An Efficient Hardware-based Multi-hash Scheme for High Speed IP Lookup

Socrates Demetriades, Michel Hanna, Sangyeun Cho and Rami Melhem
Department of Computer Science
University of Pittsburgh
{socrates,mhanna,cho,melhem}@cs.pitt.edu

Abstract

The increasingly more stringent performance and power requirements of Internet routers call for scalable IP lookup strategies that go beyond the currently viable TCAM- and trie-based solutions. This paper describes a new hash-based IP lookup scheme that is both storage efficient and high-performance. In order to achieve high storage efficiency, we take a multi-hashing approach and employ an advanced hashing technique that effectively resolves hashing collisions by dynamically migrating IP prefixes that are already in the lookup table as new prefixes are inserted. To obtain high lookup throughput, the multiple hash tables are accessed in parallel (using different hash functions) or in a pipelined manner. We evaluate the proposed scheme using up-to-date core routing tables and discuss how its key design parameters can be determined. When compared with state-of-the-art TCAM designs, our scheme reduces area and power requirements by 60% and 80% respectively, while achieving competitive lookup rates. We expect that the proposed scheme scales well with the anticipated routing table sizes and technologies in the future.

1 Introduction

The explosion of Internet traffic has pushed the technology to seek for higher-throughput links and routers that can meet the increasing demands of packet processing rates. While current optical network technology already provides link rates in excess of 40GBps, packet forwarding and filtering engines start to become a bottleneck for routers [1]. To catch up with the rapid increase of link rates, IP lookup in high speed routers must satisfy higher lookup rates and scale with the increasing routing table sizes. Processing rates of more than 125 million lookups per second utilizing a million forwarding entries are of practical interest [1].

IP lookup is a per packet process performed by a router. Given an IP address, it aims at finding the *longest prefix match* (LPM) among all the prefixes in a routing table. Current techniques for IP lookup are divided into three categories: *Ternary Content Addressable Memory (TCAM)*, *trie-based schemes* and *hash-based schemes*. TCAMs [2–4] have been the most favorable choice for high speed routers thanks to their fast and constant lookup time. As table sizes increase however, TCAM's high cost and power consumption become problematic. While various techniques try to alleviate the scalability problems, switching to alternative solutions becomes more appealing.

Trie-based schemes use a tree-like structure to store prefixes and a lookup is performed by traversing the tree based on a portion of bits in a target IP address until an LPM is found [5–7]. Unfortunately, the potentially large and unbalanced depth of a tree structure creates multi-cycle lookup latencies and low worst-case throughput. Although many researchers have identified different solution methods to address these issues, the fundamental tie of the performance and scalability of those schemes with the IP address length remains a barrier to meeting the future demands.

Hash-based schemes use hash tables to store prefixes [8–12]. Unlike trees, hash tables are flat data structures that have the desirable property of key length independent $O(1)$ latencies and are easy to implement in hardware. Moreover, hash–based schemes can potentially scale well with the increase in table sizes, enabling a promising candidate for IP lookup in the future. However, hash based schemes have three key problems that must be addressed to be fully adopted for use in a practical high-performance IP lookup engine. First, hashing is inherently subject to collisions and requires resolution techniques that may result in unpredictable lookup performance. The second problem arises when hash keys (IP prefixes) include "don't care" bits and these bits are used as hash index bits. One would have to expand all the affected IP prefixes to eliminate "don't care" bits or simply give up using hash functions that depend on one of those bits. Lastly, constructing an effective hash-based solution for the LPM functionality is tricky. For instance, when each hash table entry holds multiple prefixes, further steps are needed to determine the LPM when a lookup results in multiple matches.

In this work, we propose a new hash-based IP lookup scheme that can achieve high lookup throughput, high space utilization and low power consumption. Our scheme is essentially a multiple hash table (MHT) scheme that employs multiple hash functions and a simple modified version of the Cuckoo hashing algorithm [13]. It can resolve collisions effectively by migrating items that are already in the hash table during an insertion operation. Our empirical results us-

ing up-to-date core routing tables show that our technique achieves a space utilization up to 90% by using less than 5% of extra victim space to support a collision-free hashing.

Unlike previous hash-based IP lookup schemes, we do not classify prefixes based on their lengths. Instead, we treat them "equally" as they are inserted into the hash tables by taking a set of fixed bit locations as hash bits independent of their prefix lengths. We propose a technique called *controlled wildcard resolution (CWR)* to handle the wildcard bits that coincide with the hash bits effectively. We demonstrate through experiments that CWR provides more flexibility and control on the hashing methods and better hashing results than existing techniques.

In summary, our proposed scheme can achieve the same high lookup performance as the currently highest-performance TCAM solution and enables 6× space reduction and 8× power savings. Compared with the most area-efficient hash-based scheme known [9], our technique has three times the lookup throughput and compares competitively in terms of area efficiency.

The rest of this paper is organized as follows. Section 2 presents a summary of related work. We present the proposed approach and techniques in Section 3, followed by an experimental evaluation in Section 4. Conclusions are presented in Section 5.

