Using Multiple Hash Functions to Improve IP Lookups

Andrei Broder, Michael Mitzenmacher

Abstract— **High performance Internet routers require a mechanism for very efficient IP address look-ups. Some techniques used to this end, such as binary search on levels, need to construct quickly a good hash table for the appropriate IP prefixes. In this paper we describe an approach for obtaining good hash tables based on using multiple hashes of each input key (which is an IP address). The methods we describe are fast, simple, scalable, parallelizable, and flexible. In particular, in instances where the goal is to have one hash bucket fit into a cache line, using multiple hashes proves extremely suitable. We provide a general analysis of this hashing technique and specifically discuss its application to binary search on levels.**

I. Introduction

We describe a new hashing approach suitable for use in network routing software and hardware. This hashing approach can be applied to improve IP lookups using the technique of binary search on levels to find the longest matching prefix. In particular, we expect that this approach will prove highly suitable for IP-v6 addresses (when combined with previous techniques such as prefix expansion), and for new programmable network processors [4]. We expect that it will also be useful for similar problems, such as packet classification and filtering, where hashing is commonly used as a subroutine to allow fast lookups [13].

The basic idea of the approach is to use multiple hash functions. The idea has been analyzed and developed in several recent theoretical works. We therefore specifically address how this approach can be used to improve performance on the real problem of IP lookups. In particular, we emphasize that by properly structuring the data, one can parallelize memory accesses so that using multiple hash functions is desirable.

We test the performance of our approach through extensive simulation, including experiments on real data from a snapshot of the MaeEast database.

Andrei Broder: AltaVista Company, 1825 S. Grant Street, Suite 410, San Mateo, CA 94402, USA. This work was done while at Compaq Systems Research Center, Palo Alto.
E-mail: `andrei.broder@altavista.com`.
Michael Mitzenmacher: Harvard University, Computer Science Department, 33 Oxford St., Cambridge, MA 02138. Part of this work was done while visiting Compaq Systems Research Center.
E-mail: `michaelm@eecs.harvard.edu`.

A. Hashing for IP lookups

The standard approach used by an IP router to forward a packet is to keep a forwarding table based on IP destination address prefixes. Each prefix is associated with the next hop towards the destination. The IP router looks in its table for the longest prefix that matches the destination address of the packet, and forwards according to that match.

One attack for solving the longest matching prefix problem is to perform binary search on levels [14], [17]. We briefly review the main ideas.[1] Prefixes are divided according to length, with all prefixes of a given length in a table. We then perform a binary search for matching prefixes of the destination address according to prefix lengths. 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. Tables for each prefix length can be stored as a hash table. In this case, if there are W different possible prefix lengths and n different prefixes, the search requires $O(n \log_2 W)$ memory and $O(\log_2 W)$ time. This technique is enhanced by using the process of controlled prefix expansion in order to reduce the number of distinct prefix lengths, as described by Srinivasan and Varghese [14]. If the number of distinct prefix lengths used is only ℓ instead of the W possible, then only $\log_2 \ell$ table lookups are required, instead of $\log_2 W$. This reduces the search to $O(\log_2 \ell)$ time; the amount of memory used depends on the increase in the number of prefixes. Srinivasan and Varghese suggest that from experiments on real data, the possible increase in the number of prefixes does not lead to large increases in memory requirements [14].

The binary search on levels scheme depends on being able to create suitable hash tables in order to minimize the number of memory accesses. Since a memory access requires reading in a cache line, a natural goal is to ensure that the number of items that fall in a bucket corresponds to the capacity of a single cache line, so that each hash bucket corresponds to a cache line of memory. This ensures that each level examined during the binary search only requires a single memory access. Srinivasan and Varghese therefore suggest searching for a "semi-perfect" hash function where each bucket has only c collisions, where c is the number of items that can fit in a single cache line [14]. In

[1] Of course there are other possible attacks for this problem as well, as detailed for example in [2], [14].

their case, $c = 6$.

One potential problem with the above method is that finding a suitable semi-perfect hash function can be a slow process. As reported in [14], for the MaeEast database of IP addresses, constructing such a hash function took almost 13 minutes. The authors argue that this time may not be a problem, as prefixes change rarely enough that this computation can be done off-line. Note that if one attempts to handle table modifications on-line, there is the possibility that the capacity of a bucket could be exceeded by an unfortunate selection of values to be hashed. Such a problem could be handled by choosing a new hash function and re-hashing all entries; however, if finding a suitable hash function requires significant time, this is not desirable.

A related potential problem is that the above scheme potentially wastes significant memory. When some buckets have fewer than six elements, space is wasted for cache lines that do not hold their full contingent of items.

Our hashing scheme is designed to solve the problems introduced by searching for a semi-perfect hash function, by instead using multiple hash functions. The approach is very general and hence should prove highly suitable for IP-v6 addresses (when combined with previous techniques such as prefix expansion), as well as other similar lookup problems that use hashing.

B. Multiple hash functions

For some time it has been known that using multiple hash functions can lead to different performance behavior than using a single hash function. One of the first analyses suggested using multiple tables, with a separate function for each table. Elements that collide in one table percolate to the next. The tables shrunk in size and the hashes could be computed in parallel [3].

A seminal result in the area considered the following natural hashing scheme [1], which we here call the *d-random scheme*. Suppose that n items are hashed sequentially into a table with n buckets, in the following manner. Each item is hashed using d hash functions, which we assume yield independent and identically distributed buckets for each item. The item is placed in the least loaded bucket (that is, the bucket with the fewest items); ties are broken arbitrarily. A search for an item now requires examining the d possible buckets; however, as shown in [1] the maximum load in a bucket (with high probability) is $\frac{\log \log n}{\log d} + O(1)$. This compares quite favorably to the situation where just one hash function is used, in which case the maximum load is $\frac{\log n}{\log \log n}(1 + o(1))$ (with high probability). The key point of this result is that using two hash functions leads to a completely different behavior than using a single hash function, while three is not too much different from two. Besides improving the maximum load, using two hash functions in this way leads to a more equal distribution of the load across buckets. A numerical analysis of this hashing process is given in [11], and extensions to queueing models are presented in [10], [9], [16].

