and n claims that even if the number of new BFs, b_2, is increased in a column in new base-b_2, the total memory size for the column remains the same. The reason is that although the number of elements to hash for each new BF in the column is reduced, the total number of items for the new column in base-b_2 is the same as that for base-b_1.

In general, considering two MBHTs the total memory usage in bits for the base-b number system as a function of a requirement of $f{=}2^{-w_M}$ is calculated as follows:

$$\begin{aligned} M_b(f){=}2 \times C \times B &= 2 \times \left\{ \log_b n \right\} \times \left\{ b\,(1.44(n/b)\log_2(1/f_M)) \right\} \\ &{=}2 \cdot \left\{ \frac{\log_2 n}{\log_2 b} \right\} \cdot \left\{ 1.44n \log_2(1/f_M) \right\} {=} 2.88 \frac{n w_M \log_2 n}{\log_2 b}, \end{aligned} \tag{5.1}$$

where w_M is the precision of query operation, C is the number of columns, and B is the number of bits for a series of MBFs in one column. From this equation, denominator term $\log_2 b$ makes M_b smaller as it increases provided that n and f are constants. This is manifested in Fig. 17 showing the total memory usage in bits considering only MBFs in several number systems based on Eq. (5.1). Along with n axis of $b{=}2$, M_b increases greatly for a given f due to $\log_2 n$ and n terms in Eq. (5.1). Similarly, the change rates in axes of f and n for a smaller b are much larger than those of a larger b. Furthermore, the gap of M_bs among different bs is large enough that the saved memory can be used to reduce an f-positive of each MBF. Thus, rather than using base-b_1, using a larger base-b_2 number system is advantageous because of $\log_2 b_2 / \log_2 b_1$ times on-chip memory saving, that is, $M_{b_1}/M_{b_2}{=}\log_2 b_2 / \log_2 b_1$. However, choosing the appropriate base system depends on the current technology

of memory hardware. For example, b=2^{20}, the largest base system, could be the best choice for n=2^{20} because that gives the highest memory efficiency. In contrast to a theoretical benefit, in real hardware implementation it is very hard to probe sequentially all 2^{20} bits in a word indicated by a hash function to find a bit position having value 1. Even worse, such k words by k hash functions need to be checked for returning multi-predicate value for the involved MBF, taking an unsupportable time in hardware.

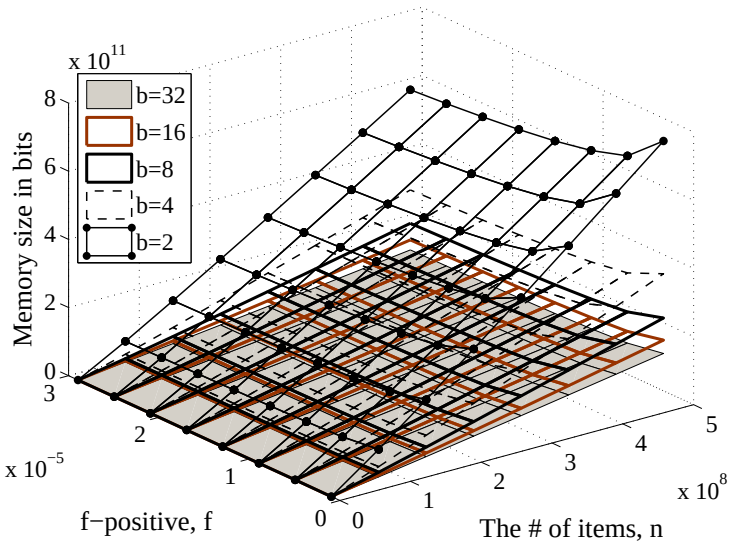


Fig. 17. Memory size M_b for $b = 2, 4, 8, 16,$ and 32 with f and n.

C. Insert Operation in an MBHT

The detailed procedure of the `insert` operation is described in **Algorithm** `insert` where r on-chip memory M_{on}s of m words of b bits, m=$1.44nw_M$ by Eq. (3.2), are involved. Address A fed to **Algorithm** `insert` is provided by a queue of free addresses. The first vertically-lined **for** loop in it is executed *in parallel* at each column. Also, the second **for** loop is done *in parallel*, as long as a conventional

BF support k hash functions *in parallel*. [55] asserts that fabrication of 6 to 8 read ports in on-chip memory is attainable. Even if the needed hash functions are larger than the attainable number of ports, splitting into several on-chip memories with the attainable port numbers is a solution as suggested in [12]. Therefore, the time complexity in on-chip memory is $\Theta(1)$ on the conditions that hash functions return indexes in constant time, and each column conducts hashing in parallel. Moreover, the number of off-chip memory accesses through $M_{off}[A]$ is exactly 1 because a rule for item e is saved in the designated address A as shown in the last line. Thus, the complexity of **Algorithm** `insert` for off-chip memory access is $\Theta(1)$. In contrast, an FHT was calculated to be a time complexity of $\mathcal{O}(nk^2/m + k)$, which is not suitable for a dynamic update in packet processing [8].

Algorithm 4: `insert`(x, e, $rule$, A)

Input: x-MBHT $x \in \{l, r\}$, key e, its $rule$, and address $A = a_0a_1 \cdots a_{r\text{-}1}$ in base-b
Result: Encoded MBHT for key e

```
1 for column i = 0 to r − 1 do                          /* On-chip Op.  */
2 |   for t = 0 to k − 1 do
3 |   |   g = h_t(e);                                  /* hashing, g ∈ {0, ··· m-1} */
4 |   |   M^i_on[g][a_i] = 1;                          /* the size of M^i_on=m × b */
5 |   end
6 end
7 M_key[A] = e ;                                        /* a key table in on-chip */
8 M_off[A] = rule ;                                     /* Off-chip memory access */
```

D. Query Operation in an MBHT

After all elements are saved contiguously in off-chip memory and encoded in a set of MBFs in on-chip memory, the remaining and ultimate goal of an HT is to search for an item by a fast `query` operation. There are two kinds of search patterns: a successful search (SS) in which an item is found; and an unsuccessful time-consuming search (US) for an item that does not exist in an HT.

Before two kinds of searches with possible false access to off-chip memory are examined, let definitions of a true index and a false index introduced.

Definition 3 (True Index)

A true index, or t-index, is defined as a series of MBFs assigned to encode a key. They are interconnected and back-to-back of each other from column 0 *to column r-1, where r is the number of columns in the base-b number system, making a sequence of full index address bits. The sequence of bits is also matched with an arbitrary memory address associated with a key saved in key and rule tables.*

For instance, key e is to be saved at address 122_4 in base-4 and 32_8 in base-8 as shown in Fig. 16 (b) and (c), respectively. In base-4, for sequence 122_4 1-BF^0, 2-BF^1, and 2-BF^2 are involved to save the key e while in base-8 for sequence 32_8 3-BF^0 and 2-BF^1 are for a t-index of key e. From the definition of a t-index, the following corollary can be concluded:

Corollary 1 *Once key e is saved at index A with a series of* $r = \log_b n$ *MBFs, i.e.* a_0-$BF^0 \cdots a_{r\text{-}1}$-$BF^{r\text{-}1}$*, in base-b, the involved BFs should return 'yes' making* $a_0 \cdots a_{r\text{-}1}$ *in the* **`query`** *of key e as a legacy BF always returns 'yes' for true membership.*

Due to the independent and identically distributed (i.i.d) property of BFs, it is possible that irrelevant BFs could return their f-positives in the `query` operation. Thus, due to the irrelevant f-positives, a false index, which is defined as the following, happens:

Definition 4 (False Index)

In **`query`** *of key e, in each column i of an MBHT, a group of MBFs in column i not pertaining to a t-index for* **`insert`** *of the key can return their f-positives. By*

the f-positives, the indexes in a word of b bits in the column i could lead to a false index, f-index, with other MBF^js, $j \neq i$ involved in the ***insert****. Thus, an f-index is a combination of index values of MBFs irrelevant and relevant to the* ***insert*** *of the key and MBFs responding to a membership test for the key return their indexes in a word of b bits. Also, the length of an f-index should be $r = \log_b n$.*

Given `query` of a key, a set of a t-index and f-indexes should be probed to guarantee that the key exists or not in a key table, implying *perfect* match unlike *approximate* match in [12, 14, 17]. In our `query` for perfect match, the numbers of f-indexes for an SS and a US are at most *n-1* and n, respectively. These numbers are comparable to the numbers of memory accesses for an SS and a US in a linked list of an LHT and an FHT. However, the difference between an indexing to a key table in an MBHT and a sequential access in a linked list of an FHT is that a key table resides in on-chip while a pair of a key and its rule exists in off-chip, so that at most one off-chip memory access is made in a MBHT while more than one access are necessary in an FHT. Most importantly, to find a matched key a few number of indexes to a key table in an MBHT can be processed in one cycle in parallel while in a sequential access in a linked list of length t, at most t cycles are necessary. Thus, it is possible that although an MBHT with less memory can give more f-indexes, they are processed in one cycle, so that less memory can be used while high bandwidth is preserved with low collision rate. The next important step is to recognize a t-index and annul a series of false positives randomly scattered in an MBHT so that the possibility of an f-index can be reduced.

1. False indexing for an SS in an MBHT

We have explained the definitions of a t-index and an f-index and how they can both occur in the `query` operations on an MBHT. Now, we derive and calculate the probability of the number of false accesses, i.e. f-indexes, in an SS. In a `query` for an SS, at least one MBF in each column needs to return its index value in k words, so that a sequence of $a_0 a_1 \cdots a_{r\text{-}1}$ of length r forms the full address A, i.e. t-index. Furthermore, in case of an f-index, false addresses can be created through f-positives by each irrelevant MBF in each column.

Suppose X_i^s is a random variable of the number of f-positives from MBF^i irrelevant to a t-index of an SS. Due to the i.i.d f-positives of b-1 BFs in a MBF, the probability density function of X_i^s is a binomial distribution, $B(b\text{-}1, f)$. Also, assume that the random variable X^s for an SS denotes for the total number of f-indexes in a given `query` operation. Then, random variable X^s is defined as the product of random variable X_i^ss, i.e. $(\prod_{i=0}^{r-1}(X_i^s+1))$-1, because of the i.i.d property of each column and the probability of $X^s = x$ is the following

$$\begin{aligned} Pr\{X^s{=}x\} &= \sum_{(x_0+1)\cdots(x_{r\text{-}1}+1)=x+1} Pr\{X_0^s{=}x_0, \cdots, X_{r-1}^s{=}x_{r-1}\} \\ &= \sum_{(x_0+1)\cdots(x_{r\text{-}1}+1)=x+1} Pr\{X_0^s{=}x_0\} \cdot Pr\{X_1^s{=}x_1\} \cdots Pr\{X_{r\text{-}1}^s{=}x_{r\text{-}1}\}. \end{aligned} \tag{5.2}$$

For example, $X^s{=}0$ means that each layer i does not have any random variable X_i^s larger than 0. Therefore, the probability becomes

$$Pr\{X^s = 0\} = Pr\{X_0^s = 0\} \cdot Pr\{X_1^s = 0\} \cdots Pr\{X_{r-1}^s = 0\}. \tag{5.3}$$

Also, in the cases of one and two f-indexes their probabilities are derived from the following:

$$Pr\{X^s = v\} = \sum_{t=0}^{r-1} Pr\{X_t^s = v\} \prod_{t' \neq t}^{r-1} Pr\{X_{t'}^s = 1\},$$

where $v \in \{1, 2\}$ because prime numbers 2 and 3 can be factored in the only one way: $2 \times 1 \times \cdots \times 1$ and $3 \times 1 \times \cdots \times 1$. Also, the mean of X^s is calculated based on the i.i.d property of X_i^s as shown

$$\begin{aligned} E[X^s] &= \sum_{t=0}^{n-1} t \cdot Pr\{X^s = t\} = E[(\prod_{i=0}^{r-1}(X_i^s + 1)) - 1] \\ &= \prod_{i=0}^{r-1} E[X_i^s + 1] - 1 = [(b-1)f + 1]^r - 1. \end{aligned} \tag{5.4}$$

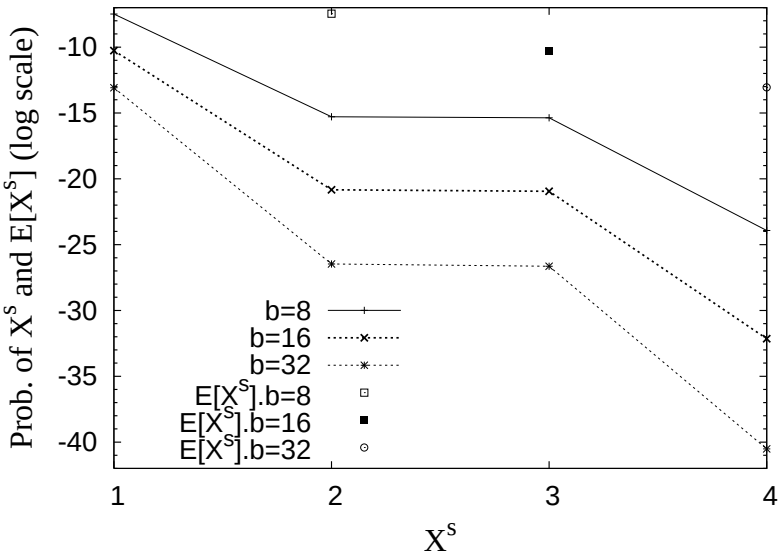


Fig. 18. Probability of X^s, the number of f-indexes, in an SS. $n = 2^{16}$. Required f=2^{-10} for b=2.

Fig. 18 shows the probabilities for three base systems (2^3, 2^4, and 2^5) derived from Eqs. (5.2) and (5.4). For a fair comparison, each memory size of M_{2^3}, M_{2^4}, and M_{2^5} are set equally so that inequality $f_{2^3} > f_{2^4} > f_{2^5}$ is satisfied based on Lemma 2 where f_{2^3}, f_{2^4}, and f_{2^5} are f-positives of each MBF in base-2^3, base-2^4, and base-

2^5, respectively. The lines in Fig. 18 are not shown in monotonic decrease due to binomial coefficient in binomial distribution $B(b\text{-}1, f)$. However, the average value of X^s from Eq. (5.4) is decreased as the number system of base-b increases, and in case of b>32 probability of Eq. (5.2) decreases monotonically due to memory gain by a larger base number system.

2. False indexing in a US in an MBHT

In addition to ensuring a low probability of more than one access to a key table in an SS, a design of an HT must also ensure the low probability of a US is. Unlike an SS, a US has no valid index, which means that all MBFs returning 'yes' make f-positives. However, by definition of an f-index, each column should have at least one BF returning 'yes' as an f-positive, otherwise a group of f-positives can not constitute an f-index. Therefore, we expect a much lower probability because of the product of each independent f-positive probability of BFs.

Let X_i^u denote a random variable of the number of f-positives from BFs at column i. Then, the probability density function of X_i^u follows a binomial distribution $B(b, f)$ due to the i.i.d f-positives of the BFs. Also, suppose random variable X^u is the number of f-indexes in a US on an MBHT. Then, random variable X^u can be formulated with random variable X_i^u into $\prod_{i=0}^{r\text{-}1} X_i^u$. In general, the probability of X^u becomes

$$\begin{aligned} Pr\{X^u = x\} &= \sum_{x_0 \cdots x_{r\text{-}1}=x} Pr\{X_0^u = x_0, \cdots, X_{r\text{-}1}^u = x_{r\text{-}1}\} \\ &= \sum_{x_0 \cdots x_{r\text{-}1}=x} Pr\{X_0^u{=}x_0\} \cdot Pr\{X_1^u{=}x_1\} \cdots Pr\{X_{r\text{-}1}^u{=}x_{r\text{-}1}\}. \end{aligned} \tag{5.5}$$

For example, the probability that there is no f-index can be calculated in the complementary way, as in the following:

$$Pr\{X^u = 0\} = 1 - \prod_{t=0}^{r-1} Pr\{X_t \geq 1\} = 1 - \prod_{t=0}^{r-1} (1 - Pr\{X_t{=}0\}).$$

Also, in the cases of one, two, and three f-indexes, their probabilities are derived as $Pr\{X_s\}$ is by factorization. Similarly, in the case of four f-indexes, there are two possibilities of factoring, i.e. $4{\times}1{\times}\cdots{\times}1$ and $2{\times}2{\times}1{\times}\cdots{\times}1$. Thus, the probability becomes the summation of two cases as follows:

$$Pr\{X^u{=}4\} = \sum_{t_1,t_2,t_1\neq t_2}^{r-1} \left(Pr\{X^u_{t_1}{=}2\} \cdot Pr\{X^u_{t_2}{=}2\} \prod_{t'\neq t_1,t_2}^{r-1} Pr\{X^u_{t'}{=}1\} \right) + \sum_{t=0}^{r-1} \left(Pr\{X^u_t = 4\} \prod_{t'\neq t}^{r-1} Pr\{X^u_{t'} = 1\} \right).$$

Finally, the mean of random variable X^u can be calculated with i.i.d property:

$$E[X^u] = \sum_{t=0}^{n} t \cdot Pr\{X^u = t\} = E[\prod_{i=0}^{r-1} X^u_i] = [bf]^r. \tag{5.6}$$

Fig. 19 shows the probabilities for three base systems (2^3, 2^4, and 2^5) derived from Eqs. (5.5) and (5.6) as Fig. 18 does.

Algorithm 5: `query(MBHT,e)`

Input: An MBHT and key e
Output: Set of $A^b = a_0 \cdots a_{r-1}$ including false indexes

```
for column i = 0 to r − 1 in an MBHT do
    for t = 0 to b − 1 do
        if e ∈ t-BF^i then                          /* i.e. M^i_on[][t]==1 */
            S_{A_i} = S_{A_i} ∪ {t};
        end
    end
end
S_A = ∅;                                  /* Set of an i-index and f-indexes */
S_A = make_paths(S_{A_0}, ··· , S_{A_{r−1}});
return S_A;                                  /* No off-chip memory access */
```

The `query` operation shown in the **Algorithm** `query` only considers on-chip

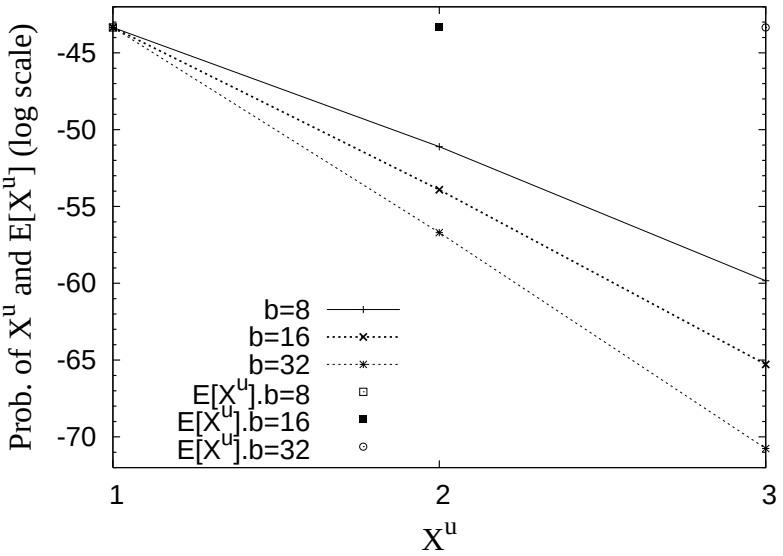


Fig. 19. Probability of X^u, false memory access, in a US. $n = 2^{16}$. Required f=2^{-10} for b=2.

operation and it needs to be called twice on *l-MBHT* and *r-MBHT*. Therefore, the average of random variables $\mathcal{X}^u$ and $\mathcal{X}^s$ for a US and an SS, respectively, using two MBHTs are the following based on Eqs. (5.6) and (5.4)

$$E[\mathcal{X}^u] = 2 \cdot E[X^u] \qquad \text{and} \qquad E[\mathcal{X}^s] = E[X^u] + E[X^s], \tag{5.7}$$

because for a US both MBHTs do not have a wanted key and for an SS one of MBHTs does not have a wanted key. Function `make_paths` makes f-indexes based on set S_{A_i}, $0 \leq i \leq r$-1. For example, given inputs $\{2_4\}, \{1_4, 3_4\}, \{0_4\}$ for S_{A_0}, S_{A_1}, and S_{A_2} in base-4 system, it returns address set $\{210_4, 230_4\}$ by concatenating each member from all S_{A_i}. The time complexity of overall `query` is $\Theta(1)$ on the condition that function `make_paths` is performed in constant time, which is possible. Also, time complexities of accessing off-chip memory depend on Eq. (5.4) and (5.6) for an SS and a US.

