Hash-Based Techniques for High-Speed Packet Processing

Adam Kirsch, Michael Mitzenmacher, and George Varghese

Abstract Hashing is an extremely useful technique for a variety of high-speed packet-processing applications in routers. In this chapter, we survey much of the recent work in this area, paying particular attention to the interaction between theoretical and applied research. We assume very little background in either the theory or applications of hashing, reviewing the fundamentals as necessary.

1 Introduction

This chapter surveys recent research on hash-based approaches to high-speed packet processing in routers. In this setting, it is crucial that all techniques be amenable to a hardware implementation, as high-performance routers typically must operate at *wire speed*, meaning that the time that a router can operate on a packet is at most the time that it takes the link to deliver a packet of some minimal size (e.g., 40 bytes, corresponding to a TCP acknowledgement with no payload). Software-based approaches are simply inadequate for this task.

Since this topic is quite broad, we start by specifying some boundaries for our coverage. There is a very large and growing body of work on the more general theme of algorithmic approaches to important packet processing applications. The most comprehensive reference for these techniques is the text by Varghese [62],

Adam Kirsch
Harvard School of Engineering and Applied Sciences, 33 Oxford Street, Cambridge, MA 02138
e-mail: kirsch@eecs.harvard.edu

Michael Mitzenmacher
Harvard School of Engineering and Applied Sciences, 33 Oxford Street, Cambridge, MA 02138
e-mail: michaelm@eecs.harvard.edu

George Varghese
Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92040 e-mail: varghese@cs.ucsd.edu

published in 2004. We therefore focus our attention on work done since then. Even limiting ourselves to the relevant research in the last few years, however, would leave an enormous number of papers to cover! We therefore further focus our attention on two key application areas for high-speed routers: hash tables and related data structures, and hash-based schemes for network measurement. While we aim to be comprehensive, given the huge recent growth of interest in this area, this survey should be considered a guide to the literature rather than a full account.

Before diving into the literature, we offer some of our high-level perspective that guides this survey. First, as will become apparent in the body of this chapter, there is an enormous amount of potential for interplay between theoretical and applied techniques. It follows that the relevant literature spans the spectrum from very theoretical to very applied. We aim our attention on the middle of this range, but we emphasize that any particular piece of work must be considered relative to its place in the spectrum. For instance, when we discuss hash table designs for high-speed routers, we consider several theoretical papers that focus on the design of hash tables generally, rather than on some particular application. In such work, the primary goal is often to present and evaluate general data structure design principles that appear to have broad potential to impact the implementation of practical systems. The evaluation in these works usually takes the form of establishing various guarantees on the data structure outside the context of any particular application. For example, a work in this vein that proposes a novel hash table design may place significant emphasis on the sorts of theoretical and numerical guarantees that can be obtained under the new design, with simulations serving in a mostly supporting role. Naturally, then, a major challenge in this area is to design data structures that are amenable to a compelling evaluation of this sort. Of course, since this approach is very general, it typically does not speak directly to how such a data structure might perform when appropriately adjusted and implemented in a real application. When properly interpreted, however, the results of these more theoretical works can be highly suggestive of increased performance for a broad range of settings.

Similarly, very applied works should be considered with respect to the concrete results that they demonstrate for the specific application of interest. And, of course, works in the middle of the spectrum typically should be considered with respect to some combination of these goals, for instance showing that a particular theoretical intuition seems to lead to compelling results for some class of related applications.

In the rest of the survey, we first give the necessary background and history in Section 2. We then consider three fairly broad application settings: hash table lookups for various hardware memory models (Sections 3 and 4), Bloom filter-type applications (Section 5), and network measurement (Section 6).

2 Background

We review some of the key concepts underlying the hash-based data structures commonly proposed for high-speed packet processing. We describe the performance measures relevant to these applications and the resulting hardware models, and also give a brief history of the earlier literature on these applications.

2.1 Hash-Based Data Structures

We begin with a brief review of the relevant history, constructions, and issues in the design of hash-based data structures. We describe some of the tension between the theory an practice of hash functions that informs our analyses, review the standard Bloom filter data structure and its variants, and discuss multiple-choice hash tables.

2.1.1 Hash Functions

Intuitively, a hash function $h : U \rightarrow V$ is a function that maps every item $u \in U$ to a hash value $h(u) \in V$ in a fashion that is somehow random. The most natural mathematical model for a hash function is that it is *fully random*; that is, it is a random element of V^U, the set of functions with domain U and codomain V. Under this assumption, the hash values $\{h(x) : x \in U\}$ corresponding to the items in U are independent random variables that are each uniformly distributed over V. Clearly, this is a very appealing scenario for probabilistic analysis.

Unfortunately, it is almost always impractical to construct fully random hash functions, as the space required to store such a function is essentially the same as that required to encode an arbitrary function in V^U as a lookup table. From a theoretical perspective, this sort of thinking quickly leads to compromises between the randomness properties that we desire in a hash function and the computational resources needed to store and evaluate such a function. The seminal work along these lines is the introduction of *universal hash families* by Carter and Wegman [9, 67], which introduces the idea of choosing a hash function h randomly (and efficiently) from a set $\mathscr{H}$ of potential hash functions, chosen so that the joint distribution of the random variables $\{h(x) : x \in U\}$ satisfies limited but intuitively powerful properties. As a matter of terminology, the terms *hash functions* and *hash families* are often used interchangeably when the meaning is clear.