The hashing scheme we examine here is a variation of the d-random scheme that offers better performance and is more suitable for the IP lookup problem. It was first introduced and analyzed theoretically by Vöcking [15]; a simpler analysis more relevant to our discussion was developed by Vöcking and Mitzenmacher [12].

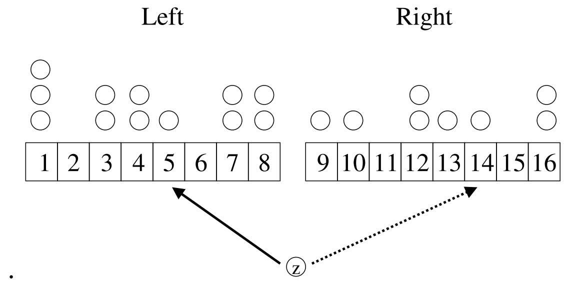

Fig. 1. The 2-left scheme. A newly inserted item, labeled z, is placed in the less filled of two random buckets, one from the left and one from the right. Ties are broken to the left. A search for z may require searching both of the buckets in which z might have been placed.

II. Multiple hashes: the d-left scheme

We begin by focusing on the case of two hash functions. The scheme we describe was introduced by Vöcking in [15] and is referred to as the *2-left* scheme in [12]. Our hash table consists of n buckets. (We assume n is even.) We split the n buckets into two disjoint equal parts, which for convenience we call the left and the right.[2] When an item is inserted, we call both hash functions, where each hash function has a range of $[1, n/2]$. The first hash function determines a bucket on the left, the second a bucket on the right. The item is placed in the bucket with the smaller number of existing items; in case of a tie, the item is placed in the bucket on the left. In order to do a lookup, one must examine the contents of the two possible buckets corresponding to the two hashes of an item.

An obvious disadvantage of this approach is that it requires two hash table lookups for each level. Note, however, that these lookups are independent, in that they can be performed in parallel. Specifically, if the hash table is placed into memory so that the left and right parts of the table are guaranteed to map to different memory areas,

[2] We emphasize that "left" and "right" are terms chosen simply for convenience; the point is simply that the table consists of two disjoint parts.

then accessing the two buckets corresponding to an item can naturally be pipelined. For example, in software one might arrange so that the left side of the table corresponds to even cache lines and the right side to odd cache lines. Alternatively, in hardware one could store different parts of the table in distinct memory bank subsystems. Hence we do not feel that the requirement that two memory accesses are required will have an important negative performance impact. (Similarly, if the machine can issue multiple instructions, then the two buckets may be searched for the item in parallel as well.) We show that in return for this price, we obtain significant benefits.

We may generalize the above to more hash functions, with the d-left scheme using d hash functions. Initially the n buckets of the hash table are divided into d groups of n/d buckets. (Again, we assume n/d is an integer.) We think of the groups as running consecutively from left to right. An incoming item is placed in the bucket with the smallest number of existing items; in case of a tie, the item is placed in the bucket of the leftmost group with the smallest number of items. In order to search for an item in the hash table, the contents of d buckets must be checked. Again, the corresponding memory lookups can easily be pipelined. We show that by increasing the number of hash functions used, one can reduce the memory required for the hash table at the potential expense of more (pipelined) memory accesses and computation.

An interesting question is why we suggest that ties be broken towards the left, rather than breaking ties randomly as in the d-random scheme. Surprisingly, the asymmetry introduced by breaking ties toward the left actually improves performance, in that the maximum number of items placed in a bucket is smaller (in a probabilistic sense) when one breaks ties in this manner. The intuition for this improvement is that as items are added, the cases where there are ties are extremely significant. For example, suppose the largest load thus far is four. In order to obtain a bucket with load five, we must choose two buckets with load four. Ties are therefore necessary to push the maximum load to new, higher levels.

By breaking ties asymmetrically, one reduces the number of ties during the course of the process, and this improves the overall balance. To see this, again suppose the system is in a state with several buckets of load four. Buckets with load five are created when two buckets of load four are chosen; subsequently, buckets of load six are created when two buckets of load five are chosen. If ties are broken randomly, the buckets of load five are spread evenly on the left and right sides. If, however, ties are broken asymmetrically, the buckets of load five initially are all placed on the left hand side. Since our random bucket choices are taken one from each side using d-left, this causes it to take longer before a bin of load six can arise.

In the context of IP lookups, this asymmetry is also helpful in that it can be used to slightly reduce the average lookup time, in the case where the item being searched for is actually in the table. As the leftmost groups are more likely to hold more items, they can be examined first. If the pattern is found in one hash bucket, the other need not be searched. Hence, for the 2-left scheme, more than half the time the second (pipelined) memory access for each level will not have to be examined when the item is to be found in the table.

A. A Basic Analysis

We provide a simple approximate fluid limit analysis of the d-left scheme. For this section we follow [12]; however, we present the analysis here for completeness. The fluid limit analysis captures the behavior of the system as the number of buckets grows to infinity. The analysis depends on viewing the insertion of items as a deterministic process, where loads behave essentially according to their expectations. Appropriate large deviation theory yields that for sufficiently large systems, this approach is quite accurate; Chernoff-like bounds can be obtained, using theory that dates back to Kurtz [6], [7], [8]. Essentially, the theory demonstrates that the law of large numbers applies to these systems. Hence, from these Chernoff-like bounds, the probability of deviating significantly from the loads given by the differential equations falls exponentially in the size of the system, in terms of the number of buckets n. In practice, as we shall see, this analysis proves accurate even for systems of reasonable size, as the theory would suggest.

For convenience we begin with the case $d = 2$; thus we have two groups and $n/2$ buckets. Let $y_i(t)$ be the fraction of the n hash buckets that contain at least i items and are in the first, that is, leftmost, group when nt items have been placed. Similarly, let $z_i(t)$ be the fraction of the n hash buckets that contain at least i items and are in the second group when nt items have been placed. Note that $y_i(t), z_i(t) \leq 1/2$ and that $y_0(t) = z_0(t) = 1/2$ for all t. We will drop the explicit reference to t and simply use y_i and z_i where the meaning is clear.

