Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems

Sailesh Kumar
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-4306
sailesh@arl.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

ABSTRACT
Hash tables provide efficient table implementations, achieving O(1), query, insert and delete operations at low loads. However, at moderate or high loads collisions are quite frequent, resulting in decreased performance. In this paper, we propose the segmented hash table architecture, which ensures constant time hash operations at high loads with high probability. To achieve this, the hash memory is divided into N logical segments so that each incoming key has N potential storage locations; the destination segment is chosen so as to minimize collisions. In this way, collisions, and the associated probe sequences, are dramatically reduced. In order to keep memory utilization minimized, probabilistic filters are kept on-chip to allow the N segments to be accessed without in-creasing the number of off-chip memory operations. These filters are kept small and accurate with the help of a novel algorithm, called selective filter insertion, which keeps the segments balanced while minimizing false positive rates (i.e., incorrect filter predictions). The performance of our scheme is quantified via analytical modeling and software simulations. Moreover, we discuss efficient implementations that are easily realizable in modern device technologies. The performance benefits are significant: average search cost is reduced by 40% or more, while the likelihood of requiring more than one memory operation per search is reduced by several orders of magnitude.

Categories and Subject Descriptors
B.3.3 [**Performance Analysis and Design Aids**]: Simulation

General Terms
Algorithms, Design, Experimentation.

Keywords
Hash Table, lookup.

1. INTRODUCTION
Hash tables are used in a wide variety of applications. In networking systems, they are used for a number of purposes, including: load balancing [1, 2, 3, 4, 11], intrusion detection [27, 28], TCP/IP state management [24], and IP address lookups [12, 13]. Hash tables are often attractive since sparse tables result in constant-time, *O(1)*, query, insert and delete operations [6, 9]. However, as the table occupancy, or load, increases, collisions will occur which in turn places greater pressure on the collision resolution policy and often dramatically increases the cost of the primitive operations. In fact, as the load increases, the average query time increases steadily and very large worst case query times become more likely.

In this paper, we propose a hash table architecture, called a *segmented hash table,* that improves the performance of any collision resolution policy while minimizing memory bandwidth requirements. A segmented hash table uses multiple logical hash tables, or segments, to improve the distribution of keys. On an insert operation, for example, each segment contains a potential location for the key; of these available choices, one is chosen that minimizes collisions. To avoid the cost of multiple probes of the physical table, and hence increased memory traffic, efficient probabilistic filters are maintained in on-chip memory for each segment. With these filters, each segment may be considered on each operation without increasing the number of off-chip memory operations required. The filters can be kept small and accurate due to a novel algorithm, selective filter insertion, that directs insertion decisions in order to maintain balanced segments while reducing the number of false positives in the filters.

Considering multiple alternatives in this way significantly reduces the total number of collisions [1, 2, 3] and hence reduces the average and worst-case hash operation costs. Additionally, overall hash table performance is very deterministic despite load variations. This may result in lower latencies, higher system throughput, reduced memory requirements and reduced ingress command queue sizes to handle worst case conditions.

In a segmented hash table, the *average* performance is improved by 40% or more, depending on the table configuration. In fact, the average number of memory operations required is very near one, suggesting that no other scheme could improve on this average performance. The improvement in *worst-case* performance is more dramatic: with many segments, e.g. 16 or more, the likelihood of requiring more than one memory reference per search is reduced by several orders of magnitude and, in practical terms, becomes negligible.

The rest of the paper is organized as follows. Hash tables, collision resolution policies, and related work are discussed in Section 2. Section 3 describes the segmented hash table architecture and presents an analytical model for characterizing its performance. Sections 4 and 5 describe the segment filter implementations and algorithms for inserts and searches, respectively. Experimental results and comparison with few other hash table schemes are discussed in Section 6. Concluding remarks are found in Section 7.

2. Background and Motivation

In this section we present a brief description of hash tables. We borrow the notation, terminology and pseudo-code from [15] where further background can be found.

A table data structure supports insert, query and delete operations on some information *v* with an associated unique key *k*. When the range of keys is large, e.g., with the 96-bit keys used to uniquely identify TCP/IP flows, direct array-based implementations make inefficient use of memory and are infeasible. Hash tables efficiently implement tables in which a unique key *k* is used to store the key-information pair *(k, v)*. Hash tables use a *hash function h* [7] to map the universe *U* of keys into the buckets of a *hash table T*[0 . . *m*–1]:

$h : U \rightarrow \{0, 1, \ldots, m-1\}.$

the range of hash indices $h(k)$ are chosen to allow direct addressing into an array (entries in such an array are often called *hash buckets*). In this paper, we say that an element with key *k* hashes, or maps, to bucket $h(k)$; we also say that $h(k)$ is the hash value of key *k*. We let *m* denote the total number of buckets in the table and *n* the total number of keys to be inserted. With this, n/m is referred as the load factor α.

If the range of hash indices is smaller than the range of keys, it is possible that more than one key will map to same bucket; such occurrences are called *collisions*. There are systematic ways to resolve collisions, as discussed below. When keys are drawn from a random distribution (or when a cryptographic hash function is used), the likelihood of a collision depends on the load of the table. Under low loads hash tables provide constant *O(1)* average query time. The excellent average query time makes hash tables the implementation of choice in a wide variety of applications.

2.1 Collision resolution policies

Hash algorithms are distinguished by their collision resolution policies [8, 15]. Broadly, two categories exist: linear chaining and open addressing.

Linear chaining stores all the elements that hash to the same bucket in a linked-list, and the pointer to the head of the linked-list is kept at the hashed index. Below we summarize the insert, search and delete operations in chaining.

CHAINED-HASH-INSERT(T, x)
Insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, x)
Search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
Delete x from the list $T[h(key[x])]$

The simplicity of linear chaining comes at a cost, however. Pointers must be maintained in every bucket to provide links. This creates overhead in the form of additional storage and bandwidth for recording and fetching the pointers. Open addressing policies, on the other hand, do not use list pointers and thus avoid their overheads.

The simplest open addressing policy is *linear probing*, in which the key is inserted in the first empty subsequent table entry from the hashed bucket In a linear order). Thus, given the primary hash function $h' : U \rightarrow \{0, \ldots, m-1\}$, linear probing uses following hash function to successively index the table

Table I: Average number of probes per search

Hashing policy	Successful	Unsuccessful
Linear probing	$\frac{1}{2}+\frac{1}{2(1-\alpha)}$	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$
Double hashing	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$
Linear chaining (assuming zero pointer overhead)	$1+\frac{\alpha}{2}$	α
Linear chaining (pointer size = k times the key size)	$1+\frac{(1+k)\alpha}{2}$	$(1+k)\alpha$

$h(k, i) : (h'(k) + i) \mod m$

for i =.0, 1, . . . , m – 1. While easy to implement, linear probing suffers from primary clustering, where long runs of occupied buckets build up, increasing the average search time. *Double hashing* is another well known open addressing scheme which ensures nearly perfect random permutations [29] and avoids clustering. Double hashing uses two hash functions h_1 and h_2 and the table is successively indexed with the following function

$h(k, i) : (h_1(k) + i h_2(k)) \mod m$,

for i =.0, 1, . . . , m – 1. However, the catch is that $h_2(k)$ results must be relatively prime to the hash-table size m else the entire table can't be covered. An easy way to achieve this is to let m be a power of 2 and to ensure that h_2 produces an odd number. Another way is to let m be a prime number itself.