Specifically, a family of hash functions $\mathscr{H}$ with domain U and codomain V is said to be *2-universal* if, for every pair of distinct items $x, y \in U$, we have that for a properly sampled hash function $h \in \mathscr{H}$,

$$\mathbf{Pr}(h(x) = h(y)) \leq \frac{1}{|V|}.$$

That is, the probability of a collision between any pair of items after being hashed is at most that for a fully random hash function. A family of hash functions is said to be *strongly 2-universal*, or more commonly in modern terminology *pairwise independent*, if for every pair of distinct items $x, y \in U$ and any $x', y' \in V$, we have

$$\mathbf{Pr}(h(x) = x' \text{ and } h(y) = y') = \frac{1}{|V|^2}.$$

That is, the behavior for any pair of distinct items is the same as for a fully random hash function. Historically, in some cases the term *universal* is used when *strongly universal* is meant. Pairwise independence generalizes naturally to k-wise independence for collections of k items, and similarly one can consider k-universal hash functions, although generally k-wise independence is more common and useful. More information can be found in standard references such as [46].

Since Carter and Wegman's original work [9], there has been a substantial amount of research on efficient constructions of hash functions that are theoretically suitable for use in data structures and algorithms (e.g., [48,55] and references therein). Unfortunately, while there are many impressive theoretical results in that literature, the constructed hash families are usually impractical. Thus, at least at present, these results do not seem to have much potential to directly impact a real implementation of hash functions.

Fortunately, it seems that in practice simple hash functions perform very well. Indeed, they can be implemented very efficiently. For example, Dietzfelbinger et al. [18] exhibit a hash function that can be implemented with a single multiplication and a right shift operation and is almost universal. For scenarios where multiplications are undesirable, Carter and Wegman's original work [9] provides a universal hash function that relies on XOR operations. Some practical evaluations of these hash functions and others, for both hardware and software applications (including Bloom filters, discussed in Section 2.1.2), are given in [24,52–54,59]. Overall, these works suggest that it is possible to choose very simple hash functions that work very well in practical applications.

There is also theoretical work that strives to explain why simple hash functions seem to perform well in practice. One common approach is to examine a particular theoretical analysis that uses the assumption of fully random hash functions, and then attempt to modify the analysis to obtain a comparable result for a class of simple hash functions (e.g., universal hash functions), or a particular family of hash functions. For instance, it is an easy exercise to show that partitioned Bloom filters (described in Section 2.1.2) can be implemented with any universal hash function, albeit with a small increase in the false positive probability. As an example of this technique that works only for a specific hash family, Woelfel [68] shows that one can implement d-left hashing (described in Section 2.1.3) using a particular type of simple hash function. In a different direction, Mitzenmacher and Vadhan [47] show that for certain applications, if one is willing to assume that the set of items being hashed satisfies certain randomness properties, then any analysis based on the assumption that the hash functions are fully random is also valid with universal hash functions (up to some small, additional error probability). From a practical perspective, this work shows that it may be possible to construct some sort of statistical test that would provide a theoretical explanation for how well applications built on simple hash functions will work on a particular source of real data. Alternatively, if one is willing to assume that the set of items being hashed has a certain amount of entropy, then one can expect the same performance as derived from an analysis with fully random hash functions.

Having reviewed the approaches to hashing most related to this work, we now articulate our perspective on hash functions. This is essentially just the standard view in the networking literature, but it bears repeating. Since we are primarily concerned with real-world systems, and since it is usually possible to choose a simple, practical hash function for an application that results in performance similar to what we would expect for a fully random hash function, we allow ourselves to assume that our hash functions are fully random in our theoretical analyses. Thus, we take the perspective of *modeling* the hash functions for the sake of predicting performance in a statistical sense, as opposed to explicitly constructing the hash functions to satisfy concrete theoretical guarantees. Furthermore, since we assume that simple hash functions work well, we generally do not think of the cost of hashing as a bottleneck, and so we often allow ourselves to use hash functions liberally.

2.1.2 Bloom Filters and Their Variants

A *Bloom filter* [2] is a simple space-efficient randomized data structure for representing a set in order to support membership queries. We begin by reviewing the fundamentals, based on the presentation of the survey [7], which we refer to for further details. A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n items from a large universe U consists of an array of m bits, initially all set to 0. The filter uses k independent (fully random) hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$. For each item $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. (A location can be set to 1 multiple times.) To check if an item y is in S, we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an item not in the set, or the *false positive probability*, can be estimated in a straightforward fashion, given our assumption that the hash functions are fully random. After all the items of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$p' = (1 - 1/m)^{kn} \approx \mathrm{e}^{-kn/m}.$$

In this section, we generally use the approximation $p = \mathrm{e}^{-kn/m}$ in place of p' for convenience.

If ρ is the proportion of 0 bits after all the n items are inserted in the Bloom filter, then conditioned on ρ the probability of a false positive is

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k = \left(1 - \mathrm{e}^{-kn/m}\right)^k.$$

These approximations follow since $\mathbf{E}[\rho] = p'$, and ρ can be shown to be highly concentrated around p' using standard techniques. It is easy to show that the expression $\left(1 - \mathrm{e}^{-kn/m}\right)^k$ is minimized when $k = \ln 2 \cdot (m/n)$, giving a false positive probability

f of

$$f = \left(1 - \mathrm{e}^{-kn/m}\right)^k = (1/2)^k \approx (0.6185)^{m/n}.$$

In practice, k must be an integer, and a smaller, sub-optimal k might be preferred since this reduces the number of hash functions that have to be computed.