The `query` described above returns set S_A of candidates indexes to probe in a key table. However, the size of set S_A is geared to be probabilistically 1 to sustain

the bandwidth requirement of a high-speed router. According to Eq. (3.2), once each MBF has the memory size appropriate to w_M for a given bandwidth requirement, it does not give any f-positives, resulting in no f-index. That is, the requirement of 160Gbps needs deterministic lookups of 500M keys (packets) in a second without an f-positive, implying that a f-positive rate should be as low as 1/500M=2n without consideration of binomial coefficients in X^u and X^u.

3. Hardware consideration for pipelining

Previsously in `insert` and `query`, parallel MBFs have been used. However, in hardware configuration, pipelining on MBFs from column 0 to column r-1 is better than parallel on MBFs in terms of operational power regardless of an SS and a US. The reason is that in one column shown in Fig. 16 only one MBF among b MBFs of base-b is involved for `insert` and to return 'yes' in a query, while the rest are to return 'no'. Also, in case of a US, all b MBFs in one column are to return 'no' in a query. These situations are true in all MBFs in a column-wise view. That is, although an SS needs to search all columns, in case of a US if anyone of MBFs at previous columns does not give 'yes' ensuing MBFs in the next column do not have to perform `query` and there is no t-index and f-index by Definitions 3 and 4. Thus, two ways of pipelining, i.e. in order of MBFs in one column and then in order of columns, can maximize the power efficiency rather than probing all MBFs at the same time.

Fig. 20 shows the benefit of a pipelined MBF by measuring the average number of steps to proceed and the average number of bits to probe under one MBF. Given a required precision w=k for an MBF based on Eq. 3.2, k_s-bit locations are probed in one step so that there are k/k_s steps to proceed in a query of a US. The upper figure shows how many steps to proceed until the last step returns 'no' in a US. By virtue

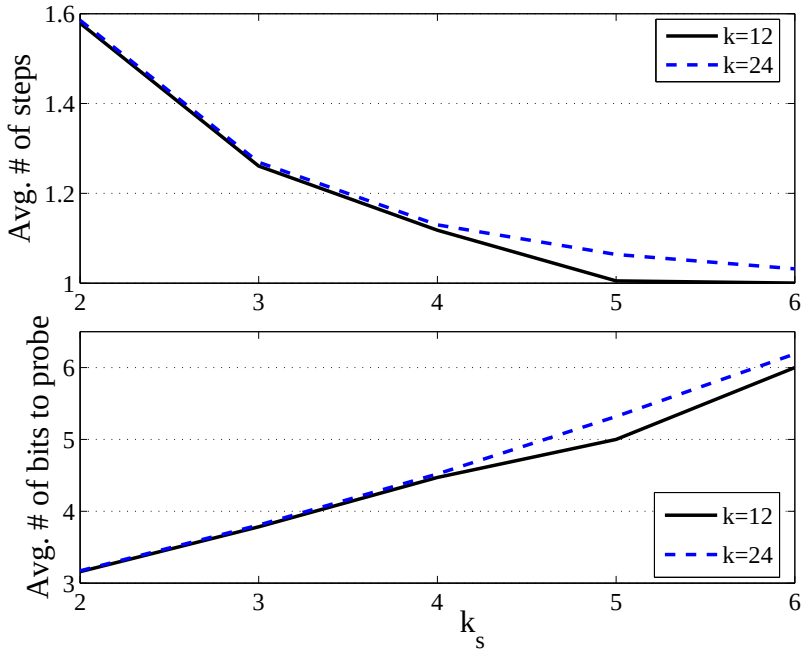


Fig. 20. The benefit of pipeline in an MBF returning 'no' in a query for two cases of k=12 or 24.

of principle of a BF, i.e. several hash functions, using a larger value of k_s shows the less number of steps to proceed. However, the average number of total bits to probe until the last step of 'no' shows the reverse way as shown in the bottom figure where the average number of bits to probe is calculated $k_s \times$*(average # of steps)*. That is, probing a smaller number of bits step by step shows the less number of memory reads, implying that pipelining in an MBF needs less power. Note that this benefit happens only to an MBF returning 'no' in a query. However, this benefit is multiplied by the rest of b-1 MBFs in one column as well as other MBFs in other columns, so that the total benefit of a pipelined MBHT becomes (b-1)$\times r$ times in an SS and $b \times r$ times in a US larger than a single pipelined MBF, where r=$\log_2 n$.

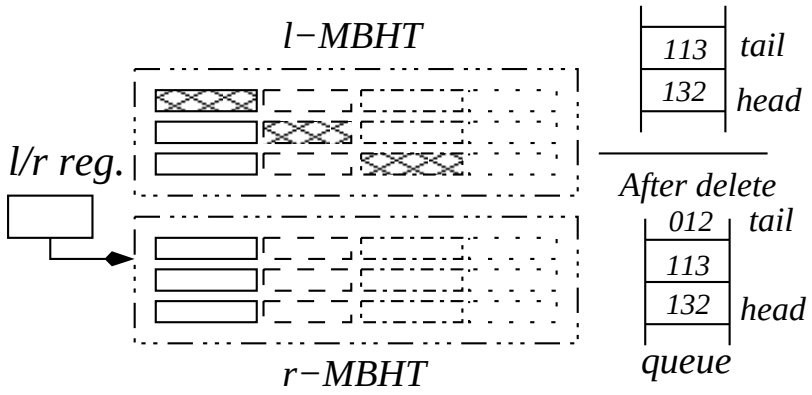


Fig. 21. An example of `delete` for item e located at 012_4 in base-4.

E. Delete Operation in MBHTs

Unlike the two kinds of searches in `query` operations, we consider `delete` operation for a successful deletion. The `delete` operation needs two `query` operations on both *l-MBHT* and *r-MBHT*, where only one of MBHTs has a relevant key. Fig. 21 shows an example with n=64 for `delete`. Initially, *l-MBHT* has been fully used for `insert`, *l/r-reg.* indicates the *r-MBHT* for future n `insert`. Now after deletions of keys located at indexes 113_4 and 132_4 of a key table, the stack has 113_4 and 132_4 as candidate indexes further `insertions`. Suppose key e was inserted in 0-BF^0, 1-BF^1 and 2-BF^2 in checked boxes as shown in the figure. Once key e for `delete` operation is confirmed by accessing a key table with the address 012_4, the address is to be put on the stack for future `insert`.

Like the `query` operation, if there are an f-index and a t-index associated with a key, two accesses to a key table in `delete` are necessary. Therefore, when random variable $\mathcal{Z}$ is denoted as the number of accesses to a key table with both MBHTs, the average memory access for a `delete` operation on the condition that the item exists, i.e. a successful *delete*, is

$$E[\mathcal{Z}] = \left(\sum_{v=1}^{n} v \cdot Pr\{X^u = v\}\right) + \left(1 + \sum_{v=1}^{n-1} v \cdot Pr\{X^s = v\}\right) \tag{5.8}$$
$$= [bf]^r + [1 + (b-1)f]^r,$$

where the first term accounts for a US in one of MBHTs based on Eq. (5.2) while the second term explains the an SS in the other based on Eq. (5.5).

The detailed procedure of the `delete` operation is shown in **Algorithm** `delete`. The complexity in on-chip memory is $\mathcal{O}(1)$ because the complexity of `query` used in the algorithm is $\mathcal{O}(1)$. The complexity of memory access is $\mathcal{O}(E[\mathcal{Z}])$ on average for a successful *delete*, and it is to be constant as $E[\mathcal{X}^s]$ is $\mathcal{O}(1)$ while the complexity for an FHT is $\mathcal{O}(nk^2/m + k)$.

Algorithm 6: `delete`(l-$MBHT$,r-$MBHT$,e)

Input: Two MBHTs and key e
Result: Update associated BF in each column

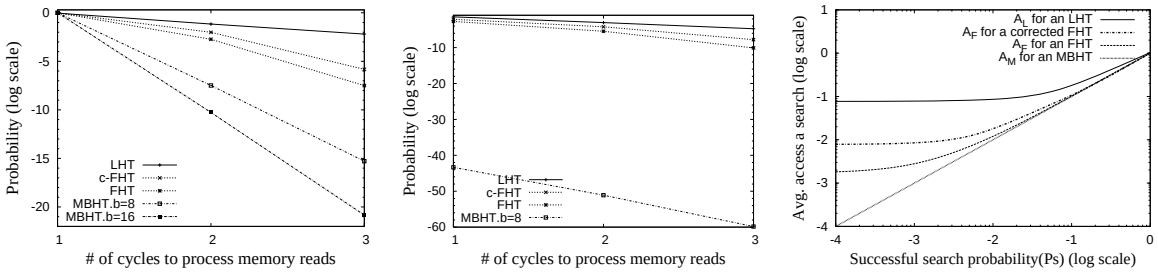
```
1 S_{l-MBHT}=query(l-MBHT,e) ;                    /* Only on-chip Op.  */
2 S_{r-MBHT}=query(r-MBHT,e) ;                    /* Only on-chip Op.  */
3 for A ∈ S_{l-MBHT} ∪ S_{r-MBHT} do              /* A = a_0 a_1 ··· a_{r-1} */
4     if M_{key}[A] == e then                     /* On-chip Mem. Acc. */
5         M_{key}[A] = ∅;
6         push(A,queue);                          /* push A to Q. for insert */
7     end
8 end
```

F. Analysis and Simulation for an MBHT

This section presents analyses of memory efficiency and the average access time per query to a key table for four schemes; an LHT, an FHT, and an MBHT. Also, a phenomenon of duplicated keys in an FHT is analyzed. Finally, one simulation is performed for determining the on-chip memory usage for IP lookup application with BGP tables available from [51, 52]. Among a class of universal hash functions, a

hardware scheme in [59] is adapted for simulation.

1. Average access time of query



(a) Probability of cycles to process mem. reads to a table or a linked list in an SS. For an MBHT $Pr\{\mathcal{X}^s\}$.

(b) Probability of cycles to process mem. reads to a table or a linked list in a US. For an MBHT $Pr\{\mathcal{X}_u\}$.

(c) Avg. access time as a function of successful-search rate.

Fig. 22. Probabilities of memory access in an SS and a US and the average access time to off-chip for an LHT, an FHT, and an MBHT with the same memory $128K \log_2 n$ to fully utilize the saved memory for increase in precisions of base-8 and base-16. k=10, and n=64K.

Let us define the average access time to off-chip as the number of accesses to off-chip given `query` operation. For an LHT with chaining, the load factor, α_L, can be given as n/m_L where n is the number of items and m_L is the number of buckets used to point an address of off-chip memory after hashing. Let T_L^s and T_L^u denote the average access time for an SS and a US, respectively, which are defined in [60] as

$$T_L^s = 1 + \alpha_L/2 - 1/2m_L, \qquad T_L^u = \alpha_L.$$

To evaluate the average access time regardless of an SS and a US, another parameter p_s, which denotes the frequency of an SS of a key in off-chip memory, is introduced. With these notations, the average access time T_L for an LHT can be expressed as

$$T_L = p_s T_L^s + (1\text{-}p_s) T_L^u = p_s \left(1 + \frac{n-1}{2m_L}\right) + (1\text{-}p_s)\frac{n_1}{m_L}. \tag{5.9}$$

For an FHT, let E_p be the expected length of a linked list in the FHT for an item in a positive match and E_f be the expected length of a linked list in the FHT for a f-positive match. E_p can be derived from the average number of items for which all buckets' length $> j$, or $n \cdot B((n\text{-}1) \cdot k, 1/m_F, > (j\text{-}1))^k$ where $B(n, 1/m_F, > j) = 1 - \sum_{i=0}^{j} \binom{n}{i} (1/m_F)^i (1\text{-}1/m_2)^{n-i}$ and m_F is the number of buckets in an FHT. Also, E_f can be derived from Eq. (9) in [15]. Therefore, the average access time T_F for an FHT is

$$T_F = p_s E_p + (1\text{-}p_s) f E_f = p_s E_p + (1\text{-}p_s)\,(1/2)^{(m_F/n)\ln 2}\, E_f, \tag{5.10}$$

where f is the f-positive probability in a shared linked list.

Finally, for an MBHT, Eqs. (5.7) are used to get an average access time T_M as the following

$$T_M = p_s E[\mathcal{X}_s] + (1 - p_s) E[\mathcal{X}_u]. \tag{5.11}$$

The average access times of other schemes can be calculated as easily as Eq. (5.11) of an MBHT. Based on [60] and [15], those of an LHT and an FHT are related with the load factor defined as the number of keys over the number of buckets, i.e. n/m. Note that an FHT considers neither buckets in bits used for pointers to off-chip memory nor counters of linked lists in bits.

Fig. 22 shows the probabilities of memory access in an SS and a US and the average access time calculated from Eq. (5.11), in terms of the number of off-chip memory accesses for four schemes under different successful search rates. Note that a modified FHT is marked as `c-FHT`, considering counters and the existing FHT is marked as `FHT`, not considering counters as memory. In Fig. 22 (a) it is shown that $Pr\{\mathcal{X}_s\}$ of an MBHT is always less than those of an LHT and an FHT. Fig. 22 (b)

shows the probabilities $Pr\{\mathcal{X}_u\}$ of an MBHT and other schemes. Particularly, the result in Fig. 22 (c) indicates that the lower the successful search rate, the better the performance of the proposed MBHT is than those of an LHT and an FHT.

Table IV. Complexities of operations to off-chip in four schemes.

Operation	`insert`	`query`	`delete`
LHT	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
FHT	$\mathcal{O}(nk^2/m+k)$ *	$\mathcal{O}(1)$	$\mathcal{O}(nk^2/m+k)^*$
MBHT	$\Theta(1)^\circ$	$\Theta(1)$ †	$\Theta(1)$

* In optimal configuration, $\mathcal{O}(k)$. ° In detail, that is just 1.
† In detail, $\Theta(p_s(1+E[\mathcal{X}^s])+(1-p_s)E[\mathcal{X}^u])$.

Table IV summarizes the complexities of off-chip memory access regarding `insert`, `query` and `delete` operations in an LHT, an FHT, and an MBHT. The big difference is in an FHT, which involves the labored complexities of `insert` and `delete` operations depending on variables n, k, b, and m. In contrast, the complexities of an MBHT and an LHT are constant.

2. Memory usage

This section presents and compares the on-chip and off-chip memory usage for each scheme. Given f-positive f=2^{-w} and the number of elements n, the memory usages in bits of an FHT are the following:

$$M_L = \log_2 n \times 1.44nw_L, \text{ and} \qquad M_F = \left\{\log_2 n + 4\right\} \times \left\{1.44nw_F\right\},$$

respectively, where 4 in M_F accounts for the number of bits in a counter, and w_F=w_B=w. Memory efficiency ratio $R_{M,F}$ of M_M to M_F whose value is derived from Eq. (5.1) becomes

$$R_{M,F} = \frac{M_F}{2\{\log_b n \cdot 1.44n(\log_2(1/f)+\alpha)\}} = \frac{x(\log_2 n+4)w}{2 \cdot \log_2 n(w+\alpha)} \approx \frac{x(\log_2 n+4)}{2\log_2 n}, \quad (5.12)$$

where α=$\log_2(b\text{-}1)$ due to coefficients in the binomial functions of X^s and X^u, and the size of a queue for free addresses in off-chip, $n \log n$ is not considered. Also, the memory efficiency ratio R_{MB} of M_M to M_B are

$$R_{M,L}=\frac{M_L}{2 \times \{\log_b n \cdot 1.44n(\log_2(1/f)+\alpha)\}} = \frac{x}{2} \cdot \frac{w}{w+\alpha} \approx \frac{x}{2}. \quad (5.13)$$

where w_F and w_M need to be set to w of given precision requirement for a fair comparison to each other.

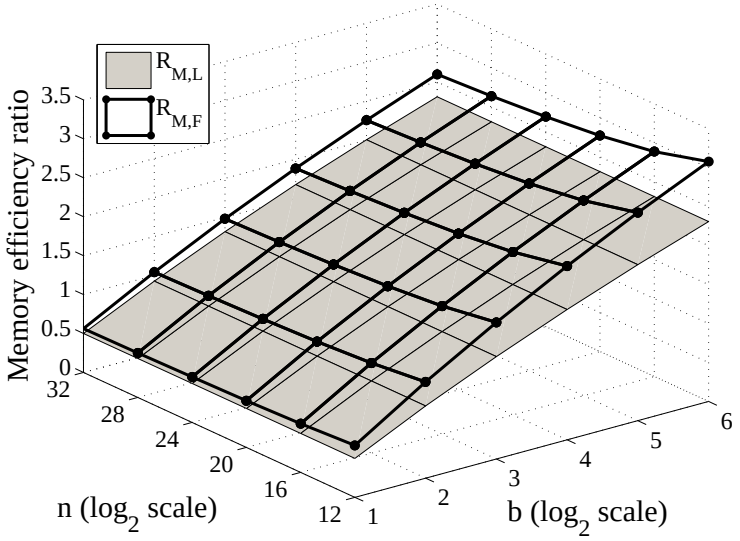


Fig. 23. Memory efficiency ratios of $R_{M,L}$ and R_{MF} with various b and n. w_F=w_M=20. Note that although an MBHT is set to have the same average access as others, the actual average access times are different each other as shown in Fig. 22.

Fig. 23 shows two ratios, R_{MF} and R_{ML}, calculated from Eqs. (5.12) and (5.13) in the range [2:2^6] for base-b and in the range [2^{10}:2^{30}] for n. The figure shows that

without a doubt the turning point for a better memory efficiency ratio surely begins at b=2^3 due to a set of two MBHTs. Also, even with large values of coefficients in binomial functions $B(b\text{-}1, f)$ and $B(b, f)$ the acquired memory gains of four ratios increase as b increases. Given b, the memory gain in a range of n does not change much as shown in the figure, although the change rate of memory gain for a given n is manifested along the b axis. Thus, compared to an LHT and an FHT, Fig. 23 proves that an MBHT approach can gain much memory as long as a larger base number system is used.

As one application of an MBHT to packet processing, i.e. URL switching, we used NePSim [61] for URL switching where all the incoming packets to a switch are parsed and forwarded according to URL. This kind of switching is a commonly used content-based load balancing mechanism [62, 63]. Kachris *et al.* [63] used a simple XOR hash to reduce the collisions among Block RAMs in connection manager for web switching, and Prodanoff *et al.* in [64] proposed URL signatures using CRC32 to reduce the size of routing tables and aggressive hashing with chaining of a linked list to speed-up routing lookups in large-scale content distribution networks.

Table V shows the memory size in bytes of an LHT, an FHT, and an MBHT for three trace databases on the condition that requirement of f is 2^{-20} and the load factor becomes 0.034 accordingly. Each trace of UC Berkeley, NLANR, and CA*netII has 149,344, 504,967, and 2,552,045 URLs, respectively. The result shows at most 1.7 times on-chip memory reduction at an MBHT in base-16 against an LHT as shown in the table. If comparison is set for an FHT, about 2 times of the memory reduction is observed due to consideration of counter bits in an FHT.

While authors in [61] validated NePSim with SDRAM, SRAM and six micro-engines against the IXP 12000 architecture in terms of performance and power, the number of accesses to SDRAM with NLANR trace is measured on the condition that

Table V. On-chip memory usage for three traces. The load factor is 0.034, K=1024.