Thus with open addressing schemes, every key k results in the following probe sequence

$\{h(k,0), h(k,1), \ldots, h(k,m-1)\}$

which is a permutation of $\{0, 1, \ldots m-1\}$ The resulting pseudo-code for inserts in open addressing is shown below

---

OPEN-HASH-INSERT(T, k)
$i \leftarrow 0$
repeat $j \leftarrow h(k, i)$
  if $T[j]$ = NIL | DELETED
    then $T[j] \leftarrow k$
      return j
    else $i \leftarrow i + 1$
  until $i = m$
error "hash table overflow"

A search operation naturally probes the same sequence of buckets that the insertion algorithm examined. A search terminates unsuccessfully [20] when an empty bucket is encountered, however, this makes the deletion from an open-addressed hash table difficult since when a key from a bucket is deleted, the bucket can't simply be marked as empty. A popular solution is to mark the bucket by storing in it a special DELETED marker. The search procedure then has to be modified such that it keeps on looking when it sees the value DELETED, while the insert procedure treats such a bucket as if it is empty. Some earlier works [17, 18, 19] also suggests the need of periodic rehashing for probing or double hashing. This is because, at high loads, keys are inserted farther from the index *h(k)* and when keys closer to indices are deleted, query time remains high. Table I summarizes the average search times [16] of hash tables with double hashing, probing and chaining.

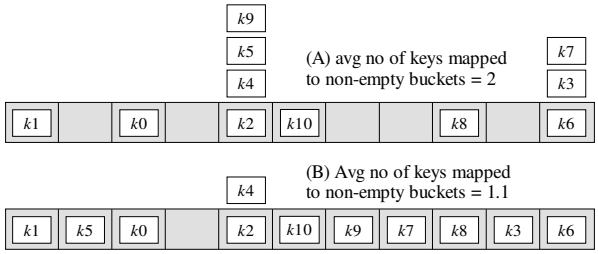

Figure 1: 11 keys in 11 buckets A) placed randomly, B) placed with a reduced collision probability. Keys shown outside are inserted as per the collision policy

2.2 Discussion

The two classes of collision resolution policies: open addressing and linear chaining have their own advantages and disadvantages. Linear chaining ensures that both successful and unsuccessful search takes $\Theta(1+\alpha)$ time on the average. It has also been shown that, in linear chaining, the expected length of the longest chain, when $n = m$, is $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$, which is also the expected worst-case search time. Linear chaining, therefore, results in good average and worst-case performance however it incurs the overheads of storing and managing the pointers and heads of the chain.

With open addressing, the extra memory freed by not storing pointers provides the hash table with a larger number of buckets for the same amount of memory, potentially yielding fewer collisions and faster searches. However the flip side is that, in open addressing, every collision creates an opportunity for another collision since the colliding element is put into another empty bucket. The expected worst-case performance of open addressing schemes is also worse when compared to that of chaining. It can be shown that, for less than half the maximum load, i.e. for $n < m/2$, the expected length of the longest probe sequence is $\Theta(\lg n)$ [5], which is lg lg m times that of chaining at full load. As $n > m/2$, the expected worst-case performance of open addressing gets even worse.

2.3 Opportunities for Improving Performance

The average and worst-case performance of a hash table depends on the way keys get hashed to buckets; the average number of hashed keys presents a lower limit on the average probe sequence, or collision, length. If the average and the worst-case numbers of keys mapped to a bucket were reduced by some mechanism, then performance can be improved. Consider a case when 11 keys are randomly hashed into 11 buckets. A naive hash table results in an average of 2 keys at every non-empty bucket as shown in Figure 1a. If by some mechanism, every bucket had, for example, only 1.1 keys mapped (also illustrated in Figure 1b), the average query time can be reduced to 12/11. The foundation of the segmented hash table is to ensure lower average and worst-case keys per bucket.

This is achieved by logically segmenting the hash table into several smaller tables, and considering a bucket from each segment for every arriving key. This results in multiple choices, which can be exploited to reduce collisions. In our scheme, these multiple choices are considered in efficient fashion via on-chip filters implemented with the high speed embedded memory technologies available in modern ASICs and FPGAs. For example, it is now possible to integrate several megabits of low latency, high bandwidth embedded memories into chips. IBM, for instance, has recently advertised an ASIC fabrication technology, which boasts of up to 300M bits of embedded memory [30]. Modern FPGAs like the Xilinx Virtex-4 contains up to 400 memory blocks each with 18K bits [31]. These embedded memories can be used to implement our desired filters.

2.4 Related Work

A considerable amount of related work has focused on the theoretical aspects of improving hash table performance. The power of multiple choices has been studied by several researchers; [1, 2, 3, 4] are some of the earliest results to suggest that the number of collisions can be reduced significantly with two random choices. These studies argue that using two hash functions to index the hash table, and subsequently choosing the one with fewest keys, reduces collisions and hence probe sequence lengths. Another study [18] examines the use of two independent tables, accessed serially, and shows that proper placement and access strategies can enhance overall performance. Parallel hashing has also been proposed. For example, [21, 22] briefly considers using multiple parallel indices (like a set associative cache array) to reduce the number of collisions and therefore the search time.

Using multiple hash functions to improve the hashing performance has also been considered in [23] where it is argued that if the underlying hardware allows the both the parallelization of hash function computations and memory reads to different banks then lookups can be performed in a single memory cycle. Another study [13] describes an approach for obtaining good hash tables for IP lookups based on using multiple hashes of each input key.

While most of these results are compelling, they focus on latency while ignoring memory bandwidth. In the packet processing context, where memory bandwidth is at a premium, it is important to minimize the number of memory references needed for each hash operation. A recently proposed architecture [10], called the fast hash table, ensures with high probability that a single memory reference is used on every search (i.e. one bucket is read).

Our proposed segmented hash table also ensures with high probability the use of a single off-chip memory reference for every hash operation. Furthermore, with the aid of the algorithms used to perform the insertions, our scheme greatly improves performance and significantly reduces the cost of implementation, measured in on-chip memory bits, as compared to fast hash table. We present this comparison in Section 6.

3. Segmented hash

A traditional hash table maps an incoming key to a single hash bucket. An N-way segmented hash table maps an incoming key onto a set of N potential buckets, one from each segment; this choice allows a segmented hash table to spread keys more evenly over the table and reduce collisions. We now consider in greater detail how hash operations are implemented.

3.1 Hash Operations

An N-way segmented hash table with capacity m consists of N equally sized logical segments, T_i, each containing m/N buckets. An incoming key, k, is hashed with a function, h, that maps the key space U onto the range of segment indices. That is, $h : U \rightarrow \{0, 1, \ldots, m/N - 1\}$. The key may be inserted at position $h(k)$ in

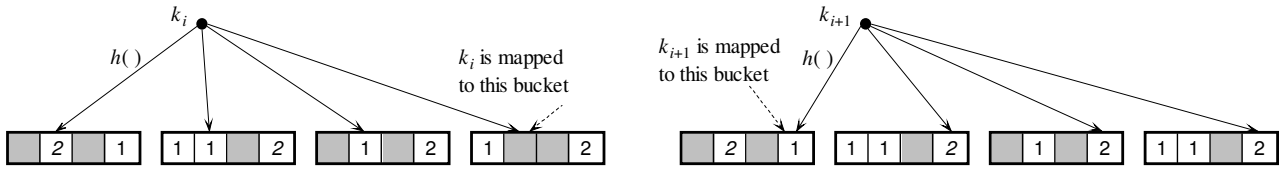

Figure 2: Schematic of a 4-way segmented hash. Values in buckets indicate the number of keys mapped to them. Grey buckets have no keys mapped. Insertion policy chooses the bucket with minimum number of mapped keys

any one of the N segments. If exactly one segment has an empty bucket at $h(k)$, then the key will be inserted there. However, a further decision must be made if either none are empty or multiple buckets are free.

If all segments are occupied at entry $h(k)$, then a collision is unavoidable. The new goal is to add the key to the shortest collision chain. Thus, the minimum collision chain length must be determined among all segments. If exactly one segment has a collision chain of minimum length, the key will be inserted there. However, if multiple segments have min-length collision chains, then a further decision must be made.