This analysis provides us (roughly) with the probability that a single item $z \notin S$ gives a false positive. We would like to make a broader statement, that in fact this gives a false positive *rate*. That is, if we choose a large number of *distinct* items not in S, the fraction of them that yield false positives is approximately f. This result follows immediately from the fact that ρ is highly concentrated around p', and for this reason, the false positive probability is often referred to synonymously as the *false positive rate*. (However, note that if we choose a large number of items not in S that may contain repeats, the repeated items may cause the number of false positives to exceed the predicted rate.)

Before moving on, we note that sometimes Bloom filters are described slightly differently, with each hash function having a disjoint range of m/k consecutive bit locations instead of having one shared array of m bits. We refer to this variant as a *partitioned* Bloom filter. Repeating the analysis above, we find that in this case the probability that a specific bit is 0 is

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

and so, asymptotically, the performance is the same as the original scheme. In practice, however, the partitioned Bloom filter tends to perform slightly worse than the non-partitioned Bloom filter. This is explained by the observation that

$$\left(1 - \frac{1}{m}\right)^{kn} > \left(1 - \frac{k}{m}\right)^n$$

when $k > 1$, so partitioned filters tend to have more 1's than non-partitioned filters, resulting in larger false positive probabilities.

The standard Bloom filter naturally supports insertion operations: to add a new item x to the set represented by the filter, we simply set the corresponding bits of the filter to 1. Unfortunately, the data structure does not support deletions, since changing bits of the filter from 1 to 0 could introduce false negatives. Of course, if we wish to support deletions, we can simply replace each bit of the filter with a counter, initially set to 0. In this case, the filter naturally represents a *multi-set* S of items, rather than a set. To insert an item x into the filter, we now increment its corresponding counters $h_1(x), \ldots, h_k(x)$, and to delete an item known to be in the multi-set represented by the filter, we decrement those counters. To test whether an item y is in S, we can simply check whether all the counters $h_1(y), \ldots, h_k(y)$ are positive, obtaining a false positive if $y \notin S$ but none of the counters are 0. (More generally, we can test whether an item y occurs in S with multiplicity at least $\ell \geq 1$

by testing whether the counters $h_1(y),\ldots,h_k(y)$ are at least ℓ, with some probability of a false positive.)

This Bloom filter variant is called a *counting Bloom filter* [24]. Clearly, all of our prior analysis for standard Bloom filters applies to counting Bloom filters. However, there is a complication in choosing the number of bits to use in representing a counter. Indeed, if a counter overflows at some point, then the filter may yield a false negative in the future. It is easy to see that the number of times a particular counter is incremented has distribution $\text{Binomial}(nk,1/m) \approx \text{Poisson}(nk/m) = \text{Poisson}(\ln 2)$, by the Poisson approximation to the binomial distribution (assuming $k = (m/n)\ln 2$ as above). By a union bound, the probability that some counter overflows if we use b-bit counters is at most $m\mathbf{Pr}(\text{Poisson}(\ln 2) \geq 2^b)$. As an example, for a sample configuration with $n = 10000$, $m = 80000$, $k = (m/n)\ln 2 = 8\ln 2$, and $b = 4$, we have $f = (1/2)^k = 2.14\%$ and $m\mathbf{Pr}(\text{Poisson}(\ln 2) \geq 2^b) = 1.78 \times 10^{-11}$, which is negligible. (Of course, in practice k must be an integer, but the point is clear.) This sort of calculation is typical for counting Bloom filters.

We now describe a variant of counting Bloom filters that is particularly useful for high-speed data stream applications. The data structure is alternately called a *parallel multistage filter* [22] or a *count-min sketch* [12] ([22] applies the data structure to network measurement and accounting, while [12] shows how it can be used to solve a number of theoretical problems in calculating statistics for data streams). The input is a stream of *updates* (i_t, c_t), starting from $t = 1$, where each *item* i_t is a member of a universe $U = \{1,\ldots,n\}$, and each *count* c_t is an integer. The state of the system at time T is given by a vector $\mathbf{a}(T) = (a_1(T),\ldots,a_n(T))$, where $a_j(T)$ is the sum of all c_t for which $t \leq T$ and $i_t = j$. The input is typically guaranteed to satisfy the condition that $a_j(T) > 0$ for every j and T. We generally drop the T when the meaning is clear.

The structures consists of a two-dimensional array Count of counters with width w and depth d: $\text{Count}[1,1],\ldots,\text{Count}[d,w]$. Every entry of the array is initialized to 0. In addition, there are d independent hash functions $h_1,\ldots,h_d : \{1,\ldots,n\} \to \{1,\ldots,w\}$. (Actually, it is enough to assume that the hash functions are universal, as shown in [12]; the argument below also holds with this assumption.) To process an update (i,c), we add c to the counters $\text{Count}[1,h_1(i)],\ldots,\text{Count}[d,h_d(i)]$. Furthermore, we think of $\hat{a}_i = \min_{j\in\{1,\ldots,d\}} \text{Count}[j,h_j(i)]$ as being an estimate of a_i. Indeed, it is easy to see that $\hat{a}_i \geq a_i$ (using the assumption that $a_j > 0$ for every j).