If we choose a random hash bucket on the left, the probability that it has at least i items is $\frac{y_i}{1/2} = 2y_i$. Analogously, if we choose a random hash bucket on the right, the probability that it has load at least i is $2z_i$.

The fluid limit behavior expresses the deterministic behavior the system would follow in the limit as the number of buckets n and the number of items nt grow to infinity. It is expressed by a family of differential equations, where for $i \geq 1$:

$$\frac{dy_i}{dt} = 2\,(y_{i-1} - y_i)\,(2z_{i-1})\ ;$$
$$\frac{dz_i}{dt} = 2\,(z_{i-1} - z_i)\,(2y_i).$$

These equations express the following natural intuition. Let dt represent the amount of time during which one item is placed in the hash table. For y_i to increase over some interval dt, the newly inserted item must choose a bucket on the left with exactly $i-1$ items and a bucket on the right with at least $i-1$ items. The probability of this occurring is simply $2\,(y_{i-1} - y_i)\,(2z_{i-1})$. Similarly, for z_i to increase over some interval dt, the newly inserted item must choose a bucket on the left with at least i items and a bucket on the right with exactly $i-1$ items.

It will be somewhat more convenient to generalize to the case of general d if we write these equations all in terms of a single sequence x_i. If we substitute x_{2i} for y_i and x_{2i+1} for z_i, the equations above nicely simplify to the following (for $i \geq 2$):

$$\begin{aligned}\frac{dx_i}{dt} &= 2\,(x_{i-2} - x_i)\,(2x_{i-1}) \\ &= 4\,(x_{i-2} - x_i)\,x_{i-1}.\end{aligned} \tag{1}$$

For the d-left scheme, we may think of x_{jd+k} as representing the fraction of the buckets that have at least j items in the kth group from the left (where the leftmost group is the 0th group from the left). Then the fluid limit model yields the following family of differential equations:

$$\frac{dx_i}{dt} = d^d\,(x_{i-d} - x_i) \prod_{j=i-d+1}^{i-1} x_j. \tag{2}$$

We will use these equations to derive the approximate behavior when multiple hash functions are used. It is also worth noting what these families of differential equations tell us about the distribution of items to hash buckets. For example, suppose we have n items and n hash buckets (so that we can think of the equations as running until time $t = 1$). How do the x_i behave?

As in [15], [12], to describe this behavior, we define the generalized Fibonacci number $F_d(k)$ by $F_d(k) = 0$ for $k \leq 0$, $F_d(1) = 1$. and $F_d(k) = \sum_{i=1}^{d} F_d(k-i)$ when $k > 1$. Note that for $d = 2$ the generalized Fibonacci numbers are just the standard Fibonacci numbers. Then the behavior of the x_i is essentially

$$x_i(1) \sim 2^{-F_d(i)}.$$

We provide a loose justification. From equation 2, we have

$$\frac{dx_i}{dt} \leq d^d \prod_{j=i-d}^{i-1} x_j,$$

so by integrating

$$\begin{aligned} x_i(1) &\leq d^d \int_0^1 \prod_{j=i-d}^{i-1} x_j(t)dt \\ &\leq d^d \prod_{j=i-d}^{i-1} \int_0^1 x_j(t)dt \\ &\leq d^d \prod_{j=i-d}^{i-1} x_j(1). \end{aligned}$$

Now suppose $x_j(1) \leq 2^{-F_d(j)-1}/d$ for $i - d \leq j \leq i - 1$. Then

$$\begin{aligned} x_i(1) &\leq d^d \prod_{j=i-d}^{i-1} \frac{2^{-F_d(j)-1}}{d} \\ &\leq \prod_{j=i-d}^{i-1} 2^{-F_d(j)-1} \\ &\leq 2^{-d} 2^{-\sum_{j=i-d}^{i-1} F_d(j)} \\ &\leq \frac{2^{-F_d(i)}}{d}. \end{aligned}$$

Hence, once the tails become sufficiently small, a simple induction can be used to show the tails decrease faster than $2^{-F_d(i)}$; that is, the decrease has a generalized Fibonacci number in the exponent.

Because x_{jd+k} represents the fraction of the buckets that have at least j items in the kth group from the left, the fraction of buckets with load at least i is $\sum_{k=0}^{d-1} x_{id+k} \approx 2^{-F_d(di)}$. Recall that for large i, $F_d(k)$ grows exponentially; that is, $F_d(k) \approx \phi_d^k$ for some constant ϕ_d. In fact ϕ_2 is the golden ratio $\frac{1+\sqrt{5}}{2} = 1.618\ldots$, and the ϕ_d form an increasing sequence satisfying $2^{(d-1)/d} < \phi_d < 2$. (For reference, $\phi_3 = 1.839\ldots$ and $\phi_4 = 1.927\ldots$) So, for example, when $d = 2$ the fraction of buckets with load at least i falls approximately like $2^{-2.6^i}$; note that the i is in the exponent of the exponent. Intuitively, this implies that the x_i fall extremely quickly with i, and hence the maximum load is very small.

Indeed, an alternative proof technique based on witness trees demonstrates that the maximum load is $\frac{\log\log n}{d \log \phi_d} + O(1)$ with high probability [15]. The analysis based on differential equations is not completely suitable for obtaining such fine bounds [11]; however, it does yield accurate numerical information useful for predicting the behavior of the hash function in practice.

B. Modeling Dynamic Deletions and Additions

In the section, we extend previous work by showing how to modify the basic equation (1) to handle dynamic additions and deletions to the table. Our goal here is to suggest that additions and deletions of addresses can be handled on-line with our suggested hashing scheme. We emphasize, however, that when attempting to handle table additions on-line there is always the possibility that the load on a bucket will exceed the maximum capacity, as given by the cache line size. In such a case, one must be prepared to take an action such as re-hashing the data using new hash functions. An advantage of our multiple hash function approach is that finding suitable new hash functions is very quick, and our analysis demonstrates that the need for such emergency procedures can generally be made so rare that it is not a significant issue.