2. Related Work

In this section, we focus only on hash-based techniques that are directly related to our work. Many other previous works are mentioned throughout the paper.

The first hash-based LPM proposal is the *binary search on prefix lengths* [10]. Since hash functions can't operate on wildcard bits, prefixes are grouped based on their lengths and stored in separate hash tables (or possibly in a single hash table) using "valid" prefix bits as hash index. A lookup then visits the tables in search for the longest matching prefix sequentially or using a binary search method. This scheme is intuitive and does not involve prefix expansion. However its drawbacks include potentially long unpredictable lookup latency, area inefficiency caused by load imbalance among the tables, and lack of scalability to IPv6.

A way to "virtually" reduce the number of unique prefix lengths and consequently the number of tables is to apply *controlled prefix expansion (CPE)* [7]. CPE converts a prefix of length L into a number of prefixes of longer length $(L + l), l \geq 1$ by expanding l of its wildcard bits into their 2^l possible derivatives. Therefore, an IP prefix table having many distinct prefix lengths can be transformed into a table with more IP prefixes that have less distinct prefix lengths. The inflation of the number of prefixes however affects the storage space adversely and the fewer the unique prefix lengths desired, the larger is the inflation. Typically, an optimistic CPE will expand a core routing table 2× when targeting three classes of distinct prefix lengths and 4× when targeting two of them. In addition, CPE will produce identical keys in the hash tables that come from different original prefixes. Accordingly, an IP lookup may result in multiple matches and longest prefix resolution is required. For that, extra information (original prefix length) is needed for each stored prefix, causing even larger space overheads.

Kaxiras and Keramidas [9, 14] proposed a space efficient multi-hashing scheme called *IPStash*. It achieves excellent space utilization and fewer expansions by classifying prefixes according to their lengths (e.g., ≤ 16, $17-21$, ≥ 21) and using different hash indices to hash the prefixes of each class to a single multi-hash table. As a result, wildcard bits are excluded from hashing and at the same time good hashing results are achieved. The reported area efficiency of IPStash beats all previously proposed hash-based schemes. However, its worst-case lookup time is sensitive to the number of classes, as the LPM searching requires iterative table accesses until a match is found.

Dharmapurikar *et al.* [12] propose to use Bloom filters to reduce the number of probes to hash tables. The idea is that by programming each distinct length group of prefixes in a separate on-chip Bloom filter and checking if the prefix is a member of a specific group or not, one can economically determine the exact off-chip hash table to access *a priori*. Similarly, Song *et al.* [11] use Extendend Bloom filters to reduce the memory accesses within a hash table. While these techniques can significantly improve the average lookup latency and power consumption, they fail to guarantee a constant lookup latency due to possible false positive responses from the Bloom filters.

Most previous proposals are characterized by techniques that divide prefixes into classes, based on their length. In highly efficient schemes, however, the number of classes affects the achievable throughput adversely [9]. Unlike previous studies, our work aims to improve lookup performance by avoiding prefix classes. At the same time, we opt for high area efficiency by employing powerful hashing techniques.

3. Our Approach

3.1. Basic architecture

Our scheme is based on a multi-level hash table (MHT) technique [15] that uses d equal-size sub-tables, $T_1, \ldots, T_d$. They can be accessed in a parallel or pipelined manner by using independent indexing or hash functions, $h_1, \ldots, h_d$. To support fast and flexible hash key calculation and matching, each sub-table T_i is implemented using an efficient row-oriented associative memory architecture [16].

Figure 1 depicts such a memory architecture. Each sub-table T_i is a regular memory array (SRAM or DRAM) having 2^R rows that can be indexed independently by an R-bit index generated by a hash function. Each row or *bucket* is usually comprised of multiple search keys, which are fetched and accessed simultaneously on a lookup. We conveniently call the total number of key entries in the whole architecture "available capacity" or "available space."