We now derive a probabilistic upper bound on $\hat{a}_i$. For $j \in \{1,\ldots,d\}$, let

$$X_{i,j} = \sum_{i' \neq i : h_j(i') = h_j(i)} a_{i'}.$$

Since the hash functions are fully random, $\mathbf{E}[X_{i,j}] \leq \|\mathbf{a}\|/w$, where $\|\mathbf{a}\| = \sum_k a_k$ (the L_1 norm of $\mathbf{a}$, assuming all of the entries in $\mathbf{a}$ are nonnegative). Markov's inequality then implies that for any threshold value $\theta > 0$, we have $\mathbf{Pr}(X_{i,j} \geq \theta) \leq \|\mathbf{a}\|/w\theta$. Now we note that $\hat{a}_i = a_i + \min_{j\in 1,\ldots,d} X_{i,j}$ and use independence of the h_j's to conclude that

$$\mathbf{Pr}(\hat{a}_i \geq a_i + \theta) \leq \left(\frac{\|\mathbf{a}\|}{w\theta}\right)^d.$$

In particular, if we fix some parameters $\varepsilon, \delta > 0$ and set $w = \lceil \mathrm{e}/\varepsilon \rceil$, $d = \lceil \ln(1/\delta) \rceil$, and $\theta = \varepsilon \|\mathbf{a}\|$, then we obtain

$$\mathbf{Pr}(\hat{a}_i \geq a_i + \varepsilon \|\mathbf{a}\|) \leq \left(\frac{1}{\mathrm{e}}\right)^{\ln(1/\delta)} = \delta.$$

In essence, we have that $\hat{a}_i$ is likely to be a fairly good estimate of a_i as long as a_i is not too small.

Under the assumption that all c_t are nonnegative, we can optimize this data structure further using a technique called *conservative updating* [22]. The basic idea is to never increment a counter more than is strictly necessary in order to guarantee that $\hat{a}_i \geq a_i$. Formally, to process an update (i, c), we let $c' = \min_{j \in \{1,\ldots,d\}} \mathrm{Count}[j, h_j(i)]$ and then set $\mathrm{Count}[j, h_j(i)] = \max(c + c', \mathrm{Count}[j, h_j(i)])$ for $j = 1, \ldots, d$. In particular, a counter is not updated if it is already larger than the count associated with the item, which is c' (the minimum over all the counters associated with i before the update) plus the update value c. We define the estimate $\hat{a}_i$ of a_i as before. It is easy to see that we still have $\hat{a}_i \geq a_i$ under this approach, but now the counters are not incremented as much. Intuitively, this technique further reduces the impact of small items on the estimates for large items. Experiments show that the improvement can be substantial [22].

A further variant of a Bloom filter extending the paradigm in a different direction is the *Bloomier filter*, which keeps a function value associated with each of the set items, thereby offering more than set membership [10]. More specifically, for each item x in a set S, there can be an associated fixed r-bit value $f(x) \in \{1, \ldots, 2^r - 1\}$; the 0 value is meant for items not in S. Given any item $x \in S$, the Bloomier filter correctly returns $f(x)$, and for any item $y \notin S$, the Bloomier filter should return 0. Here, a false positive occurs when $y \notin S$ but the Bloomier filter returns a non-zero value. As shown in [10], a Bloomier filter can be implemented near-optimally using a cascade of Bloom filters, although there are somewhat more complex constructions with better asymptotic performance.

We emphasize that the performance of a Bloom filter does not depend at all on the size of the items in the set that it represents (except for the additional complexity required for the hash functions for larger items). Thus, although Bloom filters allow false positives, the space savings over more traditional data structures for set membership, such as hash tables, often outweigh this drawback. It is therefore not surprising that Bloom filters and their many variations have proven increasingly important for many applications (see, for instance, the survey [7]). As just a partial listing of additional examples of the proliferation of Bloom filter variations, *compressed Bloom filters* are optimized to minimize space when transmitted [44], *retouched Bloom filters* trade off false positives and false negatives [20], and *approximate concurrent state machines* extend the concept of a Bloomier filter by tracking the dynamically changing state of a changing set of items [3]. Although recently more complex but asymptotically better alternatives have been proposed (e.g., [4, 49]), the Bloom filter's simplicity, ease of use, and excellent performance make it a standard data structure that is and will continue to be of great use in many applications.

2.1.3 Hash Tables

The canonical example of a hash table is one that uses a single hash function (which is assumed to be fully random) with chaining to deal with collisions. The standard analysis shows that if the number of buckets in the table is proportional to the number of items inserted, the expected number of items that collide with a particular item is constant. Thus, on average, lookup operations should be fast. However, for applications where such average case guarantees are not sufficient, we also need some sort of probabilistic worst case guarantee. Here, the qualifier that our worst case guarantees be *probabilistic* excludes, for instance, the case where all items in the table are hashed to the same bucket. Such situations, while technically possible, are so ridiculously unlikely that they do not warrant serious consideration (at least from a theoretical perspective). As an example of a probabilistic worst case guarantee, we consider throwing n balls independently and uniformly at random into n bins. In this case, a classical result (e.g., [46, Lemmas 5.1 and 5.12] or the original reference by Gonnet [29]) shows that the maximum number of balls in a bin is $\Theta((\log n)/\log\log n)$ with high probability. This result translates directly to a probabilistic worst case guarantee for a standard hash table with n items and n buckets: while the expected time to lookup a particular item is constant, with high probability the longest time that *any* lookup can require is $\Theta((\log n)/\log\log n)$.

