Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing

Haoyu Song
Applied Research Lab
Washington University in St. Louis
St. Louis, MO, 63130
hs1@arl.wustl.edu

Sarang Dharmapurikar
Applied Research Lab
Washington University in St. Louis
St. Louis, MO, 63130
sarang@arl.wustl.edu

Jonathan Turner
Applied Research Lab
Washington University in St. Louis
St. Louis, MO, 63130
jst@arl.wustl.edu

John Lockwood
Applied Research Lab
Washington University in St. Louis
St. Louis, MO, 63130
lockwood@arl.wustl.edu

ABSTRACT

Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, per-flow state management and network monitoring. These applications, which typically occur in the data-path of high-speed routers, must process and forward packets with little or no buffer, making it important to maintain wire-speed throughout. A poorly designed hash table can critically affect the worst-case throughput of an application, since the number of memory accesses required for each lookup can vary. Hence, high throughput applications require hash tables with more predictable worst-case lookup performance. While published papers often assume that hash table lookups take constant time, there is significant variation in the number of items that must be accessed in a typical hash table search, leading to search times that vary by a factor of four or more.

We present a novel hash table data structure and lookup algorithm which improves the performance over a naive hash table by reducing the number of memory accesses needed for the most time-consuming lookups. This allows designers to achieve higher lookup performance for a given memory bandwidth, without requiring large amounts of buffering in front of the lookup engine. Our algorithm extends the multiple-hashing Bloom Filter data structure to support exact matches and exploits recent advances in embedded memory technology. Through a combination of analysis and simulations we show that our algorithm is significantly faster than a naive hash table using the same amount of memory, hence it can support better throughput for router applications that use hash tables.

SIGCOMM'05, August 22–26, 2005, Philadelphia, Pennsylvania, USA.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers

General Terms

Algorithms, Design, Performance

Keywords

Hash Table, Forwarding

1. INTRODUCTION

A hash table is a versatile date structure for performing fast associative lookups, which requires $O(1)$ average memory accesses per lookup. Indeed, due to its wide applicability in network packet processing, some of modern network processors provide built-in hashing units [17]. A survey of recent research literature on network packet processing reveals that hash tables are common to many applications including per-flow state management, IP lookup and packet classification. These applications typically appear in the data-path of high-speed routers. Hence, they must be able to process packets at line speed, which makes it imperative for the underlying hash tables to deliver a good lookup performance.

1.1 Hash Tables For Packet Processing

Following is a short discussion of how various network processing applications use hash tables and why their lookup performance is important.

Maintaining Per-flow Context: One of the most important applications of hash tables in network processing is in the context of maintaining connection records or per-flow state. Per-flow state is useful in providing QoS for flows, recording measurements, and monitoring and payload analysis in Intrusion Detection Systems (IDS).

For instance, intrusion detection systems like Bro [21] and Snort [2] maintain a hash table of connection records for TCP connections. A record is created and accessed by computing a hash over the 5-tuple of the TCP/IP header. This record contains the certain information describing the connection state and is updated upon the arrival of each packet of that connection. Efforts are under way to implement intrusion detection systems in hardware for

line speed packet processing [23][12]. In these implementations, connection records are maintained in DRAM. Similarly, hardware-based network monitoring systems such as NetFlow [1] or Adaptive NetFlow [13] maintain a hash table of connection records in DRAM.

IP Route Lookup: Efficient hash tables are important for some IP routing lookup algorithms. In particular, the Binary Search on Prefix Lengths [28] algorithm, which has the best theoretical performance of any sequential algorithm for the best-prefix matching problem, uses hash tables. The algorithm described in [11], uses parallel search of on-chip Bloom filters to identify which of a number of off-chip hash tables must be searched to find the best-matching prefix for a given packet.

In [28], prefixes are grouped according to their lengths and stored in a set of hash tables. A binary search on these tables is performed to find the matching prefixes of the destination IP address. Each search step probes a corresponding hash table to find a match. By storing extra information along with the member prefixes in hash tables, a match in a given table implies that the longest matching prefix is at least as long as the size of prefixes in the table, whereas a failure to match implies the longest matching prefix is shorter. In the worst case, if there are W different possible prefix lengths, the search requires at most $\log W$ probes of the hash table. For IPv4 lookup, this means we need to perform lookups in five hash tables. Even with the controlled prefix expansion [25] we need multiple hash table lookups depending on the resulting number of unique prefix lengths. This algorithm critically demands better hash tables to preserve the performance gained by binary search.

In [11], the authors present a hardware based LPM algorithm for IP lookup. The technique improves the performance of a regular hash table using Bloom filters. In this algorithm prefixes are also grouped by length. Each group is programmed in a Bloom filter and All the prefixes are kept in a hash table. Bloom filters are maintained in a high-bandwidth and small *on-chip* memory while the hash table resides in the slow and high volume *off-chip* memory. Before a search is initiated in the off-chip table, the on-chip Bloom filter is probed to check if the item exists in the table. This typically allows one to perform just a single probe of the off-chip table. However, if the probe of the off-chip table requires multiple memory accesses, the performance of the algorithm can suffer.

The *BART* scheme [20] also uses hash tables for routing table lookup. It constructs simple hash functions by picking a few bits in the IP address. To bound the collisions in a hash bucket, it selects the bits for use in the hash function, based on an exhaustive search of the space of possible bit sets. This makes configuration of the lookup engine for a particular set of address prefixes cumbersome and time-consuming.

Packet Classification: Hash tables are also used for some packet classification algorithms. Fundamentally, many packet classification algorithms first perform a lookup on a single header field and leverage the results to narrow down the search to a smaller subset of packet classifiers [18, 4, 19, 15]. Since a lookup on the individual fields can also be performed using one of the hash table based algorithms mentioned above, improving the hash table performance also benefits packet classification algorithms.

The tuple space search algorithm [24] groups the rules into a set of "tuples" according to their prefix lengths specified for different fields. Each group is then stored in a hash table. The packet classification queries perform exact match operations on each of the hash tables corresponding to all possible tuples, given the rule set. While the algorithm analysis in [24] was centers on the number of distinct tuples, the hash table lookup performance also directly affects the classification throughput.

Exact flow matching is an important subproblem of the general packet classification problem, where the lookup performs an exact match on the packet 5-tuple header fields. In [26], exact filters are used for reserved bandwidth flows and multicast in high performance routers as an auxiliary component to general packet classification. The search technique described in [26] employs a hash lookup with chaining to resolve collisions. A hash key based on low-order bits of the source and destination address is used to probe an on-chip hash table containing "valid" bits. If the appropriate bit for the packet being processed is set, the hash key is used to index a table in off-chip Static Random Access Memory (SRAM). Off-chip table items are chained together if multiple filters hash to the same bucket. The hash table performance directly impacts the system throughput.

The above mentioned applications illustrate the role of hashing in a variety of network processing applications and make it clear that the performance of the hash table lookup has a direct impact on their performance.

1.2 Related Work

A hash table lookup involves hash computation followed by memory accesses. While memory accesses due to collisions can be moderately reduced by using sophisticated cryptographic hash functions such as MD5 or SHA-1, these are difficult to compute quickly. In the context of high-speed packet processing devices, even with specialized hardware, such hash functions can take several clock cycles to produce the output. For instance, some of the existing hardware implementations of the hash cores consume more than 64 clock cycles [16], which exceeds the budget of minimum packet time. Moreover, the performance of such hash functions is no better than the theoretical performance with the assumption of uniform random hashing.