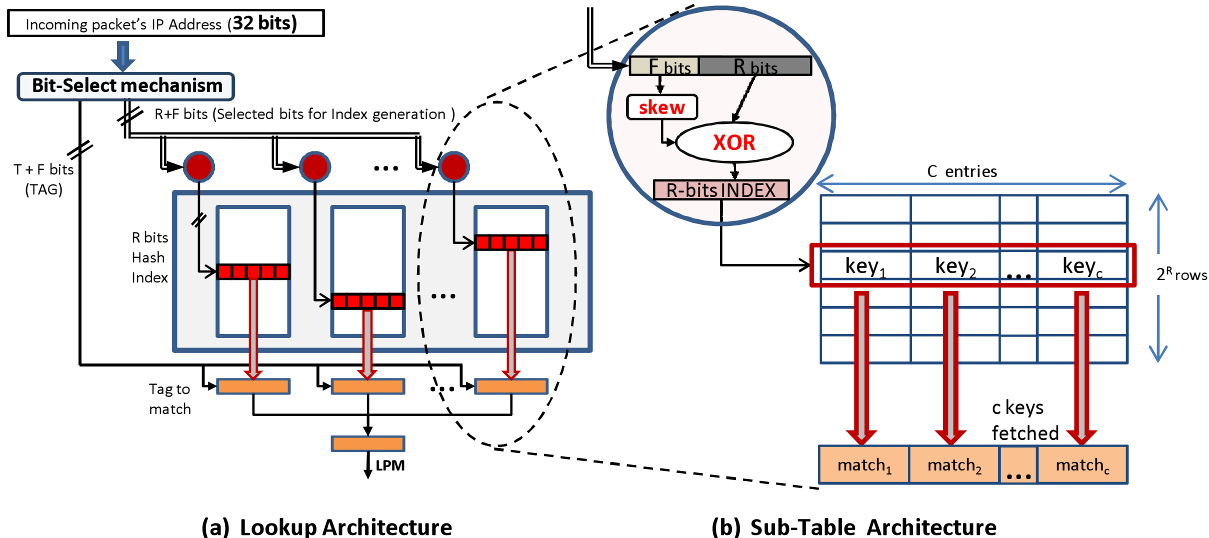

Figure 1. The Proposed IP Lookup Architecture

Each table is accessed using a hardware based index generator and a number of match processors compare the fetched keys with the search key in parallel, resulting in constant-time matching. The "Bit-Select" component is a reconfigurable mechanism that decides which bits participate in the index generation ($(R + F)$ bits in Figure 1). When inserting a prefix, wildcard bits may appear in the selected bits for indexing and hence, the Bit-Select mechanism needs to resolve this by prefix expansion and performing a set of insertions. Section 3.3 discusses the Bit-Select mechanism and hash functions further.

For a lookup in a MHT scheme, d independent hash indices are created to search the d hash tables. In each table, a multiple entry bucket is fetched into match processors, which then compare the fetched keys (IP prefixes) with the search key (destination IP address). If multiple prefixes match the search key, the match processors must determine the LPM among all the matching keys. The operation requires that the length of the stored prefixes be known and hence this information is kept with each prefix in the table. Assuming that tables are accessed in parallel, a single lookup will be satisfied after one memory access cycle followed by the matching process. The two operations are easily pipelined and a throughput of one IP lookup per cycle is obtained.

3.2. Hashing IP prefixes using multiple hash functions

In general, hashing with multiple hash functions is known to perform better than hashing with a single hash function since each item has multiple possible choices to be hashed in the tables. Figure 2(a) and (b) show an example of how prefixes of an IP table are distributed in a single- or multi-hashing scenario respectively. The table has 185K prefixes from AS1103, found in [20]. The x-axis represents the table's buckets (total of 4096) and the y-axis depicts the load

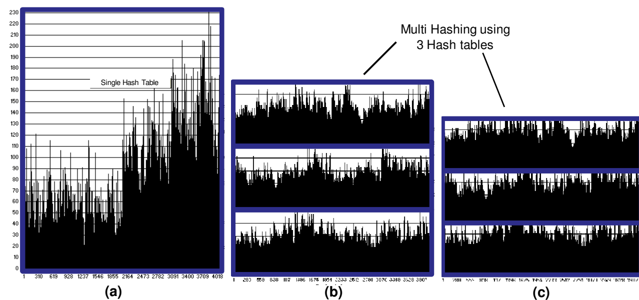

Figure 2. Space Utilization of (a) Single Hashing (b) MHT (c) MHT with migrations allowed.

of the buckets. Hence, the whole plot area represents the total storage capacity needed and the dark area its utilization. When single hashing is used, the max bucket load is 234 items. When multi-hashing is applied, the bucket size is reduced to $3\times\ 49 = 147$ without any collision, therefore improving the space utilization. Note that when an item is inserted in the multi-hashing scheme, the least loaded bucket is chosen among the three possible options.

All hash functions in our scheme use the same bit positions in a prefix or a destination address, and they generate a skewed hash index by using an XOR-folding technique as depicted in Figure 1(b). XOR-folding [17] has been widely used in hardware-base hash index generation mainly due to its simplicity. It folds an $(F + R)$-bit key into an R-bit hash index through a simple process of XORing every F bit with the R bits into a final hash bit. Skewness can be applied by a simple rearrangement of the F bits that are XORed. Using such a fast hash function allows us to hide the index generation latency by pipelining with the actual memory access.

While employing multiple hash functions improves over-

all hashing results, Figure 2(b) shows that the distribution of hashed IP prefixes is far from uniform. Even with three hash tables, much of the space remains unutilized. An attempt to improve hashing with more powerful or adaptive hash functions may slow down the lookup speed. In what follows, we present a technique that improves the space usage and yet, does not affect the critical lookup path.

Improving hashing with rehashing during insertion. We improve the multi-hash scheme by applying a rehashing technique that allows the migration of an existing item in the hash table to a new location during an insertion operation. The goal of this rehashing is to distribute the hashed items more evenly for higher space utilization (Figure 2(c)).

Our rehashing technique is similar in spirit to the Cuckoo hashing algorithm [13], which resolves collisions by rearranging keys into other possible "nests" to make space for the key being inserted. The original Cuckoo hashing is however an iterative algorithm and its complexity poses a serious hurdle for an efficient hardware implementation of a high-speed IP lookup engine. On the other hand, our proposed technique considers only a single iteration of the Cuckoo algorithm.