The previous example illustrates a connection between hashing and *balanced allocations*, where some number of balls is placed into bins according to some probabilistic procedure, with the implicit goal of achieving an allocation where the balls are more-or-less evenly distributed among the bins. In a seminal work, Azar et al. [1] strengthened this connection by showing a very powerful balanced allocation result: if n balls are placed sequentially into $m \geq n$ bins for $m = O(n)$, with each ball being placed in one of a constant $d \geq 2$ randomly chosen bins with minimal load at the time of its insertion, then with high probability the maximal load in a bin after all balls are inserted is $(\ln\ln n)/\ln d + O(1)$. In particular, if we modify the standard hash table with chaining from above to use d hash functions, inserting an item into one of its d hash buckets with minimal total load, and performing a lookup for an item by checking all d of its hash buckets, then the expected lookup time is still constant (although larger than before), but the probabilistic worst case lookup time drops exponentially. This scheme, usually called *d-way chaining*, is arguably the simplest instance of a *multiple choice hash table*, where each item is placed according to one of several hash functions.

Unsurprisingly, the impact of [1] on the design of randomized algorithms and data structures, particularly hash tables and their relatives, has been enormous. For details and a more complete list of references, we refer to the survey [45]. Before moving on, however, we mention an important improvement of the main results in [1] due to Vöcking [65]. That work exhibits the *d-left* hashing scheme, which works as follows. There are n items and m buckets. The buckets are partitioned into d groups of approximately equal size, and the groups are laid out from left to right. There is one hash function for each group, mapping the items to a randomly chosen bucket in the group. The items are inserted sequentially into the table, with an item being inserted into the least loaded of its d hash buckets (using chaining), with ties broken to the left. Vöcking [65] shows that if $m = n$ and $d \geq 2$ is constant, then the maximum load of a bucket after all the items are inserted is $(\ln \ln n)/d\phi_d + O(1)$, where ϕ_d is the asymptotic growth rate of the d-th order Fibonacci numbers. In particular, this improves the factor of $\ln d$ in the denominator of the $(\ln \ln n)/\ln d + O(1)$ result of Azar et al. [1]. Furthermore, [65] shows that d-left hashing is optimal up to an additive constant. Interestingly, both the partitioning and the tie-breaking together are needed to obtain this improvement.

Both d-way chaining and d-left hashing are practical schemes, with d-left hashing being generally preferable. In particular, the partitioning of the hash buckets into groups for d-left hashing makes that scheme more amenable to a hardware implementation, since it allows for an item's d hash locations to be examined in parallel. For high-speed packet processing applications, however, hashing schemes that resolve collisions with chaining are often undesirable. Indeed, for these applications it is often critical that almost everything be implemented cleanly in hardware, and in this case the dynamic memory allocation requirements of hashing schemes that use chaining are problematic. Thus, we prefer *open-addressed* hash tables where each bucket can store a fixed constant number of items (typically determined by the number of items that can be conveniently read in parallel). Of course, we can simulate a hashing scheme that uses chaining with an open-addressed hash table as long as no bucket overflows, and then we just need to ensure that it is highly unlikely for a bucket to overflow. Alternatively, we can work directly with open-addressed hashing schemes that are explicitly designed with a limit on the number of items that can be stored in a bucket. In this case, for the sake of simplicity, we typically assume that each bucket can hold at most one item. The results can usually be generalized for larger buckets in a straightforward way. The potential expense of using open-addressed hash tables in these ways is that many buckets may be far from full, wasting significant space.

The standard open-addressed multiple choice hash table is the multilevel hash table (MHT) of Broder and Karlin [6]. This is a hash table consisting of d subtables $T_1, \ldots, T_d$, with each T_i having one hash function h_i. We view these tables as being laid out from left to right. To insert an item x, we find the minimal i such that $T_i[h_i(x)]$ is unoccupied, and place x there. As above, we assume that each bucket can store at most one item; in this case the MHT is essentially the same as a d-left hash table with the restriction that each bucket can hold at most one item, but the correspondence disappears for larger bucket sizes. If $T_1[h_1(x)], \ldots, T_d[h_d(x)]$ are all occupied, then we declare a *crisis*. There are multiple things that we can do to handle a crisis. The approach in [6] is to resample the hash functions and rebuild the entire table. That work shows that it is possible to insert n items into a properly designed MHT with $O(n)$ total space and $d = \log\log n + O(1)$ in $O(n)$ expected time, assuming only 4-wise independent hash functions.

In Section 3, we discuss more recent work that describes ways to design MHTs so that no rehashings are necessary in practice. Essentially, the idea is that if the T_i's are (roughly) geometrically decreasing in size, than the total space of the table is $O(n)$. If the ratio by which the size of T_{i+1} is smaller than T_i is, say, twice as large as the expected fraction of items that are not stored in $T_1, \ldots, T_i$, then the distribution of items over the T_i's decreases doubly exponentially with high probability. This double exponential decay allows the choice of $d = \log\log n + O(1)$. For a more detailed description of this intuition, see [6] or [32].