In fact, this is a generalization of the case of multiple empty buckets, in which multiple segments have collision chain lengths of 0. At this point in the discussion, we are free to break this tie arbitrarily; i.e., one can be chosen at random. However, we will shortly leverage this decision to optimize our implementation. Two inserts into a 4-way segmented hash table are illustrated in Figure 2. The pseudo-code of the insert procedure with open addressing is shown below.

```
INSERT(T, k)
i ← 1
repeat j ← h(k, i)
    if SEGMENT-INSERT(k, j) then
        return 1
    else i ← i + 1
  until i = m/N
error "All segments are full"

SEGMENT-INSERT(k, j)
i ← 1
repeat
    if T_i[j] = NIL | DELETED
        then T_i[j] ← k
            return 1
        else i ← i + 1
  until i = N
return −1
```

Given the preceding discussion of inserts, query operations are readily understood. To query for key k, each segment must be probed at position $h(k)$. If k is not found at that position, the collision chains in each segment are followed. Either the key is found or the query is unsuccessful. Delete operations are handled in a similar fashion. We now build an analytical model to quantify the benefits of using multiple segments to reduce collisions.

3.2 Analysis of Collision Conditions

A collision in a segmented hash table occurs when a key k arrives, but the bucket indexed by $h(k)$ is occupied in all segments. Thus a collision occurs when at least N+1 keys k_i exist whose $h(k_i)$ are the same. This suggests that collisions should reduce as N increases. However, note that the hash function $h : U \rightarrow \{0, 1, \ldots, m/N - 1\}$, now maps keys to a set which is N times smaller, which may suggest that collisions should increase. Hence we present a brief analysis of collision conditions in the segmented hash table. First we arrive at the expected number of buckets which receive a given number of keys, x. The total number of keys in buckets receiving x keys will be the product of the number of these buckets and x. We use this key result to arrive at the equations for the average distance and the distribution of keys in a segmented hash table.

Lemma 1. If n identical keys are randomly inserted into m buckets, then the expected number of keys lying in all buckets which have exactly x keys mapped into them is

$$< f(x) > \;=\; xm\binom{n}{x}\frac{(m-1)^{n-x}}{m^n}$$

Proof: Refer to the Appendix.

Theorem 1. In a N-way segmented hash table, the expected number of keys mapped to all buckets which has exactly x keys mapped to them is

$$< e(x) > \;=\; \sum_{i=xN}^{(x+1)N}\left[\frac{im}{N}\times\binom{n}{i}\frac{N^i(m-N)^{n-i}}{m^n}\right]$$

Proof: A segmented hash with n keys, and hash function $h : U \rightarrow \{0, 1, \ldots, m/N - 1\}$, can be viewed as hashing n identical keys randomly to m/N bucket groups (a bucket group has N buckets, one from each segment). With this, segmented hash table insertion policy ensures that, every set of N arriving keys into a bucket group will be mapped uniformly across the N buckets. Thus, i^{th} key is mapped to a bucket only when iN keys are already preset in its bucket group. Hence, inserting $m = m/N$ in lemma 1 and summing $< f(i) >$ from $i = xN$ to $(x+1)$ N, we get the above theorem. ■

The average number of keys mapped to a bucket determines the lower bound on the average distance of keys from their indices. From theorem 1, the expected number of keys at a distance more than d is the probability that a key is inserted at a distance more than d, and is given by n_{d+}/n, where

$$n_{d+} \;=\; \sum_{x>d}\sum_{i=xN}^{(x+1)N}\left[\frac{(i-dN)m}{N}\times\binom{n}{i}\frac{N^i(m-N)^{n-i}}{m^n}\right]$$

$$=\; \sum_{i>dN}\left[\frac{(i-dN)m}{N}\times\binom{n}{i}\frac{N^i(m-N)^{n-i}}{m^n}\right]$$

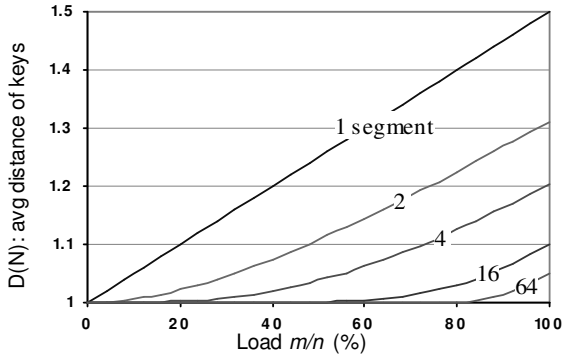

Figure 3: Average distance of keys from hash buckets

We plot this probability in Figure 3 for *d* equal to 1 and in Figure 4 for *d* equal to 2. It is apparent that as *N* increases, the likelihood that keys are inserted far from their indices reduces rapidly. Note that linear chaining ensures that keys remain at a minimum distance from its indices (i.e., the head of its chain), hence the above probability directly applies to it. However, the average distance of keys in open addressing hashing schemes are slightly higher, as we will see later.

Theorem 2:

The lower bound of the average chain length (i.e. distance of a key from its index) in a segmented hash table is

$$D(N) = \frac{1}{n}\sum_{i=0}^{n} \frac{<e(i)>}{i}\left[\left(1+\left\lfloor\frac{i}{N}\right\rfloor\right)\times\left\lfloor\frac{i}{N}\right\rfloor\times\left(\frac{N}{2}\right)+(i \bmod N)\times\left\lceil\frac{i}{N}\right\rceil\right]$$

Proof: In a segmented hash table, keys with the same hashed index $h(k)$ are uniformly distributed across the *N* segments. Thus, every set of *N* keys that hashes to the same index is put in the *N* buckets (one from each table) at the hash index and the subsequent sets sees a minimum increase in the distance by 1. Thus, *x* keys hashing to an index results in $\lceil x/N \rceil$ sets of *N* keys, of which the i^{th} set, $i \in [1, \lfloor x/N \rfloor]$ is *i* hops away from the index and the last set of (*x mod N*) keys is $\lceil x/N \rceil$ hops away. Therefore, the average distance of the *x* keys which hashes to the same index in a segmented hash table is

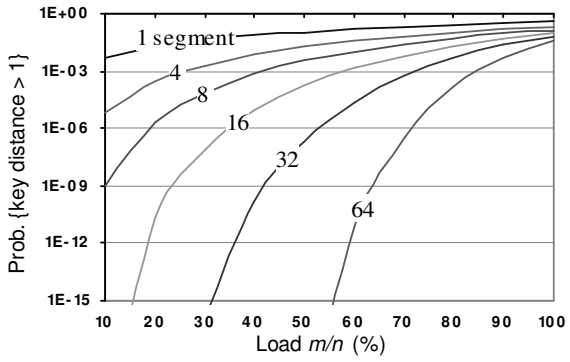

Figure 4: Plotting Prob. {distance of a key exceeds 1}

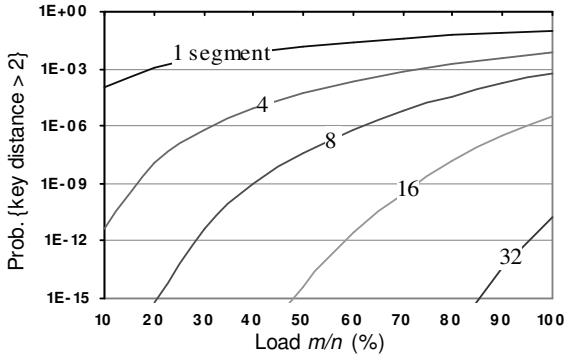

Figure 5: Plotting Prob. {distance of a key exceeds 2}

$$<d(x)> = \frac{1}{x}\left[\sum_{i=1}^{\lfloor x/N \rfloor} i \times N + (x \bmod N)\times\left\lceil\frac{x}{N}\right\rceil\right]$$

Summing with an arithmetic progression yields,

$$<d(x)> = \frac{1}{x}\left[\left(1+\left\lfloor\frac{x}{N}\right\rfloor\right)\times\left\lfloor\frac{x}{N}\right\rfloor\times\left(\frac{N}{2}\right)+(x \bmod N)\times\left\lceil\frac{x}{N}\right\rceil\right]$$

Now, $<d(x)>$ is the average distance of keys in buckets with exactly *x* keys and $<e(x)>$ is the number of such keys. Multiplying them and taking an average by summing over all *n* keys, we get the above theorem. ■

Since we were unable to get a closed form solution of $D(N)$ for *N* greater than 1, we have plotted D(*N*) in Figure 5. It essentially presents the lower bound on the average number of probes required for search operations in a segmented hash table. For *N = 1,* inserting *x = 1/m and y = (m-1)/m*, and using the binomial theorem:

$$(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k}$$

and its derivatives with respect to *x*, the average distance of keys from their hashed index in a hash table with single segment can be shown to be

$$D(1) = 1+\left(\frac{n-1}{2m}\right)$$

which is the average distance of keys in a single hash table.