The insertion algorithm works as follows. Assume an insertion policy that keeps the load among the d sub-tables balanced by placing each new item into the sub-table with the least loaded bucket. In the case of a tie, the item is inserted in the leftmost table according to the d-left scheme [18]. Then, at some point, a collision may appear if an item x cannot be inserted into tables $T_i,\ i = 1, \ldots, d$ because buckets $T_i[h_i(x)],\ i = 1, \ldots, d$ are full. In such case, a single iteration of the Cuckoo algorithm will try to resolve the collision by checking whether an item y at some bucket $T_i[h_i(x)],\ 1 \leq i \leq d$, can migrate to some other table $T_j \neq T_i$. This requires that a bucket $T_j[h_j(y)]$ has an empty entry to accommodate the item y. If such an item y is found, then it can be moved and x will take its place. Otherwise, the colliding insertion of x cannot be resolved–a situation that is called *unresolved collision* or *crisis*.

According to this algorithm, prefixes may begin to migrate only after an inserted prefix finds all its possible "nests" full–a situation that will occur more frequently as tables become more occupied. When an insertion activates a rehashing process, its insertion time will vary depending on how many items are subject to migration; the worst-case scenario occurs when inserting an item fails even after searching for all possible $c \times d$ moves, where c is the number of items per bucket.

3.3. Selecting hash bits from keys

A hash function uses bits extracted from predefined locations in a prefix, independently of its length. We call the actual hardware logic that extracts those bits from the hash key *Bit-Select mechanism*. We discuss in this section how this mechanism can be configured to select a set of bits that are generally desirable to participate in the index generation. We call such a set of bits a *Bit-Select configuration*. Wrong decisions on how many and which bits should be used for the index generation may result in inefficient space usage.

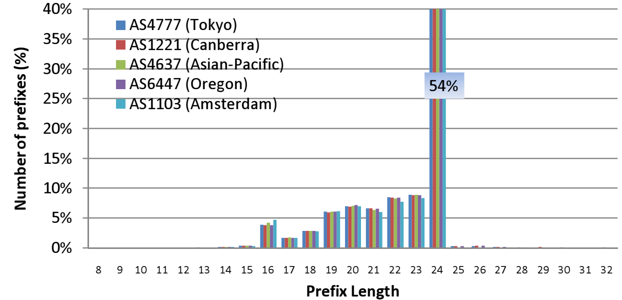

Figure 3. Prefix Length Distribution

Researchers have made two interesting observations from different IP routing tables in the Internet's core, which have been found to be constant over the years. The first key observation is the relatively fixed distribution of the prefixes according to their lengths that is depicted in Figure 3. Most prefixes ($\sim$98%) have lenghts between 16 and 24 bits and 24-bit prefixes comprise about 54% of the table. Prefixes longer than 24 bits are very few ($<$1%) and there are no prefixes less than 8 bits. The second key observation concerns each bit's apparent randomness or entropy–how unbiased it is towards one or zero in a routing table. As a study in [9] pointed out, regardless of prefix length, high entropy bits start from the 6th bit and reach the prefixes' maximum length.

According to the first observation, only the 8 leftmost bits of the prefix should be used for hashing since "don't care" bits naturally cannot participate in the hash index construction. However, such a restriction not only creates unbalanced hashing results (because of the bits' low entropy), but also sets a limit on the number of bits that can be selected for indexing, thereby narrowing the design and configuration space. A straightforward method of including wildcard bits in hashing is to "expand" the affected prefixes using a systematic method like CPE [7] such that the wildcard bits are eliminated. Once prefixes are expanded, the new expanded prefix database can be hashed. Obviously, the more wildcard bits selected, the larger the inflation of the IP routing table and hence the storage overhead. Therefore, it is generally desirable to restrict the selection to leftmost bits.

On the other hand, hash functions that use high entropy bits are expected to demonstrate probabilistically random indices and hence, balanced hashing result. Obviously, such knowledge is useful when deciding which bits to use. Based on the second observation, seeking high entropy bits leads to the rightmost bits of the prefixes, which conflicts with the original attempt of selecting leftmost bits.

Consequently, choosing a Bit-Select configuration requires a fine balance between efficient hashing and storage overhead. The flexibility of the Bit-Select logic in selecting any combination of bits allows us to examine the trade-off in more detail. For that, we need first a mechansim to support wildcard bits effectively.

Prefix	Method	Prefixes Insertion	Indexes Generated	Final inserted items
101***	CPE	10100*	110	0** @ 110
		10101*	111	0** @ 111
		10110*	110	~~0** @ 110~~
		10111*	111	~~0** @ 111~~
101***	CWR	11*	110	0** @ 110
			111	0** @ 111

Figure 4. CPE and CWR example

Dealing with wildcard bits. Traditionally, the CPE technique expands the prefixes up to a certain desired length L and then all the L leftmost bits can be used for the hash index construction. If we consider using only a subset of the expanded bits in hashing, some of the expanded prefixes become redundant and do not need to be hashed. On one hand, such a case is desirable since it may reduce the storage overhead without affecting the hashing performance much. On the other hand, CPE cannot exclude such redundant prefixes in the process of expansion and hence, they must be detected and omitted during the prefix insertion process.