URL Traces	LHT	FHT	MBHT base-8	MBHT base-16
UC Berkeley[65]	9024KB	11124KB	6860KB	5393KB
NLANR[66]	33634KB	40735KB	25570KB	20102KB
CA*netII[67]	190953KB	226841KB	1145171KB	114127KB

an LHT, an FHT, and an MBHT were implemented in SRAMs. Especially, given a query an MBHT is to return indexes with a set of SRAMs. Table VI shows the

Table VI. AAS in a successful search of NLANR trace for three schemes. f=2^{-10}.

Schemes	LHT	FHT	MBHT (b=8)	MBHT (b=16)
AAS*	1.026306	1.002472	1.002411	1.000092
# of Acc.°	968861.7	946231.9	946303.9	944114.4

° It means the total number of off-chip accesses provided the URL queries of NLANR.

measured accesses to SDRAM in NePSim with NLANR. The first row is the average access for a successful search. While an FHT needs $2.4E$-3 extra accesses on average for a successful search, the proposed MBHT with b=16 asks $9.2E$-5. Although this value could be minuscule, when it comes to the difference between the numbers of off-chip accesses in an FHT and an MBHT, the gap between them is 2117.

CHAPTER VI

A HIERARCHICALLY INDEXED HASH TABLE

Unlike an FHT using a BF, with a set of BFs a *hierarchical indexing tree* (HIT) is conceptually embedded into an HT of less memory size than an FHT. That is, memory area for an HT used for pointers to a key table is partitioned to make an HIT. An HIT for n keys in power of 2 is composed of s=$\log_2 n$ layers (i.e. SRAM modules) and partitions the address space in a rectangle of $n \times s$ 0/1 bits, so that a BF covers a column group of the same bits, either 0 or 1, in the index address space to a key table. The detail of how to build an HIT is as follows

A. Building a Conceptual HIT in Stacked SRAMs

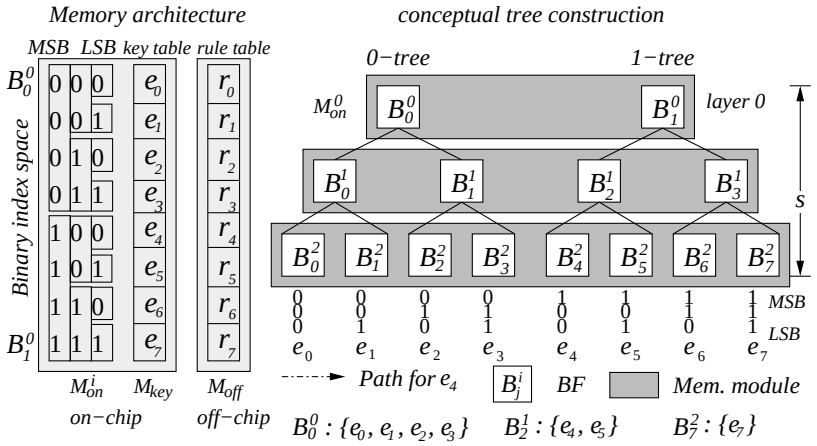


Fig. 24. Basic configuration of hierarchical indexing tree of 0- and 1-tree.

Fig. 24 shows a hierarchical partition for an HIHT. Let B^i_j denotes j-th BF in layer i, hereinafter $0 \le i \le s\text{-}1$, and let all n keys be filled in a key table in on-chip memory sequentially from index $0_0...0_{s\text{-}1}$ to index $1_0...1_{s\text{-}1}$. If key $e \in S$ is to be inserted at index address $A = a_0 a_1 ... a_{s\text{-}1}$, where $a_t \in \{0, 1\}$, $0 \le t \le s\text{-}1$, a BF, denoted $B^i_{a_0 \cdots a_i}$ at

each layer i, is involved to encode key e as a legacy BF does. In this hierarchical partitioning and encoding, B_j^i at each layer i takes care of $n^i{=}n/2^{i+1}$ keys of set S. That is, B_j^i covers $n^i{=}n/2^{i+1}$ keys starting from $j{\cdot}2^{s\text{-}1\text{-}i}$ to $(j{+}1){\cdot}2^{s\text{-}1\text{-}i}\text{-}1$ in index address space. For instance, B_0^0, B_2^1, and B_7^2 take care of sets $\{e_0, \cdots e_3\}$, $\{e_4, e_5\}$ and $\{e_7\}$, respectively. Eq. (3.2) states that m is linearly proportional to n in a given f. Thus, given $f^i{=}2^{\text{-}w^i}$ for a BF on layer i, where w^i is a precision of a BF on layer i, the total size in bits of memory M_{on}^i for BFs on layer i is $2^{i+1}(1.44n^iw^i){\times}1$. Finally an embedded HIT is comprised of a 0-tree and 1-tree covering half of n keys located in $0x_1...x_{s\text{-}1}$ of a key table and the remaining half in $1x'_1...x'_{s\text{-}1}$, where x_t, $x'_t{\in}\{0,1\}$, $1{\le}t{\le}s\text{-}1$.

B. Insert Operation in an HIT

Fig. 24 shows a basic structure of our HIT consisting of 3 layers of BFs for 8 keys. The left side of Fig. 24 shows the binary address space with a set of BFs partitioning the address space of a key table, and the right side shows the transformed dual trees, 0-tree and 1-tree, where each node represents B_j^i. For an example of the insertion of key e_4 at index address 100_2, B_1^0 at layer 0, B_2^1 at layer 1, and B_5^2 at layer 2 are involved.

Procedure `insert` shows the detailed insert operation and is as simple as that for a BF. Although conceptually all BFs are separate each other in an HIT, for hardware implementation assume that BFs on layer i are embedded in one on-chip memory module M_{on}^i as shown in Fig. 24, and there are $s{=}\log_2 n$ memory modules. Finding a base address for B_j^i is easily calculated as shown in line 3. The first vertically-lined **for** loop in **Procedure** `insert` is executed in *parallel* and *pipelining* at each layer. Also, the second **for** loop is done in *parallel*, as does a legacy BF.

Procedure `insert`

Input: key e, rule r, and its given address $A = a_0 a_1 \cdots a_{s-1}$ in a binary mode
Output: Encoded HIT for key e

```
1 for layer i = 0 to s − 1 do                                   /* On-chip Op.  */
2 |  m^i=1.44n^i w^i;   j=a_0 ··· a_i;
3 |  idx=j · m^i;                          /* Find right base index idx for B^i_j */
4 |  for t=0 to k-1 do                              /* One M. for BFs on layer i */
5 |  |  M^i_on[idx+h_t(e)]=1;
  |  |                          /* h_t(e) ∈ {0,··· m^i-1}, M^i_on of 2^{i+1}m^i×1 bits */
6 |  end
7 end
8 M_off[A]=r ; M_key[A]=e;
```

Therefore, the time complexity in on-chip memory is $\Theta(1)$ on the condition that hash functions return indexes in a constant time, and each layer conducts hashing in parallel. Moreover, the number of off-chip memory accesses to M_{off}, is exactly 1 because key e and its associated rule are saved in M_{key} and M_{off} at the designated address A, as shown in line 8. Thus, the complexity of **Procedure** `insert` for off-chip memory access is $\Theta(1)$. In contrast, an FHT claimed a time complexity of $\mathcal{O}(nk^2/m + k)$.

C. Delete Operation in dual HITs

`delete` operation is not as easy as `insert` because a basic BF does not support deletion of a key which was inserted in the BF. However, dual HITs, an l-HIT and a r-HIT, in on-chip memory is used to rotate a target HIT for `insert` operation and another target HIT for `delete` operation, as shown in Fig. 25. Once one HIT is full for previous n keys, `query` operation stays with the HIT. But if set S is dynamic, but limited in size n, a new HIT takes care of `insert` for a new key by setting BFs in a new HIT as well as a bit in a *valid bit array* (VBA). An index for the new key is indicated by *'next'* which is updated every time from a *free address stack* (FAS) in

off-chip memory. Also, the old HIT handles `delete` operation by simply setting off a bit in a VBA coupled with the corresponding HIT. Updating *'next'* and an FAS is not a burden because whenever there are `insert` or `delete` operations, these operations need off-chip access, thus, *'next'* and an FAS can be updated without another cost. Checking V_l and V_r with indexes given by two HITs makes it sure that an unnecessary access to off-chip memory is blocked within on-chip for HITs. Also, when all n keys are encoded in one HIT, i.e. the moment that an FAS is empty, the other HIT needs to be initialized 0 for the next set of `insert` operations with the initialized BF.

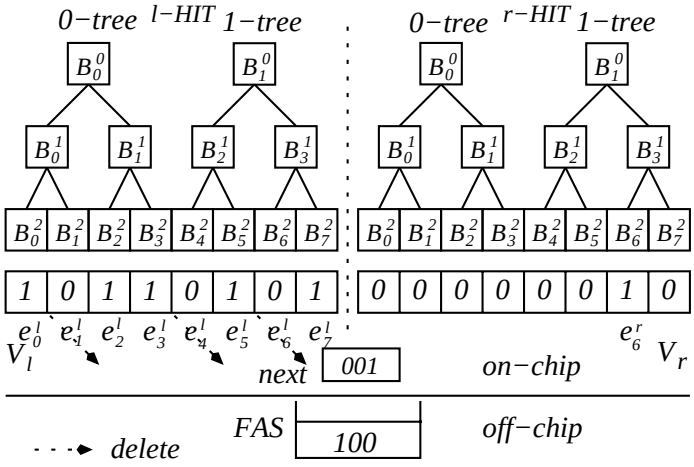


Fig. 25. Dual configuration of HITs for `delete` operation.

Suppose, for example, 8 keys, e_0^l, ..., and e_7^l, are inserted in an l-HIT as shown in Fig. 25. Then, after the 8 keys, a target HIT for new insertion becomes a r-HIT. Now the ensuing operations are deletions of e_4^l, e_1^l, and e_6^l. After the deletions, suppose the next operation is insertion of e_6^r in a r-HIT with proper setting a bit for e_6^r in array V_r as shown in Fig. 25 where *next* and an FAS have 001 and 100, respectively. By rotating a target HIT for insertion among dual HITs and confirming an index returned by each HIT with a VBA, the operations of `insert` and `deletion` are processed seamlessly. Also, by using two rotated HITs, an HIT does not need

counters in each BF, i.e. a counting BF, which costs 4 times more memory size than a BF. Thus, using two HITs saves 2 times the memory. The detailed procedure and complexity of `delete` are provided in Sec. C.

D. Query Operation Making Index Paths in Dual HITs

Once all keys are saved in a key table and encoded in a set of BFs in on-chip memory, the remaining and ultimate goal of an HIT is to search a key in it by `query` operation fast. There are two kinds of search patterns, an unsuccessful search (US) in which a key is relentlessly searched although it is not in an HIT, and a successful but time-consuming search (SS) in which a key is to be searched out in an HIT. Before a discussion of these two kinds of searches, let definitions of index path, index segment, false index path, and false segment introduced.

Definition 5 (Index Path)
In an HIT, an index path, or ***i-path****, is defined as a series of* $B_j^i s$ *used in* ***`insert`*** *operation and hierarchically connected each other from layer* 0 *to layer s-1, making a sequence of address bits. The sequence of indexing bits in* $B_j^i s$ *is also matched with an arbitrary index for a key saved in a key table and the size of the sequence of bits from the series of* $B_j^i s$ *must be* s*.*

As a corollary, it can be concluded that in `query` for key e previously encoded by `insert` for the key e, an i-path for the key e should show up as a BF returns 'yes' in true membership testing.

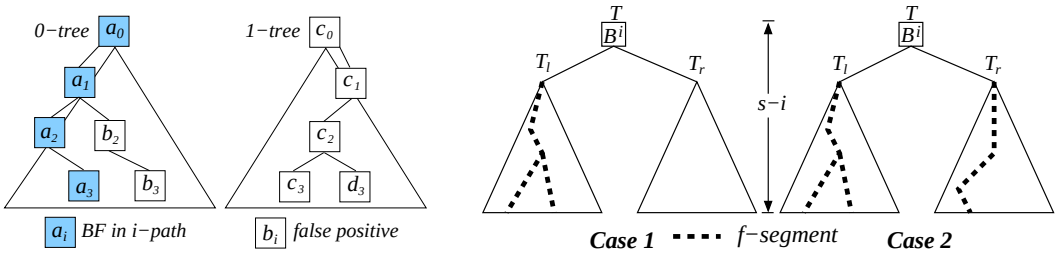
In an HIT, besides an i-path dedicated to a key, due to f-positives from irrelevant BFs a false index to a key table is possible. For example, suppose key e_4 is inserted with i-path $1_0 0_1 0_2$ in Fig. 24 and then a query to e_4 is requested. The result of the query may give an ambiguous $1_0 0_1 x_2$, $x_2 \in \{0, 1\}$, due to an f-positive of B_5^2. Thus,

this ambiguity needs two accesses to a key table. Given a query for an i-path of size s, there are totally 2^s*-1* false indexes because each B_j^i is independent and identically distributed, i.i.d. Besides the definition of an i-path, a false index path is defined by result of `query` operation, leading to a false indexing to a key table in on-chip memory.

Definition 6 (False Index Path and False Segment)
In ***`query`****, from hierarchically consecutive layers, a group of B_j^is not pertaining to an i-path can be formed in a series of at most size s, and to become a false index path, or* ***f-path****, this series needs to be either connected to an i-path or a completely different path of size s, i.e. independent of an i-path in an HIT. Also, the group attached to an i-path is called a false segment, or f-segment. The number of f-paths plus an i-path is compatible with the length of shared linked list used for query of a key in an FHT.*

Even if it is possible that there is a set of BFs giving f-positives in `query`, BFs that are only hierarchically connected to each other and an i-path can be part of an f-segment. Thus, f-positives from the rest BFs can be ignored. For example of previous 100_2 for e_4 in Fig. 24, even if B_1^1 and B_3^1 randomly make f-positives right after `query`, there is no f-segment starting from the B_1^1 and B_3^1. By the definition of an f-path, the probability of the f-path is cumulatively calculated as the product of f-positives from BFs along the f-path.

Figs. 26(a) and 26(b) show an example of the calculation of probability of an f-path in an HIT with one i-path and three f-paths. A series of $a_0a_1a_2a_3$ in the dark boxes is an i-path. The probability of the f-segment b_2b_3 foaming f-path $a_0a_1b_2b_3$ is $\prod_{t=2}^{3} f^t$ where f^t is the f-positive of a BF on layer t. Also, the probabilities of the remaining f-paths, $c_0c_1c_2c_3$ and $c_0c_1c_2d_3$, are the same as $\prod_{t=0}^{3} f^t$ because the probabilities of f-positives of BFs on the same layer are the same each other.



(a) Examples of an i-path, f-segments, and f-paths.

(b) Probability of cumulative f positives in an HIT.

Fig. 26. Examples of an i-path, f-segments, and f-paths. Probability of f-paths.

Once the probability of an individual f-path is known, the final attention is paid to the probability that an HIT has t f-paths, $0<t<n$. Suppose binary tree T of height l has two sub-trees, T_l and T_r of height l-1 with cumulative false positives F_{T_l} and F_{T_r}, respectively, as shown in Fig. 26(b). Also, let T_l and T_r have n_l and n_r f-segments of size l-1. To have n_l+n_r f-segments of size l, the binary tree T rooted at B^i with height l needs itself to be an f-positive. Therefore, the probability F_T that the binary tree T with its sub-trees has n_l+n_r f-paths is the product of three: the probability that T needs to be an f-positive, the probability that T_l has n_l f-paths, and the probability that T_r has n_r f-paths, i.e. $f^i \cdot F_{T_l} \cdot F_{T_r}$. Fig. 26(b) shows the cases of the probability that tree T has 2 or 3 f-paths as follows:

Case 1 : The right child tree T_r does not have any f-segment. By definition, for an f-segment to exist in T, B^i rooted in T must be an f-positive. Now that B^i is an f-positive, f-segment in T_l of size *l-1* become part of an f-segment of size l. Therefore, automatically T has the same number of f-segments from T_l due to its f-positive.

Case 2 : Both T_r and T_l have a few f-segments and contribute f-segments of tree T in the summation of f-segments in T_l and T_r, in total 3 f-segments, because

B^i is an f-positive and both T_l and T_r have their own f-segments.

Suppose $P^i(t)$ is defined as the probability of t f-segments starting on layer i in an HIT of height s and it is calculated as the following in a recursive way:

$$P^i(t) \geq \sum_{v=0}^{t} P^{i+1}(t) \cdot P^{i+1}(t-v) \cdot f^i \qquad \text{if} \qquad t \leq 2^{i+1}, \tag{6.1}$$

where base cases of $t > 2^{i+1}$ and $i{=}s$ are 0 and 1, respectively. Also, as f^i based on Eq. (3.1) is bounded and has a global minimum at $k^i{=}m^i \ln 2/n^i{=}w^i$ of Eq. (3.3), an inequality in Eq. (7.1) for layer i can be removed at the optimal configuration of k^i.

1. False indexing to a key table in on-chip for a US

Besides the design issue of low probability of more than one access to a key table for an SS, it is also equally important that the probability of f-indexes in a US is lower. Unlike an SS, in a US there is no i-path for a given key, meaning that all BFs in `query` return 'yes' as f-positives. However, there is a chance that each of 0-tree and 1-tree can give plural f-paths. In contrast to one f-positive in an FHT [15] leading to off-chip memory accesses, one f-path by a series of f-positives of *hierarchically* connected B^i_js in each layer i of an HIT becomes one index access to a key table. Thus, far less probability is expected because of the product of f-positive probabilities of BFs.

Suppose random variable $\mathcal{X}_u$ is the number of f-paths in a US on an HIHT. Then, it is the number of entries in a key table and equally the number of memory accesses if one memory access can do one memory read to an entry. The probability $Pr\{\mathcal{X}_u = v\}$, $v{>}0$ can be easily derived based on Eq. (7.1) of an optimal configuration of $k{=}w$ as the following

$$Pr\{\mathcal{X}_u = v\} = \sum_{v=t_0+t_1+t_2+t_3} P^0(t_0) \cdot P^0(t_1) \cdot P^0(t_2) \cdot P^0(t_3) \tag{6.2}$$

because there are two HITs, $l\text{-}HIT$ and $r\text{-}HIT$, each having 0- and 1-trees. The sum in Eq. (7.2) accounts for the combination of becoming v among t_0, t_1, t_2, and t_3. That is, if v=1, there are four cases: 0+0+0+1, 0+0+1+0, 0+1+0+0, and 1+0+0+0.

Although choosing a large value for precision w^i for layer i>0 is possible, the number of BFs on each layer in a HIT must be upper bounded, so that the total number of memory reads to M^i_{on} for the BFs must be sustainable in hardware implementation. The expected number of BFs to probe on layer i>0 for a US becomes

$$N^i_{B,U} = 2 \times \{2^{i-1} \prod_{t=0}^{i-1} f^t \times 2\} = 2 \times \{2^{i-1} \times 2^{-2i} \times 2\} = 2^{1-i}, \tag{6.3}$$

where f^i=2^{-w^i}=2^{-2} except layer s-1. On layer s-1, f^{s-1}=$2^{-w^{s-1}}$ is set a collision rate as low as one for a high-speed router like 2^{-29} for 160Gbps. In convergence,

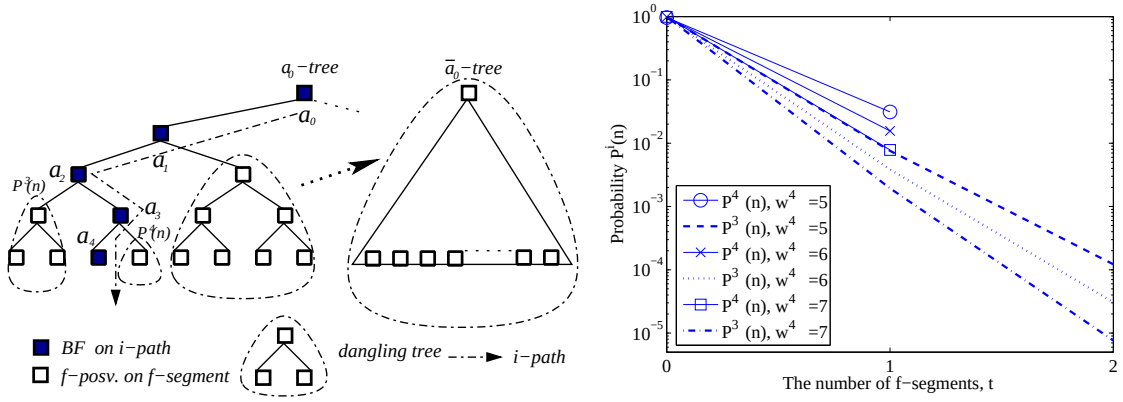
$$\lim_{i \to \infty} N^i_{B,U} = 0, \tag{6.4}$$

meaning that as long as i for a layer index increases, the expected number of BFs on the layer i is minuscule enough that simple memory hardware can support the request of a small number of memory reads.