Note that if we are required to handle dynamic additions only, equation (1) still holds. One only needs an upper bound on the number of items to be hashed, and the equation can be used to determine the distribution when the number of items hashed reaches this upper bound.

If there are additions and deletions, we must model how deletions occur. Two important points are the rate of deletions compared with the rate of additions, and how the items to be deleted are chosen. For the first issue, a natural breakdown is to assume that items are added only up to some point in time, and then additions and deletions vary. We let the probability that an event is an insertion be p and the probability that an event is a deletion be $1-p$. For the second issue, we can vary our equations to analyze the case where, when a item is to be deleted, the item is chosen uniformly at random from all items. More concretely, we can model the situation where all addresses have lifetimes that are exponentially distributed with the same mean. More general deletion models, such as models where the age of an item can affect its probability of being deleted, can be handled using the analysis of [15], although this approach does not give the numerical answers we desire here. The model where a random bucket is chosen and an item is deleted from that bucket can also be handled using these techniques, however [9].

We modify the equation (1) to account for deletions by noting that the total number of balls is $\sum_{i\geq 0} i(x_{2i}+x_{2i+1})$, and the number of balls that can be deleted that cause a reduction in x_i is $\lfloor\frac{i}{2}\rfloor(x_i - x_{i+2})$. Hence the equations that describe the behavior of the system are given by

$$\frac{dx_i}{dt} = 4p\,(x_{i-2}-x_i)x_{i-1} - \frac{(1-p)\lfloor i/2\rfloor(x_i-x_{i+2})}{\sum_{j\geq 0} j(x_{2j}+x_{2j+1})}. \quad (3)$$

Intuitively, the final distribution is likely to be smoother when deletions occur in this manner, as heavily loaded buckets are more likely to incur a deletion than lightly loaded buckets.

III. Data

A. Evaluating the differential equations

Number of items

	$n/2$	n	$2n$	$3n$	$4n$
0	6.1e-01	3.7e-01	1.4e-01	5.0e-02	1.8e-02
1	3.0e-01	3.7e-01	2.7e-01	1.5e-01	7.3e-02
2	7.6e-02	1.8e-01	2.7e-01	2.2e-01	1.5e-01
3	1.3e-02	6.1e-02	1.8e-01	2.2e-01	2.0e-01
4	1.6e-03	1.5e-02	9.0e-02	1.7e-01	2.0e-01
5	1.6e-04	3.1e-03	3.6e-02	1.0e-01	1.6e-01
6	1.3e-05	5.1e-04	1.2e-02	5.0e-02	1.0e-01
7	9.4e-07	7.3e-05	3.4e-03	2.2e-02	6.0e-02
8	5.9e-08	9.1e-06	8.6e-04	8.1e-03	3.0e-02
9	3.3e-09	1.0e-06	1.9e-04	2.7e-03	1.3e-02
10	1.6e-10	1.0e-07	3.8e-05	8.1e-04	5.3e-03
11	7.4e-12	9.2e-09	6.9e-06	2.2e-04	1.9e-03
12	3.1e-13	7.7e-10	1.2e-06	5.5e-05	6.4e-04
13	1.2e-14	5.9e-11	1.8e-07	1.3e-05	2.0e-04
14	4.2e-16	4.2e-12	2.5e-08	2.7e-06	5.6e-05
15	1.4e-17	2.8e-13	3.4e-09	5.5e-07	1.5e-05

TABLE I
LOADS IN THE FLUID LIMIT (n BUCKETS, 1 CHOICE). ENTRIES REPRESENT THE FRACTION OF BUCKETS WITH THE EXACT LOAD GIVEN IN THE LEFT COLUMN.

We first demonstrate what results we obtain by evaluating the fluid limit system given by the family of differential equations. The results obtained here were found by simulating the progress of the differential equations using discrete time steps of $5 \cdot 10^{-7}$, which prove more than sufficient for this level of accuracy. For example, to obtain a result for $n/2$ items and n buckets, we run the differential equations up to $t = 1/2$. Values of less than 1e-100 are left blank in our tables.

For comparison purposes, we include in Table I equivalent results in the case where a single hash function is used, assuming that the hash function distributes items independently and uniformly at random into buckets. We note the well-known fact that as n grows to infinity the fraction of buckets with load k when the average load is μ approaches a Poisson random variable, and hence the fraction with load k is simply $\frac{e^{-\mu}\mu^k}{k!}$.

Two important points are manifest from Tables I, II, and III. First, when using two or more hash functions, the fraction of buckets with a given load decreases remarkably quickly with the load, especially in comparison with the single choice. This is to be expected given our previous discussion. As an example, consider when n items

	Number of items				
	$n/2$	n	$2n$	$3n$	$4n$
0	5.3e-01	2.3e-01	3.4e-02	4.6e-03	6.2e-04
1	4.4e-01	5.5e-01	2.1e-01	4.0e-02	6.9e-03
2	3.0e-02	2.2e-01	5.0e-01	2.0e-01	4.3e-02
3	8.6e-06	4.4e-03	2.6e-01	4.8e-01	1.9e-01
4	9.2e-16	5.2e-08	9.1e-03	2.7e-01	4.7e-01
5	1.4e-42	1.2e-21	5.0e-07	1.2e-02	2.8e-01
6		5.3e-58	7.2e-19	1.1e-06	1.3e-02
7			1.5e-50	6.6e-18	1.6e-06
8				5.7e-48	1.8e-17
9					8.4e-47

TABLE II

LOADS IN THE FLUID LIMIT (n BUCKETS, 2 CHOICES). ENTRIES REPRESENT THE FRACTION OF BUCKETS WITH THE EXACT LOAD GIVEN IN THE LEFT COLUMN.