In this work, we propose *controlled wildcard resolution (CWR)* to handle wildcard bits in hashing, which has several desirable properties such as real-time wildcard bit resolution at insertion time and high flexibility. Unlike CPE, CWR does not initially expand the prefixes before performing insertions. Instead, it performs expansion at insertion (index generation) time if there are wildcard bits that actually participate in hashing. According to this technique, the hashing bits can be selected to be any of the 32 (or 128 for IPv6) bits of the prefix, a property that allows us to do sensitivity analysis on several combinations of bits for a Bit-Select configuration.

Figure 4 shows an example of how CPE and CWR work differently. The example prefix has three bits, followed by three wildcard bits. The hash function uses the three bits that are underlined. According to CWR, the expansion rate is two since only a single wildcard bit appears in the index generator. In the case of CPE, the single IP prefix would have to be pre-expanded at least up to the LSB that is used by the hash indexing–in our example, up to the 5th bit–and then all the four expanded IP prefixes have to pass through an insertion process, one by one (although they might not actually be inserted in the table). Although both techniques will eventually result in the same IP routing table inflation, CWR spends less time for an insertion, it is more flexible, and it is amenable to simple and fast hardware implementation.

Bit-Select sensitivity analysis. Having a flexible Bit-Select mechanism and the CWR technique that can support it, we can explore how different Bit-Select configurations affect the hashing results and the storage requirements. Before we continue, we define as *extra provided capacity* (C_p) the difference between the total capacity of a hash scheme and the nominal (unexpanded) size of the routing table that is to be hashed. Our sensitivity analysis determines which Bit-Select configuration should be used, given a certain storage capacity limit.

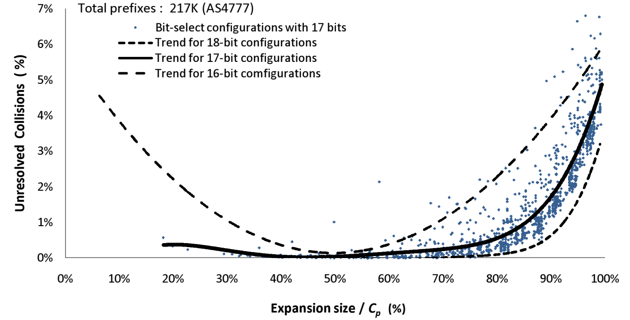

Figure 5. Effectiveness of Different Bit-Select Configurations

Figure 5 shows how different Bit-Select configurations behave when we hash a routing table of nominal capacity N into our 3-level multi-hash scheme with the total capacity of $2 \times N$ (i.e., $C_p = N$). For this example, we use table AS4777 from [20] for which $N = 217K$. We examine all possible configurations that result from selecting 17 bits out of the total 32 bits of each prefix. Each configuration is represented on the graph by a point (x, y) where x represents the total expansion of the hashed prefixes that is produced by the specific configuration as a percentage of the extra provided capacity C_p. y indicates the hashing performance using as metric the number of unresolved collisions. Note that combinations that result in expansions larger than the limit of the extra provided capacity (i.e., >100%) are not considered. As expected, configurations that produce expansions close to the limits of the available space perform poorly since no hash function is capable of fully utilizing the available space perfectly. On the other hand, configurations that result in small expansions imply a selection of bits with poor entropy (leftmost selection) and hence, poor hashing performance.

From the experiments, we find that any configuration that can keep the expansion between 20% and 90% of the extra provided capacity results in less than 5% of unresolved collisions. It is important to mention here that it is preferable to select configurations that use as many bits as possible while their expansion remains in the desirable interval. To make this clearer, we show in Figure 5 the resulting trend lines[1] for three different sets of combinations, where each of them combines a different number of bits to form a configuration (e.g., 16, 17 or 18 bits). Obviously, bigger sets feed the hash function with more information and therefore they are expected to improve the hashing performance. For example, 18-bit configurations result in less unresolved collisions in general ($< 2\%$) and perform better than 17 or 16-bit configurations. However, bigger sets tend to shrink fast the "window" of the feasible configurations. Based on the same

[1] Trend lines are a polynomial interpolation of the data points.

example, 18 bits is the maximum set of bits that gives a practically applicable configuration. If 19 bits are used instead, the expansion will be more than 100% for any configuration and hence, such a selection should be avoided. Note that the number of the selected bits ($(R+F)$ bits in Figure 1) does not necessarily change the indexing length (R bits). These are the bits that participate in the hash index generation where a 12-bit index can be constructed by folding 4 bits from a 16-bit set or by folding 6 bits from an 18-bit set.