2. False indexing to a key table in on-chip for an SS

The probability of f-paths in a US has been derived. Now, the probability of the number of f-paths in an SS is derived and calculated. The situation in an SS is very different from that of a US because there must be one i-path and several possible f-paths while there is no i-path in a US. Fig. 27(a) shows an example of 5 layers for 2^5 keys where along an i-path there are 5 dangling trees, d-trees, contributing

f-paths, if any. All d-trees except one rooted on layer 0 are attached to the i-path and they contribute a number of f-paths with different probabilities related to $P^i(n)$ of Eq. (7.1).



(a) An HIT of 5 layers with an i-path and dangling trees.

(b) $P^i(t)$ of Eq. (7.1). w^j=3, j<4, s=5, and n=2^5. Note that as to $P^4(t)$, there is only one possible f-segment, i.e. t=1.

Fig. 27. An i-path and d-trees in an SS, and $P^i(n)$ of Eq. (7.1) for each d-tree in an HIT.

Fig. 27(b) shows the $P^i(t)$ in an HIT with various w^4 for n=2^5 keys. Two lines for w^4=5 have a big difference of 4 times between them, meaning that $P^4(t)$ is the dominant one contributing the number of f-segments. This is true for all other cases of w^4=6, 7. Now, for comparison of the difference in precision choices about w^{s-1}, i.e. w^4, look at three solid lines with markers located at the top of the figure. The difference ratio of $P^4(1)$ between two precision choices for layer 4, w_1 and w_2, is determined by 2 to power of w_1-w_2. Therefore, no matter what precision of w^j is chosen for the layer j, $0 \le j \le s$-2, a dominant probability of f-segments in a HIT comes from the layer 4. Based on this result, it can be concluded that choosing reasonably lower precision for layer j, $0 \le j \le s$-2, does not affect the probability of the number of f-paths, but results in saving memory for these layers.

Besides decision rule of w^i for layer i, it is necessary to calculate the number of partially found f-segments on each layer in an SS. Two times this number is considered as the number of BFs needed to probe on the next layer, like the expected number of BFs in Eq. (6.3) for a US. Note that in an SS, there is an i-path and at least two children BFs probe from a BF on the i-path due to a binary property. The expected number of BFs to probe on layer i>0, except layer s-1, for an SS becomes

$$\begin{aligned} N_{B,S}^{i} &= 2 + f^{i-1} \cdot 2 + 2 \cdot f^{i-1} f^{i-2} \cdot 2^2 + \cdots + 2^{i-1} \cdot f^{i-1} \cdots f^0 \cdot 2^i \\ &= 2 + \sum_{t=0}^{i-1} 2^t \prod_{v=0}^{t} f^{i-v-1} 2^{t+1} = 2 + \left(1 - 2^{-i}\right), \end{aligned} \tag{6.5}$$

where 2 accounts for two BFs on an i-path and f^i=2^{-2}. With Eq. (6.3), the maximum in convergence among the expected numbers of BFs to probe in an SS and a US becomes 3 as

$$\lim_{i \to \infty} \max\{N_{B,U}^{i}, N_{B,S}^{i}\} = 3. \tag{6.6}$$

Thus, the total number of memory reads to M_{on}^{i} for layer i, 0<i<s-1, is k^i times 3 because for each BF k^i hash functions are necessary. In conclusion, in a query, either unsuccessful or successful, the expected number of memory reads on M_{on}^{i} for layer i, 0≤i<s-1, is upper bounded to $3k^i$, where k^i is set 2. On layer s-1, $k^{s\text{-}1}$=$w^{s\text{-}1}$ is set to the log of reverse collision rate, i.e. 29 for 160Gbps, so that bandwidth requirement is secured by our deterministic $\Theta(1)$ lookup. The $k^{s\text{-}1}$ memory reads of random locations by $k^{s\text{-}1}$ hash functions can be supported by a simple switching circuitry in one cycle. Even if a commodity of SRAM has a limit in the number of memory reads, a large SRAM can be divided into a smaller SRAM with the less number of memory reads without worsening an f-positive as suggested in [15].

Based on the observation from Fig. 27(b), calculating the number of f-paths, i.e. false indexes to a key table in on-chip memory in an SS, is necessary. Let random variable $\mathcal{X}_s$ be the number of f-paths from an HIHT, given an i-path for a key in

`query` operation. Then, $\mathcal{X}_s+1$ is the total number of entries in a key table to check in an SS and is equal to the length of searched linked list of an FHT [15]. The detailed probability of $\mathcal{X}_s$ for an SS without f-paths is defined as following

$$Pr\{\mathcal{X}_s = 0\} = P^0(0) \cdot P^1(0) \cdot P^2(0) \cdots P^{s-1}(0), \tag{6.7}$$

because each d-tree along an i-path and two of 0- and 1-trees are independent to each another. In general, the $Pr\{\mathcal{X}_s{=}v\}$ is calculated based on the independent property of each d-tree along a i-path as the following

$$Pr\{\mathcal{X}_s = v\} = \sum_{v=t_0+\cdots+t_{s\text{-}1}} P^0(t_0) \cdot P^1(t_1) \cdot P^2(t_2) \cdots P^{s\text{-}1}(t_{s\text{-}1}). \tag{6.8}$$

3. Detailed procedures for `query` and `delete`

Complete `query` operation consists of `query-i` shown in **Procedure** `query-i` only considering on-chip operation for layer `i`. The time complexity of this procedure is $\Theta(1)$ on the condition that $2\|L\|$, 2 times of the size of L, is bounded to the number of memory reads that hardware supports without burden. The reason for this is that given candidates for partial addresses in L, the number of BFs to probe is doubled due to two children of each node in a binary tree. Thus, by pipelining on each layer starting layer 0, `query` is performed in one cycle, so that `query-s-1` returns complete indexes to a key table for a given query. On the last layer s-1, the average numbers of complete indexes are calculated as $E[\mathcal{X}_s]+1$ or $E[\mathcal{X}_u]$ on average for an SS and a US, respectively, where $E[\mathcal{X}_s]$ and $E[\mathcal{X}_u]$ can be derived from Eqs. (7.4) and (7.2) as

$$E[\mathcal{X}_s] = \sum_{t=0}^{n-1} t \cdot Pr\{\mathcal{X}_s = t\}, \qquad E[\mathcal{X}_u] = \sum_{t=0}^{n} t \cdot Pr\{\mathcal{X}_s = t\}, \tag{6.9}$$

and they are considered $\Theta(1)$ because $Pr\{\mathcal{X}_s{=}1\}{\ll}1$ and $Pr\{\mathcal{X}_u{=}1\}{\ll}1$. Moreover, $Pr\{\mathcal{X}_s = 1\}{\gg}Pr\{\mathcal{X}_s{=}t\}$ and $Pr\{\mathcal{X}_u = 1\}{\gg}Pr\{\mathcal{X}_u{=}t\}$, $t > 1$ as recognized in Fig. 27(b).

Procedure `query-i`

Input: M^i_{on} for layer `i`, list L of partial indexes found on up to layer i-1, including i-path, and key e
Output: A set of partial $A = a^0 \cdots a^i$ of $i{+}1$ bits, including f-segments

```
1  S = ∅;  n=‖L‖;          /* S: Set of partial paths.  L={A_0, ··· A_{n-1}} */
2  for t = 0 to n-1 do
3      m^i=1.44n^i w^i;   A_t = L[t];   idx_0=2A_t · m^i;   idx_1=(2A_t + 1) · m^i;
       cnt_0=cnt_1=0;
4      for t=0 to k^i-1 do  /* One Mem. for BFs on layer i.  k^i hash funcs.
       */
5          if M^i_on[idx_0+h_t(e)]==1 then cnt_0++;  /* idx_0, idx_1 indicate B^i_j */
6          if M^i_on[idx_1+h_t(e)]==1 then cnt_1++;   /* Two sets of k^i accesses
           due to binary children BFs in an HIT */
7      end
8      if cnt_0==k^i then S=S∪ A_t · 0;   /* concatenate 0 or 1 bit to the end
       of A_t */
9      else if cnt_1==k^i then S=S∪ A_t · 1;
10 end
11 return S;                                  /* No off-chip memory access */
```

As with `delete`, like `query` operation, if there is an f-path beside an i-path associated with a key, two accesses to a key table are necessary. Thus, when random variable $\mathcal{Z}$ is denoted as the number of accesses to a key table in on-chip memory, the average memory access for `delete` operation on the condition of existence of a target key, i.e. in an successful deletion, is

$$E[\mathcal{Z}] = 1 + \sum_{v=1}^{n-1} v \cdot Pr\{\mathcal{X}_s = v\} = 1 + E[\mathcal{X}_s].$$

The detailed procedure is shown in **Procedure** `delete`. The complexity of it in on-chip memory is $\Theta(1)$ based on the complexity of `query` is $\Theta(1)$. The complexity of indexes to access a key table is $\Theta(E[\mathcal{Z}])$ on average for a successful deletion and

it is to be constant as $E[\mathcal{X}_s]$ is $\Theta(1)$.

Procedure `delete`

Input: Two HITs of l- and r-HIT and key e
Output: Updated *valid bit array* V_l or V_r for l- and r-HIHT

```
1 S_l=query(l-HIT,e) ;
2 S_r=query(r-HIT,e) ;                                  /* Only on-chip Op.  */
3 if ||S_l ∪ S_r|| == 1 then
4     A ∈ S_t; V_t[A] = 0;                                /* t s.t.  ||S_t||==1 */
5 else if ||S_l ∪ S_r|| > 1 then
6     foreach A in S_t, t ∈ {l, r} do                   /* Off-chip M. Acc.  */
7         if V_t[A]==1 and M_key[A]==e then
8             V_t[A] = 0;                          /* Save A in FAS via 'next' */
9     end
```

4. Parallel accesses to a key table in an interleave way

In a linked list used in an LHT and an FHT, accessing the last key in the linked list of t keys takes t cycles in sequence, because memory address of key e is known after a previous key e' with a pointer to the next key e is obtained in the previous cycle. In contrast, in an HIHT, the candidate indexes to a key table are generated at the last layer s-1 in every cycle, so that parallel accesses to the key table are possible, as shown in `query-i`. Thus, unlike the sequential access in a linked list, our memory access is different as follows: every cycle, a set of indexes is generated for a given query and there is no other index to access the key table for the query as an i-path exists in the set. Also, by a simple interleaving or switching circuitry in SRAM for a key table, a group of memory reads to entries in the key table via the indexes can be processed in parallel in one cycle because indexes are different from each other and known in advance. For instance of interleaving, Garcia *et al.* [68] provide worst-case bandwidth guarantee by utilizing potential of bank interleaving in a SRAM/DRAM hybrid for packet buffer. Once perfect match is complete with a set of indexes to a key

table in on-chip in one cycle, an HIHT can provide a deterministic lookup needing only one access to off-chip memory to know rule about every query of an SS. If a query is a US, there is no access to off-chip memory. The detailed analysis of the size of a set of indexes in SS and US queries is shown in the following section.

E. Simulation Result for an HIHT

This section presents an analysis on a memory efficiency for three schemes: an FHT [15], a BFHT [16], and an HIHT. As to a perfect hash function, Mitzenmacher and Vadhan [69] claimed that simple hash function can provide a truly random hash function. A class of universal hash functions are suitable for hardware implementation and thus a scheme from [59] is chosen.

1. Memory comparison with other hash mechanisms

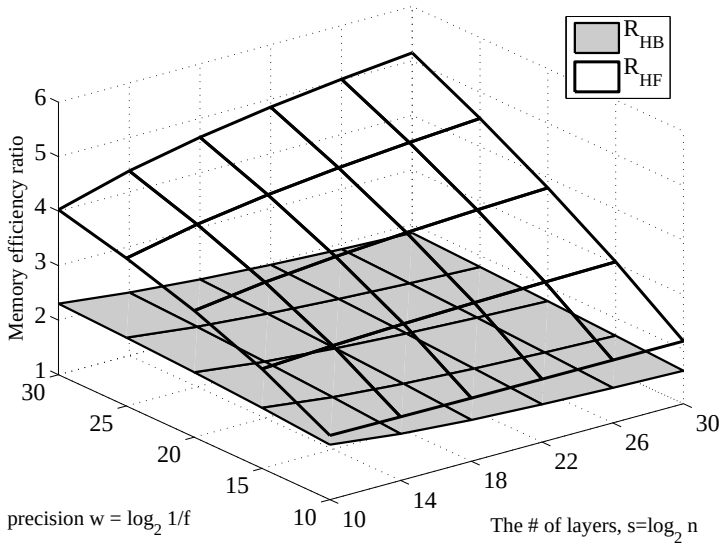


Fig. 28. Memory efficiency ratios of $R_{H,B}$ and $R_{H,F}$ with various s and w. Note a corrected-FHT is considered.

To maximize memory efficiency of an HIHT, the precision of layer t is set to 2, $0 \le t < s\text{-}1$ and that of layer s-1 is set to w as the same as precisions of an FHT and a BFHT. The precision value 2 is chosen based on the hardware consideration for pipelining and memory read ports as stated in Sec. 2. Also, for a fair comparison, the on-chip memory tables were considered in a BFTH and an HIHT. According to Eq. (3.2) for the requirement of $f=2^{-w}$, the total HIHT memory usage, M_3, counts two copies of HITs plus two VBAs and a key table and is calculated as the following

$$\begin{aligned} M_H &= 2\{2\beta n_0 w_0 + \cdots + 2^{s-1}\beta n_{s-1} w_{s-1} + n\} + n\log_2 n \\ &= 2\beta n \sum_{i=0}^{s-1} w_i + 2n + n\log_2 n = 2\beta n \left(\sum_{i=0}^{s-2} 2 + w_{s-1} \right) + 2n + n\log_2 n, \end{aligned} \tag{6.10}$$

where $\beta=1.44$. In contrast, the FHT memory usage is $M_F=\beta wns+4\beta wn=\beta w(s+4)n$ by considering 4-bit counters while an FHT in [15] did not consider the counters for a fair memory comparison. The memory size M_B for a BFHT is 2 times $kn(2\log_2 n+\log_2 k+w)$ due to the seamless update in $O(n\log n)$ complexity. Thus, considering $n\log_2 n$ for an on-chip key table, memory efficiency ratios $R_{H,F}$ and $R_{H,B}$ of M_H over M_F and M_B become

$$R_{H,F} = \frac{\beta nw(s+4)}{2\beta n \sum_{i=0}^{s-1} w_i + 2n + n\log_2 n} \tag{6.11}$$

$$R_{H,B} = \frac{2kn(\log_2 n + \log_2 k + w) + n\log_2 n}{2\beta n \sum_{i=0}^{s-1} w_i + 2n + n\log_2 n}. \tag{6.12}$$

Fig. 28 shows two ratios, $R_{H,F}$ and $R_{H,B}$, calculated from Eqs. (6.12) and (6.11) in the range [*10:30*] for w with the required precision of each scheme and in the range [*10:30*] for s. Note that the LHT memory size against M_H is not considered nor drawn in the figure, because with given high precision w of 29, the LHT memory size is more than a hundred times larger than that of an HIHT. The change in w provides

a greater benefit in $R_{H,F}$ than the change in s does, implying that as long as the collision rate stays lower for a high bandwidth, our HIHT maintains multi-integer-fold, not fractional-fold, efficiency gain. For instance, 160Gbps needs to set w to be at least 29 ($\approx \log_2 500M$). However, $R_{H,B}$ shows the minor change according to w and s, and it gives around two times memory efficiency over all ranges. These results on $R_{H,F}$ and $R_{H,B}$ support our claim that our HIT offers a better space-efficient hashing architecture shown in the previous section.

2. Power comparison with TCAM for IP lookup

In addition to memory comparison among hash mechanisms, power comparison between an HIHT-based IP lookup and TCAM-based IP lookup is made. Although the detailed hash-based IP lookup architecture with proposed hash schemes, an HIHT and an FPHT, will be shown in Sec. VIII, this section and the following section show preliminary power and memory comparisons with TCAM and a trie for IP lookup.

Fig. 29 shows the consumed energy per read clock in two IP lookup schemes: a TCAM-based IP lookup and an HIHT-based IP lookup. The consumed energy per read clock is measured by CACTI [35] and a tool [70]. The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random. It is found that the proposed HIHT-based IP lookup scheme consumed 51 times less power compared to the TCAM-based IP lookup. It is shown that the TCAM uses a tremendous power amount as the table size is increased while the proposed HIHT-based schemes uses a small power amount.

3. Memory comparison with Trie for IP lookup

Fig. 30 shows the total memory for IP Lookup in three schemes: a Tree Bitmap [71] and an HIHT. As Table I discussed, trie-based IP lookup schemes suffer $\mathcal{O}(W)$

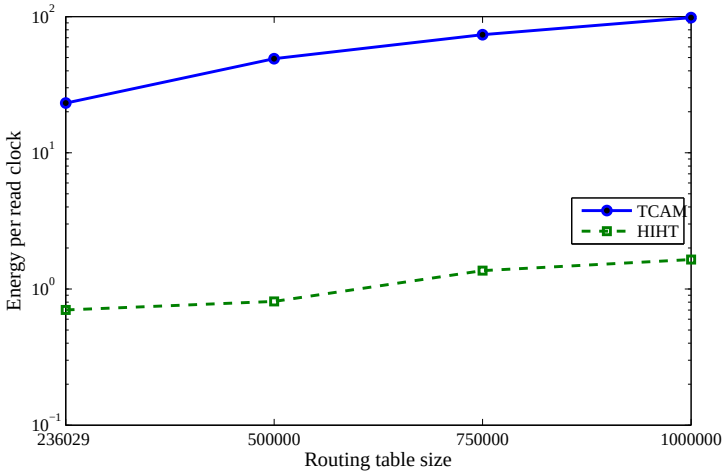


Fig. 29. Consumed energy per read clock in $0.09\mu m$ process technology.

lookup complexity where W is the IP address length while hash-based schemes provide $\mathcal{O}(W)$ lookup complexity. Comparing memory efficiencies of IP lookup schemes with different lookup complexities is not fair because hash-based schemes can provide a higher throughput. However, the proposed HIHT-based hash scheme can provide $\mathcal{O}(1)$ lookup complexity as well as memory efficiency as follows. The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random as in the previous section. In each memory calculation, other bitmaps and pointer memory overhead for hash-based and trie-based approaches are considered. It is found that the HIHT-bases IP lookup scheme consumed 1.75 times less memory compared to Tree bitmap. Furthermore, as the table size increases, the HIHT-based IP lookup scheme saves at most 2.15 times memory.