We defer the details of the various ways to construct MHTs to Sections 3 and 4, where MHTs play a critical role. For the moment, however, we simply note that MHTs naturally support deletions, as one can just perform a lookup on an item to find its location in the table, and then mark the corresponding item as deleted. Also, MHTs appear well-suited to a hardware implementation. In particular, their open-addressed nature seems to make them preferable to approaches that involve chaining, and their use of separate sub-tables allows for the possibility that all of the hash locations for a particular item can be accessed in parallel. Indeed, these considerations are part of the original motivation from [6].

There is also a substantial amount of work in the theory literature on open-addressed multiple choice hashing schemes that allow items in the table to be moved during an insertion in order to increase space utilization [13, 19, 28, 34, 50, 51]. The most basic of these schemes is *cuckoo hashing* [28, 50], which works as follows. There are d sub-tables $T_1, \ldots, T_d$, with each T_i having one hash function h_i. When attempting to insert an item x, we check if any of its hash locations $T_1[h_1(x)], \ldots, T_d[h_d(x)]$ are unoccupied, and place it in the leftmost unoccupied bucket if that is the case. Otherwise, we choose a random $I \in \{1, \ldots, d\}$ and evict the item y in $T_I[h_I(x)]$, replacing y with x. We then check if any of y's hash locations are unoccupied, placing it in the leftmost unoccupied bucket if this is the case. Otherwise, we choose a random $J \in \{1, \ldots, d\} - \{I\}$ and evict the item z in $T_J[h_J(y)]$, replacing it with y. We repeat this procedure until an eviction is no longer necessary.

Cuckoo hashing allows for a substantial increase in space utilization over a standard MHT with excellent amortized insertion times (even for small d, say, $d = 4$). Unfortunately, however, in practice a standard cuckoo hash table occasionally experiences insertion operations that take significantly more time than the average. This issue is problematic for high-speed packet processing applications that have very strict worst case performance requirements. We address this issue further in Section 4.

2.2 Application Performance Measures and Memory Models

Roughly speaking, the quality of the algorithms and data structures in this chapter can be measured by their space requirements and speed for various operations.

In conventional algorithm analysis, speed is measured in terms of processing steps (e.g., instructions). However, as a first approximation, we count only memory accesses as processing steps. In a hardware design, this approximation is usually justified by the ability to perform multiple complex processing steps in a single cycle in hardware (using combinatorial logic gates, which are plentiful). In a software design, we can often ignore processing steps because instruction cycle times are very fast compared to memory access times.

Thus our main application performance measures are usually the amount of memory required and the number of memory accesses required for an operation of interest. Unfortunately, these measures are more complicated than they may first appear because there are different types of memories: *fast memory* (cache in software, SRAM in hardware), and *slow memory* (main memory in software, DRAM in hardware). The main space measure is typically the amount of fast memory required for a technique. If a design only uses slow memory, then the amount of memory used is often irrelevant because such memory is typically cheap and plentiful. Similarly, if a design uses both fast and slow memory, the main speed measure is typically the number of slow memory accesses (because fast memory accesses are negligible in comparison). If a design uses only fast memory, then the speed measure is the number of fast memory accesses.

To make this abstraction more concrete, we give a brief description and comparison of SRAM and DRAM. Typical SRAM access times are 1-2 nsec for on-chip SRAM and 5-10 nsec for off-chip SRAM; it is possible to obtain on-chip SRAMs with 0.5 nsec access times. On-chip SRAM is limited to around 64 Mbits today. The level 1 and level 2 caches in modern processors are built from SRAM.

In order to refresh itself, an SRAM bit cell requires at least 5 transistors. By comparison, a DRAM cell uses only a single transistor connected to an output capacitance that can be manufactured to take much less space than the transistors in an SRAM. Thus SRAM is less dense and more expensive (per bit) than memory technology based on DRAM. However, the compact design of a DRAM cell has an important negative consequence: a DRAM cell requires higher latency to read or write than the SRAM cell. The fastest off-chip DRAMs take around 40-60 nsec to access (latency) with longer times such as 100 nsec between successive reads (throughput). It seems clear that DRAM will always be denser but slower than SRAM.

Moving on, three major design techniques are commonly used in memory subsystem designs for networking chips and can be used for the algorithms and data structures in this chapter. While we will not dwell on such low-level issues, it is important to be aware of these techniques (and their limitations).

- **Memory Interleaving and Pipelining:** Many data structures can be split into separate banks of an interleaved memory, where each bank is a DRAM. Memory accesses can then be interleaved and pipelined to facilitate parallelism. For example, consider a binary search tree data structure, and suppose that it is split into separate banks based on nodes' locations in the tree. If we wish to perform multiple lookups on the tree, we can boost performance by allowing operations at different nodes of the tree to occur in parallel. Parallelism is quite feasible in modern ASICs up to a point (less than 50-fold parallelism).

- **Wide Word Parallelism:** A common theme in many networking designs is to use wide memory words that can be processed in parallel. This can be implemented using DRAM and exploiting the *page mode*, or by using SRAM and making each memory word wider. In software designs, wide words can be exploited by aligning data structures to the cache line size. In hardware designs, one can choose the width of memory to fit the problem (up to say 5000 bits or so, after which electrical issues may become problematic).