	Number of items				
	$n/2$	n	$2n$	$3n$	$4n$
0	5.1e-01	1.6e-01	9.1e-03	4.6e-04	2.3e-05
1	4.9e-01	6.8e-01	1.6e-01	1.0e-02	6.0e-04
2	6.8e-03	1.6e-01	6.6e-01	1.5e-01	1.1e-02
3	5.5e-15	1.1e-05	1.7e-01	6.6e-01	1.5e-01
4	2.9e-92	4.4e-33	2.0e-05	1.8e-01	6.6e-01
5			2.2e-31	2.2e-05	1.8e-01
6				4.6e-31	2.3e-05
7					5.6e-31

TABLE III

LOADS IN THE FLUID LIMIT (n BUCKETS, 3 CHOICES). ENTRIES REPRESENT THE FRACTION OF BUCKETS WITH THAT LOAD.

are hashed into n buckets, for large n. Our results show that 1e-06 of the buckets will have load at least 9 if a single hash function is used; with two hash functions, only about 5.2e-08 + 1.2e-21 + 5.3e-58 $\approx$ 5.2e-08 of the buckets will have load four or greater, and similarly with three hash functions, only 4.4e-33 of the buckets will even have load four!

Second, when tn items are placed, the loads are strongly centered around the integers nearest to t. This follows naturally from the above, since the average bucket load is of course t, and the probability of high bucket loads decreases so quickly. These two effects are exactly what we desire from our hash table. We wish the probability of having a heavily loaded bucket should be small, so that we do not overload a cache line; however, we wish most cache lines to be reasonably full.

It is worth noting that there is a noticeable gain in moving from two hash functions to three. The difference follows from the Fibonacci decrease of the tails; the tails decrease significantly faster with each additional choice. (From our theoretical analysis, we have that when $d = 2$ the fraction of buckets with load at least i falls approximately like $2^{-2.6^i}$; for $d = 3$, the fraction of buckets with load at least i falls instead like $2^{-6.2^i}$.) Hence one can trade off the number of memory accesses required in order to improve the memory usage. Using more hash functions requires more memory accesses (although they can still be pipelined in a straightforward fashion); in return, more entries can be stored without violating the constraint given by the number of entries that can fit on a cache line.

B. Comparing the differential equations and simulations

Items	Buckets	Results
32000	64000	Max. load 5 for 3992 trials Max. load 6 for 5375 trials Max. load 7 for 598 trials Max. load 8 for 34 trials Max. load 9 for 1 trials
32000	32000	Max. load 6 for 675 trials Max. load 7 for 6487 trials Max. load 8 for 2485 trials Max. load 9 for 320 trials Max. load 10 for 30 trials Max. load 11 for 3 trials
32000	16000	Max. load 8 for 233 trials Max. load 9 for 4437 trials Max. load 10 for 4075 trials Max. load 11 for 1040 trials Max. load 12 for 178 trials Max. load 13 for 29 trials Max. load 14 for 7 trials Max. load 15 for 1 trials
32000	8000	Max. load 11 for 2 trials Max. load 12 for 1105 trials Max. load 13 for 4354 trials Max. load 14 for 3018 trials Max. load 15 for 1139 trials Max. load 16 for 287 trials Max. load 17 for 74 trials Max. load 18 for 15 trials Max. load 19 for 2 trials

TABLE IV

SIMULATION RESULTS, RANDOM INSERTIONS, 1 CHOICE.

Items	Buckets	Results
32000	64000	Max. load 2 for 5826 trials Max. load 3 for 4174 trials
32000	32000	Max. load 3 for 9980 trials Max. load 4 for 20 trials
32000	16000	Max. load 4 for 9911 trials Max. load 5 for 89 trials
32000	8000	Max. load 6 for 9895 trials Max. load 7 for 105 trials

TABLE V
SIMULATION RESULTS, RANDOM INSERTIONS, 2 CHOICES.

Items	Buckets	Results
30000	60000	Max. load 2 for 10000 trials
30000	30000	Max. load 2 for 7154 trials Max. load 3 for 2846 trials
30000	15000	Max. load 3 for 7441 trials Max. load 4 for 2559 trials
30000	7500	Max. load 5 for 8462 trials Max. load 6 for 1538 trials
30000	6000	Max. load 6 for 8735 trials Max. load 7 for 1265 trials

TABLE VI
SIMULATION RESULTS, RANDOM INSERTIONS, 3 CHOICES.

Because the results given by the differential equations describe asymptotic behavior, it is worth comparing their behavior to simulations of the underlying random process. In particular, we are interested in whether the differential equations accurately predict the maximum load of a bucket for numbers of items and buckets likely to arise in practice. For this reason, we focus on instances where the number of buckets and items are in the small tens of thousands. Our differential equations would better match larger systems, and give less accurate results for smaller systems.

For the case of one or two hash functions, we simulated systems with 32,000 items with varying numbers of buckets: 8,000, 16,000, 32,000, and 64,000. In order to divide groups evenly, we used slightly different numbers of buckets for the case of three choices (see Table VI). These simulations are idealized, in that the buckets for each item were chosen independently and uniformly at random from the left and right sides (using the pseudo-random generator *drand48*). We emphasize that this idealization does not necessarily correspond to the data itself being random in practice, but rather that the *hashes* of the initial data appear random. Using a computationally expensive but powerful hash function such as MD5 could approximate this

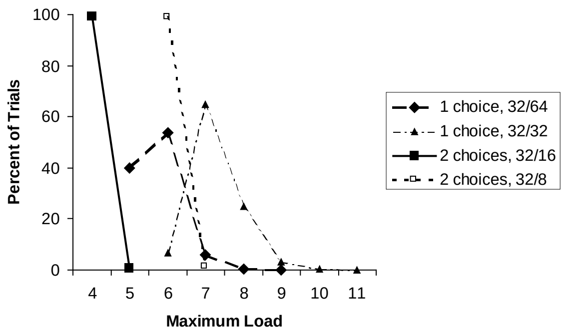

Fig. 2. One vs. two hash functions, over 10,000 trials. In the legend, the the number of items (in thousands) is followed by the number of buckets (in thousands).

behavior. In practice, we suggest simpler hash functions, as described in Section IV.