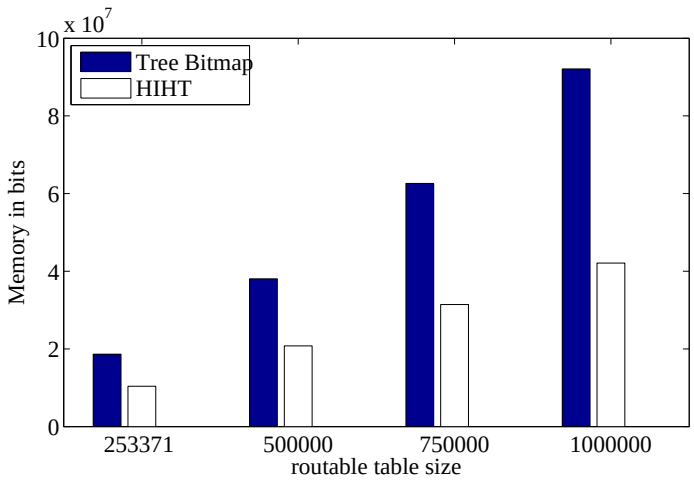


Fig. 30. Memory comparison of Tree Bitmap and an HIHT in different table sizes.

CHAPTER VII

HASHING USING BLOOM AND FINGERPRINT FILTERS

Sec. III.B shows that an SBF and an FF are memory- and power-efficient approximate membership testers. Such membership testers are used to propose an FPHT in this chapter. In an FPHT, a group of SBFs is used to do a pipelined binary search for a key's FP, and subsequently the found FP of w bits is used to access a key table with a desired lookup precision w. Suppose there exists no f-positive in a SBF and an FP. Then, there exists only one found FP in a FPHT's binary search. However, since a SBF and an FP can produce an f-positive, there exist candidate FPs and key indexes in the binary search result. Since the number of key table accesses through FPs is considered as the reverse of the lookup throughput, the probability of the fewer number of key table access is desirable for performing a high speed packet processing.

Fig. 31 illustrates our FPHT pipeline architecture where a set of SRAM modules is in pipeline and key and rule tables are accessed by an FPHT's generated indexes for a perfect match lookup. Based on a key's query results from SBFs which are designed through a SRAM module in Sec. A, each stage logic decides SBFs' base addresses based on Eq. (3.2) and accesses SBFs in the next stage.

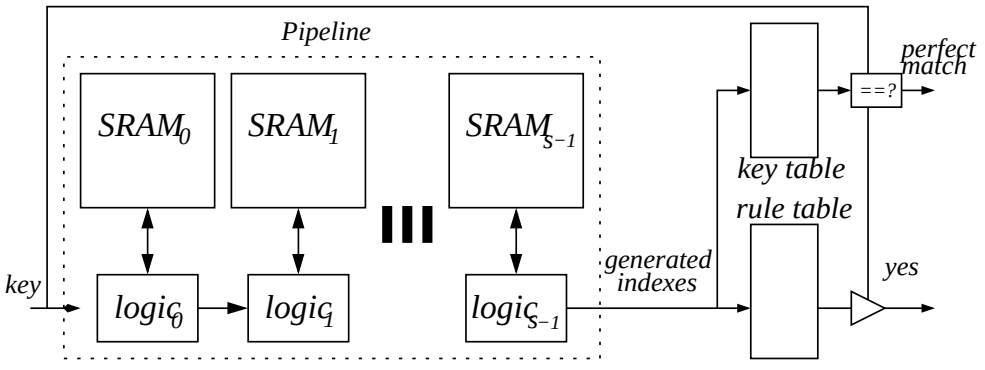


Fig. 31. An pipelined FPHT architecture with s stages.

In a logic view, a tree in multiple memory modules with a set of SBFs is conceptually embedded into a memory (Hereafter a BF or an SBF is used interchangeably). In the tree, nodes (or BFs) are used for a binary search where a BF's key query result choose the next BF nodes to probe. With BFs' and an FP' query results, the embedded tree generates indexes to a key table. The tree, called an indexing tree (IT), is memory efficient while preserving a required lookup speed, because it never uses pointers in implementing a tree for FP and key table accesses through the same indexes. In addition, the memory cost for BFs and an FF in an IT is far less than other schemes because BFs' role is limited to a binary search for a key's FP and an FF. This ensures one access to a key table with a high probability. This memory saving is beneficial to modern fast packet processing that are challenged with a scalability issue. The detailed IT build is as follows.

A. Building a Conceptual IT of a Binary Prefix Tree

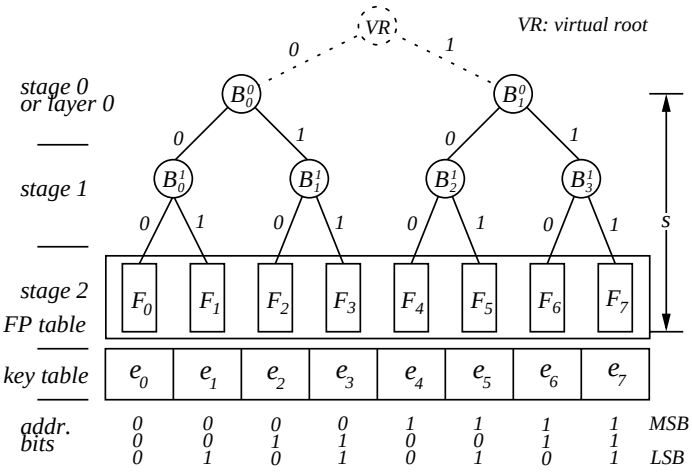


Fig. 32. Conceptual IT construction with BFs and tables of FPs and keys.

Suppose n is in power of 2. Then, an IT in a binary prefix tree is built as follows: An IT for n keys is composed of s=$\log_2 n$ stages and a prefix tree is built

based on index bits bits for a key table, so that a BF in the prefix tree is shared by a prefix among indexes. Fig. 32 shows the IT partition where keys are stored in a key table consecutively and the keys' index addresses are partitioned by BFs on stage j, $0 \leq j \leq s\text{-}2$ or FPs on stage s-1, so that each key has its own BF-FP path in the IT. In general, n keys are filled in an key table sequentially from index $0_0...0_{s\text{-}1}$ to index $1_0...1_{s\text{-}1}$. Let B_j^i denotes j-th BF in stage i, hereafter $0 \leq i \leq s\text{-}1$ while $F_j^{s\text{-}1}$ denotes j-th FP on stage s-1. Then, if key e is to be inserted at index $A = a_0a_1...a_{s\text{-}1}$, where $a_t \in \{0,1\}$, $0 \leq t \leq s\text{-}1$, a BF, denoted $B^i_{a_0 \cdots a_i}$ at each layer i, is involved to encode key e just like a legacy BF. In this hierarchical binary encoding, B_j^i covers $n_i = n/2^{i+1}$ keys whose indexes in a key table range from $j \cdot 2^{s\text{-}i\text{-}1}$ to $(j+1) \cdot 2^{s\text{-}i\text{-}1}\text{-}1$. For instance, B_0^0 and F_5^3 cover key sets $\{e_0, \cdots, e_3\}$ and $\{e_5\}$, respectively.

Regarding memory hardware, although BFs are conceptually partitioned in a layer (or a stage) for their key sets, they concatenate each other in a SRAM module while separated by their base addresses. That is, as Eq. (3.2) specifies the required BF memory size for a bounded f-positive, B_j^i has base address $j \cdot 1.44n_i$ in each memory bank of $M_{on}^i[k]$. Also, as it states that for a given f m is linearly proportional to n based on Eq. (3.2), given $f_i = 2^{\text{-}w_i}$ for a BF on stage i, the total memory of M_{on}^i for BFs on layer i is of size $2^{i+1}(1.44n_iw_i)$. Finally, the index order of keys' FPs in an FP table is exactly corresponding to that of keys in a key table, and the an FP table size in bits is nw based on Eq. (3.5).

An IT is named after a tree outlook because each stage in pipelining has a sequence of bits and sub-block of bits for a BF on stage i makes a binary relationship with sub-blocks on stage i-1 and i+1. Yet, to maintain a tree an IT does not use explicit pointers as in a binary search tree [60], but an implicit index for each BF sub-block in M_{on}^i. That is, B_j^i is located at $j \cdot 1.44n_iw_i$ in memory M_{on}^i. Also, all 2^{i+1} BFs, independent each other, on layer i contribute the memory size of M_{on}^i. Thus, a

large memory volume reserved for pointers is saved.

B. Insert Operation in an IT

Fig. 32 shows the binary address space with a set of BFs that *hierarchically* partition a key table's address space. In this tree structure, the insertion of key e_0 at index 100_2, for example, means B_1^0, B_0^1, and F_0^2 of layer 0, 1, and 2 are involved. **Algorithm** `insert-i` shows the detailed insert operation on stage i as simple as that for a BF.

Algorithm 10: `insert-i`()

Input: key e, rule r, and partial index A=$a_0a_1\cdots a_i$ in binary bits
Output: Encoded IT for key e on stage i

1 m_i=$1.44n_iw_i$; j=$a_0\cdots a_i$;
2 **for** t=0 **to** k-1 **do**
  // $h_t(e)\in\{0,\cdots,m_i'\text{-}1\}$, M_{on}^i `of` $2^{i+1}m_i\times$ `bits`
3 $M_{on}^i[k][h_t(e)][j]$==1; // M_{on}^i: `BFs on layer` i
4 **end**

Albeit conceptually all BFs are separate from each other in an IT, their hardware implementation assumes that BFs on layer i are embedded in one memory M_{on}^i and there exist s memory modules. Finding base address for B_j^i is easily calculated as shown in line 1. The first **for** loop is done in *parallel*, as does a legacy BF and **Algorithm** `insert-i` works on stage i in pipeline. Thus, the time complexity is $\mathcal{O}(1)$ under the condition that hash functions return indexes within a constant time, and each layer conducts hashing in parallel. This condition is made possible in hardware implementation as noted in [15]. After the last stage s-1, key e and its associated rule are saved as $M_{rule}[A]$=r and $M_{key}[A]$=e, where A is the designated address. The complexity of **Algorithm** `insert` for memory access to key and rule tables is $\mathcal{O}(1)$, because key e and its associated rule are saved in M_{key} and M_{rule} with A. In contrast, an FHT claims a time complexity of $\mathcal{O}(nk^2/m+k)$, while a BFHT does

$\mathcal{O}(n \log n)$.

C. Query Operation Making Indexes in an IT

Once all the keys are saved in a key table and encoded in a set of BF and FP memory modules, the ultimate remaining IT goal is to search a key by performing fast query operation. There are two kinds of search patterns, an unsuccessful search (UL) in which a key is relentlessly searched although it does not exist in an IT, and a successful but time-consuming lookup (SL) in which a key is to be searched in an IT. Before a discussion of these two kinds of searches, let definitions of an index path, a false index path, and a false segment introduced.

DEFINITION 7 (INDEX PATH)
In an IT, an index path, or ***i-path****, is defined as a series of* $B_j^i s$ *used in* `insert` *operation and hierarchically connected each other from layer* 0 *to layer* s-1 *to produce an index bit sequence. The sequence of indexing bits in* $B_j^i s$ *is also matched with an arbitrary index of a key saved in a key table and the size of the bit sequence from the series of* $B_j^i s$ *must be* s.

As a corollary, it can be concluded that in `query` for key that is e previously encoded by `insert`, an i-path for the key e should show up as BFs return 'yes' for their true membership.

A false index to a key table, other than an i-path dedicated to a key, is made possible due to the f-positives from irrelevant BFs or FPs in an IT. For example, suppose key e_4 is inserted with i-path $1_0 0_1 0_2$ in Fig. 32 and then a query to e_4 is requested. This query result may give an ambiguous $0_0 x x'$, $x, x' \in \{0, 1\}$, due to f-positives of B_1^1 and other FPs. Thus, this ambiguity needs to be resolved with other accesses to a key table. Given a query for an i-path of size s, there are totally

2^s*-1* false indexes because each B_j^i is independent and identically distributed, or i.i.d. Besides the i-path definition, I define a false index path in `query` operation, leading to a false index to a key table.

Definition 8 (False Index Path and False Segment)

In ***`query`****, a group of B_j^is or F_j^i not pertaining to an i-path can be formed in a series of at most size s from hierarchically consecutive layers. To become a false index path, or* ***f-path****, this series needs to be either connected to an i-path or a completely different path of size s, i.e. independent of an i-path in an IT. Also, the group attached to an i-path is called a false segment, or f-segment. The number of f-paths plus an i-path is the total number of key table accesses which was is the shared-linked-list length used for an FHT key query.*

Even if it is possible that there is a set of BFs giving f-positives in `query`, BFs that are only hierarchically mutually connected to BFs and an i-path can be a part of an f-segment. Thus, f-positives from the rest BFs can be ignored. For the previous example of 100_2 for e_4, even if B_1^1 and B_3^1 randomly make f-positives right after `query`, there is no f-segment starting from the B_1^1 and B_3^1. By the definition of an f-path, the probability of the f-path is cumulatively calculated as the product of f-positives from BFs along the f-path.

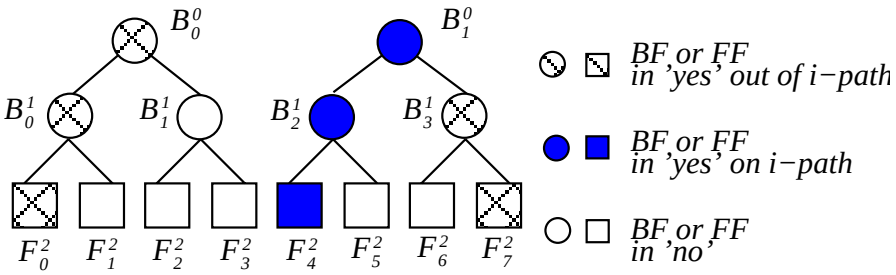


Fig. 33. Examples of an i-path and f-paths for a given query of key e_4 in an IT without a virtual root.

Fig. 33 shows an example of calculating the probability of an f-path in an IT with one i-path and two f-paths. A series of $B_1^0B_2^1F_4^2$ in the dark boxes is an i-path. The probability of the f-segment $B_3^1F_7^2$ forming f-path $B_1^0B_3^1F_7^2$ is $\prod_{t=1}^{2} f_t$ where f_t is the f-positive of a BF or an FP on layer t. Also, the probability of the remaining f-path, $B_0^0B_0^1F_0^2$, is $\prod_{t=0}^{2} f_t$ since the probabilities of f-positives of BFs or FPs on the same layer are the same each other.

Once the probability of an individual f-path is known, the final attention is paid to the probability that an IT may have t f-paths, $0<t<n$. Suppose a binary tree T of height ℓ has sub-trees T_l and T_r of height ℓ-1 which have n_l and n_r f-segments of size ℓ-1. Also, let T_l and T_r have probabilities F_{T_l} and F_{T_r} for their f-segments. To obtain n_l+n_r f-segments of size ℓ, the binary tree T with height ℓ needs to be an f-positive. Thus, the probability F_T of the binary tree T with its sub-trees having n_l+n_r f-segments is the product of three: the probability that T needs to be an f-positive, the probability that T_l has n_l f-segments, and the probability that T_r has n_r f-segments, i.e. $f_i \cdot F_{T_l} \cdot F_{T_r}$. Based on this recursive way, the probability $P_i(t)$ of t f-segments starting on layer i for an IT of height s is calculated as the following:

$$P_i(t) = \sum_{v=0}^{t} P_{i+1}(v) \cdot P_{i+1}(t-v) \cdot f_i \qquad \text{if} \qquad t \leq 2^{i+1}, \tag{7.1}$$

where base cases of $t > 2^{i+1}$ and $i=s$ are 0 and 1, respectively.

1. False indexing to a key table for a UL

Besides the design issue of producing a low probability of multiple accesses to a key table in an SL, it is equally important that the probability of f-indexes in a UL is also lower. Unlike an SL, there is no i-path for a given key in a UL, meaning that all BFs in `query` return 'yes' as f-positives. However, there is a chance that an IT may give plural f-paths. In contrast to an f-positive in an FHT [15] leading to off-chip memory

accesses, an f-path by a series of f-positives with *hierarchically* connected B_j^is in each layer i becomes one index access to a key table. Thus, a far less probability is expected due to the product of f-positive probabilities of BFs.

Suppose random variable $\mathcal{X}_u$ is the number of f-paths in a UL on an IT. Then, the probability $Pr\{\mathcal{X}_u = v\}$, v>0 can be easily derived based on Eq. (7.1) as the following

$$Pr\{\mathcal{X}_u = v\} = \sum_{v=t_0+t_1} P_0(t_0) \cdot P_0(t_1) \tag{7.2}$$

because an IT has two children trees on layer 0. The sum in Eq. (7.2) accounts for the combination of deriving v among t_0 and t_1. That is, if v=1, there are two cases: 0+1 and 1+0.

2. False indexing to a key table for an SL

The probability of f-paths in a UL is derived. Now, the probability of the f-paths number in an SL is calculated. The situation in an SL is very different from that of a UL, because there must exist one i-path with possible f-paths of highly low probability while there is no i-path in a UL. Fig. 34 shows an example of 3 layers for 2^3 keys where along an i-path there are 3 dangling trees, labeled as d-trees, contributing to f-paths, if any. All d-trees except one rooted on layer 0 are attached to the i-path and they contribute to the f-paths number with different probabilities related to $P_i(n)$ of Eq. (7.1).

Based on the observation from Fig. 34, to calculate the f-paths number, i.e. false indexes to a key table in an SL, is necessary. Let random variable $\mathcal{X}_s$ be the number of f-paths in `query` operation with an i-path for a key. Then, $\mathcal{X}_s$+ 1 is the total indexes in an SL which equals to the searched-linked-list length of an FHT [15]. The detailed probability of $\mathcal{X}_s$ for an SL without f-paths is defined as following:

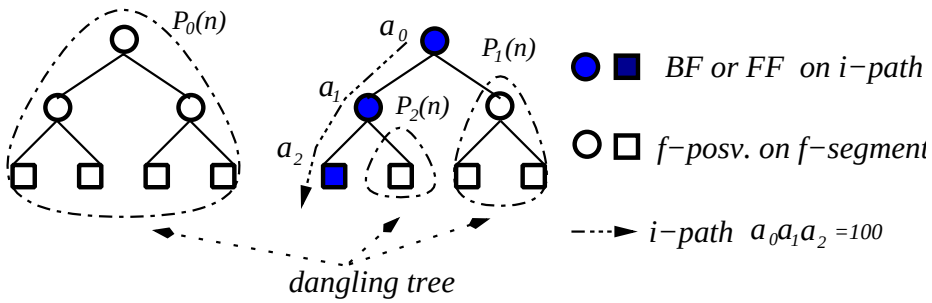


Fig. 34. An IT of 3 layers (or stages) with an i-path and dangling trees.

$$Pr\{\mathcal{X}_s = 0\} = P_0(0) \cdot P_1(0) \cdot P_2(0) \cdots P_{s\text{-}1}(0), \tag{7.3}$$

because each d-tree along an i-path are mutually independent to each another. In general, the $Pr\{\mathcal{X}_s{=}v\}$ is calculated based on the independent property of each d-tree along an i-path as the following

$$Pr\{\mathcal{X}_s = v\} = \sum_{v=t_0+\cdots+t_{s\text{-}1}} P_0(t_0) \cdot P_1(t_1) \cdot P_2(t_2) \cdots P_{s\text{-}1}(t_{s\text{-}1}). \tag{7.4}$$

3. Detailed algorithm for query

A complete `query` operation consists of `query-i` for layer i, $0{\le}i{\le}s$-1, shown in **Algorithm** `query-i`. The time complexity of this algorithm is $\Theta(1)$ under the condition that $\|L\|$ is bounded by the number of memory reads supported by the hardware without overhead. The reason for this is that given candidates for partial indexes in L, the number of BFs to probe doubles due to having two children to each binary tree node.