Another avenue to improve the hash table performance would be to devise a perfect hash function based on the items to be hashed. While this would deliver the best performance, searching for a suitable hash function can be a slow process and needs to be repeated whenever the set of items undergoes changes. Moreover, when a new hash function is computed, all the existing entries in the table need to be re-hashed for correct search. This can impede the normal operations on the hash table making it impractical in high-speed processors. Some applications instead settle on using a "semi-perfect" hash function which can tolerate a predetermined collision bound. However, even searching for such a hash function can require time in the order of minutes [25, 20].

Multiple hash functions are known to perform better than single hash functions [6]. When we have multiple hash tables each with different hash function, the items colliding in one table are hashed into other tables. Each table has smaller size and all hash functions can be computed in parallel. Another multi-hashing algorithm, *d-random scheme*, uses only one hash table but d hash functions [3]. Each item is hashed by d independent hash functions, and the item is stored into the least loaded bucket. A search needs to examine d buckets but the bucket's average load is greatly reduced. A simple variation of *d-random*, which is called the *d-left scheme* is proposed to improve IP lookups [7]; this approach generalizes the *2-left* scheme in [27]. In this scheme, the buckets are partitioned into d sections, each time a new item needs to be inserted, it is inserted into the least loaded bucket (left-most in case of a tie). Simulation and analysis show the performance is better than *d-random*. While these ideas are similar to our fast hash table algorithm, our approach uses on-chip Bloom filters to eliminate the need to search multiple buckets in an off-chip memory.

A Bloom filter [5] can be considered a form of multi-hashing.

Counting Bloom Filter [14] extend the simple binary Bloom filter by replacing each bit in the filter with a counter. This makes it possible to implement a deletion operation on the set represented by the Bloom filter. Some lookup mechanisms schemes use a Bloom filter to avoid unnecessary searches of an off-chip hash table table [11, 10]. While this is useful, it does nothing to reduce the time needed to perform the search of the off-chip table when a search is called for. Thus lookup performance can still be unpredictable. In contrast, our fast hash table algorithm fully utilizes the information gained from an extended Bloom filter to optimize the exact match lookup.

1.3 Scope for Improvement

From a theoretical perspective, although hash tables are among the most extensively studied data structures with almost saturated improvements, from an engineering perspective designing a good hash table can still be a challenging task with potential for several improvements. The main engineering aspect that differentiates our hash table design from the rest is the innovative use of the advanced embedded memory technology in hardware. Today it is possible to integrate a few mega bits of Static Random Access Memory (SRAM) with multiple access ports into a very small silicon. Moreover, multiple such embedded memory cores can be incorporated in the same VLSI chip. For instance, most of the modern Field Programmable Gate Array (FPGA) devices contain multiple on-chip embedded SRAM with two read/write ports. Some of the high-end FPGAs such as Xilinx Virtex II Pro contain 512 memory blocks each with 18K bits [29]. We exploit the high lookup capacity offered by such memory blocks to design an efficient hash table.

At the same time it is important to note that embedded memory on its own is not sufficient to build a fast hash table when we need to maintain a large number of items having significant size. For instance, we can not squeeze 100,000 TCP connection records each of 32 bytes into a hash table built with only 5Mbits of on-chip memory. Thus, we must resort to using the commodity memory such SDRAM to store the items in the hash table. Since, DRAM is inherently slow, use of commodity memory makes it imperative to reduce the off-chip memory access resulting either from collision or due to unsuccessful searches for efficient processing. This leads us to the question: Can we make use of the small but high bandwidth on-chip memory to improve the lookup performance of an off-chip hash table? The answer to this question forms the basis of our algorithm.

We use the well-known data structure Bloom filter [5], and extends it to support *exact match* and reduce the time required to perform this exact match. We use a small amount of on-chip multiport memories to realize a counting-Bloom-filter-like data structure such that it not only answers the membership query on the search items but also helps us reduce the search time in the off-chip table.

The rest of the paper is organized as follows. Section 2 illustrates our algorithms and architecture of fast hash table. In Section 3, we provide a detailed mathematical analysis of the proposed hash table algorithm. We also provide comparisons on the average search time and the expected collision list length of the naive hash table and our fast hash table, theoretically and experimentally, in Section 3 and 4. Finally, Section 5 concludes the paper.

2. FAST HASHING

For the purpose of clarity, we develop our algorithm and hash table architecture incrementally starting from a naive hash table (NHT).

We consider the hash table algorithm in which the collisions are resolved by chaining since it has better performance than open addressing schemes and is one of the most popular methods [9].

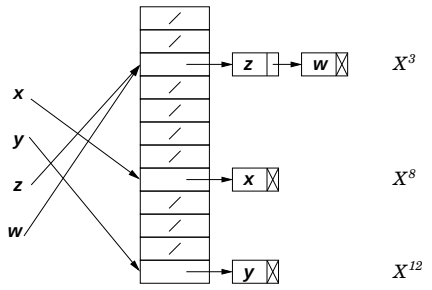

Figure 1: A Naive Hash Table

An NHT consists of an array of m buckets with each bucket pointing to the list of items hashed into it. We denote by X the set of items to be inserted in the table. Further, let X^i be the list of items hashed to bucket i and X^i_j the j^{th} item in this list. Thus,

$$X^i = \{X^i_1, X^i_2, X^i_3, ..., X^i_{a_i}\}$$
$$X = \bigcup_{i=1}^{L} X^i$$

where a_i is the total number of items in the bucket i and L is the total number of lists present in the table. In the Figure 1, $X^3_1 = z$, $X^3_2 = w$, $a_3 = 2$ and $L = 3$.

The insertion, search and deletion algorithms are straightforward:

InsertItem$_{NHT}(x)$
1. $X^{h(x)} = X^{h(x)} \cup x$

SearchItem$_{NHT}(x)$
1. if ($x \in X^{h(x)}$) return true
2. else return false

DeleteItem$_{NHT}(x)$
1. $X^{h(x)} = X^{h(x)} - x$

where $h()$ is the hash function based on uniform random hashing.

2.1 Basic Fast Hash Table

We now present our Fast Hash Table (FHT) algorithm. First we present the basic form of our algorithm which we call Basic Fast Hash Table (BFHT) and then we improve upon it.

We begin with the description of Bloom filter which is at the core of our algorithms. A Bloom filter is a hash-based data structure to store a set of items compactly. It computes *k* hash functions on each item, each of which returns an address of a bit in a bitmap of length *m*. All the *k* bits chosen by the hash values in the bitmap are set to '1'. By doing this, the filter essentially programs the bitmap with a signature of the item. By repeating the same procedure for all input items, Bloom filter can be programmed to contain the summary of all the items. This filter can be queried to check if a given item is programmed in it. The query procedure is similar—the same *k* hash functions are calculated over the input and the corresponding *k* bits in the bitmap are probed. If all the bits are set then the item is said to be present, otherwise it is absent. However, since the bit-patterns of multiple items can overlap within the bitmap, Bloom filter can give false-positive result to a membership query.

For the ensuing discussion, we will use a variant of a Bloom filter

called *Counting Bloom Filter* [14] in which each bit of the filter is replaced by a counter. Upon the insertion of an item, each counter indexed by the corresponding hash value is incremented. Due to its structure, a counter in this filter essentially gives us the number of items hashed in it. We will show how this information can be used effectively to minimize the search time in the table.

We maintain an array C of m counters where each counter C_i is associated with bucket i of the hash table. We compute k hash functions $h_1(), ..., h_k()$ over an input item and increment the corresponding k counters indexed by these hash values. Then, we store the item in the lists associated with each of the k buckets. Thus, a single item is stored k times in the off-chip memory. The following algorithm describes the insertion of an item in the table.