As an example of how to compare these results with the fluid limits, consider the case of 32,000 items and 32,000 buckets. The fluid limit suggests that a fraction 5.2e-08 of the buckets will have load 4 (or greater) in this case. Hence, over 10,000 runs, we would expect to see around 16 or 17 buckets with load 4. In simulations we see a maximum load of 4 only 14 times, suggesting the fluid limit provides an excellent guide to the behavior of realistic sized systems.

We provide a graphical representation of the difference between using one and two hash functions in Figure 2. The legend gives the number of items (in thousands) followed by the number of buckets (in thousands). The main point here is that using two hash functions allows greater predictability and a smaller maximum load, even while using much less memory.

The power of using three hash functions is rather surprising. Consider the case where there are tens of thousands of items, and six items can fit into a cache line; this is essentially the situation considered in [14]. With 30,000 items and 6,000 buckets using three hash functions, even though the average load is five items per bucket, the maximum load is only six! Using two hash functions, we see that with 32,000 items and 8,000 buckets the maximum load is very likely to be six. Hence we can achieve an average load of four and a maximum load of six, using two hash functions. In general, we see that for parameters that appear reasonable for the IP routing scenario, we can achieve a very good utilization of memory with our hash table using a small number of hash functions.

For a more direct comparison between our simulations and the fluid limit calculation, we provide detailed results for each of our sets of 10,000 trials. We present the fraction of buckets with each load. The results of Tables VII and VIII are almost exactly the same as predicted by our analysis as given in Tables II and II. The small differences might simply be the statistical effect of having too small a sample for rare events. Alternatively, the analysis might slightly underestimate the fraction of buckets with the largest load for our simulations; for larger numbers of items and buckets this discrepancy would shrink.

The results are strongly robust. For example, we ran 1,000,000 experiments with 32,000 items and 8,000 buckets, using two choices. The maximum load was 6 for 987,296 of these trials, and 7 for the remaining 12,704 trials.

Again, the results make clear that using two or three hash functions can drastically reduce the maximum load and the variance in the maximum load, leading to better and more predictable hashing performance. Further, using multiple hash functions can dramatically improve upon the total space used to store the hash table by reducing the amount of unused space.

Load	Number of buckets: 64,000	32,000	16,000	8,000
0	5.3e-01	2.3e-01	3.4e-02	6.3e-04
1	4.4e-01	5.5e-01	2.1e-01	6.9e-03
2	3.0e-02	2.2e-01	4.9e-01	4.3e-02
3	8.6e-06	4.5e-03	2.6e-01	1.9e-01
4		6.3e-08	9.1e-03	4.7e-01
5			5.6e-07	2.8e-01
6				1.3e-02
7				1.3e-06

TABLE VII
LOADS FOUND BY SIMULATIONS ($32,000$ ITEMS, VARYING NUMBERS OF BUCKETS, 2 CHOICES). ENTRIES REPRESENT THE FRACTION OF BUCKETS WITH THAT EXACT LOAD.

C. Simulations for Deletions and Additions

The differential equations (3) describe the behavior of a system with insertions and random deletions. Such equations can be used to determine the end state of the system. However, what is important in the setting of deletions is not the end state, but the amount of time until the number of items hashed to a single bucket becomes too large. At such time, a cache line cannot store a bucket, and we are forced to do a potentially expensive re-hash to create a new hash table.

The results from the differential equations can be used to obtain very loose approximations for the probability that some bucket exceeds its capacity during the course of a process. Since $x_{2i} + x_{2i+1}$ is meant to approximate the fraction of buckets with load at least i as the number of buckets and buckets grows large, the total expected number of buckets with load at least i over the first T steps can approximately be upper bounded by

$$\sum_{t=0}^{T-1} x_{2i}(t) + x_{2i+1}(t) \leq T \max_{0 \leq t \leq T-1} x_{2i}(t) + x_{2i+1}(t).$$

The expected number of buckets with load at least i over the first T steps is certainly larger than the probability of seeing a bin with load at least i over the first T steps. Hence, if this expectation is small, we obtain a bound on the corresponding probability.

We emphasize that the point here is not so much to get accurate upper bounds for the probability a bin ever exceeds some load. Rather, the point is that the x_i shrink so fast that we would expect to run a significant number of steps before needing to re-hash if we choose our parameters appropriately. We consider a specific example: suppose we start by inserting 32,000 items into 16,000 buckets using two choices. We then either insert or delete an item, each with equal probability, until we see a bucket with load six. For convenience, we refer to each insert or delete operation as a step.

From Table II, the asymptotic fraction of buckets with load at least six is 7.2e-19 after the insertion stage. As deletions tend to reduce the number of highly loaded buckets, we would therefore expect that our hash table could deal with insertions and deletions for a long time before a bucket with load six appears. In practice, however, with such a small number of bins, the variance has a very large effect.

We simulated the process with 32,000 items and 16,000 bins, stopping when we saw a bucket with load six or the number or when we had performed 10,000,000 steps. In one hundred trials, we reached 10,000,000 steps without

Load	Number of buckets: 60,000	30,000	15,000	7,500
0	5.1e-01	1.6e-01	9.1e-03	2.4e-05
1	4.9e-01	6.8e-01	1.6e-01	5.9e-04
2	6.8e-03	1.6e-01	6.6e-01	1.1e-02
3		1.1e-05	1.7e-01	1.5e-01
4			2.0e-05	6.6e-01
5				1.8e-01
6				2.3e-05
7				

TABLE VIII
LOADS FOUND BY SIMULATIONS ($30,000$ ITEMS, VARYING NUMBERS OF BUCKETS, 3 CHOICES). ENTRIES REPRESENT THE FRACTION OF BUCKETS WITH THAT EXACT LOAD.

seeing a bucket with load six seventy-five times. Of the remaining twenty-five trials, the smallest number of steps was only 121,805, but the average was approximately 4.54 million. In all of these twenty-five trials, the number of hashed items was greater than 32,000 when the process stopped; the average was over 34,500 items. Hence the maximum number of items that one expects to be in the system should be a major concern when deciding the appropriate size of the hash table. These results justify our assertion that our hashing schemes are highly robust under deletions and insertions.