By pipelining starting layer 0, a query is performed in one cycle, so that `query-s-1` returns complete indexes to a key table for a given query. On the last layer s-1, the average numbers of complete indexes are calculated as $E[\mathcal{X}_s]{+}1$ or $E[\mathcal{X}_u]$ on average

Algorithm 11: query-i()

Input: M^i_{on} for layer i$\leq s$-1, list L of partial indexes found on up to layer i-1 including i-path, and key e

Output: A set of partial $A = a_0 \cdots a_i$ of i+1 bits, including f-segments

```
// S: Set of partial paths.  L={A_0, ... A_{n-1}}
S = ∅;  n=||L||;                                  // ||L|| is the size of L
for t = 0 to n-1 do
    m_i=1.44 n_i w_i;  A_t = L[t];  j_0=A_t·0;  j_1=A_t·1; cnt_0=cnt_1=0;
    // One M. for BFs on layer i.  k_i hash funcs.
    for t=0 to k_i-1 do
        // idx_0, idx_1 indicate B^i_j
        if M^i_on[t][h_t(e)][j_0]==1 then cnt_0++;
        if M^i_on[t][h_t(e)][j_1]==1 then cnt_1++;
    end
    // concatenate 0 or 1 bit at the end of A_t
    if cnt_0==k_i then S=S∪ A_t · 0;
    else if cnt_1==k_i then S=S∪ A_t · 1;
end
return S;                                  /* No memory access for a key table */
```

for an SL and a UL, respectively, where $E[\mathcal{X}_s]$ and $E[\mathcal{X}_u]$ can be derived from Eqs. (7.4) and (7.2) as

$$E[\mathcal{X}_s] = \sum_{t=0}^{n-1} t \cdot Pr\{\mathcal{X}_s = t\}, \qquad E[\mathcal{X}_u] = \sum_{t=0}^{n} t \cdot Pr\{\mathcal{X}_s = t\}. \tag{7.5}$$

These equations are considered $\mathcal{O}(1)$ because having one f-path, $Pr\{\mathcal{X}_s{=}1\} \ll o(1)$ and $Pr\{\mathcal{X}_u{=}1\} \ll o(1)$, is very unlikely because of the way a high-speed router is designed.

D. Delete Operation with Counting BFs

A BMF in [27] suffers from a dynamic membership change, because an index table stores a key's k hash values of based on its neighborhood with other keys and this neighborhood is collected by avoiding collision with other keys' hash values. Thus, an index-table setup in a Blooimer filter takes $\mathcal{O}(n \log n)$ complexity, implying that

a BFHT using a Bloomier filter needs the same time complexity for updating keys. However, our FPHT takes $\mathcal{O}(1)$ for update. Unlike an MBHT [31] and an HIHT [32], CBFs for a dynamic update is adopted. Since CBFs are used for `delete` operation, `insert` operation needs to be modified at line 3 as `query` operation does at lines 5 and 6 for counter operations. The detail is the following.

To remove a key with an i-path, all CBFs on the i-path need to delete the key. Deleting the key in a CBF is as easy as decreasing counters indexed by hash functions. Also, an FP for the key is reset to 0 to indicate an empty FP. Since an i-path for the key is known, resetting the FP is easy. If a membership of a key to remove is not known, a lookup on the key is necessary to find an associated i-path. In this `query`, if there are any f-paths besides an i-path associated with a key, the necessary number of key table accesses is one plus the number of f-paths. Once CBFs and an FP is updated for the key deletion, the i-path (or the FP index) is saved in an index pool, so that when a new key insertion is asked, one from the index pool is used for the next key's insertion as an i-path.

When random variable $\mathcal{Z}$ is denoted as the number of accesses to a key table, the average memory access for `delete` operation on the condition of a target key's existence, i.e. a successful deletion, is

$$E[\mathcal{Z}] = 1 + \sum_{v=1}^{n-1} v \cdot Pr\{\mathcal{X}_s = v\} = 1 + E[\mathcal{X}_s]. \tag{7.6}$$

The `delete` complexity is $\mathcal{O}(1)$ based on the `query`'s $\mathcal{O}(1)$ complexity. The complexity of indexes to access a key table is $\Theta(E[\mathcal{Z}])$ on average for a successful deletion and it is to be constant as $E[\mathcal{X}_s]$ is $\mathcal{O}(1)$.

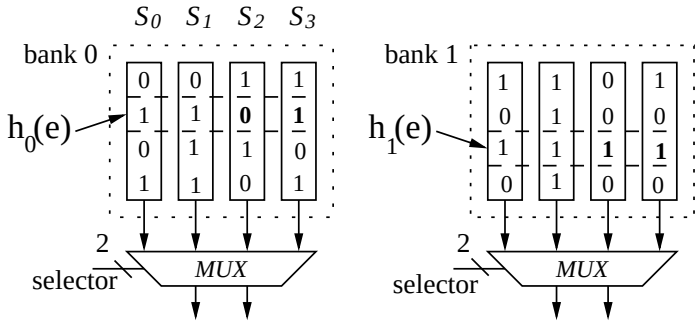


Fig. 35. A sample configuration of a 4-SBF in k=2 banks. A 4-SBF represents S_0 through S_3. The memory size is 2×4×4.

E. FPHT Optimization in a b-ary Prefix Tree

The IT so far is built as a binary prefix tree in a 2-base number system. Since a BF acts as a binary-predicate in an IT, a BF assigned for bit 0 in its index bits returns 'yes' like a BF assigned for bit 1 does. However, when a b-ary prefix tree, $b \in \{2^2, 2^3, \cdots\}$, is adopted in an IT, a BF is assigned for bit x, $x \in \{0, \cdots, b\text{-}1\}$, and a node in a b-ary prefix tree is implemented in a b-SBF as a 4-SBF is shown in Fig. 35. Also, the IT height, i.e. the number of pipeline stages, is reduced to $s = \log_2 n$, thereby the total IT memory is. According to b-base number, using a b-ary prefix tree requires a bit change in an key table index of 2-base number system. For instance, index 0100_2 for e_4 in a binary prefix tree is simply transformed to 10_4 in a 4-ary prefix tree. However, this change does not create index addressing disturbance. Thus, without any key table change memory saving is observed by adopting a b-ary prefix tree.

F. Simulation Results for an FPHT

This section presents an analysis on memory efficiency for three schemes; an MBHT [31], an HIHT [32], and an FPHT.

1. Memory size in consideration of speed and scalability

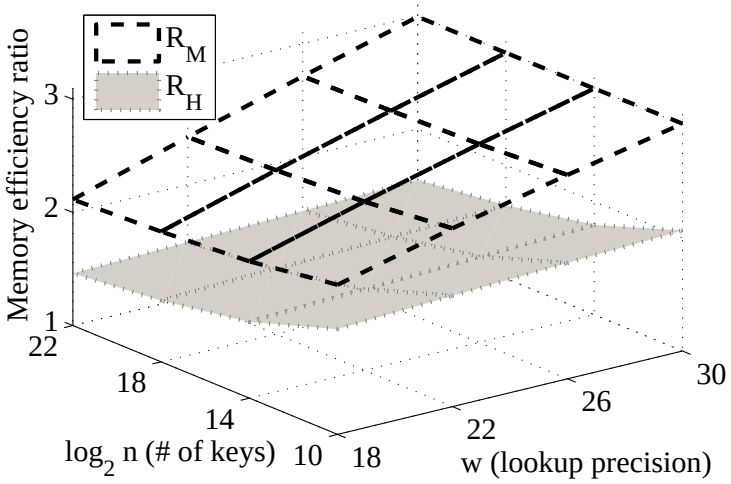


Fig. 36. Memory efficiency ratios of an FPHT over an MBHT and an HIHT at various n and w. In an FPHT, a lookup precision of a CBF is set to 6 for a 16-ary prefix tree.

This section shows calculation of the memory efficiency ratio among an MBHT, an HIHT, and an FPHT to properly address speed and scalability. Since authors in [31, 32] made memory efficiency by comparison of their schemes against an FHT [15] and an BFHT [16], the memory comparison is not considered again. Also, a 16-ary prefix tree is used for an FPHT optimization. The MBHT memory size is M_M=$2\beta n(w+\log_2 b)\log_b n+2n+n\log_2 n$, b=16, and the HIHT memory size, M_H, is calculated as $2\beta n\cdot(3(\log_2 n\text{-}1)+w)+2n+n\log_2 n$. In contrast, the FPHT memory size becomes $\beta n(3+\log_2 b)\times(\log_b n\text{-}1)\times C+nw+n\log_2 n$ where $\log_b n$ is a prefix tree height and C=3 for counter bits.

Fig. 36 shows two memory efficiency ratios, R_M and R_H of an FPHT over an MBHT and a HIHT based on Eqs. (3.2) and (3.5). As shown in this figure, an MBHT is not suitable for speed and scalability concerns. Although R_M at small w and n values is smaller than that of higher w and n values, it is evident that in the

overall range of w and n an FPHT approximately needs smaller memory size than a MBHT, and the highest memory efficiency is 3.0. In case of R_H, 2.1 times memory efficiency is shown. The memory capacities of an MBHT and an FPHT at the highest efficiency with n=2^{10} and w=30 are 262,964 and 87,409 bits, respectively.

2. Power comparison with TCAM for IP lookup

In addition to memory comparison among hash mechanisms, power comparison between an FPHT-based IP lookup and TCAM-based IP lookup is made. Although the detailed hash-based IP lookup architecture with proposed hash schemes, an HIHT and an FPHT, will be shown in Sec. VIII, this section and the following section show preliminary power and memory comparisons with TCAM and a trie for IP lookup.

Fig. 37 shows the consumed energy per read clock in two IP lookup schemes: a TCAM-based IP lookup and an FPHT-based IP lookup. The consumed energy per read clock is measured by CACTI [35] and a tool [70] The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random. It is found that the proposed FPHT-based IP lookup scheme consumed 51 times less power compared to the TCAM-based IP lookup. It is shown that the TCAM uses a tremendous power amount as the table size is increased while the proposed FPHT-based schemes uses a small power amount.

3. Memory comparison with Trie for IP lookup

Fig. 38 shows the total memory for IP Lookup in three schemes: a Tree Bitmap [71] and an FPHT. As Table I discussed, trie-based IP lookup schemes suffer $\mathcal{O}(W)$ lookup complexity where W is the IP address length while hash-based schemes provide $\mathcal{O}(W)$ lookup complexity. Comparing memory efficiencies of IP lookup schemes with different lookup complexities is not fair because hash-based schemes can provide

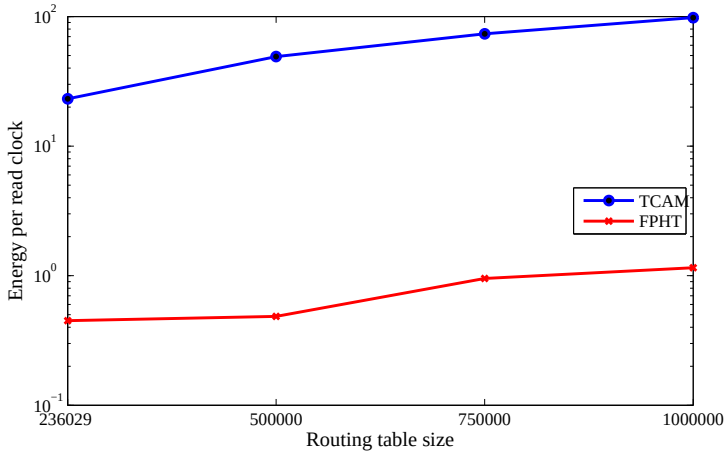


Fig. 37. Consumed energy per read clock in $0.09\mu m$ process technology.

a higher throughput. However, the proposed FPHT-based hash scheme can provide $\mathcal{O}(1)$ lookup complexity as well as memory efficiency as follows. The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random as in the previous section. In each memory calculation, other bitmap and pointer memory overhead for hash-based and trie-based approaches are considered. It is found that the FPHT-bases IP lookup scheme consumed 1.75 times less memory compared to Tree bitmap. Furthermore, as the table size increases, the FPHT-based IP lookup scheme saves at most 2.4 times memory.

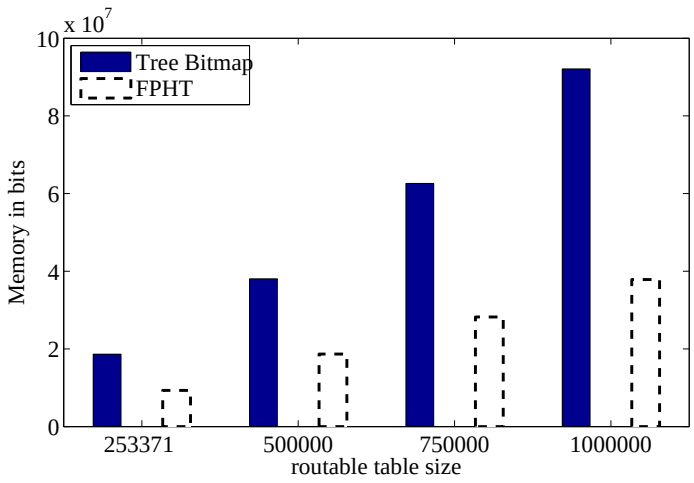


Fig. 38. Memory comparison of Tree Bitmap and an FPHT in different table sizes.

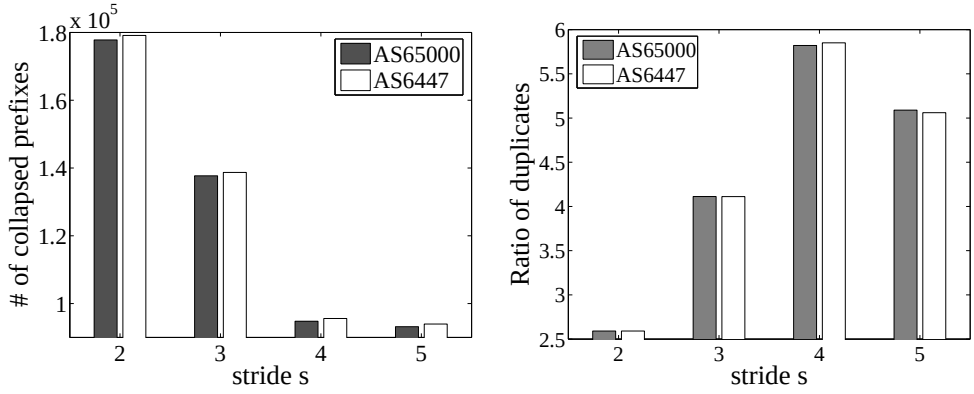
CHAPTER VIII

HASH-BASED IP LOOKUP ARCHITECTURE

This chapter presents HIHT and FPHT IP lookup architectures based on the proposed hashing schemes and compares their performances with contemporary IP lookup architectures in terms of power consumption and memory overhead.

A. Hash-based IP Lookup Architecture Build

Authors in [15, 16] show that hash-based IP lookup schemes are capable of providing better memory and power performance. However, since a hash only supports a singleton match, either a prefix collapse in Sec. III.C.2 or a controlled prefix extension in Sec. III.C.1 is necessary if hash schemes are applied to IP lookup. Since a controlled prefix extension inflates the number of next-hops, a prefix collapse scheme is a better way in build hash-based IP lookup architecture when proposed HIHT and FPHT schemes are applied to IP lookup.



(a) The # of collapsed prefixes (b) Avg. ratio of duplicate next-hops

Fig. 39. The number of collapsed prefixes and the average number of duplicate next-hops at various stride s. The prefix number for AS 65000 and AS 6447 are 233451 and 235307, respectively.

Fig. 39(a) shows the benefit of using the prefix collapse. In the figure, the number of collapsed prefixes gets smaller than the number of the original prefixes at various stride s. As the stride size increases, the number of collapsed prefixes reduces and it is 2.7 times smaller than that of an original prefix set at stride 5 as shown in Fig. 39(a). However, just as the stride size increases, there exists a problem with the number of next-hops. When a bitmap for the prefix collapse is used, the ratio of next-hop duplicates is increased as shown in Fig. 39(b). For example, the duplicate ratio of 5.8 at stride 4 indicates that a bitmap of size 2^4 has 5.8 times duplicate next-hops on average. The used BGP tables, AS 65000 and AS 6447, are obtained from [51] and other BGP tables also show the similar pattern. However, since this dissertation aims for power and memory efficiencies in hash itself, we leave the issue of next-hop inflation open for hash-based IP lookup.

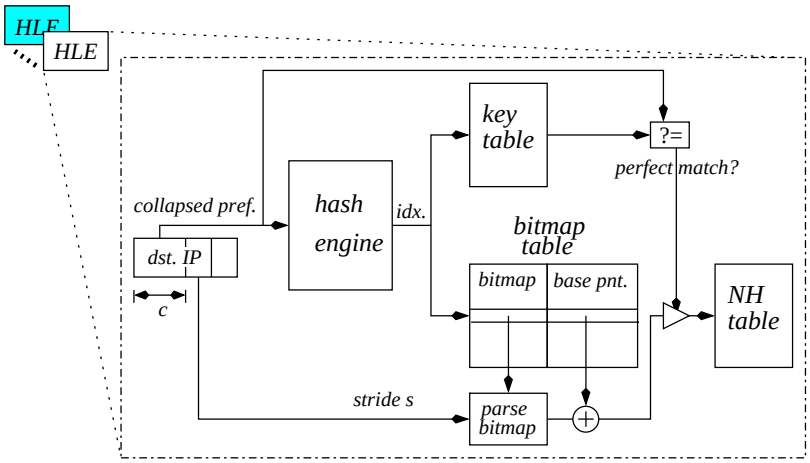


Fig. 40. IP lookup architecture with parallel Hash Lookup Engines (HLEs) for a wild-card support. Each HLE has different c and s values.

Fig. 40 illustrates a general hash-based IP lookup architecture using the prefix collapse and the bitmap scheme. Prefixes are divided into collapsed prefixes and bitmaps. Later, each HLE saves collapsed prefixes of the same length c in a key table for a perfect match and its corresponding bitmaps in a bitmap table in order to index

the next-hop table. In the figure, an HIHT or an FPHT is substituted for an HLE. For each IP lookup operation, an HLE strips the first c bits and the following s bits from a destination IP, does hash based on c bits, and accesses a next-hop table by parsing an indexed bitmap, if perfectly matched. A match with a longest collapsed prefix is the final match for a given IP lookup among perfect matches.

B. Simulation Result of HIHT and FPHT-based IP Lookup Schemes

This section shows comparison result of HIHT and FPHT-based IP lookup schemes against contemporary schemes in terms of power and memory. For a scalability issue of routing table size, we consider four sizes: 236,029, 500K, 750K, and 1M.

1. Power-efficient hash-based IP lookup

Fig. 41 shows the consumed energy per read clock in three schemes: a TCAM, an HIHT shown in Sec. VI, and an FPHT shown in Sec. VII. We use CACTI [35] and a tool [70] to measure the consumed energy per read clock. The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random. It is found that the proposed scheme consumed 51 times less power compared to the TCAM-based IP lookup and 1.5 times less power compared to the HIHT-based scheme for the first table. It is shown that the TCAM uses a tremendous power amount as the table size is increased while our hash-based schemes of an HIHT and an FPHT use a small power amount. Furthermore, an FPHT-based scheme always uses less power amount than an HIHT-based scheme in all table sizes since an FF uses a smaller power than an BF as discussed in Sec. B.

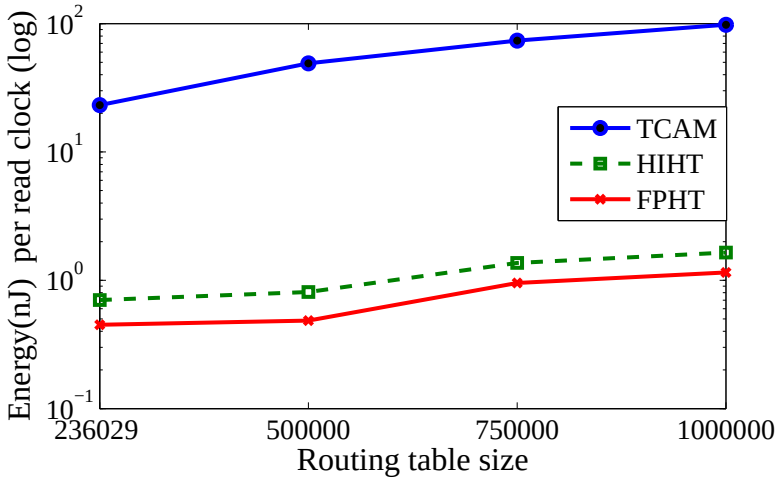


Fig. 41. Consumed energy per read clock in $0.09\mu m$ process technology.