Using the above approach, other segmented hash table performance primitives can be easily quantified. We note that collision conditions in a segmented hash table improve dramatically when compared to a naïve, single segment table.

3.3 An Obvious Deficiency

The previous section clearly shows that multiple segments dramatically reduce the number of collisions. However, the presence of multiple segments requires that they be probed for each operation. Consider, for example, a key query in an *N*-way table. If we assume that each segment is equally loaded, then, on average, *N*/2 segments will need to be probed before the key is found. This is not likely to be an improvement on the O(1) average case performance of a naïve hash table. Of course, if each segment can be searched in parallel, the query time will be

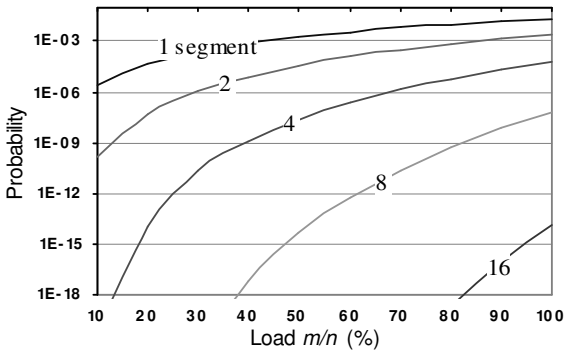

Figure 6: Probability that a 2-bit counter for the chain length overflows in segmented hash table for various *N*. This is also the probability that chain length exceeds 3

excellent, but at the cost of an *N*-fold increase in memory bandwidth. Increasing bandwidth by this amount is not feasible in most applications.

For these reasons, the segmented hash table maintains filters for each segment so that, on average, only one segment is examined per operation. To support insert operations, a bit vector is used to maintain bucket occupancies. To support queries, a modified Bloom filter is used at each segment to determine if a key is present. The following two sections describe these mechanisms.

4. Ensuring *O*(1) Inserts

Recall that insert operations require finding the segment with the shortest collision chain. To avoid probing multiple segments, we use a supporting data structure. The implementation of this structure differs based on the collision resolution policy, so we describe them separately beginning with open addressing.

For open addressing, we build a bit vector that keeps a bit for every bucket in the hash table. The bit vector is organized as words of *N*-bits, where each bit indicates the availability of the bucket in the corresponding segment. For instance, in an 8-way segmented hash table, words will have 8-bits and a word 11001101 at an address *h*, indicates that the bucket indexed by *h* is free in segments 1, 4 and 5. These status bits can be read instead of probing the segments. If none of the bits are zero, i.e., none of the indexed buckets are free, the subsequent set of buckets is probed by reading the availability status bits along the probe sequence. Thus, by reading a single word of memory, the occupancy of all segments is known.

However, linear chaining is not amenable to such a bit vector, because the buckets in the linked-list chain at different segments may be located at physically different addresses even if the linked list itself are for the buckets with the same hash index. This holds for any free list-based chain implementation, where free buckets at every segment are put into and allocated from a free linked list. This might lead to probing *N* status words.

Therefore, we use a vector of counters for chaining in which a counter is kept at each bucket storing the length of the linked-list chains beginning there. The counter is incremented whenever a key is added to the linked-list chain and decremented whenever a key is deleted. For each insert, the counters at the target bucket in each segment are read, and the segment with the smallest counter is chosen. The corresponding counter is then incremented and stored back.

The data structure layout is similar to that of the bit vector. If the width of the counter is *w,* then the word length for the vector of counters needs to be $N \times w$ and so many bits are read and written back for every insert. Fortunately, *w* can be limited to 2 (thus chain lengths can either be 0, 1, 2 or 3) and it can be shown that the probability of occurrence of a counter overflows is sufficiently small when there are large number of segments, *N*.

From Theorem 1, the expected number of keys at distance of more than 3 from the hashed bucket will be determine the probability of counter overflow. Thus, the probability that a 2-bit counter will overflow is n_{3+}/n, where,

$$n_{3+} = \sum_{x>3} \sum_{i=xN}^{(x+1)N} \left[\frac{(i-3N)m}{N} \times \binom{n}{i} \frac{N^i (m-N)^{n-i}}{m^n} \right]$$

$$= \sum_{i>3N} \left[\frac{(i-3N)m}{N} \times \binom{n}{i} \frac{N^i (m-N)^{n-i}}{m^n} \right]$$

We plot this overflow probability in Figure 4 for different values of *N*. It is apparent that the probability of an overflow event is almost negligible once 16 segments are used. For smaller *N*, the counter width can be scaled to either 3 or 4 to achieve a low overflow probability (e.g. with 8 segments, a 3-bit counter overflows with a probability of 2.22E-41, at 100% load). In the unlikely event of counter overflow, all that happens is that arriving keys are inserted into a suboptimal segment, where the chain at the hashed bucket is already long.

4.1 Storing the Vector

The status vectors can be easily stored in an off-chip memory. The amount of memory required will be 1 bit per hash bucket in open addressing and *w* bits per bucket in chaining, where *w* is the counter width. The memory word width can be set at either *N* or $N \times w$ bits respectively; consequently for an insert, the average number of probes into this memory will be equal to the average probe sequence length. Since the average probe sequence lengths are nearly one with a segmented hash table, the performance of inserts remains highly deterministic. For example, when there are 64 logical segments, the probability that an insert requires more than one memory probe is less than 1E-6 at 80% load.

5. Ensuring *O*(1) SearchES

For key searches, instead of searching the segments directly, we maintain on-chip predictive filters that can be quickly queried to determine which segment, if any, holds the key. For this purpose, we use a modified Bloom filter at every segment. In this section, we describe Bloom filters and our enhancement which dramatically reduces the probability of false positives.

5.1 A Brief Description of Bloom filters

Bloom filters [25] support space-efficient membership queries. A set, $S = \{x_1, x_2, \ldots, x_n\}$, of *n* elements is represented with an array, *v,* of *m* bits, initially all set to 0. A set of *k* independent hash functions, $h_1, h_2, \ldots, h_k$, each with range $\{1, 2, \ldots, m\}$, are used to set the bits at positions $h_1(x), h_2(x), \ldots, h_k(x)$ for all *x* in set *S*. In other words, for each item in the set, *k* random but deterministic bits in *v* are set. Clearly, a particular bit might be set to 1 multiple times. To search for item *k*, bits at positions $h_1(k)$,

$h_2(k)$, . . . , $h_k(k)$ are checked; if any of these are 0, then k is certainly not present in the set. Otherwise, it is assumed that k is in the set. However, since random hash functions are used, false positives are possible. For example, imagine that enough keys are inserted that all bits in v have been set; in this case, every search will be successful whether or not the key was actually inserted. The false positive rate is a clear consequence of the design parameters chosen, namely, n, m, and k. Specifically, the probability of a false positive, or the false positive rate, is

$$\left(1-\left(\frac{1}{m}\right)^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k$$

The right-hand side is minimized for $k = \ln 2 \times m/n$, which implies that an optimum k exists for each choice of n and m. The false positive probability for optimum k becomes

$$\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}$$

In practice, of course, k must be an integer, and a smaller than optimum k might be preferred to reduce the number of hash computations required.