3.4. Handling unresolved collisions with a victim TCAM

Practically, a "good" hash scheme will result in a very low probability of having unresolved collisions on updates. To guarantee deterministic lookup rates, however, it is necessary to ensure that for certain loads, no unresolved collision will ever occur. Although theoretical analysis can give us probabilistic bounds related with the hashing performance of the d-left scheme [18] or Cuckoo variations [13, 19], it requires certain assumptions that might not accurately reflect the real behavior of the IP routing tables and the real design organization, and often suggests worst-case scenarios that lead to completely impractical implementations.

Instead, our work aims to derive a feasible solution that can be effectively suitable for high speed IP lookup. To deal with unresolved collisions, we use a small special victim space that can accommodate the small number of colliding items as some previous hashing solutions proposed, e.g., [9].

The victim space can be a small TCAM or some overflow/extra SRAM buckets that can be plugged in the MHT scheme as an extra small table. Adding such a small table in the parallel access path is not expected to increase the power consumption significantly. Moreover, we believe that the extra table along with the tables of our MHT scheme are suitable for implementing in pipeline structures and can be optimized for power savings.

4. Evaluation

4.1. Experimental setup

For a comprehensive experimental evaluation of the proposed scheme, we have developed and used a set of software tools that can report in detail how the studied hashing techniques and design configurations perform when IP routing tables are incrementally mapped and updated. We use a number of real IP routing tables obtained from [20] that differ in geographical as well as chronological position. For fair comparison of different IP lookup solutions in terms of their power and area, we base our study on product-grade implementation results reported in [22–24], that use the same $130nm$ process technology. Our baseline design consists of 3 sub-tables indexed by 12-bit skewed hash indices generated using 16 selected bits. Note that all the hash-based approaches in comparison are based on the same architecture and have identical access latency since we assume that they carry the same index generators in the critical access path.

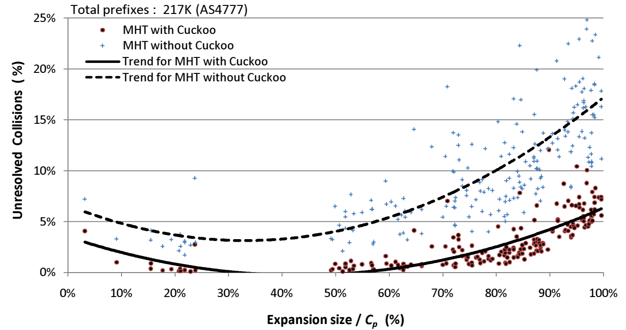

Figure 6. Our proposed MHT scheme vs standard d-left MHT scheme. Behavior of different Bit-Select configurations.

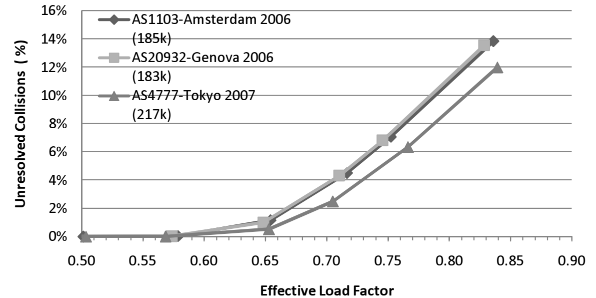

Figure 7. Space Efficiency. The size of unresolved collisions implies the required size of victim TCAM.

4.2. Results

Figure 6 is similar in concept to Figure 5, except that it shows a comparison between the hashing behavior of the Cuckoo-enhanced proposed scheme and the regular d-left scheme (for 16-bit configurations). Again the total capacity available is 2 times the nominal capacity of the original routing table. As expected, the enhanced scheme outperforms the regular d-left MHT scheme since the migration of prefixes can resolve most of the collisions that occur. The remaining unresolved collisions are below 5% of the original load N for any configuration in the interval of 20% to 80%. Such a bound could quite accurately define the appropriate size of the victim space that could be used to accommodate the colliding items.

The size of a victim TCAM is dependent on the *effective load factor*, which is defined as the ratio of the the nominal routing table size to the available space capacity. For example, the previous discussion suggests that the victim size should not be more than 5% of the original load (N) when a load factor of $N/2N = 0.5$ is given. Figure 7 shows the expected victim space requirements as a function of the effective load factor for three different routing tables. The Bit-Select configuration was chosen to be somewhere in the desirable interval. As expected, further shrinking of the avail-

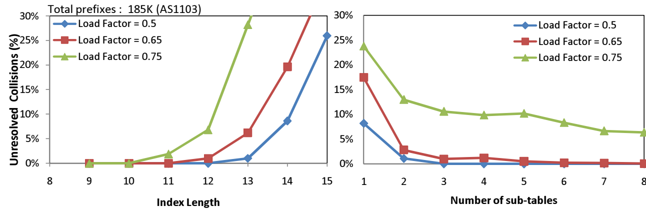

Figure 8. Design Space Exploration.

able space (larger effective load factor) increases the number of collisions and larger victim TCAM is required to handle them. On the other hand, offering more capacity (smaller effective load factor) lessens the need for extra storage and hence a trade-off appears between the effective load factor and the TCAM size requirements.