InsertItem$_{BFHT}(x)$
1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $C_{h_i(x)} + +$
4. $X^{h_i(x)} = X^{h_i(x)} \cup x$

Note that if more than one hash functions map to the same address then we increment the counter only once and store just one copy of the item in that bucket. To check if the hash values conflict, we keep all the previously computed hash values for that item in registers and compare the new hash value against all of them (line 2).

The insertion procedure is illustrated in the Figure 2. In this figure, four different items, x, y, z and w are shown to have been inserted, sequentially. Thus, each of the items is replicated in $k = 3$ different buckets and the counter value associated with the bucket reflects the number of items in it.

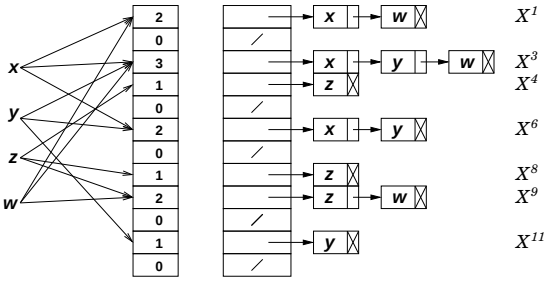

Figure 2: Basic Fast Hash Table (BFHT). The data structure after inserting x, y, z and w

Search procedure is similar to the insertion procedure: given an item x to be searched, we compute k hash values and read the corresponding counters. When all the counters are non-zero, the filter indicates the presence of input item in the table. Only after this step, we proceed to verify it in the off-chip table by comparing it with each item in the list of items associated with one of the buckets. Indeed, if the counters are kept in the fast on-chip memory such that all of the k random counters associated with the item can be checked in parallel then in almost all cases we avoid an off-chip access if the table does not contain the given item. Given the recent advances in the embedded memory technologies, it is conceivable to implement these counters in a high speed multi-port on-chip memory.

Secondly, the choice of the list to be inspected is critical since the list traversal time depends on the length of the list. *Hence, we choose the list associated with the counter with smallest value* to reduce off-chip memory accesses. The speedup of our algorithm comes from the fact that it can choose the smallest list to search where as an NHT does not have any choice but to trace only one list which can potentially have several items in it.

As will be shown later, in most of the cases, for a carefully chosen value of the number of buckets, the minimum value counter has a value of 1 requiring just a single memory access to the off-chip memory. In our example shown in Figure 2, if item y is queried, we need to access only the list X^{11}, rather than X^3 or X^6 which are longer than X^{11}.

When multiple counters indexed by the input item have the same minimum value then somehow the tie must be broken. We break this tie by simply picking the minimum value counter with the smallest index. For example, in Figure 2, item x has two bucket counters set to 2, which is also the smallest value. In this case, we always access the bucket X^1.

The following pseudo-code summarizes the search algorithm on BFHT.

SearchItem$_{BFHT}(x)$
1. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
2. if ($C_{min} == 0$)
3. return false
4. else
5. i = SmallestIndexOf(C_{min})
6. if ($x \in X^i$) return true
7. else return false

Finally, if the input item is not present in the item list then clearly it is a false positive match indicated by the counting Bloom filter.

With the data structure above, deletion of an item is easy. We simply decrement the counters associated with the item and delete all the copies from the corresponding lists.

DeleteItem$_{BFHT}(x)$
1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $C_{h_i(x)} - -$
4. $X^{h_i(x)} = X^{h_i(x)} - x$

2.1.1 Pruned Fast Hash Table (PFHT)

In BFHT, we need to maintain up to k copies of each item which requires k times more memory compared to NHT. However, it can be observed that in a BFHT only one copy of each item — the copy associated with the minimum counter value — is accessed when the table is probed. The remaining $(k-1)$ copies of the item are never accessed. This observation offers us the first opportunity for significant memory optimization: all the other copies of an item except the one that is accessed during the search can now be deleted. Thus, after this pruning procedure, we have exactly one copy of the item which reduces the memory requirement to the same as that of the NHT. We call the resulting hash table a Pruned Fast Hash Table (PFHT).

The following pseudo-code summarizes the pruning algorithm.

PruneSet(X)
1. for (each $x \in X$)
2. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
3. i = SmallestIndexOf(C_{min})
4. for ($l = 1$ to k)
5. if ($h_l(x) \neq i$) $X^{h_l(x)} = X^{h_l(x)} - x$

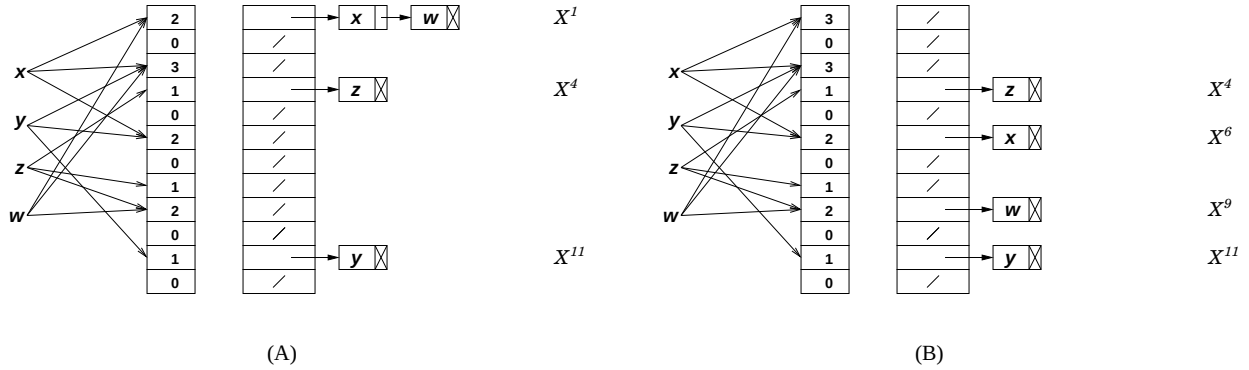

Figure 3: Illustration of Pruned Fast Hash Table (PFHT) (A) The data structure after execution of pruning (B) List-balancing. Items x and w are re-adjusted into different buckets by incrementing counter 1.

The pruning procedure is illustrated in Figure 3(A). It is important to note that during the Pruning procedure, the *counter values are not changed*. Hence, after pruning is completed, the counter value no longer reflects the number of items actually presenting in the list and is usually greater than that. However, for a given item, the bucket with the smallest counter value always contains this item. This property ensures the correctness of the search results. Another property of pruning is that it is independent of the sequence in which the items are pruned since it depends just on the counter values, which are not altered. Hence, pruning in sequence x-y-z-w will yield the same result as pruning it in z-y-x-w.

A limitation of the pruning procedure is that now the incremental updates to the table are hard to perform. Since counter values no longer reflect the number of items in the list, if counters are incremented or decremented for any new insertion or deletion respectively then it can disturb the counter values corresponding to the existing items in the bucket which in turn will result in an incorrect search. For example, in Figure 3(A), the item y maps to the lists $\{X^3, X^6, X^{11}\}$ with counter values $\{3, 2, 1\}$ respectively. If a new item, say v, is inserted which also happens to share the bucket 11 then the counter will be incremented to 2. Hence, the minimum counter value bucket with smallest index associated with y is no longer the 11 but now it is 6 which does not contain y at all. Therefore, a search on y will result in an incorrect result. With this limitation, for any new insertion and deletion the table must be reconstructed from scratch which can be make this algorithm impractical for variable item sets.