2. Memory-efficient hash-based IP lookup

Fig. 42 shows the total memory size for IP Lookup in three schemes: a Tree Bitmap [71], an HIHT, and an FPHT. As Table I discussed, trie-based IP lookup schemes suffer $\mathcal{O}(W)$ lookup complexity where W is the IP address length while hash-based schemes provide $\mathcal{O}(1)$ lookup complexity. Comparing memory efficiencies of IP lookup schemes with different lookup complexities is not fair because hash-based schemes can provide a higher throughput. However, our hash-based schemes can provide $\mathcal{O}(1)$ lookup complexity as well as memory efficiency as follows. The table size was varied from 236K to 1M entries. The first table of 236K entries is taken from [51] and the rest tables are created by random as in the previous section. In each memory size calculation for hash-based and trie-based approaches, other bitmaps, pointer memory overhead, and hash-engine memory are considered. It is found that the HIHT-based scheme consumed 1.8 times less memory compared to Tree Bitmap scheme and the FPHT-based scheme used 1.1 less memory compared to the HIHT-based scheme for the first table. In conclusion, it is shown that the FPHT-based scheme is the most

memory-efficient IP lookup scheme in this result. Furthermore, as the table size increases, the FPHT-based scheme saves at most 2.4 times memory.

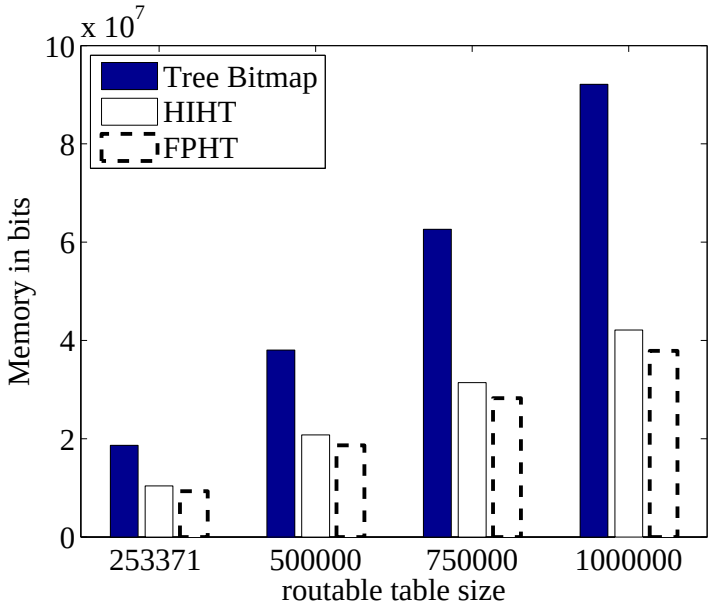


Fig. 42. Memory size comparison of Tree Bitmap, an HIHT, and an FPHT in different table sizes.

CHAPTER IX

HYBRID CAMS OF CAM AND SRAM FOR IP LOOKUP

In this chapter, we propose a hybrid CAM (HCAM) IP lookup architecture for high throughput and power efficiency. Our approach adopts both a prefix collapse scheme and a circuit level redundancy in multi-ports to a Bloom filter (BF). A prefix collapse reduces the number of prefixes while a collapsed prefix does not have a prefix feature. In such prefix collapse, the collapsed prefixes (CPs) can be put in a deterministic lookup-capable CAM to demonstrate further hardware efficiencies on power and the number of transistors per cell than a TCAM. The detail is the following.

A. HCAM-based IP Lookup Architecture

Using TCAM for a prefix match has been considered as a prohibitive scheme despite TCAM's advantages in a deterministic lookup and partitioning for multi-lookups. This section presents the detail of HCAM-based architecture with high throughput and power efficiency.

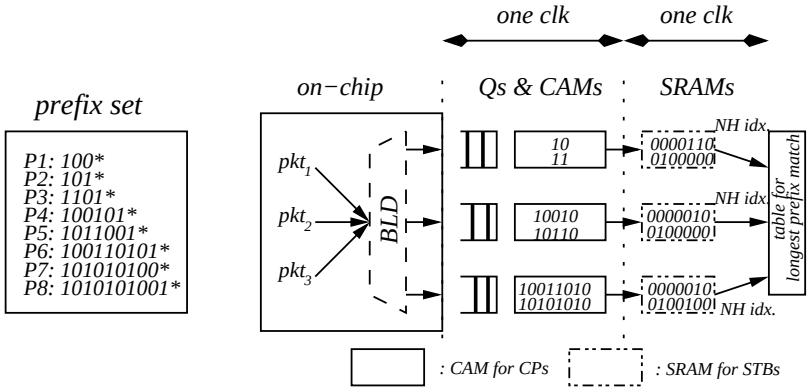


Fig. 43. HCAM-based IP lookup architecture for a prefix set. Stride s=2. The collapsed prefix lengths,d_1, d_2, d_3, are 2,5, and 8, respectively.

A pipeline in an HCAM-based scheme has three stages *in pipeline*; a distributor

with BFs, CAMs with queues, and SRAMs as in Fig. 43. To provide a high throughput, multiple pipelines can be used working *in parallel*. In the figure, 3 packets are fed into a distributor together, and the distributor disseminate the packets to their associated queues. A queue is buffer zone between a fast-distributor stage and a slow-CAM stage as in [36, 38]. A Bloom filter is well known for a binary *approximate* membership query [49], and it removes irrelevant lookup queries to collapsed prefixes which are saved in a CAM block. Thus, a high-power-consuming CAM query is avoided. Once a CAM block entry is *perfectly* matched with a collapsed prefix, we retrieve an SRAM block entry at the same index. The retrieved entry indicates an STB associated with the collapsed prefix for stride match. Thus, the prefix match is achieved by performing CAM and STB matches.

As to completing the longest prefix match, a table is used to record all CAM matches for a given lookup. Once a lookup is forwarded to associated queues by a distributor, a record of match statuses in all pipelines is created in the table. Whenever a match is found in any pipeline, the match is recorded in the lookup's record. Once a found match is considered as the longest prefix match in the record, a packet associated with the lookup is forwarded without waiting for other query results of the lookup.

By IP lookup policy, a router forwards packets based in a prefix set while preserving a packet order. Although packet disorder can happen due to queues' delay in parallel lookups, the disorder does not disturb an order of packets belonging to a single flow. A flow is defined as a set of packets between applications on two end hosts identified by two IP addresses, and a prefix represents a set of IP addresses to forward packets if the prefix is the longest. As long as a match order of packets which are associated with the longest prefix is preserved, the same order of the outgoing packets is. Since our HCAM scheme preserves an packet order in a pipeline queue by

selectors, the match order after a SRAM is the same as the order of packets in a flow.

B. Prefix Transformation with CAM & SRAM

A proposed PC uses one bitmap and one pointer to encode a subtrie in a uni-bit trie, so that the number of prefixes is reduced and the need of 'don't care' (or *) bit comparison is eliminated. Now, the collapsed prefixes can be put in a CAM which has less hardware complexity than a TCAM and provides a singleton match, but the same parallel lookups through partitioning. As to comply with a prefix match on stride bits, we build an STB in a SRAM, so that the overhead of TCAM usage disappears. The details of a PC and an STB are shown in the following sections.

1. Prefix collapse

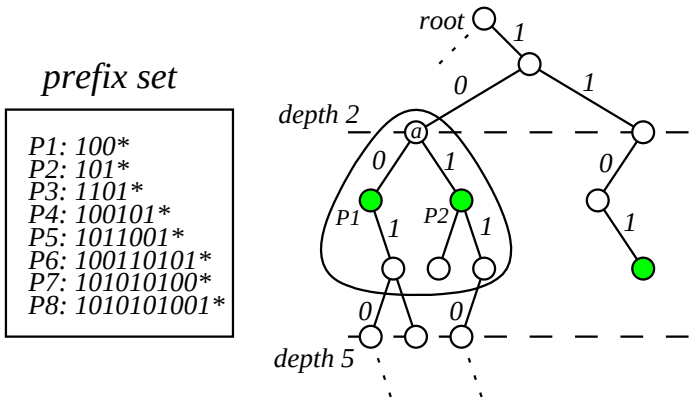
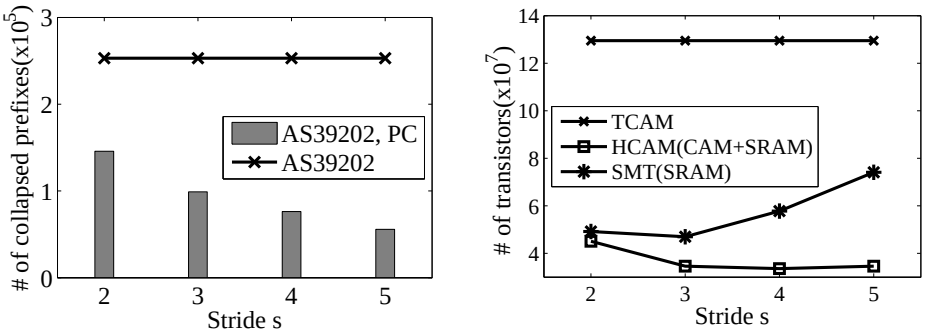


Fig. 44. A sample prefix set and a subtrie in a uni-bit trie for the set.

Given a uni-bit trie for a prefix set, our PC encodes every subtrie which is rooted at specific trie-depth and whose depth is s, so that prefixes in a subtrie share a common path from a trie root to the subtrie root. Suppose there are 8 prefixes and a subtrie with root node a at trie-depth 2 is encoded as in Fig. 44. Since prefix P1 and P2 share a common prefix part, i.e. '10', one collapsed prefix '10' is used in a

CAM. Thus, the number of collapsed prefixes for a CAM can be far smaller than the number of original prefixes.



(a) The # of collapsed prefixes (b) Memory size in terms of transt.

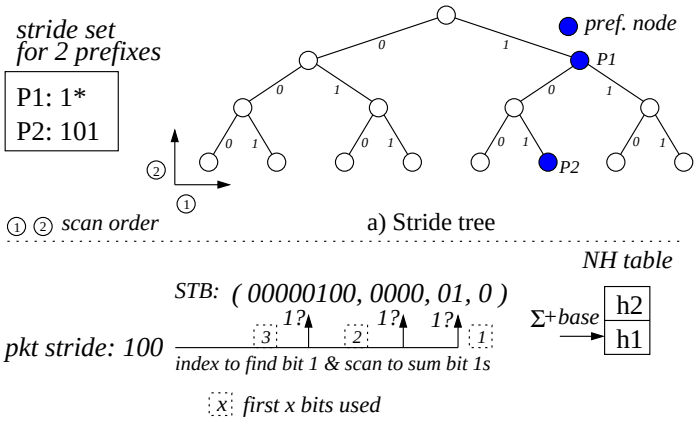
Fig. 45. The number of collapsed prefixes and the number of transistors at various stride s.

Such PC's benefit is shown in Fig. 45(a) by counting collapsed prefixes from a routing table AS 39202 [72] whose prefix number of prefixes is 252,951. The relationship between the stride size and the number of collapsed prefixes is that as the stride size gets larger, the number of collapsed prefixes is getting smaller. At stride 5, the number of collapsed prefixes is 4.5 times fewer than that of the original prefixes marked as a line, and for the stride 5 66, 3376, 52254, and 63 collapsed prefixes are found at depth 7, 13, 19, and 25, respectively. Other stride sizes do not cause any significant reduction.

2. A complete prefix match through an STB in SRAM

Since a CAM does not support a prefix match, a supplementary match is necessary even after a CAM match occurs. Given a collapsed prefix, there are 2^{s+1}-1 possible prefixes at stride s, and they can be presented at a subtrie bitmap. Fig. 46 a) shows two prefix strides at stride s=3 and a stride tree for them. In a stride tree, a node

is marked as '1' when there is a corresponding prefix stride. Thus, when scanning nodes' bits in the horizontal order followed by the vertical order, we get an 15-bit STB (00100000,0010,01,0) for three prefix strides.



b) Index calculation for a given packet stride

Fig. 46. A stride tree for 2 prefix strides and an index method to an NH table.

Given an STB for the stride s, there are s+1 groups of bits, each designated for bits on the same layer in a stride tree. In each group, bits are scanned while the number of bits of value 1 is counted, and when a bit indexed by the most significant bits in a stride is 1, the counting stops. Then, the summed number of bits of 1, $\sum$, becomes a relative index to an NH table. Fig. 46 b) shows such an index calculation in STB (00100000,0010,01,0) for the packet stride 100. Once a CAM block match happens, the match's index in the CAM block is used to access an STB in the corresponding SRAM block. Once the STB is known by one SRAM access, calculating an index $\sum$ can be made shortly at one CPU-clock speed. **Procedure** `stride_match` shows the detailed steps.

As to subtrie memory in Fig. 44, a proposed PC needs an (2^{2+1}-1)-bit STB with one base pointer to an NH table for a 2-bit stride subtrie. In general, the STB size is 2^{s+1}-1 bits for a subtrie of s stride bits. However, the subtrie size for a

Procedure `stride_match`

Input: Stride S of $a_0 \cdots a_{s-1}$, stride size s, and STB B of $b_0 \cdots b_{2^{s+1}-2}$
Output: Relative index $\sum$ to an NH table, or "no match"

```
1  for (idx_B=s-1, sum=0; idx_B≥0 ; idx_B−−) do
2      idx_S = a_0 ··· a_{idx_B};
3      if idx_B=s-1 then scan_B=0;
4      else scan_B=Σ_{t=0}^{idx_B-1} 2^{s−t};                    // Set base in B to scan
5      for (idx_scan=0;idx_scan≤idx_S+scan_B;idx_scan++) do
6          if B[idx_scan]==1 then sum++;
7      end
8      if B[idx_scan]==1 then                                    // match happens
9          return sum;
10 end
11 return "no match";
```

segmented multibit trie (SMT) in [20] is 19 bits which is 2.7 times more than the STB size. Generally, an SMT needs $3k+1(=2k+k+k+1)$ bits for k neighboring nodes. In addition, an SMT needs two pointers to maintain connectivities among SMTs and an NH table.

Such pointer overhead is manifest in Fig. 45(b). In general, as the stride is larger, the numbers of transistors for an SMT scheme and an HCAM are reduced significantly except at stride 5, and the numbers are, at most 3.9 times, smaller than that of a native TCAM scheme. In comparison between a SMT and an HCAM, an HCAM uses 1.7 times less memory at stride 4 because an SMT is designed to encode a lightly loaded subtrie and to maintain connectivity with others through pointers.

C. A Bloom Filter-based Lookup Distributor

STCAM [37] and BTCAM [38] schemes use a distributor forwarding multiple packets per clock cycle. Such a packet distribution to corresponding pipelines is necessary for high throughput. Such a distributor adopts a multi-tiered BLD with a set of n_c BFs [30], each forwarding a packet to a corresponding pipeline. Such a BLD is designed

to distribute lookups for multi-lookups per cycle and remove unnecessary lookups in queues for a power efficiency lookup.

The total memory usage of various contemporary schemes is differentiated from a proposed HCAM-based scheme as shown in Fig. 47. TCAM or SRAM blocks of other schemes are only considered, and not memory block selectors, to store prefixes for prefix match in this comparison. However, BFs' memory is included in the HCAM memory calculation. Although the BTCAM shows the highest throughput among other contemporary schemes, our HCAM uses 2.8 times less memory while achieving the same throughput as the BTCAM.

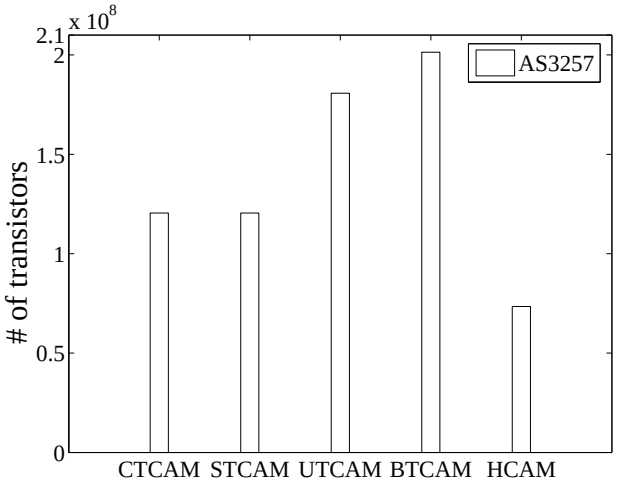


Fig. 47. The memory comparison of all schemes in terms of a transistor. Lookup precision w=10. Note that 'HCAM' includes all CPs, STBs, and BFs.

D. Experimental Results for an HCAM-based Scheme

This section presents an analysis on throughput and power efficiencies for an HCAM-based scheme and other contemporary schemes.

1. Throughput

It is difficult to theoretically analyze the HCAM's throughput performance because the non-determinacy of the lookup traffic. However, the upper and the lower bound of its performance can be estimated based on the following lookup traffic assumptions. 1) Queuing theory is used to model the lookup engine and assume that the arrival process of the incoming IP addresses is a Poisson process with the average arrival rate λ. 2) The service process of the lookup operation follows a deterministic distribution with a constant service rate μ due to CAM's deterministic lookup. Then, a service time to process lookups in a queue becomes $T_s{=}1/\mu$, and it is independent of the arrival processes. 3) The queue size in each pipeline is finite with n_q lookup requests.

It is obvious that if n_c CAM blocks perform independent IP lookups, the system can be modeled as an $M/D/n_c/n_q n_c$ queuing. In this case, the lookup traffic can be always balanced among all n_c CAMs. Thus, the $M/D/n_c/n_q C$ queuing model should be the upper throughput bound.

However, the lower throughput bound is more interesting since it affects the practicality of the proposed scheme to a real field. By neglecting the adaptive load balancing process and assuming that the traffic is evenly distributed to n_c CAMs by BFs, an HCAM can be modeled as n_c independently and identically distributed $M/D/1/n_q$ queuing network for n_c pipelines as in Fig. 48.

Now, an analysis is made on one of the identically distributed $M/D/1/n_q$ queues by considering an increased arrival rate in a queue due to a BF's f-positive. That is, the arrival rate λ/n_c is increased by a probabilistic value f because a BF falsely assigns a lookup to each queue due to a BF's f-positive. Once such a look exists in a queue, a CAM block needs to proceed a lookup operation for the unsuccessful look, and this consequently undermines throughput and wastes power. Thus, the traffic

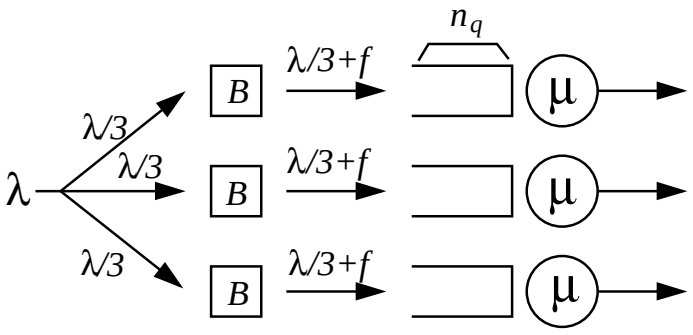


Fig. 48. Queuing model of n_c pipelines in an HCAM. n_c=3.

intensity of each queue is defined as

$$\rho' = (1+f)\lambda/n_c \times T_s, \tag{9.1}$$

while a successful lookup' traffic intensity to a queue is $\rho=\lambda/n_c\times T_s$.

Let $\{Q_i\}_{i=1}^{\infty}$ be the stochastic process of the number of the IP addresses in the queue at the time of the i-th arrival. Then, a queue's loss probability which can be derived from [36, 73, 74] is the following:

$$P_L = P(Q = n_q) = \{1 + (\rho'\text{-}1)\alpha_{n_q}(\rho')\}/\{1+\rho'\alpha_{n_q}(\rho')\}, \tag{9.2}$$

where

$$\alpha_{n_q}(\rho') = \sum_{i+m=n_q-2} \frac{e^{\rho'(i+1)}(-1)^m \rho'^m (i+1)^m}{m!}, \; n_q \geq 2. \tag{9.3}$$

Now, since processing ULs is considered as wasting the lookup time in a CAM block, a throughput of our concern, *Goodput*, is defined as

$$\textit{Goodput} = \rho(1 - P_L), \tag{9.4}$$

because under the probability that a queue is not full, 1-P_L, a CAM block processes successful lookups in traffic intensity ρ, not ρ'. Also, the overall goodput with n_c CAM blocks is calculated by multiplying Eq. (9.4) with n_c.