D. Implications

It is worth summarizing some of the benefits and the new tradeoffs that our approach yields.

One important benefit is that under the assumption that hash functions are sufficiently random (which we discuss below), the performance of these hashing schemes for various values of the memory size, cache line size, etc. can easily be tested numerically using the appropriate differential equations. Although the results obtained in this fashion are asymptotic, they appear quite accurate for systems of reasonable size (say, in the tens of thousands). This is not surprising, given that Chernoff-like bounds apply.

Similarly, when a fixed number of items are to be inserted in the hash table, one can use the asymptotic results to predict the probability of success for a given cache line size. This number can be used to trade pre-processing time for space. In particular, in order to use less memory, it may be suitable to aim for a setup where the probability that no cache line size is exceeded is, say, only 20%. In this case, trying several combinations of hash functions may be necessary; the set of items can be re-hashed offline until a suitable hash table is produced. Knowing the probability of success allows one to estimate the time to find an appropriate combination. The search for good hash functions is likely to be very efficient, as we describe in Section IV.

There are tradeoffs between the number of hash functions used, the memory used, and the applicable cache line size. Increasing the number of hash functions decreases the maximum load, and hence allows smaller cache lines. While two hash functions appear generally sufficient, three can be used to improve memory utilization. Similarly, increasing the hash table size reduces the maximum load while increasing the total memory used.

Our hash scheme also performs well when items are inserted and deleted from the table. Deletions have a tendency to decrease more full buckets, and therefore the system can handle a significant number of insertion and deletion steps before unfortunate circumstances necessitate a re-hashing of the data.

Finally, we reiterate that all memory look-ups required by this scheme can be done in parallel, in either hardware or software, since each hash function yields buckets that can be stored in completely separate areas of memory.

IV. Implementation Details

In practice one cannot simply obtain a perfectly random hash function; instead one generally chooses a hash function from a small family of hash functions. Our analysis thus far has assumed that our hash functions are perfectly random, and unfortunately we don't know how to analyze the use of smaller hash families (e.g., *2-universal* families [3], [5]) in this context, although our belief is that standard hash families will provide performance similar to the analysis in practice. Our belief is centered on the fact that in practice we will not have adversarially chosen worst case data, and hence our hash functions are likely to be "sufficiently random" that our analysis describes actual behavior. An interesting question that is outside the scope of this paper is to consider what the best hash functions to use on IP routing data would be. A related question is how random does IP routing data appear.

A simple hash function (for both hardware and software) that one can use is to treat the input as an element in an appropriate finite field $Z[2^k]$ and multiply by a random element in the field $Z[2^k]$, that is, modulo a given irreducible prime polynomial. This is simply implemented as a multiplier without carries and a CRC (cyclic redundancy check). Each hash function can be based on a different random multiplier and a different irreducible prime polynomial. Using a more complex and larger family of hash functions based on using several random multipliers (see, e.g., [3]) more closely approximates the family of all possible hash functions, if this is desired.

An IP router that needed to build a hash table could simply choose two random elements of the field, using one element as a multiplier for each hash function. If the hash table is found suitable, in that the maximum number of items in a bucket fits on a cache line, these multipliers are used; otherwise, new random elements are chosen. The process is repeated until a suitable hash function is found.

To test how realistic hash functions perform, we implemented a simple scheme that derives two hash values from prefixes by computing the standard 16 bit CRCs, CRC-16 and CRC-CCITT, on them. (Hence we have not even bothered with random multipliers for the hash function.) Note that if we assume that our prefixes are, for example, 32 bit strings generated uniformly at random, then it is as though our hashes give two uniform, independent values for each hash function. (This follows simply from the Chinese remainder theorem, applied over this polynomial domain.) We checked our implementation by testing it on 32 bit strings generated uniformly at random, and found that it indeed behaves entirely similarly to the simulations based on hashes being perfectly random.[3]

Consecutive prefixes (which may be likely to arise in practice) naturally land in distinct buckets for each hash function, which should actually improve performance. We tested this with the following experiment. Items are divided into *blocks*. The first 32 bit string for each block is generated randomly; the rest of the bit strings in the block are just consecutive integers. The results appear in Table IX. Although performance appears quite similar to our simulations where items are hashed independently and uniformly at random when the block size is small, when the block size is large performance actually improves. This is because the small stride ensures that all items within a block hash to different buckets.

We performed similar tests using different strides; for example, we tried having consecutive elements in the same block differ by 256 or 173. For most strides, performance was similar to that of our simulations where items are hashed independently and uniformly at random. However, for a stride of 256, performance degraded for large block sizes. We believe that this particular stride interacts with the hash function in some way that some buckets tend to be repeated. Further tests suggested that there may be a small number of stride values that have worse performance than expected. This problem disappears, however, when we introduce random multipliers as described above, as shown in Table X.

Items	Buckets	Block size	Results
32000	16000	10	Max. load 4 for 9925 trials Max. load 5 for 75 trials
32000	16000	100	Max. load 4 for 9966 trials Max. load 5 for 34 trials
32000	16000	1000	Max. load 3 for 1919 trials Max. load 4 for 8075 trials Max. load 5 for 6 trials
32000	8000	10	Max. load 6 for 9866 trials Max. load 7 for 134 trials
32000	8000	100	Max. load 6 for 9942 trials Max. load 7 for 58 trials
32000	8000	1000	Max. load 5 for 3128 trials Max. load 6 for 6870 trials Max. load 7 for 2 trials

TABLE IX
SIMULATION RESULTS, 2 CRCS AS HASH FUNCTIONS, WITH BLOCKED INPUTS (STRIDE 1).

Items	Buckets	Block size	Results
32000	16000	10	Max. load 4 for 9902 trials Max. load 5 for 98 trials
32000	16000	100	Max. load 4 for 9700 trials Max. load 5 for 300 trials
32000	16000	1000	Max. load 4 for 668 trials Max. load 5 for 8565 trials Max. load 6 for 765 trials Max. load 7 for 2 trials
32000 with	16000 random	1000 multiplier	Max. load 4 for 9562 trials Max. load 5 for 436 trials Max. load 6 for 2 trials

TABLE X
SIMULATION RESULTS, 2 CRCS AS HASH FUNCTIONS, WITH BLOCKED INPUTS (STRIDE 256).