We now describe a version of InsertItem and DeleteItem algorithms which can be performed incrementally. *The basic idea used in these functions is to maintain the invariant that out of the k buckets indexed by an item, it should always be placed in a bucket with smallest counter value.* In case of a tie, it should be placed in the one with smallest index. If this invariant is maintained at every point then the resulting hash table configuration will always be the same irrespective of the order in which items are inserted.

In order to insert an item, we first increment the corresponding k counters. If there are any items already present in those buckets then their corresponding smallest counter might be altered. However, the counter increments do not affect all other items. Hence, each of these items must be re-inserted in the table. In other words, for inserting one item, we need to reconsider all and only the items in those k buckets.

The following pseudo-code describes the insertion algorithm.

InsertItem$_{PFHT}(x)$

1. $Y = x$
2. for (i = 1 to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $Y = Y \bigcup X^{h_i(x)}$
5. $X^{h_i(x)} = \phi$
6. $C_{h_i(x)} + +$
7. for (each $y \in Y$)
8. $C_{min} = min\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
9. i= SmallestIndexOf(C_{min})
10. $X^i = X^i \cup y$

In the pseudo-code above, Y denotes the list of items to be considered for insertion. It is first initialized to x since that is definitely the item we want to insert (line 1). Then for each bucket x maps to, if the bucket was not already considered (line 3), we increment the counter (line 6), collect the list of items associated with it (line 4) since now all of them must be reconsidered and also delete the lists form the bucket (line 5). Finally, all the collected items are re-inserted (lines 8-10). It is important to note that we do not need to increment the counters while re-inserting them since the items were already inserted earlier. Here we just change the bucket in which they go.

Since the data structure has n items stored in m buckets, the average number of items per bucket is n/m. Hence the total number of items read from buckets is nk/m requiring as many memory accesses. Finally $1 + nk/m$ items are inserted in the table which again requires as many memory accesses. Hence the insertion procedure has a complexity of the order $O(1 + 2nk/m)$ operations totally. Moreover, for an optimal Bloom filter configuration, $k = mln2/n$. Hence, the overall memory accesses required for insertion are $1 + 2ln2 \approx 2.44$.

Unfortunately, incremental deletion is not as trivial as insertion. When we delete an item we need to decrement the corresponding counters. This might cause these counters to be eligible as the smallest counter for some items which hashed to them. However, now that we keep just one copy of each item we can not tell which items hash to a given bucket if the item is not in that bucket. This can be told with the help of only pre-pruning data structure i.e. BFHT in which an item is inserted in all the k buckets and hence we know which items hash to a given bucket. Hence in order to perform an incremental deletion, we must maintain an *off-line BFHT* like the one shown in Figure 2. Such a data structure can be maintained in router software which is responsible for updates to the table.

In order to differentiate between the off-line BFHT and on-line PFHT we denote the off-line lists by χ and the corresponding counter by ζ. Thus, χ^i denotes the list of items associated with bucket i, χ^i_j the j^{th} item in χ^i and ζ_i the corresponding counter. The following pseudo-code describes the deletion algorithm.

DeleteItem$_{PFHT}$(x)

1. $Y = \phi$
2. for (i = 1 to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $\zeta_{h_i(x)} - -$
5. $\chi^{h_i(x)} = \chi^{h_i(x)} - x$
6. $Y = Y \bigcup \chi^{h_i(x)}$
7. $C_{h_i(x)} - -$
8. $X^{h_i(x)} = \phi$
9. for (each $y \in Y$)
10. $C_{min} = min\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
11. i= SmallestIndexOf(C_{min})
12. $X^i = X^i \cup y$

When we want to delete an item, we first perform deletion operation on off-line data structure using DeleteItem$_{BFHT}$ algorithm (line 2-5). Then we collect all the items in all the affected buckets (buckets whose counters are decremented) of BFHT for reinsertion. At the same time, we delete the list of items associated with each bucket from the PFHT since each of them now must be reinserted (line 7-8). Finally, for each item in the list of collected items, we re-insert it (line 9-12) just as we did in InsertItem$_{PFHT}$. Notice the resemblance between the lines 6-12 of DeleteItem$_{PFHT}$ with lines 4-10 of InsertItem$_{PFHT}$. The only difference is that in DeleteItem$_{PFHT}$, we collect the items to be re-inserted from the BFHT and we decrement the counter instead of incrementing it.

Before we derive the expressions for the complexity of DeleteItem algorithm, we notice that we have two types of operations involved: on the BFHT and on the PFHT. We derive the complexity for only the PFHT operations since the BFHT operations can be performed in the background without impeding the normal operations on PFHT. With this consideration, we note that the number of items per non-empty bucket in BFHT is $2nk/m$ since only half the buckets in the optimal configuration are non-empty (see Section 3). Since we collect the items from k buckets, we have totally $2nk^2/m$ items to be re-adjusted in the loop of line 9. For readjustment, we need to read as well as write each item. Hence the overall complexity of the deletion operation is $O(4nk^2/m)$. With optimal configuration of the table it boils down to $4kln2 \approx 2.8k$.

2.1.2 Optimizations

After the pruning procedure, more than one items can still reside in one bucket. We show a heuristic balancing scheme to further balance the bucket load by manipulating the counters and a few items. The reason that a bucket contains more than one items is because *this bucket is the first least loaded bucket indicated by the counter values for the involved items that are also stored in this bucket.* Based on this observation, if we artificially increment this counter, all the involved items will be forced to reconsider their destination buckets to maintain the correctness of the algorithm. There is hope that by rearranging these items, each of them can be put into an actually empty bucket. The feasibility is based on two facts: first, analysis and simulations show that for an optimal configuration of Bloom filter, there are very few collisions and even fewer collisions involving more than 2 items. Each items has k possible destination buckets and in most case the collided bucket is the only one they share. The sparsity of the table provides a good opportunity to resolve the collision by simply giving them second choice. Secondly, this process does not affect any other items, we need to only pay attention to the involved items in the collided bucket.

However, incrementing the counter and rearranging the items may potentially create other collisions. So we need to be careful to use this heuristics. Before we increment a counter, we first test the consequence. We perform this scheme only if this action does not result in any other collision. The algorithm scans the collided buckets for several rounds and terminates if no more progress can be made or the involved counters are saturated. We will show that this heuristics is quite effective and in our simulations all collisions are resolved and each non-empty bucket contains exactly one item. Figure 3(B) illustrates this list balancing optimization. By simply incrementing the counter in bucket X^1 and re-inserting the involved items x and w, we resolve the collision and now each non-empty bucket contains exactly one item.

2.1.3 Shared-node Fast Hash Table (SFHT)

In the previous section, we saw that in order to perform incremental updates, we need an off-line BFHT. However, with the assumption that the updates are relatively infrequent compared to the query procedure, we can afford to maintain such a data structure in control software which will perform updates on the internal data structure (which is slow) and later update the pruned data structure accordingly. However, some applications involve time critical updates which must be performed as quickly as possible. An example is the TCP/IP connection context table where connections get set up and broken frequently and the time for table query per packet is comparable to time for addition/deletion of connection records [12].

We present an alternative scheme which allows easy incremental updates at the cost of a little more memory than the required for PFHT but significantly less than that of BFHT. The basic idea is to allow the multiple instances of the items to share the same item node using pointers. We call the resulting hash table as Shared-node Fast Hash Table (SFHT). The lookup performance of the resulting scheme is the same as that of the BFHT but slightly worse than the PFHT. Moreover, with the reduced memory requirement, this data structure can be kept on-line.

The new algorithm can be illustrated with the help of Figure 4.