In general, there is a clear inverse tradeoff between amount of memory and false positive rate. For instance when m/n is 8, i.e. 8 bits are allocated for every key, an ideal value of $k = 5$ (i.e., 5 hash functions) achieves a false positive rate of about 2%. Instead, if we allocate 16-bits for every key, the optimal choice of 10 hash functions yields a false positive rate of 0.04%, whereas using 5 hash functions yields 0.1%.

Bloom filters do not support delete operations, which precludes their use when the set S changes over time. The problem is that a given bit may have been set by multiple keys, and to unset that bit would effective delete all of them, rather than the single key that should be deleted. An extension has been developed called the counting Bloom filter [26] that solves this problem by keeping a counter at each entry of the vector v (instead of just a bit). When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. In order to avoid counter overflow, counters must to be dimensioned properly. It has been shown [26] that for k less than or equal to the optimum number of $\ln 2 \times m/n$, the probability that the counter exceeds any given number i is

$$\Pr\{\max(c) \geq i\} \leq m\left(\frac{e \ln 2}{i}\right)^i$$

Taking $i = 32$, which implies a 5-bit counter, we obtain

$$\Pr\{\max(c) \geq 32\} \leq 4.5 \times 10^{-42} \times m$$

Thus 5 bits per counter is amply sufficient. Note that when searching, all that matters is whether a given counter is non-zero. Furthermore, the counters are only modified on inserts and deletes, not during searches. Therefore, on-chip implementations of counting bloom filters can often benefit from maintaining a bit vector that records the status (i.e., non-zero or not) of each counter. Provided that searches are significantly more frequent than inserts/deletes, this bit vector can be kept on-chip, and the counters can be kept in off-chip memory.

This is the scheme used in the segmented hash table Bloom filters. We maintain both a counter array, $\boldsymbol{c}_i$ and a bit array $\boldsymbol{v}_i$, for each segment i.

5.2 False Positive Rate with Multiple Segments

The impact of using accurate filters at each segment is profound: the benefit of dramatically improved key distribution is now decoupled from the need to probe multiple segments. If the filters are perfectly accurate, the results from Section 3.2 suggest that there is an extremely low probability that more than one memory reference will be required, even for tables with loads of 80-90%. The now critical metric is the false positive rate; a high false positive rate could instigate spurious memory references.

The discussion in the preceding section directly relates to the false positive rate of a single Bloom filter. Our scheme uses N independent Bloom filters, one for each segment. Therefore, for any given key search, the effective false positive rate has increased by a factor of N since there are now N opportunities for a false positive. In fact, the increase is somewhat greater than N since many combinations must be considered. The probability of exactly i false positives is as follows.

$$\Pr\{i \text{ false positives}\} = \binom{N}{i} \times f^i \times (1-f)^{N-i},$$

where f^i is the false probability rate of a single segment. Consequently the overall probability of a false positive is

$$\Pr\{\text{false positive}\} = \sum_{i=1}^{N} \binom{N}{i} \times f^i \times (1-f)^{N-i}.$$

The consequence of this increase in false positive rate is that a designer would need to increase the size of the Bloom filters for a segmented hash table in order to maintain a given false positive rate. Fortunately, we can modify our insertion algorithm in such a way that we dramatically reduce the chance of false positives. As described in the next section, our insert algorithm achieves false positive rates *lower* than those seen in a single Bloom filter.

5.3 Selective Filter Insertion Algorithm

The key idea behind our insertion scheme, which we refer to as *selective filter insertion*, is to minimize the number of non-zero counters in each Bloom filter while keeping the segments equally balanced. Recall that in our discussion of inserts in Section 3.1, if multiple segments had min-length collision chains at the target bucket, one could be chosen at random. In place of this policy, selective filter insertion chooses the segment that will create the fewest number of non-empty Bloom filter counters. The intuition is simple: the false positive rate grows with the percentage of non-zero counters. Selective filter insertion keeps that percentage minimized. We now describe the algorithm in greater detail.

A segmented hash table maps every arriving key k to one bucket from each segment, that is to the set $\{h^{T1}(k), h^{T2}(k), \ldots, h^{TN}(k)\}$. As explained earlier, the goal is to choose the bucket from that set that has the smallest collision chain. Since several segments may have a bucket with a min-length collision chain, we let $\{minSet\}$ denote those segments.

To choose the target segment from $\{minSet\}$, we next examine the contents of each segment's Bloom filter. To reduce the probability of false positives, we choose the segment in $\{minSet\}$ for which the insertion of key k creates the fewest number of non-zero counters in its Bloom filter. This approach will act to maximize the number of counters with value zero, and, hence, will minimize the false positive rate.

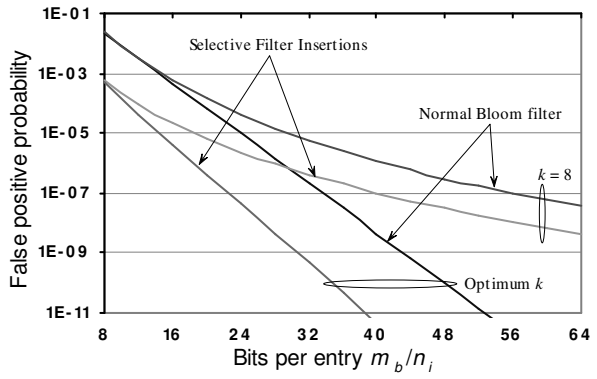

Figure 7: Plotting false positive probability of individual hash table segments. It is apparent that using Selective Filter Insertions reduces the false positive probability of each segment by 2 to 4 orders of magnitude

A careful examination suggests that the number of segments in {*minSet*} will be roughly proportional to the number of empty buckets in the entire table. We use this important property to choose segments such that the false positive rate of the associated Bloom filters is reduced dramatically. Below, we present the outline of the segment selection algorithm.

Low false positive rates can be achieved by ensuring that only a few bits in the Bloom filter array v_i are set to 1. Recall that a bit in v_i is only 1 of the corresponding counter is non-zero. For every arriving key, our algorithm chooses a segment i from the {*minSet*} such that the associated Bloom filter array v_i has the minimum number of bit transitions. If the i^{th} segment were chosen, the number of bit transitions t_i in the Bloom filter v_i will be

$t_i = k - (v_i[\mathrm{h}_1(k)] + v_i[h_2(k)] + \ldots v_i[h_k(k)])$,

where k is the number of hash functions and $v_i[j]$ is the bit value at index j in v_i. Thus a naive algorithm can simply choose a segment i from the set {*minSet*} which minimizes t_i.

While intuitively correct, the algorithm as described will lead to unbalanced segments. Consider that an arriving key is more likely to be inserted in a segment which is already heavily loaded, with more bits set in its Bloom filter. For example, when initially all segments are empty and an arriving key leads into a segment x, then all subsequent keys are more likely to lead to the same segment, as there may be fewer transitions there, at least until collisions mount and those segments exit {*minSet*}. Thus, the scheme will not achieve the desired reduction in false positive rate. Care must be taken to fill segments uniformly. To do so, we introduce an occupancy parameter, δ, to choose eligible sets from {*minSet*}, as described in the algorithm below.

1. *Label segments in the set {minSet} eligible if its occupancy is less than (1+δ) times the occupancy of the least occupied segment. δ is typically set at 0.1 to 0.01.*

2. *If no segment remains eligible, select the least occupied segment from {minSet}.*

3. *Otherwise choose a segment from {minSet}, which is labeled eligible with minimum t_i*

4. *If multiple such segments exist, choose the one which is least occupied.*

5. *If multiple such segments are again found, break the tie with a round-robin arbitration policy.*

Using software simulation, we have quantified the benefits of selective filter insertion on individual segments. Figure 7 reports the false positive rates of individual Bloom filter segments (not the total across all) in a segmented hash table with 64 segments, both with and without our algorithm. In these experiments, δ was set at 0.1 but the results are not sensitive to that value; any small value will suffice. It is apparent from the plots that the false positive rate is reduced by 2 to 3 orders of magnitude with selective filter insertion algorithm. We note that improvement diminishes, albeit slowly, as N is reduced. For instance, in the case of 16 segments with optimal k, when 40-bits are kept for every bucket, the false positive rate is reduced by about 100x, as compared to the 600x reduction seen with 64 segments.