Exploring the design space. We seek to further explore the design space in order to acquire deeper knowledge of our scheme and possible improvements. We examine the effect of different hash index lengths as well as different number of sub-tables. In Figure 8(a), for a given effective load factor and a fixed number of sub-tables equal to three, we vary the length of the hash index. For consistency, every x-bit index is constructed by using the same 18 bits, so, what actually varies is the number of the folding bits in the hash function. For instance, a 12-bit hash index is generated by folding 6 bits while a 15-bit hash index folds only 3 bits. Shorter hash index implies two effects: First, given that storage capacity is kept unchanged, the narrower the address space the larger the bucket's size and therefore more hash entries are available per bucket. Second, a shorter hash index is constructed by using more folding bits over the same amount of information and hence is expected to distribute better the hash items among the address space. Thus, based on the previous facts, a shorter hash index improves the hashing results. However, a shorter index implies having more entries per bucket, which when implemented in hardware means larger memory lines and more prefixes to be fetched simultaneously for matching comparisons. These are all important design factors that affect the complexity, area and power consumption.

In a similar plot, we vary the number of sub-tables while using the same 12-bit index length (Figure 8(b)). Each table can use a different hash function, so more tables imply more powerful MHT schemes even though each bucket has less capacity. As it can be observed, a significant improvement appears when going from a single hashing to double and triple hashing. More hash tables seem to offer diminishing benefits, which do not justify the additional complexity and area overheads.

The results shown in Figure 8 were produced by using the routing table AS1103 with 185K entries. A higher load factor implies less total capacity since the nominal routing table size remains the same. We expect that different IP routing tables follow the same trend.

Comparison with state-of-the-art TCAM designs. Our scheme can be viewed as an attempt to shift the bottleneck of a hash-based lookup scheme like IPStash from the lookup throughput to the insertion time and relieve the IP lookup critical path from any timing barriers. Specifically, our implementation achieves a very high throughput of one lookup per cycle, accelerating the lookup rate by three times compared with IPStash, and avoiding the unpredictable lookup rates of the fast but imperfect Bloom-filter-based schemes. To make our implementation efficient, however, we need to spend more time during insertions or updates of the IP routing table. In the worst-case, incremental updates require $c \times d + 1$ memory accesses, where c is the bucket's size and d the number of sub-tables. However, such cases are extremely rare, unless the tables are heavily utilized. For lighly utilized tables, most of the insertions can finish within a single memory cycle.

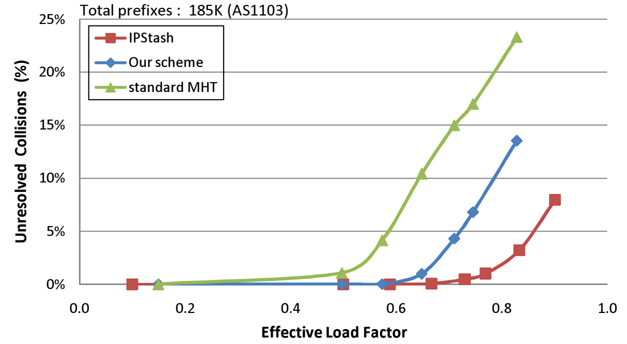

Figure 9. Space Efficiency: a comparison with IPStash and a standard d-left MHT.

In terms of space efficiency, Figure 9 shows that even though the applied hashing techniques are quite powerful and achieve high space utilization, table inflation prevents us from outperforming the highly space efficient IPStash.

TCAM-based IP lookup provides high throughput but suffers from large area requirements and power dissipation. On the other hand, IPStash has low area and power requirements but lower throughput. Our scheme combines high throughput, low area requirements and power savings. Figure 10 compares the area and power requirements of the three different schemes when their memory arrays are implemented with either static or dynamic memory cells.

Even with a load factor of 0.5 (and including a 5% victim TCAM), our implementation with dynamic memory achieves up to 60% reduction in area requirements and 80% of power savings compared with a state-of-the-art dynamic TCAM design [22–24]. Compared with the hash-based IPStash scheme, although area requirements per entry are identical, total area requirements are slightly more. This is because, we assume (for fairness) an IPStash with a high load factor of 0.8 and thus fewer total entries–since it is more space efficient than our scheme. Although IPStash is assumed to keep a routing table in a smaller memory array, its overall power consumption is be much higher since, on average, it requires 2.5 memory accesses per lookup [14].

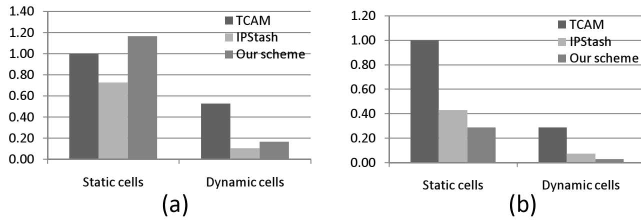

Figure 10. (a) Area and (b) Power requirements.