We start with an empty table and insert items one by one. When the first item, x is inserted we just create a node for the item and instead of inserting it separately in each of the lists corresponding to the hash buckets, we simply make the buckets point to the item. This clearly results in a great deal of memory savings. When we insert the next item y, we create the node and make the empty buckets point to the item directly. However, two of the three buckets already have a pointer pointing to the earlier item, x. Hence we make the item x point to y using the next pointer. Note that the counters are incremented at each insertion. More importantly, the counter values may not reflect the length of the linked list associated with a bucket. For instance, the first bucket has a value of 1 but there are two items, x and y in the linked list associated with this bucket. Nevertheless, it is guaranteed that we will find a given item in a bucket associated with that item by inspecting the number of items equal to the associated counter value. For instance, when we wish to locate x in the first bucket, it is guaranteed that we need to inspect only one item in the list although there are two items in it. The reason that we have more items in the linked list than indicated by the counter value is because multiple linked lists can get merged as in the case of x and y.

The insertion of item z is straightforward. However, an interest-

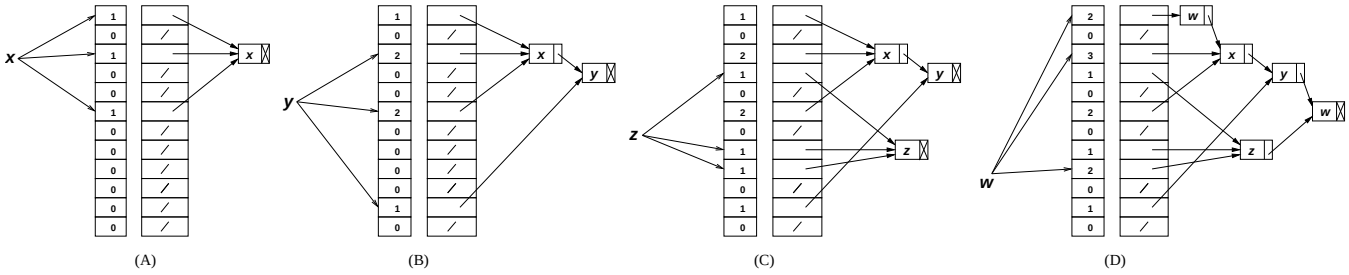

Figure 4: Illustration of Shared-node Fast Hash Table (SFHT)

ing situation occurs when we insert w. Notice that w is inserted in 1^{st}, 3^{rd} and 9^{th} bucket. We create a node for w, append it to the linked lists corresponding to the buckets and increment the counters. For 3^{rd} and 9^{th} bucket, w can be located exactly within the number of items indicated by the corresponding counter value. However, for the first bucket this is not true: while the counter indicates two items, we need to inspect three in order to locate w. This inconsistency will go away if instead of appending the item we *prepend* it to the list having the property of counter value smaller than the number of items in the list. Thus, if we want to insert w in the first bucket and we find that the number of items in the list is two but the counter value is one, we prepend w to the list. This will need replication of the item node. Once prepended, the consistency is maintained. Both the items in the first list can be located by inspecting at the most two items as indicated by the counter value.

The item node replication causes the memory requirement to be slightly more than what we need in NHT or PFHT where each item is stored just once. However, the overall memory requirement is significantly lower than the BFHT.

The following pseudo-code describes the algorithm.

InsertItem$_{SFHT}(x)$
1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. if ($C_{h_i(x)} == 0$)
4. Append(x, $X^{h_i(x)}$)
5. else
6. $l \leftarrow 0$
7. while ($l \neq C_{h_i(x)}$)
8. l++
9. read $X_l^{h_i(x)}$
10. if ($X_{l+1}^{h_i(x)} \neq NULL$) Prepend($x$, $X^{h_i(x)}$)
11. else Append(x, $X^{h_i(x)}$)
12. $C_{h_i(x)}$++

In this pseudo-code, l is used as a counter to track the number of items searched in the list. We search up to $C_{h_i(x)}$ items in the list. If the list does not end after $C_{h_i(x)}$ items (line 10) then we prepend the new item to the list otherwise we append it. Note that prepending and appending simply involves scanning the list for at the most $C_{h_i(x)}$ items. Hence the cost of insertion depends on the counter value and not on the actual linked list length. In SFHT, we have nk items stored in m buckets giving us an average counter value nk/m. We walk through nk/m items of each of the k lists and finally append or prepend the new item. Hence the complexity of the insertion is of the order $O(nk^2/m + k)$. Moreover, for an optimal counting Bloom filter, $k = mln2/n$ (see Section 3). Hence the memory accesses for deletion are proportional to k.

The extra memory requirement due to node replication is hard to compute but typically small. We use simulation results to validate our claim. Our simulation results presented in Section 3.4 show that the memory consumption is typically 1 to 3 times that of NHT (or PFHT). This, however, is significantly smaller than that of BFHT.

The pseudo-code for deletion on SFHT is as shown below. We delete an item from all the lists by tracing each list. However, since the same item node is shared among multiple lists, after deleting a copy we might not find that item again by tracing another list which was sharing it. In this case we do not trace the list till the end. We just need to consider the number of items equal to the counter value. If the list ends before that then we simply start with the next list (line 4).

DeleteItem$_{SFHT}(x)$
1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $l \leftarrow 1$
4. while ($l \neq C_{h_i(x)}$ AND $X_l^{h_i(x)} \neq NULL$)
5. if ($X_l^{h_i(x)} == x$)
6. $X^{h_i(x)} = X^{h_i(x)} - x$
7. break
8. l++
9. $C_{h_i(x)}$ − −

3. ANALYSIS

We analyze and compare the FHT algorithm with the NHT algorithm in terms of the expected lookup time and lookup time tail probability to demonstrate the merit of our algorithm. We assume that NHT and all the versions of FHT have the same number of buckets, m. As we have seen, given same number of items, PFHT should consume exactly the same amount of off-chip memory as NHT and SFHT consumes slightly more memory due to the item replication. Therefore the only extra cost of our algorithm is the use of the on-chip memory to store the bucket counters.

The lookup performance of PFHT is difficult to analyze probabilistically. Hence we analyze the performance of our algorithm before pruning, i.e. we will use algorithm SearchItem$_{BFHT}$ for the purpose of analysis. It is important to note that the post-pruning data structure will always have less number of items in each bucket than the pre-pruning data structure. *Hence the lookup time on PFHT will always be shorter than the BFHT or equal.* This will be evident later when we discuss our simulation results. We also note that the SFHT has same lookup performance as BFHT.

3.1 Expected Linked List Length

For an NHT search when the linked list is not empty, let Y be the length of searched bucket. We have:

$$Pr\{Y = j|Y > 0\} = \frac{Pr\{Y = j, Y > 0\}}{Pr\{Y > 0\}} \quad (1)$$

$$= \frac{\binom{n}{j}(1/m)^j(1-1/m)^{n-j}}{1-(1-1/m)^n} \quad (2)$$

Now we analyze the distribution of the linked list lengths of FHT. It should be recalled that in order to store n items in the table, the number of actual insertions being performed are nk (or slightly less than that if same item could be hashed into same bucket by different hash functions), each of which is independent of each other. Under the assumption of simple uniform hashing, we can derive the average length of the list in any bucket.

With nk insertions in total, the probability that a bucket received exactly i insertions can be expressed as:

$$f_i = \binom{nk}{i}\left(\frac{1}{m}\right)^i\left(1-\frac{1}{m}\right)^{(nk-i)} \quad (3)$$

The question we try to answer now is: when the Bloom filter reports a match for a given query[1](i.e. all the k' counters > 0, where k' is the number of unique buckets for an item calculated by k hash functions. We know that $1 \leq k' \leq k$), what is the probability that the smallest value of the counter is j?