- **Combining DRAM and SRAM:** Given that SRAM is expensive and fast, and DRAM is cheap and slow, it makes sense to combine the two technologies to attempt to obtain the best of both worlds. The simplest approach is to simply use some SRAM as a cache for a DRAM database. While this technique is classical, there are many more creative applications of the use of non-uniform memory models; we will see some in this chapter.

We must also point out that routers often make use of content-addressable memories (CAMs), which are fully associative memories, usually based on SRAM technology (so that improvements in SRAM technology tend to translate into improvements in CAM technology), that support a lookup of a data item in a single access by performing lookups on all memory locations in parallel. There are also ternary CAMs (TCAMs) that support wildcard bits, which is an extremely useful feature for prefix match lookups, a critical router application discussed in Section 2.3. This specialty hardware is much faster than any data structure built with commodity SRAM, such as a hash table, could ever be. However, the parallel lookup feature of CAMs causes them to use a lot more power than comparable SRAMs. Furthermore, the smaller market for this sort of technology results in CAMs being much more expensive, per bit, than SRAMs. For both peak power consumption and cost per bit, an order of magnitude difference between a CAM and a comparable SRAM would not be surprising.

In this chapter, we regard CAMs as being expensive, special-purpose hardware for table lookups that are practical only when storing small sets of items. Thus, we do not think of CAMs as being a replacement for hash tables, but we do advocate their use for parts of our hash-based data structures. In particular, in Section 4, we describe MHTs from which some small number of items are expected to overflow. There, a CAM is used to store those items (but not all items, which is prohibitively expensive), allowing excellent worst case lookup times for the entire data structure.

2.3 History of Hash-Based Techniques

As mentioned in the introduction, the primary purpose of this chapter is to survey the major recent developments in the literature on hash-based techniques for high-speed packet processing. In this section, we give a brief overview of the relevant history covered by the text [62], published in 2004. We refer the reader to [62] and the references below for further details.

Historically, hashing based on flow IDs has been used for load-balancing packets across network paths (see for example, RFC 2991 [58]) instead of schemes like round-robin that do not preserve FIFO ordering. This chapter, however, concentrates on packet processing tasks such as lookups and measurement.

The earliest directly relevant application of a hash-based technique is probably the implementation of bridge lookups in the DEC GIGAswitch system [60]. The GIGAswitch was a switch designed in early 1990's to connect up to 32 100 Mbps FDDI links, with the main technical challenge being the need for wire-speed forwarding. This requirement made the earlier bridge lookup algorithms, which were based on binary search and designed to work at lower speeds, impractical. Ultimately, the issue was addressed by using a hash table. To ensure that the hash function was suitable (i.e., that it did not hash too many items to the same bucket), the hash function was simply resampled until all items were suitably accommodated in the table. (The particular hash function used was multiplication by a random polynomial modulo a particular irreducible polynomial over a finite field of characteristic two, although the hash function itself is not important for our purposes.) For the sake of completeness, we note that there was a small content-addressable memory (CAM) that could store some overflow from the table, which was checked in parallel with the table; the hash function was actually resampled only when an item could not be inserted into its bucket in the table and the CAM was full. While this use of a CAM is common in networking applications, it is only recently being considered in the theory literature on hashing; we discuss this more in Section 4. From a theoretical perspective, this implementation of hashing was not particularly novel, but it has been enormously influential in the design of real networking products. (Like many great ideas, it seems obvious in retrospect.)

The next major application of hashing-based techniques in networking hardware described in the literature was the scheme of Waldvogel et al. [66] for longest prefix match for fast IP lookups. (The *longest prefix match* problem is essentially the task of building a data structure for a set of strings that, for a query string, finds a longest string in the data structure that is a prefix of the query string; see [62] for details.) The basic idea was to keep hash tables for all possible prefix lengths and then perform the longest prefix match by a binary search over those tables. Many optimizations were needed to make this approach practical [8, 61, 66]. For our purposes, the most significant observation was that the scalability of the scheme was basically determined by the space utilization achievable by the hash tables. Indeed, the scheme required that no hash bucket receive more than a certain number of items, and the hash function was resampled until that condition was satisfied. Thus, improving the space utilization of hash tables became a problem of direct interest for implementing networking hardware.

In particular, Srinivisan and Varghese [61] pointed out that using wide memory words (easily done in hardware) ameliorated the problems of resampling the hash function and improved space utilization. However, Broder and Karlin [6] had earlier shown that the use of parallelism via multiple hash functions was also a powerful tool for this purpose. The next logical development was to combine parallelism and wide words. This was done through an application of d-left hashing by Broder and Mitzenmacher [8], increasing the space utilization considerably. It appears that d-left hashing is the "state of the art" way to implement a hash table in hardware; many current products use this scheme.

While Bloom filters have for some time been proposed for networking applications [7], only recently have they been considered for speeding up core networking hardware. For example, Dharmapurikar et al. [15] appear to have been the first to suggest utilizing Bloom filters to speed up longest prefix matching algorithms. They keep hash tables for all possible prefix lengths in slow memory as in [66], but with the addition that a Bloom filter corresponding to each hash table is also stored in fast memory so that the hardware can quickly find the length of the longest matching prefix. Assuming no false positives, the hardware then has to do only 1 more lookup to slow memory to confirm the match. Similar ideas are applied to packet classification and string matching in [14, 16, 17].