In summary, our experiments show that our scheme can operate in a collision-free mode under a fair Bit-Select configuration and a load factor of more than 0.65. For such a load factor, we suggest the use of a victim TCAM with size no more than 5% of the initial load. Larger victim TCAMs can be used to improve the load factor. For current routing tables, a 12-bit index generated by a set of 18 bits is preferable. The design can be expanded accordingly as the size of routing tables increases. Having a single memory access per lookup allows us to achieve a lookup throughput equal to that of TCAMs. At the same time, the highly optimized hashing strategy of our scheme results in a significant reduction in power consumption and area.

5. Conclusions

This paper presented a new hash-based IP lookup scheme that is both storage efficient and high performance. The following contributions are made and findings presented:

- We introduce a multi-hash, multi-table approach to building a hardware IP lookup engine that achieves single memory access latency/throughput per lookup. One may access the hash tables in parallel for the smallest lookup latency or choose to access them in a pipelined manner to save power at the same high lookup rate.
- We find that simple fast hash functions can work sufficiently well if certain relocations of items are allowed during hashing and if key characteristics of the prefixes are taken into consideration. We have developed and used a design flow that analyzes the performance of a set of given hash configurations and automatically derives key design parameters.
- We present a controlled wildcard resolution (CWR) technique to aid hashing wildcard bits in IP prefixes. It does not require that the prefixes be expanded *en masse* before table construction. Rather, CWR resolves wildcard bits in the hash function when a prefix is actually inserted to the hash table in a simple, elegant way.
- We present a detailed empirical study that shows the efficacy of the proposed scheme using a set of real IP routing tables. Specifically, we show that the proposed design can reduce area and power requirements by 60% and 80% respectively when compared with state-of-the-art TCAM designs, and has the advantage of high and constant lookup rate when compared with other advanced hash-based IP lookup schemes.

While we recognize the potential of power savings with the pipelined table lookup in our scheme, we have not explored the design space of such an architecture in this paper. Our current research investigates the hash algorithms and hardware design strategies geared toward further reducing power consumption based on the idea of pipelining, without compromising the area efficiency and performance we obtain in this paper.

References

[1] A. Gallo, "Meeting traffic demands with next-generation Internet infrastructure." Lightwave, 2001.

[2] I. D. T. Inc. http://www.idt.com/products/.

[3] N. microsystems. http://www.netlogicmicro.com.

[4] M. Technology. http://www.micron.com.

[5] W. Eatherton *et al.* "Tree bitmap: Hardware/software ip lookups with incremental updates," *ACM SIGCOMM Computer Communications Review*, 34(2), 2004.

[6] L. Devroye, "Efficient construction of multibit tries for ip lookup," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, pp. 650–662, August 2003.

[7] V. Srinivasan and G. Varghese, "Faster ip lookups using controlled prefix expansion," pp. 1–10, ACM SIGMETRICS, June 1998.

[8] V. J. J. Hasan, S. Cadambi and S. Chakradhar, "Chisel: A storage-efficient, collision-free hash-based network processing architecture," pp. 203–215, IEEE ISCA, June 2006.

[9] S. Kaxiras and G. Keramidas, "Ipstash: A set-associative memory approach for efficient ip-lookup," pp. 992–1001, IEEE Infocom, 2005.

[10] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed ip routing lookups," pp. 25–37, ACM SIGCOMM, September 1997.

[11] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," pp. 181–192, ACM Sigcomm, August 2005.

[12] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," ACM Sigcomm, August 2005.

[13] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Lecture Notes in Computer Science*, vol. 2161, pp. 121–133, 2001.

[14] S. Kaxiras and G. Keramidas, "Ipstash: A power-efficient memory architecture for ip-lookup," pp. 361–373, IEEE Micro, November 2003.

[15] A. Z. Broder and A. R. Karlin, "Multilevel adaptive hashing," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pp. 43–53, 1990.

[16] S. Cho, J. Martin, R. Xu, M. Hammoud, and R. Melhem, "Ca-ram: A high-performance memory substrate for search-intensive applications," pp. 230–241, IEEE ISPASS, April 2007.

[17] R. Jain, "A Comparison of Hashing Schemes for Address Lookup in Computer Networks," pp. 1570–1573, IEEE Transactions on Communications, 1992.

[18] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve ip lookups," pp. 1454–1463, IEEE Infocom, 2001.

[19] M. M. A. Kirsch, "The power of one move: Hashing schemes for hardware," IEEE Infocom, 2008.

[20] RIS, "Routing information service." http://www.ripe.net/ris/, 2006.

[21] G. Huston, "Analyzing the internet's bgp routing table," *The Internet Protocol Journal*, vol. 4, 2001.

[22] F. Morishita *et al.* "A 312-MHz 16-Mb Random-Cycle Embedded DRAM Macro With a Power-Down Data Retention Mode for Mobile Applications," *IEEE J. Solid-State Circuits*, Jan. 2005.

[23] H. Noda, K. Inoue, H. J. Mattausch, T. Koide, and K. Arimoto. "A Cost-Efficient Dynamic Ternary CAM in 130nm CMOS Technology with Planar Complementary Capacitors and TSR Architecture," *Proc. Int'l Symp. VLSI Circuits*, June 2003.

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