Let X denote the value of the smallest counter value among k' counter values corresponding to a query item *when all the counters are non-zero.* Hence,

$$Pr\{X = s\} = \sum_{j=1}^{k} Pr\{k' = j\} \times Pr\{X = s|k' = j\} \quad (4)$$

Let $d(j, r)$ be the probability that the first r hashes of an item produce exactly j distinct values. To derive $d(j, r)$, we know if the first $r-1$ hashes of an item have already produced j distinct values, the r^{th} hash has to produce one of the j values with probability j/m. Otherwise, the first $r-1$ hashes of an item must have already produced $j-1$ distinct values and the r^{th} hash should produce a value which is different from the $j-1$ values. The probability of this is $(m-(j-1))/m$. Hence,

$$d(j, r) = \frac{j}{m}d(j, r-1) + \frac{m-j+1}{m}d(j-1, r-1) \quad (5)$$

with the boundary conditions $d(j > r, r) = 0$, $d(0, 0) = 1$, $d(0, r > 0) = 0$. So, based on the fact that $Pr(k' = j) = d(j, k)$, now we can write

$$Pr\{X = s\} = \sum_{j=1}^{k} d(j, k) \times Pr\{X = s|k' = j\} \quad (6)$$

Now, let

$q(r, s, j)$ = Pr{smallest counter value in any r of the j buckets is s}

$p(i, j)$ = Pr{a counter value in a set of j non-empty buckets is i}

Since there is at least one item in any non-empty bucket, j non-empty buckets contain at least j items. We consider the probability to allocate the $i-1$ out of the rest $nk-j$ items in one of the j buckets to make the bucket has exactly i items. Thus,

[1]Note that this might be a false positive

$$p(i, j) = \binom{nk-j}{i-1}(1/m)^{(i-1)}(1-1/m)^{((nk-j)-(i-1))} \quad (7)$$

With these definitions, we can write

$$q(r, s, j) = \sum_{i=1}^{r}\binom{r}{i}p(s, j)^i \times \left(1-\sum_{h=1}^{s}p(h, j)\right)^{r-i} \quad (8)$$

This is because in r buckets we can have i buckets ($1 \leq i \leq r$) with counter value s while all other $r-i$ buckets have counter values greater than s. $q(r, s, j)$ is simply sum of the probability for each choice. The boundary conditions are $q(1, s, j) = p(s, j)$.

Putting things together we get:

$$Pr\{X = s\} = \sum_{j=1}^{k} d(j, k) \times q(j, s, j) \quad (9)$$

Based on Eq. 9, Figure 5 shows the linked list length comparisons of FHT and NHT. The figure tells us once we do need to search a non-empty linked list, what is the length distribution of these linked lists. In next section we use simulations to show the pruning and balancing effects which will improve the performance significantly.

It can be seen that given a probability of the inspected linked list length being within a bound, the bound on NHT is always larger than the bound on the FHT. For instance, with a probability of 10^{-3}, the NHT have about 3 items in a list where as FHT have only 2, thus improving the performance of NHT by a factor of 1.5. The improvement of the bound keeps getting better for smaller probabilities.

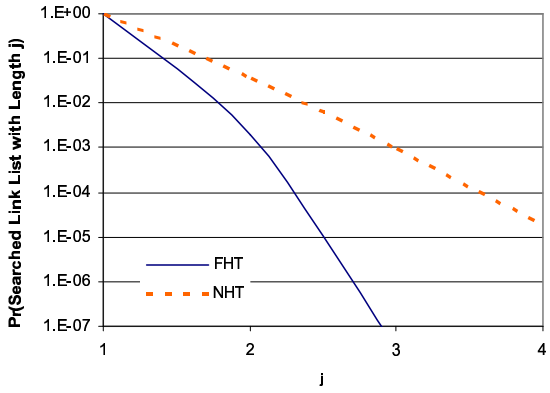

Figure 5: Probability distribution of searched linked list length: $n = 10,000$**,** $m = 128K$**.** $k = 10$ **for FHT.**

For the NHT, the expected number of buckets that the attached linked list exceeds a give length j is expressed as:

$$E(\#\ of\ buckets\ with\ length > j) = m \times B(n, 1/m, > j) \quad (10)$$

Where $B(n, 1/m, > j)$ is the the probability that a binomial random variable (or the load of a bucket) is greater than j:

$$B(n, 1/m, > j) = 1 - \sum_{i=0}^{j}\binom{n}{i}(1/m)^i(1-1/m)^{n-i} \quad (11)$$

For NHT, if the expected number of buckets with linked list length j is i, we can equivalently say that the expected number of items for which the bucket linked list lengths are j is $i \times j$. So the expected number of items for which the bucket linked list length $> j$ for an NHT can be expressed as:

$$E(\#\ of\ items\ for\ which\ bucket\ length > j) = \sum_{i=j}^{n}(i+1)(E(\#\ of\ buckets\ with\ length > i) - E(\#\ of\ buckets\ with\ length > (i+1)) = m\sum_{i=j}^{n}[(i+1) \times (B(n, 1/m, > i) - B(n, 1/m, > (i+1)))] \quad (12)$$

Now we derive the expected number of items in an FHT for which all buckets have more than j items (before pruning and balancing). We use an approximate expression for this:

$$E(\#\ of\ items\ for\ which\ all\ buckets\ length > j) = n \times B((n-1) \times k, 1/m, > (j-1))^k \quad (13)$$

The idea behind this is that if we consider a single item that hashes to k distinct buckets and a particular bucket for that item, the number of additional items that map to the same bucket is given by the binomial distribution with $(n-1) \times k$ trials. We can approximate the probability that all buckets for that item have $> j$ by raising this probability to the k-th power. This is not quite precise, since the probabilities for the sizes of the different buckets are not strictly independent. However, the true probability is slightly smaller than what we get by multiplying probabilities, so this gives us a conservative estimate. On the other hand, the expression is only approximate, since it assumes that all n items are mapped by the k hash functions to k distinct buckets. It's likely that for a small number of items, this will not be true, but we show through simulations that this does not have a significant impact on the results.

Figure 6 shows the expected number comparisons of FHT and NHT. This expected number tells us the number of items that are in link lists with at least j nodes.

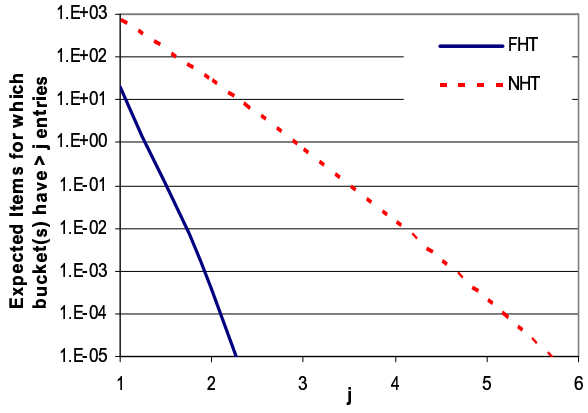

Figure 6: Expected number of items for which the searched bucket contains $> j$ items. $n = 10,000$, $m = 128K$. $k = 10$ for FHT.

The results show a definite advantage for the FHT even before the pruning and balancing optimizations. We can interpret that there is only two items in a billion for which the smallest bucket has more than 3 entries. For the NHT, there is about two items in ten thousands for which the bucket has more than 5 entries. Also in this configuration, only a few tens of items need more than 1 node access to query in FHT, but near 1000 items need more than 1 node access to query in NHT.