In addition to the theoretical analysis, a series of experiments is also made to measure an HCAM-based scheme's throughput performance. Due to a difficulty in

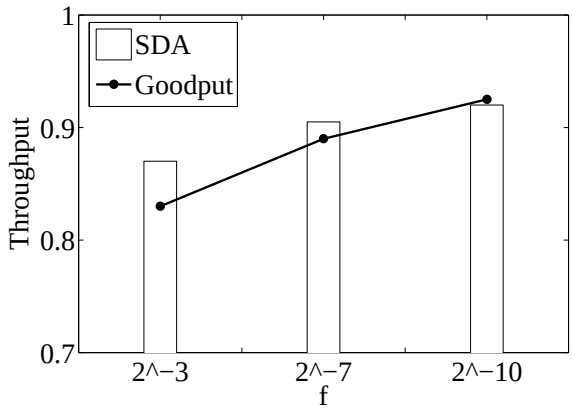
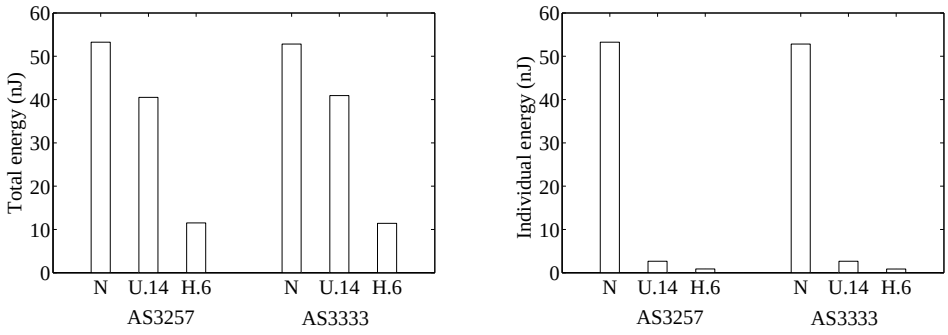


Fig. 49. Goodput vs. measured throughput of a CAM block in an SDA trace. ρ=0.95.

getting a pair of a BGP table and its corresponding IP trace, an SDA trace from [58] is utilized to extract prefixes from packet streams by considering a unique destination IP as a prefix. In experiment runs, it is assumed that a distributor disseminates lookup requests fast enough that queues of successful or unsuccessful lookups are full and a CAM block processes a lookup from a queue in one clock. Also, four CAM blocks are used, each with a queue size n_q=5. Fig. 49 shows *Goodput* defined by Eq. (9.4) and a CAM block's throughput defined by the number of SLs in a queue over the total clocks to process packets. The figure shows that the smaller an f-positive f is, the higher throughput is achieved. Also, the total *Goodput* of 4 CAM blocks is marked as 3.7.

2. Power

By using the TCAM and CAM modeling tools [35, 70], we measured the total energy in one clock and individual energy for a single lookup in a TCAM or CAM block in three approaches: a naive TCAM (NTCAM), a UTCAM, and an HCAM. Such energy consumptions are shown in Fig. 50 for AS 3257 and AS 3333 routing tables. An NTCAM in the figure provides only one lookup with the entire prefixes while



(a) Total energy per clock cycle (b) Energy in a TCAM- or CAM-block

Fig. 50. a) Total energy consumption in one clock for an NTCAM, a UTCAM, and an HCAM. Symbols 'N', 'U.14', and 'H.6' denote NTCAM with a block of whole prefixes, UTCAM with 16 blocks of 14K prefixes, and HCAM with 16 blocks of 6K prefixes, respectively. $.13\mu m$ process technology is used. b) The energy consumptions for a single lookup operation in a block for three schemes.

a UTCAM and an HCAM can provide multiple lookups with 16 TCAM or CAM blocks, respectively. To make the number of blocks in UTCAM and HCAM even, UTCAM and HCAM block sizes are set to 14K and 6K entries, respectively. In this configuration, the same throughput can be achieved. On average, an HCAM saves 3.6 and 4.6 times total energies compared to a UTCAM and an NTCAM, respectively, even if an HCAM and a UTCAM have the same number of blocks. The power usage can be easily calculated by dividing a consumed energy by a lookup access time which depends on the process technology of memory chip fabrication.

CHAPTER X

SUMMARY

A. Conclusion

It was discussed that the existing hash schemes for packet processing, like an FHT, a BFHT, and Peacock hashing suffer from key duplicates, a complicated update, and setup failure and they are not scalable in terms of scalability and speed. To overcome these problems, one packet classifier and three hashing schemes are proposed: an MPC, an MBHT, an HIHT, and an FPHT, for large-scale and high-speed packet processing.

An MPC is proposed by reconfiguring BFs into small-sized BFs and large-sized BFs in a multi-tiered way without memory overhead, compared to a PPC. By *Linear Property* 1 in Sec. III.A, it is shown that how an MPC is built with the same memory capacity as that of a PPC in Sec. A. It is observed that the number of fabricated read ports in BFs' memory as well as the MPC area cost are reduced with the same memory. In simulation with NLANR's IP traces for flow identification, an MPC shows higher efficiencies in all traces than a PPC, at most 2.0 and 4.2 times of throughput and power, respectively.

Also, an MBHT of a novel hash architecture is proposed, generating indexes to a key table with a set of MBFs in base-b number system. The MBFs work *in pipelining* in `query` so that a subset of them in row i determine A_i, which is a part of a whole index address $A^b{=}a_0 \cdots a_{r\text{-}1}$ of base-b number system. From Lemmas 1 and 2, it is realized that adapting a larger base number system saves significant on-chip memory against an LHT and an FHT, and showed that base-2^3 is the starting point of better memory efficiency for an MBHT as shown in Fig. 23. A novel hash architecture is

proposed, generating indexes to a key table with a set of MBFs in base-b number system. The MBFs work *in pipelining* in `query` so that a subset of them in row i determine A_i, which is a part of a whole index address $A^b{=}a_0 \cdots a_{r\text{-}1}$ of base-b number system. From Lemmas 1 and 2, it is realized that adapting a larger base number system saves significant on-chip memory against an LHT and an FHT, and showed that base-2^3 is the starting point of better memory efficiency for an MBHT as shown in Fig. 23.

Thirdly, a novel hash architecture with two HITs is proposed, generating indexes to a key table with a set of BFs. The BFs in two HITs work systematically, or in *pipeline* and *hierarchical* fashion to minimize the number of indexes Only one off-chip memory access is required in addition to achieving efficiency in on-chip access. For `insert`, an i-path is assigned to a key and one BF on each layer is involved in encoding the key in one of HITs. For `query`, one on-chip memory module for each layer is probed for candidate BFs having their base indexes on the memory derived from Eq. (3.2). After the last probing in layer s-1, the returned indexes are used for *perfect* match in a on-chip key table, so that a deterministic $\Theta(1)$ lookup is guaranteed. For `delete`, by rotating two HITs, seamless update of keys is provided without counters costing four times the memory, so that only half of the memory is used.

As the last hash scheme, an FPHT, by using CBFs in a binary search for a key's fingerprint and utilizing an keys' FF in a high-precision query for a high-speed router, a proposed FPHT produces an i-path and no f-path to a key table with a high probability and memory efficiency. In throughput comparison against Peacock hashing, it was shown that while Peacock hashing suffers from a lower throughput in a UL, an FPHT throughput is proportional to the number of threads regardless of lookup kinds.

In hash-based IP lookup architectures with an HIHT or an FPHT, it is observed that an FPHT-based IP lookup saves 51 times power and 1.8 times memory compared to TCAM and trie-based IP lookup, respectively.

In addition to these power- and memory-efficient hash schemes, a hybrid CAM is also proposed where a high performance lookup can be achieved by parallel lookups among CAM and SRAM blocks. In an HCAM, a prefix is broken into a collapsed prefix in CAM and a stride in SRAM. The prefix collapse reduces the number of prefixes that results in reduced memory usage by a factor of 2.8. High throughput is achieved by storing the collapsed prefixes in partitioned CAMs that perform multiple IP lookups per cycle. A stride tree bitmap with a matched collapsed prefix completes the longest prefix match.

B. Future Works

Since hashing provides only a singleton match for a one-dimension key, any hash-based packet processing application needs a lookup-key transformation for its application domain. For instance, since IP lookup needs a prefix match, the hash-based IP lookup needs prefix expansion or collapse as discussed in Sec. C. Although this dissertation proposed one packet classifier and three hashing schemes proven with memory and power efficiencies, these belong to a one-dimension singleton match. As future work, a hashing scheme for a two-dimension key in packet classification will be considered.

Power- and memory-efficient hash mechanisms have been shown in this dissertation. However, the reviewing on the importance of a throughput metric in a high-speed router implementation encourages us to consider mapping a m-trie, which was developed for IP lookup, onto multiple pipelines. Since the principle of a pipeline is to give an one-clock throughput, multiple m-trie-mapped pipelines can give multi-folds

throughput for IP lookup or packet classification.

REFERENCES

[1] K. G. Coffman and A. M. Odlyzko, *Internet Growth: Is There a "Moore's Law" for Data Traffic?, Handbook of Massive Data Sets*, Kluwer, New York, 2002.

[2] M. Gray, (1996), [Online]. Available: http://www.mit.edu/people/mkgray/net/internet-growth-summary.html.

[3] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," in *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, 2003, p. 120.

[4] V.C. Ravikumar and R.N. Mahapatra, "TCAM Architecture for IP Lookup Using Prefix Properties," *MICRO, IEEE*, vol. 24, no. 2, pp. 60–69, 2004.

[5] V. C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, "EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 521–533, 2005.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," in *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005, pp. 193–204.

[7] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, 1999.

[8] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines," *IEEE/ACM Trans. Netw.*, vol. 13, pp. 690–703, 2005.

[9] S. Sahni and K.S. Kim, "Efficient Construction of Multibit Tries for IP Lookup," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 650–662, 2003.

[10] S. Sahni and K.S. Kim, "Efficient Construction of Pipelined Multibit-trie Router-Tables," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 32–43, 2007.

[11] A.C. Snoeren, "Hash-based IP Traceback," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001, pp. 3–14.

[12] S. Dharmapurikar, P. Krishnamurthy and D.E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003, pp. 201–212.

[13] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," in *MICRO 37: Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, New York, 2004, pp. 52–61.

[14] F. Chang, W-C. Feng, and K. Li, "Approximate Caches for Packet Classification," in *INFOCOM 2004. 23rd Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Hong Kong, Chnia, 2004, vol. 4, pp. 2196–2207.

[15] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," in *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005, pp. 181–192.

[16] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A Storage-Efficient, Collision-free Hash-based Network Processing Architecture," in *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, 2006, pp. 203–215.

[17] D. Guo, J. Wu, G. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM 2006. 25th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 2006, pp. 1233–1242.

[18] J. Moscola D.V. Schuehler and J.W. Lockwood, "Architecture for a Hardware-based TCP/IP Content Scanning System," in *Hot Interconnect: IEEE Symposium on High Performance Interconnects*, 2003.

[19] S. Dharmapurikar, H. Song, J. Turner and J. Lockwood, "Fast Packet Classification Using Bloom Filters," in *ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, San Jose, 2006, pp. 61–70.

[20] H. Song, J. Turner, and S. Dharmapurikar, "Packet Classification Using Coarse-grained Tuple Spaces," in *ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, San Jose, 2006, pp. 41–50.

[21] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," in *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, Italy, 2006, pp. 315–326.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Seattle, 1997, pp. 25–36.

[23] F. Baboescu and G. Varghese, "Scalable Packet Classification," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 2–14, 2005.

[24] T. V. Lakshman and D. Stiliadis, "High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," in *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Vancouver, Canada, 1998, pp. 203–214.

[25] A.Z. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," in *INFOCOM 2001. 20th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 2001, pp. 1454–1463.

[26] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *ESA'06: Proceedings of the 14th Conference on Annual European Symposium*, 2006, pp. 684–695.

[27] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," in *SODA '04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004, pp. 30–39.

[28] S. Kumar, J. Turner, and P. Crowley, "Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking," in *INFOCOM 2008. 27th*

Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, pp. 101 – 105.

[29] A. Kirsch and M. Mitzenmacher, "Simple Summaries for Hashing with Choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, 2008.

[30] H. Yu and R. Mahapatra, "A Throughput-efficient Packet Classifier with n Bloom Filters," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, New Orleans, 2008, pp. 1 – 5.

[31] H. Yu and R. Mahapatra, "A Memory-efficient Hashing by Multi-predicate Bloom Filters for Packet Classification," in *INFOCOM 2008. 27th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Phoenix, 2008, pp. 1795 – 1803.

[32] H. Yu and R. Mahapatra, "A Space- and Time-efficient Hash Table Hierarchically Indexed by Bloom Filters," in *IPDPS 2008. IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1 – 12.

[33] H. Yu and R. Mahapatra, "A Pipelined Indexing Hash Table Using Bloom and Fingerprint Filters for IP Lookup," in *SIGCOMM 2008*, pp. 463 – 464.

[34] The Linley Group, A Guide to Search Engines and Networking Memory, (2006, Nov.), [Online]. Available: http://www.linleygroup.com/pdf/NMv4.pdf.

[35] CACTI, (2001, Feb.), [Online]. Available: http://www.hpl.hp.co.uk/personal/Norman_Jouppi/cacti5.html.

[36] K. Zheng, C. Hu, H. Lu and B. Liu, "An Ultra High Throughput and Power Efficient TCAM-based IP Lookup Engine," in *INFOCOM 2004. Proceedings*

IEEE 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, 2004, pp. 7–11.

[37] J. Akhbarizadeh, M.M. Nourani, R. Panigrahy, and S. Sharma, "A TCAM-Based Parallel Architecture for High-speed Packet Forwarding," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 58–72, 2007.

[38] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMs: An SRAM-based Parallel Multi-pipeline Architecture for Terabit IP Lookup," in *INFOCOM '08. Proceedings of IEEE 27th Annual Joint Conference of the IEEE Computer and Communications Societies*.

[39] L. Fan, P. Cao, J. Almeida, and A.Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[40] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," in *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, New York, 1998, pp. 191–202.

[41] M. Nourani and M. Faezipour, "A Single-Cycle Multi-Match Packet Classification Engine Using TCAMs," in *HOTI '06: Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, Washington, DC, 2006, pp. 73–80.

[42] M. Singhal, J. Xu and J. Degroat, "A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds," in *INFOCOM 2000. Proceedings IEEE of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2000, pp. 1445–1454.

[43] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," in *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002, pp. 47–60.

[44] A. Kumar, J. Xu and E. W. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," in *INFOCOM 2005. Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2005, pp. 13–17.

[45] D. Sy and L. Bao, "CAPTRA: Coordinated Packet Traceback," in *IPSN '06: Proceedings of the Fifth International Conference on Information Processing in Sensor Networks*, 2006, pp. 152–159.

[46] S. Cohen and Y. Matias, "Spectral Bloom Filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 241–252.

[47] F. Zane, G. Narlikar, and A. Basu, "CoolCAM: Power-efficient TCAMs for Forwarding Engines," in *INFOCOM 2003. Proceedings of IEEE the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2003, pp. 42 – 52.

[48] W. Jiang and V. Prasanna, "Parallel IP Lookup Multiple SRAM-based Pipelines," in *IPDPS '08. 22nd IEEE International Parallel and Distributed Processing Symposium*, 2008, pp. 1–14.

[49] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2002.

[50] A. Pagh, R. Pagh, and S. S. Rao, "An Optimal Bloom Filter Replacement," in *SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, 2005, pp. 823–829.

[51] BGP Routing Tables Analysis Report, (2008), [Online]. Available:http://bgp.potaroo.net.

[52] University of Oregon Route Views Project, (2005, Jan.), [Online]. Available: http://www.routeviews.org/.

[53] I. Kaya and T. Kocak, "Energy-efficient Pipelined Bloom Filters for Network Intrusion Detection," in *IEEE International Conference on Communications*, 2006, pp. 2382 – 2387.

[54] F. Nemati, H.-J. Cho, S. Robins, R. Gupta, M. Tarabbia, K.J. Yang, D. Hayes, and V. Gopalakrishnan, "Fully Planar $0.562\mu m^2$ T-RAM Cell in a 130nm SOI CMOS Logic Technology for High-density High-performance SRAMs," in *IEEE International Electron Devices Meeting '04*, 2004, pp. 273–276.

[55] B. Dipert, "Special Purpose SRAM Smooth the Ride," *Electronics Design, Strategy, News*, 1999, pp. 9–13.

[56] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1990.

[57] E. Safi, A. Moshovos, and A. Veneris, "L-CBF: a Low-Power, Fast Counting Bloom Filter Architecture," in *ISLPED '06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, Tegernsee, Germany, 2006, pp. 250–255.

[58] Passive Measurement and Analysis Project, National Laboratory for Applied Network Research (NLANR), (2006, July), [Online]. Available:http://pma.nlanr.net/traces/traces.

[59] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "A Performance Study of Hashing functions for Hardware Applications," in *Proceedings of Int. Conf. on Computing and Information*, 1994, pp. 1621–1636.

[60] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[61] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with a Power Evaluation Framework," *IEEE Micro*, vol. 24, no. 5, pp. 34–44, 2004.

[62] A. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha, "Design, Implementation, and Performance of a Content-based Switch," in *INFOCOM 2000. Proceedings of IEEE the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2000, pp. 1117 – 1126.

[63] C. Kachris and S. Vassiliadis, "Design of a Web Switch in a Reconfigurable Platform," in *ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, San Jose, 2006, pp. 31–40.

[64] Z. G. Prodanoff and K. J. Christensen, "Managing Routing Rables for URL Routers in Content Distribution Networks," *Int. J. Netw. Manag.*, vol. 14, no. 3, pp. 177–192, 2004.

[65] Monthly Log Files 2000, Computer Science Division, University of California, Berkeley.

[66] NLANR Sanitized Cache Access Logs, (2006), [Online]. Available: ftp://ircache.nlanr.net/Traces/.

[67] Sanitized Log Files from Canada's Coast to Coast Broadband Research Network (CA*netII), (2000), [Online]. Available: ftp://ircache. nlanr.net/ Traces.

[68] J. García, J. Corbal, L. Cerdà and M. Valero, "Design and Implementation of High-performance Memory Systems for Future Packet Buffers," in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, p. 373.

[69] M. Mitzenmacher and S. Vadhan, "Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream," in *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, 2008.

[70] B. Agrawal and T. Sherwood, "Modeling TCAM Power for Next Generation Network Devices," in *ISPASS '06: IEEE International Symposium on Performance Analysis of Systems and Software*, 2006.

[71] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[72] RIPE Network Coordination Centre, (2006), [Online]. Available:http://www.ripe.net/.

[73] K.S. Trivedi, *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[74] S. Alouf, P. Nain, and D. Towsley, "Inferring Network Characteristics via Moment-based Estimators," in *INFOCOM 2001. Proceedings of IEEE the 20th Annual Joint Conference of Computer and Communications Societies*, 2001, pp. 1045 – 1054.

VITA

Heeyeol Yu was born in Kimje, Korea. After completing his schooling at Pohang Jechul High School, he went on to obtain his Bachelor of Science in Computer Science from Korea Advanced Institute of Science and Technology, Taejon, Korea in February 1994. He graduated with his Master of Science in Computer Science from the University of California, Los Angeles in December 2003.

Contact address:
Department of Computer Science and Engineering
Texas A&M University
TAMU 3112
College Station, TX 77843-3112

The typist for this dissertation was Heeyeol Yu.