A. Using Real IP Data

We also examined the performance of these hash functions on real data obtained from Srinivasan and Varghese, who used this data in [14]. Our tests were based on a snapshot of the MaeEast database with 38,816 prefixes.

Our primary test was to take the input data that arises for one of the hash tables using the Binary Search on Levels with controlled prefix expansion. Using three levels (with prefixes of 16, 24, and 32 bits), the table of 24-bit prefixes has 198,734 entries. (The other tables are significantly smaller, and we ignore them here.) The hash function determined in [14] used 131,072 buckets of 32 bytes, and therefore requires four megabytes of space, in order to ensure that at most six entries were held in each cache line. The hash function took a few minutes to find on a modern Alpha system. Using just the two CRCs as hash functions and 65,536 buckets, we obtained a maximum load of five. Using 50,000 buckets suffices for a maximum load of six. Our hash table requires half the space (or less) and was found essentially instantaneously. Experiments using random multipliers along with the CRCs show essentially the behavior, although it appears that using just the two CRCs is somewhat fortunate. For 1,000 trials with random multipliers and 65,536 buckets, the maximum load was five for 835 trials and six for the remaining trials.

We repeated the experiment when the first prefix level uses 18 bits. In this case, the number of entries for the 24 bit hash table is reduced to 117,131. Again, in this instance the hash function determined in [14] requires four megabytes of space and some time to find. Using the two

[3] Because our hashes are 16 bits and our simulations use a number of buckets that is not a power of 2, some buckets are slightly more likely to be chosen. We have not found this to have a significant impact.

CRCs, we can achieve a maximum load of six with only 32,768 buckets. In this case, we require only one quarter the space, and again the first pair of hash functions we tried prove successful. In fact, when this experiment was repeated 1,000 times with random multipliers, the maximum load was six every time.

Just for fun, we tried creating a hash table using just the 38,816 prefixes, all converted into 32 bit numbers. With 9,000 buckets we achieved a maximum load of six, again just using the CRCs.

From these results, we suggest that although we cannot make statements regarding worst case behavior for using multiple hash functions when the hash functions are chosen from a small, easily implemented family, we believe that in practice a reasonable implementation will perform similarly to our analysis. The families we have tested (with a single random multiplier per hash function) perform close to the analysis and are simple to implement in hardware or software. In fact, they are quite minimal; one could undoubtedly design more complex hash functions that would improve results. Determining what hash functions are most appropriate depends in part on the underlying data and in part on the desired tradeoff between hashing complexity and performance. For the specific case of IP routing, this is an avenue for possible future study.

We note that there are also further possibilities for saving space in the hash table. For example, it may be possible not to store the entire IP prefix in the hash table. Suppose we use a 1-1 hash function (a random permutation) that maps 32 bit IP prefixes (in, say, IPv-6) to 32 bit values. We may use the first 16 bits as an index into a hash table, and identify the prefix in the table using only the remaining 16 bits from the hash.

V. Conclusions

We have suggested a hashing scheme, d-left, based on using multiple hash functions that is suitable for situations where it is important to bound the maximum number of items that fall into a bucket, such as when the bucket is meant to fit in a cache line. A key feature of the d-left scheme is that all hashes and memory lookups can be done in parallel in a straightforward manner.

We have also discussed the applicability of d-left to IP routing, using the binary search on levels approach. Important future work includes building a more complete testbed for testing the d-left hashing scheme on real data and comparing its performance against other approaches. We also believe that d-left hashing is a simple but extremely powerful technique that will prove useful in other applications as well, and we are actively seeking possible applications.

Acknowledgments

The authors would like to thank V. Srinivasan and G. Varghese for providing access to their data.

References

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

[2] A. Bremler-Barr, Y. Afek, and S. Har-Peled. Routing with a Clue. In *Proceedings of the ACM SIGCOMM '99 Conference*, 1999, pp. 203–213.

[3] A. Broder and A. Karlin. Multilevel Adaptive Hashing. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 43–53.

[4] D. Carrigan. Network Processors Help Enable the Internet Economy. Available at `//developer.intel.com/solutions/archive/issue19/stories/top3.htm`.

[5] L. Carter and M. Wegman. Universal Classes of Hash Functions. *Journal of Computer Systems and Science*, 18:2, 1979, pp. 143–154.

[6] S. N. Ethier and T. G. Kurtz. **Markov Processes: Characterization and Convergence**, 1986, John Wiley and Sons.

[7] T. G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability* Vol. 7, 1970, pp. 49-58.

[8] T. G. Kurtz, **Approximation of Population Processes**, SIAM, 1981.

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

[10] M. Mitzenmacher. Load Balancing and Density Dependent Jump Markov Processes. In *Proc. of the* 37^{th} *IEEE Symp. on Foundations of Computer Science*, 1996, pp. 213–222.

[11] M. Mitzenmacher. Studying Balanced Allocations with Differential Equations. *Combinatorics, Probability, and Computing*, vol. 8, 1999 pp. 473-482.

[12] M. Mitzenmacher and B. Vöcking. The Asymptotics of Selecting the Shortest of Two, Improved. Extended abstract available at www.eecs.harvard.edu/~michaelm/NEWWORK/papers.html. Short abstract to appear in *Proc. of the* 39^{th} *Allerton Conference*.

[13] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *Proc. of SIGCOMM '99*, pp. 135–146.

[14] V. Srinivasan and G. Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, vol. 17, no. 1, 1999, pp. 1–40.

[15] B. Vöcking. How Asymmetry Helps Load Balancing. In *Proc. of the* 40^{th} *IEEE Symp. on Foundations of Computer Science*, 1999, pp. 131–141.

[16] N.D. Vvedenskaya, R.L. Dobrushin, and F.I. Karpelevich. Queueing System with Selection of the Shortest of Two Queues: an Asymptotic Approach. *Problems of Information Transmission*, Vol 32, 1996, pp. 15–27.

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