3.2 Effect of the Number of Hash Functions

We know that for an ordinary Bloom filter, the optimal number of hash functions k is related to the number of buckets m and the number of items n by the following relation [14]

$$k = \frac{m}{n} ln2 \quad (14)$$

Now we justify analytically why the same number of hash functions is also optimal for the FHT's lookup performance. From Equation 13, we know the expected number of items for which each bucket has more than j items. It is desirable to have at least one bucket with just one item in it. Hence we wish to minimize the probability of all the buckets corresponding to an item having more than one item. This translates into minimizing the following with respect to k.

$$B((n-1)k, 1/m, > 0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{(n-1)k}\right)^k \quad (15)$$

This expression is the same as the expression for the false positive probability of the ordinary Bloom filter containing $(n-1)$ items in m buckets [14]. Hence the optimal number of hash functions for the counting Bloom filters is given by

$$k = \frac{m}{n-1} ln2 \approx \frac{m}{n} ln2 \quad (16)$$

for a large number of items. Therefore, the optimal configuration of the ordinary Bloom filter for minimizing the false positive probability is the same as optimal configuration of FHT for reducing the item access time. Figure 7 shows the performance of FHT for different optimal configurations. For a fixed number of items n, we vary k and always ensure that m is optimally allocated for FHT. For each configuration we use the same number of resulting buckets for the NHT. The performance is compared for FHT and NHT. We can make two observations from the figure. First, the performance is always better if we have more buckets per item (i.e. larger m/n). Secondly, the performance of FHT is always significantly better than the NHT. This can be observed by comparing the curves H(1,3) and H(2,3), H(1,6) and H(4,6) and so on.

We also plot the performance when we use less number of hash functions than the optimal, and fix m and n. This is shown in Figure 8. The optimal number of hash functions for the configuration used is 10. Although the performance degrades as we use less than 10 hash functions, it is still significantly better than the NHT ($k = 1$ curve). An advantage of having a smaller number of hash functions is that the incremental update cost is reduced. Moreover, the associated hardware cost is also reduced.

3.3 Average Access Time

Load factor of a hash table is defined as the average length of lists in the table [9]. For an NHT, the load factor α can be given as:

$$\alpha = n/m \quad (17)$$

Let T_1, T_1^s and T_1^u denote the time for an average, successful and unsuccessful search respectively (ignoring the hash computation time). For an NHT, the following can be shown [9]:

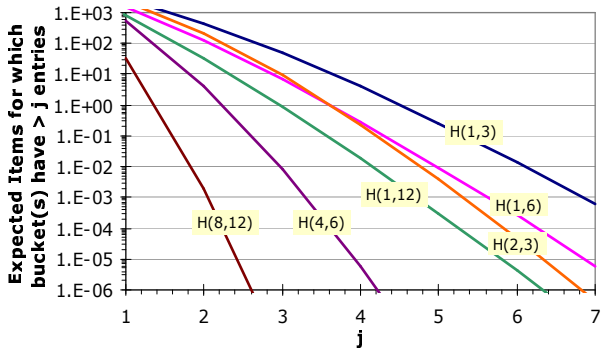

Figure 7: The effect of optimal configuration of hash table. $H(i,j)$ indicates $i = k$ and $j = m/n$. When $i = 1$, it implies an NHT.

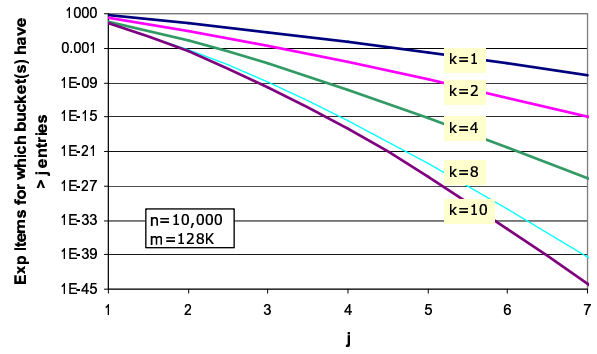

Figure 8: The effect of non-optimal configuration of FHT. $k = 1$ corresponds to the NHT.

$$T_1^s = 1 + \alpha/2 - 1/2m \tag{18}$$

$$T_1^u = \alpha \tag{19}$$

In order to evaluate the average search time, we need to introduce another parameter p_s which denotes the probability of a *true positive*, i.e., the frequency of searches which are successful. Similarly, $p_u = 1 - p_s$ denotes the frequency of issuing unsuccessful searches.

With these notations, the average search time can be expressed as:

$$\begin{aligned} T_1 &= p_s T_1^s + p_u T_1^u \\ &= p_s \left(1 + \frac{n-1}{2m}\right) + (1 - p_s)\frac{n}{m} \end{aligned} \tag{20}$$

For the FHT, let E_p be the expected length of linked list in the FHT for a member item and E_f be the expected length of linked list in the FHT for a false positive match. E_p can be derived form Equation (12) and E_f can be derived from Equation (9). So the average search time T_2 is:

$$T_2 = p_s E_p + p_u f E_f \tag{21}$$

$$= p_s E_p + (1 - p_s)\left(\frac{1}{2}\right)^{(m/n)ln2} E_f \tag{22}$$

We compare our algorithm with NHT scheme by using same set of configurations. Figure 9 shows the expected search time in terms of the number of off-chip memory accesses for the three schemes under different successful search rate.

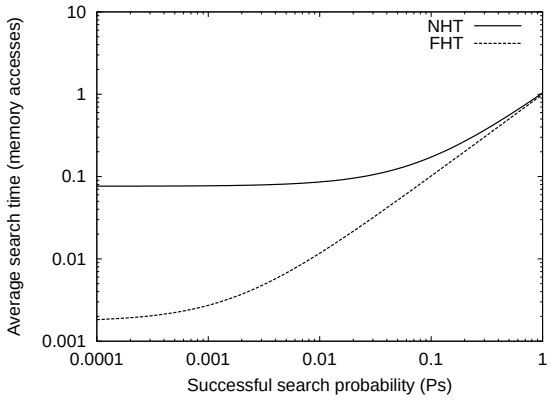

Figure 9: Expected search time for the NHT and FHT as a function of successful-search rate. $m = 128K$, $n = 10,000$. $k = 10$ for FHT

We see that the lower the successful search rate, the better the performance of our algorithm is. Note that this estimation is conservative for our algorithm. We do not take into account the potential benefit of some optimizations such as pruning and balancing.

3.4 Memory Usage

There are three distinct blocks in the FHT architecture which consume memory. The first is the on-chip counting Bloom filter. Second is the hash table buckets and the third being the actual item memory. In the analysis so far, we have always considered the same number of buckets for both the FHT and NHT. The NHT does not require on-chip memory though the FHT needs a small amount of it. Finally, while the NHT needs memory for exactly n items, the different versions of the FHT need different amount of memory depending on how many times an item is replicated. The BFHT needs to store each item k times hence needs a space of nk, and PFHT keeps exactly one node for each item hence the storage is same as the NHT. The SFHT trades off the memory for better incremental update support. We computed the memory requirement for SFHT using simulations with $m = 128K$, $n = 10,000$ and $k = 10$. Figure 10 shows the memory consumption of all the three schemes. The results show that for the chosen configuration, the SFHT uses 1 to 3 times more memory than NHT or PFHT, which is much less than the BFHT memory requirement.