In Table 2, we show the overall false positive rates across all segments in a segmented hash table for a few combinations of system parameters. The effectiveness of the selective filter insertion algorithm is profound*: the overall false positive rates of the segmented hash table are reduced by up to three orders of magnitude, even if compared to the false positive rate of individual segments in a naïve approach*.

Due to space restrictions, we are not presenting the analytical treatment of the selective filter insertion algorithm. However we plan to publish it as an extended technical report.

5.4 Bloom Filter Implementation Details

The implementation of a segmented hash table with its requisite counting Boom filters is straightforward. For every hash table segment i, a counting Bloom filter is maintained. In addition to the counter array, $\boldsymbol{c_i}$, a bit array $\boldsymbol{v_i}$ is used to record whether each entry in $\boldsymbol{c_i}$ is empty. The counter array is manipulated on inserts and deletes, while the bit array is used to support queries. The counters indexed by each of the k Bloom filter hash functions are incremented during every insert (the segments are chosen according to the selective filter insertion algorithm) and are decremented upon deletion. A non-zero counter is reflected in the bit array v_i by setting the corresponding bit to 1. Bits whose corresponding counter is zero are reset to 0. If each Bloom filter has m_b counters and counter widths are 5, the total number of bits needed for both arrays is $6 \times m_b$. Of the two arrays c_i and v_i, only v_i is accessed during search; c_i is accessed during deletes and inserts.

Array v_i is kept on-chip since these bits are looked up in the data path (i.e., on every search). For applications where the delete and insert traffic are sufficiently low (e.g. managed by the control plane), the array c_i can be stored in an off-chip memory. On the other hand, when delete and insert traffic are comparable to the search traffic, both v_i and c_i must be kept in on-chip memories. The selective filter insertion algorithm is of great help in both scenarios, as it reduces the size of the Bloom filter for a given false positive rate target. For example, a false positive rate of 1.0E-5 can be achieved by using about 32 bits per bucket in v_i for an unmodified filter. With the selective filter selection algorithm, only 20 bits per bucket are needed.

Table 2: Expected number of false positives in segmented hash

No of segments, N	Bits per bucket, m_b/n	Naïve Bloom Filter			Selective Filter Insertion Algorithm		
		1 × Pr. (1 false)	2 × Pr. (2 false)	*Expected # of false positives*	1 × Pr. (1 false)	2 × Pr. (2 false)	*Expected # of false positives*
16	8	0.2	0.08	*0.3*	0.02	0.0006	*0.02*
	16	0.007	5×10^{-5}	*0.007*	0.0002	1×10^{-7}	*0.0002*
	32	3×10^{-6}	8×10^{-12}	3×10^{-6}	4×10^{-8}	2×10^{-15}	4×10^{-8}
64	8	0.4	0.5	*1.5*	0.03	0.001	*0.03*
	16	0.03	0.0008	*0.03*	0.0002	7×10^{-12}	*0.0002*
	32	1×10^{-5}	1×10^{-10}	1×10^{-5}	3×10^{-8}	1×10^{-15}	3×10^{-8}

In order to efficiently structure the on-chip bit array, we take advantage of the fact that we use the same set of k independent hash functions for each of the N Bloom filters; i.e., for a given key, the same bit positions are accessed in each segment's Bloom filter. Thus, memory words can be composed of a bit from each segment; each row indicates the index, and each column contains a bit from a different segment. This organization is illustrated in Figure 8.

With this word construction, the memory must still allow k accesses per cycle, one for each Bloom filter hash function. This can be achieved with a single array by using memories which run at relatively higher clock frequencies. This approach is common with today's VLSI technology; however, low clock frequencies improve power consumption and reliability. Alternatively, several smaller memories can be used, each accessed in parallel. With this structure, the k independent hash functions must return an equal number of indices for each memory. We achieve this by adding an offset to any hash functions which collide within a memory. With a fixed offsetting policy, whenever the same key is searched for, the hash function collides and is resolved in the same manner; therefore, correct operation is ensured. Our simulation results suggest that this small modification results in comparable performance to the case when unconstrained hash functions are used.

The Bloom filter's counter array c_i, on the other hand, can be implemented with any memory word organization because counters of multiple segments are never accessed together (either incremented or decremented). During inserts, the selective filter insertion algorithm chooses the segment and its counter array is updated. During deletes as well, only counters in the corresponding segment are decremented. The question of where to keep the counter array, on-chip or off-chip, depends solely upon the particular system requirement. Most applications in networking, such as intrusion detection, route lookup, per flow state management, and packet classification, manage the hash table through the control plane, which is often slow compared to the data plane. Off-chip memory, for these applications, is the right place to keep the counter array. However, there are applications, like TCP offloading, where insert and delete traffic are comparable to search traffic, which necessitate on-chip counter storage.

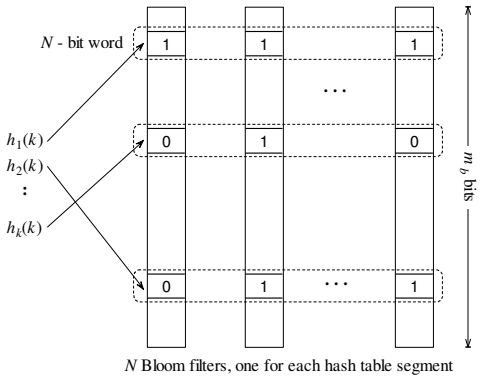

Figure 8: Memory word construction of Bloom filter array. Each word has N bits, one from each Bloom filter.

5.5 Discussion

As mentioned earlier, we have argued that {*minSet*} contains more than $N/2$ segments on an average. The intuition behind this is based on the fact that, with hash table segmentation, the average number of keys mapped to a bucket is around one. Hence, the expected number of buckets available to a randomly inserted key at load α is equal to $1 - \alpha$ times N. In practice, however, hash tables typically do not operate at full load since off-chip memory capacity can be economically increased to reduce load, therefore, the average number of bucket choices remains high. In the next section, we present experimental results conducted on the entire system where the actual characteristics of {*minSet*} are captured.

6. Simulations and Analysis

In this section, we present experimental results based on software simulation that quantify the relative performance of segmented hash tables. First, we discuss experiments that report the improvement in probe sequence lengths in support of the analytical measures derived in Section 3.2. Next, we quantify the performance of our Bloom filters with selective filter insertion.