We now elaborate on the memory usage for on-chip counters. The memory consumption for counter array depends on the number of counters and the width of each counter. While the number of counters is decided by Equation 14, counter width depends on how many items can get hashed to a counter. While in the worst case all the nk items can land up in the same bucket, this is highly improbable. Hence we calculate how many items can get hashed in a bucket on an average and choose a counter width to support it. For any counter that overflows by chance, we make a special arrangement for it. We simply keep the counter on the chip and attach the index of the counter in a small Content Addressable Memory (CAM) with just a few entries. When we want to address a counter, we check to see if it is one of the overflown counters and access it from the special hardware. Otherwise the normal operations proceed. Given the optimal configuration of counting Bloom filters (i.e. $mln2/n = k$) and $m = 128K$, we can show that the probability of a counter being > 8 to be 1.4e-6 which is minuscule for our purpose. In other words, one in a million counters can overflow

j	*Fast Hash Table*				*Naive Hash Table*	
	Analysis	*Simulation*			*Analysis*	*Simulation*
		basic	*pruning*	*balancing*		
1	19.8	18.8	5.60×10^{-2}	0	740.32	734.45
2	3.60×10^{-4}	4.30×10^{-4}	0	0	28.10	27.66
3	2.21×10^{-10}	0	0	0	0.72	0.70
4	1.00×10^{-17}	0	0	0	1.37×10^{-2}	1.31×10^{-2}
5	5.64×10^{-26}	0	0	0	2.10×10^{-4}	1.63×10^{-4}
6	5.55×10^{-35}	0	0	0	2.29×10^{-6}	7×10^{-6}

Table 1: Expected # of items for which all buckets have $> j$ entries. In the table, $n = 10,000$ and $m = 128K$. $k = 10$ for FHT

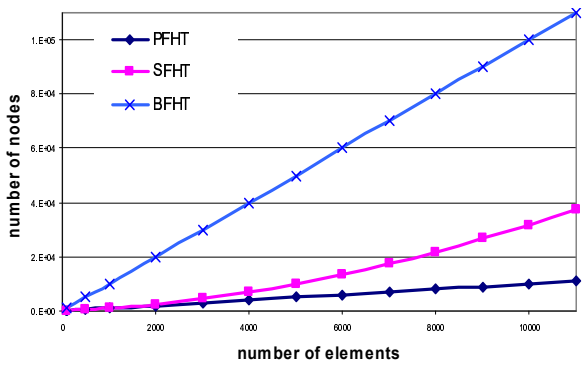

Figure 10: Item memory usage of different schemes. $m = 128K$, $n = 10,000$ and $k = 10$.

when we have only 128K counters. Hence we can comfortably choose the counter width of three bits and this consumes less than 400K bits of on-chip memories.

4. SIMULATIONS

We simulate the FHT lookup algorithm using different configurations and compare the performance with NHT under the condition that each scheme has the same number of buckets. Firstly, we need to choose a set of "good" hash functions. We will show in this section, even with a set of simple hash functions our algorithm demonstrates an appealing lookup performance that is much better than NHT. In the optimal case, our algorithm's successful lookup time is exactly 1 and the average unsuccessful lookup time equals to the false positive rate of the Bloom filter.

A class of universal hash functions described in [8] are suitable for hardware implementation [22]. For any member item X with b-bits representation as

$$X = \langle x_1, x_2, x_3, \ldots, x_b \rangle$$

the i^{th} hash function over X, $h_i(x)$ is calculated as:

$$h_i(X) = (d_{i1} \times x_1) \oplus (d_{i2} \times x_2) \oplus (d_{i3} \times x_3) \oplus \ldots \oplus (d_{ib} \times x_b)$$

where '$\times$' is a bitwise AND operator and '$\oplus$' is a bitwise *XOR* operator. d_{ij} is a predetermined random number in the range $[0 \ldots m-1]$. For NHT simulation, one of such hash functions is used.

We simulate the tail distribution of the expected number of items in a non-empty bucket which needs to be searched. The simulation ran 1,000,000 times with different seeds. In Table 1, we list both analysis results and simulation results.

From the table we can see that our analysis of the FHT and NHT are quite precise. The simulation results are very close to the analytical results and validate the correctness of our approximate analysis. More importantly, the FHT has already demonstrated advantages over the NHT. After the pruning and balancing, the improved results are indeed good: each non-empty bucket contains exactly one items. This means in the worst case, only one off-chip memory access is needed.

5. CONCLUSION

Hash tables are extensively used in several packet processing applications such as IP route lookup, packet classification, per-flow state management and network monitoring. Since these applications are often used as components in the data-path of a high-speed router, they can potentially create a performance bottleneck if the underlying hash table is poorly designed. In the worst case, back-to-back packets can access an item in the most loaded bucket of the hash table leading to several sequential memory accesses.

Among the conventional avenues to improve the hash table performance, using sophisticated cryptographic hash functions such as MD5 does not help since they are too computationally intensive to be computed in a minimum packet-time budget; devising a perfect hash function by preprocessing keys does not work for dynamic data sets and real-time processing; and multiple-hashing techniques to reduce collisions demand multiple parallel memory banks (requiring more pins and more power). Hence, engineering a resource efficient and high-performance hash table is indeed a challenging task.

In this paper we have presented a novel hash table data structure and algorithm which outperforms the conventional hash table algorithms by providing better bounds on hash collisions and the memory access per lookup. Our hash table algorithm extends the multi-hashing technique, Bloom filter, to support exact match. However, unlike the conventional multi-hashing schemes, it requires only one *external* memory for lookup. By using a small amount of multi-port *on-chip* memory we show how the accesses to the *off-chip* memory, either due to collision or due to unsuccessful searches, can be reduced significantly. Through theoretical analysis and simulations we show that our hash table is significantly faster than the conventional hash table. Thus, our Fast Hash Table can be used as a module to aid several networking applications.

6. REFERENCES

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

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

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

[4] Florin Baboescu and George Varghese. Scalable packet classification. In *ACM Sigcomm*, San Diego, CA, August 2001.

[5] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13, July 1970.

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

[7] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of IEEE INFOCOM*, 2001.

[8] L. Carter and M. Wegman. Universal classes of hashing functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1 edition, 1990.

[10] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2003.

[11] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. In *ACM Sigcomm*, August 2003.

[12] Sarang Dharmapurikar and Vern Paxson. Robust TCP stream reassembly in the presence of adversaries. In *USENIX Security Symposium*, August 2005.

[13] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better NetFlow. In *ACM Sigcomm*, August 2004.

[14] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8, March 2000.

[15] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings of IEEE INFOCOM*, 2000.

[16] HDL Design House. HCR_MD5: MD5 crypto core family, December, 2002.

[17] Intel Corporation. Intel IXP2800 Network Processor. Datasheet, 2002.

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

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

[20] Jan Van Lunteren. Searching very large routing tables in wide embedded memory. In *Proceedings of IEEE Globecom*, November 2001.

[21] Vern Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, December 1999.

[22] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[23] David V. Schuehler, James Moscola, and John W. Lockwood. Architecture for a hardware-based TCP/IP content scanning system. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2003.

[24] V. Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *SIGCOMM*, pages 135–146, 1999.

[25] V. Srinivasan and G. Varghese. Fast Address Lookup using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 1999.

[26] David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, and Jonathan Turner. System-on-chip packet processor for an experimental network services platform. In *Proceedings of IEEE Globecom*, 2003.

[27] B. Vocking. How asymmetry helps load balancing. In *Proceedings of 40th IEEE Symposium on Foundations of Computer Science*, 1999.

[28] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of ACM SIGCOMM*, 1997.

[29] Xilinx Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, November, 2004.