In these experiments, all hash tables contain a total of 64K buckets (i.e. if there are 64 segments, each segment will have 1024 buckets in it). The simulation consists of 500 phases in which periods of random searches are alternated with periods of deletes and inserts. Initially, a number of keys are inserted into the table according to the load factor. At the start of each phase, 100,000 random keys are searched. At the end of each phase,

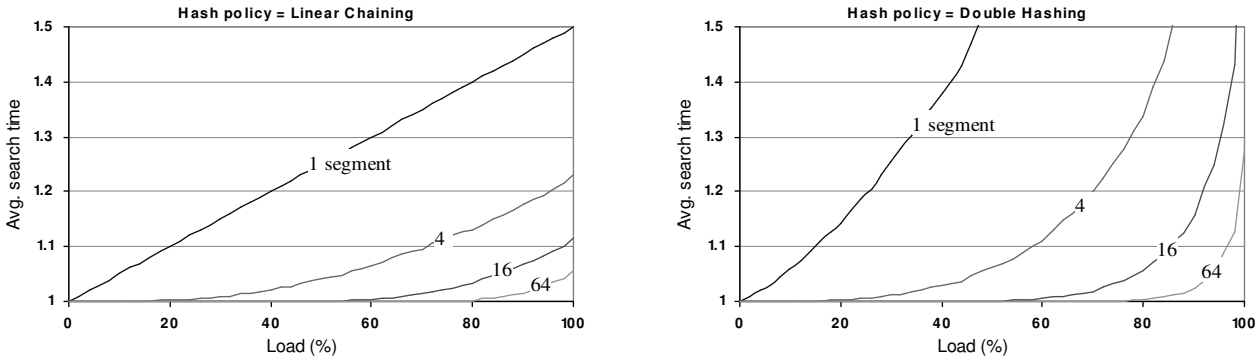

Figure 9: Plotting the average search time (in number of memory probes). Bloom filter has 32 bits/entry and uses *selective filter insertion*. There are 64K buckets in the entire table and above plots represent 500 phases. During every phase, 100,000 random searches are performed. Between every phase 10,000 random keys are deleted and inserted.

Table 3: PDF of the search time (in number of memory probes) in the above simulation setup and at a 60% load

Collision Resolution Policy	Number of Segments	1 probe	2 probes	3 probes	4 probes	5+ probes
Linear Chaining	*1*	0.753208	0.20279	0.037909	0.005424	0.000669
	4	0.933988	0.065518	0.000495	1.95×10^{-06}	0
	16	0.995744	4.26×10^{-05}	0	0	0
	64	1	0	0	0	0
Double Hashing	*1*	0.600754	0.219539	0.093876	0.043257	0.042574
	4	0.902454	0.082002	0.012407	0.002349	0.00079
	16	0.994315	0.00056	3.02×10^{-06}	0	0
	64	1	0	0	0	0

10,000 random keys are deleted and another set of 10,000 random keys are inserted. Including inserts and deletes are particularly important to evaluate double hashing performance, since the management of delete markers can be troublesome if not implemented well (as discussed in Section 2.1).

Figure 9 reports the average number of memory references required versus hash table load, for various segment counts. Results for both linear chaining and double hashing are presented. In this experiment, the selective filter algorithm is used and the Bloom filters provide 32 bits per entry. We chose 32 bits per entry to reduce the effect of false positives, which are examined in the next set of experiments. As can be seen, the performance closely matches that predicted by our analytical model in Figure 3, particularly for linear chaining. For double hashing, collisions increase exponentially with load. In both cases, increasing the number of segments greatly decreases the average. In fact, with 64 segments, both collision resolution policies maintain an ideal average of 1 to a load of 80%, yielding an improvement of 40% for linear chaining and 2-3x for double hashing, as compared to a traditional, single segment hash table.

In order to explain the distribution that yields these average values, Table 3 reports the PDF of references at 60% load. As the number of segments increases, the distribution tightens dramatically. In fact, once 64 segments are used, 100% of the 5M searches performed require only a single memory reference. With 16 segments, the likelihood of requiring 2 memory operations has been reduced by 5 and 3 orders of magnitude for linear chaining and double hashing, resp.

It is apparent from the plots that, segmenting the single hash table into multiple logical tables improves both the average and the expected worst case performance dramatically. Hashing operations become highly deterministic as the number of segments increases.

Recall, however, that a segmented hash table must have very accurate per-segment filters in order to benefit from the previous results. In order to illustrate the effectiveness of the selective filter insertion algorithm in reducing false positives, we generate traffic patterns which include a percentage of unsuccessful searches. Ideally, our filters will avoid off-chip hash table references on unsuccessful searches. However, in practice, off-chip references can't be avoided completely due to non-zero false positive rates. Clearly, the effectiveness of the filter architecture is demonstrated by how often it can avoid the off-chip reference.

Among our alternatives, we consider four hashing architectures: a) a hash table with a single segment, b) a hash table with 64 logical segments and a naïve Bloom filter set, c) a hash with 64 logical segments and a Bloom filter set which using the selective filter insertion algorithm, and d) a fast hash table [10], as introduced in Section 2.4. In all Bloom filters, either 16 or the optimal number of hash functions are used, whichever is smaller.

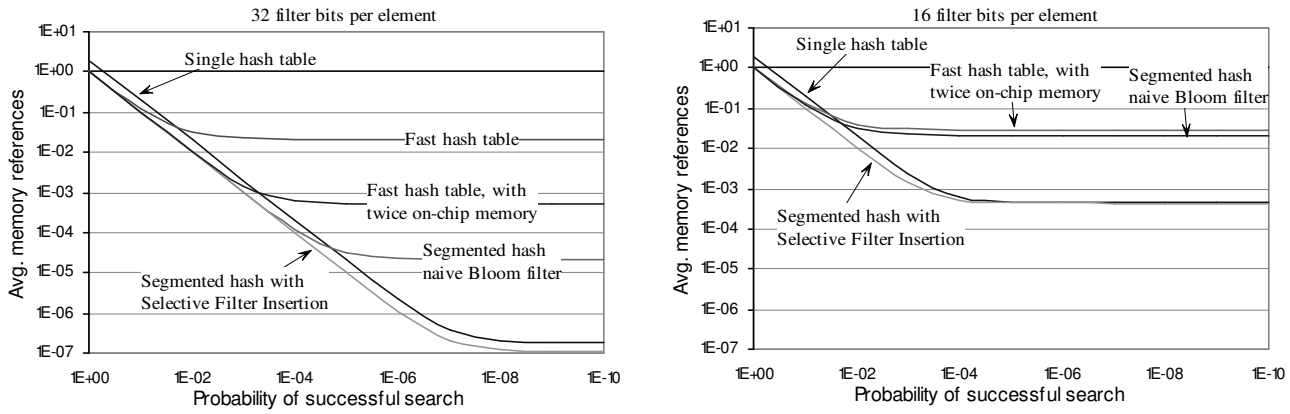

Figure 10: Plotting the average memory references at different successful search rates. Lower memory references reflect the effectiveness of the filter. Load is kept at 80%. These results are based upon simulation of the entire system. Note that fast hash table is assumed to be storing the heads of the linked-list chains on-chip, while segmented hash uses double hashing.

In order to ensure fair comparisons, we allocate an equal amount of on-chip memory to each architecture. In the first experiment, we allocate 2M bits of on-chip memory to support a hash table containing 64K keys; thus 32 bits of on-chip memory are available for every key. In the second experiment, we allocate 1M bits of on-chip memory for all policies but the fast hash table, which still has 2 M bit on-chip memory because that scheme cannot operate with 1 M bit of memory, and we keep the maximum number of keys the same. Thus, the fast hash table has 32-bits on-chip per key, while the others have 16-bits.

The results are illustrated in Figure 10. As can be seen, the segmented hash table with selective filter insertion outperforms all of the other multi-filter architectures. It should be noted that the single hash table outperforms the other schemes because it has only one Bloom filter; in fact, the selective filter insertion algorithm allows the segmented hash table to outperform the single hash table over much of the experimental range.

6.1 Linear Chaining or Double Hashing?

With a segmented hash table, the collision policies are more or less equally effective up to very high loads, around 90%. Thus, the additional memory overheads required to support pointer management in linear chaining are probably not justifiable. With this organization, double hashing will achieve equivalent performance with considerably less memory.

7. Concluding Remarks

In this paper, we have introduced the segmented hash table architecture. A segmented hash table improves the distribution of keys and, consequently, reduces collisions as compared to a traditional hash table. By dividing the hash memory into *N* logical segments, each incoming key can be placed into one of *N* possible locations; the ultimate destination is chosen so as to minimize collisions. To avoid multiple off-chip memory operations, a probabilistic filter is kept on-chip for each segment. These filters effectively minimize the off-chip bandwidth required to perform inserts, deletes and searches. The filters are efficient, i.e., small and accurate, in large part due to selective filter insertion algorithm that directs insertion decisions in order to maintain segment balance and reduce the occurrence of false positives in the filters. The resulting performance improvements are significant. When a linear chaining policy is used with 64 segments, average case performance is improved by 40% or more at high loads; with double hashing, the improvement at high loads is a factor of 2-3. The improvement in worst-case performance is even greater. With 16 or more segments, the likelihood of requiring more than one memory operation per search is reduced by many orders of magnitude, typically between 3 and 5 depending on the table configuration.

8. Acknowledgements

This work was supported in part by NSF grants CCF-0430012 and CNS-0325298 and by a gift from Intel Corp.

References

[1] Y. Azar, A. Broder, A. Karlin, E. Upfal. Balanced Allocations, Proc. 26th ACM Symposium on Theory of Computing, 1994, pp. 593–602.

[2] M. Mitzenmacher, The Power of Two Choices in Randomized Load Balancing, Ph.D. thesis, University of California, Berkeley, 1996.

[3] Y. Azar, A. Z. Broder , A. R. Karlin , E. Upfal, Balanced allocations (extended abstract), Proc. ACM symposium on Theory of computing, May 23-25, 1994, pp. 593-602.

[4] M. Adler, S. Chakrabarti, M. Mitzenmacher, L. Rasmussen, Parallel randomized load balancing, Proc. 27th Annual ACM Symposium on Theory of Computing, 1995, pp. 238-247.

[5] G. H. Gonnet, Expected length of the longest probe sequence in hash code searching, Journal of ACM, 28 (1981), pp. 289-304.

[6] M. L. Fredman, J. Komlos, E. Szemeredi, Storing a sparse table with O(1) worst case access time, Journal of ACM, 31 (1984), pp. 538-544.

[7] J. L. Carter, M. N. Wegman, Universal Classes of Hash Functions, JCSS 18, No. 2, 1979, pp. 143-154.

[8] P. D. Mackenzie, C. G. Plaxton, R. Rajaraman, On contention resolution protocols and associated probabilistic phenomena, Proc. 26th Annual ACM Symposium on Theory of Computing, 1994, pp. 153-162.

[9] A. Brodnik, I. Munro, Membership in constant time and almost-minimum space, SIAM J. Comput. 28 (1999) 1627–1640.

[10] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," SIGCOMM, Philadelphia PA, August 20-26, 2005.

[11] B. Vocking, How Asymmetry Helps Load Balancing. Proc. 40th IEEE Symposium on Foundations of Comp. Science, 1999, pp. 131-141.

[12] M. Wadvogel, G. Varghese, J. Turner, B. Plattner. Scalable High Speed IP Routing Lookups, Proc. of SIGCOMM 97, 1997.

[13] A. Broder, M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups", IEEE INFOCOM, 2001, pp. 1454-1463.

[14] W. Cunto, P. V. Poblete: Two Hybrid Methods for Collision Resolution in Open Addressing Hashing, SWAT 1988, pp. 113-119.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press, 1990.

[16] P. Larson, Dynamic Hash Tables, CACM, 1988, 31 (4).

[17] M. Naor, V. Teague. Anti-persistence: History Independent Data Structures. Proc. 33nd Symposium on Theory of Computing, May 2001.

[18] R. Pagh, F. F. Rodler, Cuckoo Hashing, Proc. 9th Annual European Symposium on Algorithms, August 28-31, 2001, pp.121-133.

[19] D. E. Knuth, The Art of Computer Programming, volume 3, Addison-Wesley Publishing Co, second edition, 1998.

[20] L. C. K. Hui, C. Martel, On efficient unsuccessful search, Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 217-227.

[21] Goto, Ida, and Gunji, "Parallel hashing algorithms", Information Processing Letters, Vol. 6, No. 1, Feb. 1977.

[22] Hiraki, Nishida, and Shimada, "Evaluation of associative memory using parallel chained hashing", IEEE Tran. on Software Engineering, Vol. 33, No. 9, pp. 851--855, Sept. 1984.

[23] A. Broder and A. Karlin, "Multilevel adaptive hashing," ACM-SIAM Symposium on Discrete Algorithm, 1990.

[24] G. R. Wright, W.R. Stevens, TCP/IP Illustrated, volume 2, Addison-Wesley Publishing Co., 1995.

[25] Burton H. Bloom, :Space/time trade-offs in hash coding with allowable errors", Communications of the ACM, v.13 n.7, p.422-426, July 1970

[26] Li Fan , Pei Cao , Jussara Almeida , Andrei Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol", IEEE/ACM Transactions on Networking (TON), v.8 n.3, p.281-293, June 2000.

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

[28] 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.

[29] George S. Lueker, Mariko Molodowitch, "More analysis of double hashing," Combinatorica 13(1): 83-96 (1993).

[30] Cu-11 standard cell/gate array ASIC, IBM.

[31] Virtex-4 FPGA, Xilinx.

Appendix A: Proof of Lemma 1

Lemma 1. If n identical keys are randomly inserted into m buckets, then the expected number of keys lying in all buckets which have exactly x keys mapped into them is

$$< f(x) > \;=\; xm\binom{n}{x}\frac{(m-1)^{n-x}}{m^n}$$

Proof: Let i be the index to buckets, and $n(i)$ be the number of keys in each bucket. We define,

$$f(x) \;=\; \sum_{i=0}^{m-1} n(i)\times\delta(n(i)\text{ - }x),$$

where the function δ = 1 if its argument is 0, and 0 otherwise. $f(x)$ is the total number of keys lying in buckets with exactly x keys, $x\in[0,n]$. The mean of f is given by

$$< f(x) > \;=\; \sum_{n} f(n)\times\Pr(n)$$

where, <> indicates expectation and the sum is over all possible configurations $\{n(i)\}$ with $\sum_{i=0}^{m-1} n(i) = n$. Below, $\{n\}$ is used to denote the set of all $n(i)$, $\{n(i)\}$.

The probability of a configuration $\Pr(i)$ is given by the multinomial distribution. Substituting for $f(n)$ and reversing the order of the sums, we have

$$< f(x) > \;=\; \sum_{i=0}^{m-1}\sum_{\{n\}} n(i)\times\delta(n(i)-x)\times\Pr(n)$$

But now, for each value of i in the first sum, we sum over all values of $n(j)$ but only those $n(j)=x$ will contribute non-zero values to these sum. Thus, those $n(j)\neq x$ can be dropped from the expression.

$$< f(x) > \;=\; \sum_{i=0}^{m-1}\sum_{n(i)} n(i)\times\delta(n(i)-x)\times\Pr(n(i))$$

where now $\Pr(n(i))$ is the marginal probability of the number of keys in basket i. The effect of the function δ is just to restrict the sum over $n(i)$ to values equal to x.

$$< f(x) > \;=\; \sum_{i=0}^{m-1}\sum_{n(i)=x} n\{i\}\times\Pr(n(i))$$

The marginal distribution of $n\{i\}$ is the same for all i by symmetry, so sum over i just introduces a factor of M

$$< f(x) > \;=\; m\sum_{n(i)=x} n(i)\times\Pr(n(i))$$

The final step is that the marginal distribution for $n(i)$ is just a binomial distribution with $p = 1/m$.

$$\Pr(n(i)) = \binom{n}{n(i)}\left(\frac{1}{m}\right)^{n(i)}\left(\frac{m-1}{m}\right)^{n-n(i)}$$

Thus the expected number of keys in buckets with exactly x keys, where $x\in[0,n]$, is

$$< f(x) > \;= x\times m\times\binom{n}{x}\left(\frac{1}{m}\right)^{x}\left(\frac{m-1}{m}\right)^{n-x}$$

$$= xm\binom{n}{x}\frac{(m-1)^{n-x}}{m^n} \qquad \blacksquare$$

Note that, this looks like the derivative of a binomial series with $p = 1/n$ and n trials with x success.