A further development beyond lookups for networking hardware was the introduction by Estan and Varghese [22] of hashing-based data structures specifically designed to facilitate network measurement and accounting. The main contribution of that work is the count-min sketch (there called a *parallel multistage filter*) and the accompanying conservative update heuristic described in Section 2.1.2. The analysis presented in Section 2.1.2 is essentially taken from the more theoretical paper [12] for generality, although it is important to note that [22] contains a significant amount of analysis for the specific applications of interest. In particular, [22] contains a detailed evaluation of how count-min sketches perform for the task of identifying large network flows in real data.

Later, Singh et al. [56] used the methods of [22] for extracting worm signatures at high speeds. The basic premise of [56] is that if hashing methods are useful for measurement, then they can also be used to measure patterns indicative of network security attacks. This theme is continued in later work. Examples include hash-based algorithms to detect denial-of-service attacks based on detecting so-called *superspreaders* [64] and algorithms to detect sources that send control packets to start sessions without corresponding end control packets [35].

We also note that there are likely countless applications of innovative hashing-based techniques in networking hardware that are not described in the literature. Unfortunately, few concrete aspects of these designs are discussed openly, primarily because the actual implementations of these techniques are widely considered to be trade secrets by router manufacturers. Broadly speaking, though, hash tables and related data structures are considered generally useful for a variety of packet classification, network monitoring, and lookup applications.

Finally, we note that while it is common for a high-speed packet processing application to be amenable (at least in principle) to a hash-based approach, there is something of a historical bias against these techniques. The issue is that hash tables and related data structures can only offer probabilistic guarantees, as opposed to the more commonly accepted deterministic worst case performance bounds offered by classical algorithmic approaches. For instance, for the longest prefix match problem described above, a hardware designer may prefer (simply as a matter of philosophy) a trie-based approach (see, for example, [62]) to a variant of the scheme of Waldvogel et al. [66], whose effectiveness ultimately relies on the efficiency of hash table lookups, which can be difficult to quantify. While this attitude may be changing, it is important to remember that the sorts of probabilistic guarantees offered by hash-based solutions always depend strongly on certain randomness assumptions (i.e., the assumption that hash functions are fully random), and so convincingly evaluating the performance of such a scheme requires extensive analysis and experimentation.

3 Lookups in a Non-Uniform Memory Model: On-Chip Summaries of Off-Chip Hash Tables

We start by examining a problem first addressed by Song et al. [57]. Recall that in a multiple choice hashing scheme with d hash functions, one performs a lookup for an item x by examining (in the worst case) all d hash locations corresponding to x. In a hardware application, it may be reasonable to implement this procedure by examining all d locations in parallel, particularly if the hash table memory is stored on the chip that is performing the lookup. However, if the hash table must be stored off-chip (due to its size), then performing all d of these lookups in parallel may introduce a prohibitively expensive cost in terms of chip I/O, particularly the number of pins on the chip that are needed to access the hash table. It then becomes natural to ask whether we can design some sort of on-chip *summary* that can reduce the number of worst case (off-chip) memory accesses to the hash table from d to, say, 1. This is the question addressed in [57].

More formally, the summary answers questions of the following form, “Is item x in the hash table, and if so, in which of its d hash locations is it?” The summary is allowed some small false positive probability (e.g., 0.5%) for items not in the hash table, since these are easily detected and so do not significantly impact performance as long as they are infrequent. However, if a queried item x is in the hash table, the summary should always correctly identify the hash location used to store x, unless some sort of unlikely failure condition occurs during the construction of the summary. (In particular, if the summary is successfully constructed, then it does not generate false negatives and the worst case number of off-chip memory accesses is 1.) The objective is now to design a summary data structure and the corresponding hash table so that they are efficient and the summary data structure is successfully constructed with overwhelming probability.

The basic scheme proposed by Song et al. [57] is as follows. (Essentially, it is a combination of a counting Bloom filter variant for the summary with a variant of d-way chaining for the hash table.) For simplicity, we start by describing the variant where the hash table is built by inserting n items, and after that it is never modified. Here, the hash table consists of m buckets and d hash functions, and the summary consists of one b-bit counter for each bucket. When an item is inserted into the hash table, it is placed in all of its d hash buckets and all of the corresponding counters are incremented. Then the hash table is *pruned*; for each item in the table, the copy in the bucket whose corresponding counter is minimal (with ties broken according to the ordering of the buckets) is kept, and the rest are deleted. A query to the summary for an item x is now answered by finding the smallest of its d counters (again with ties broken according to the ordering of the buckets). If the value of this counter is 0, then x cannot be in the table, and otherwise x is presumed to be in the corresponding bucket.

Song et al. [57] give several heuristic improvements to this basic scheme in order to reduce collisions in the underlying hash table and optimize performance for various applications. The heuristics appear effective, but they are only analyzed through simulations, and do not appear to be amenable to theoretical or numerical analyses. Insertions can be handled, but can necessitate moving items in the hash table. Deletions are significantly more challenging in this setting than in most hash table constructions. In particular, it may be necessary to keep a copy of the entire hash table before pruning (or smaller variant called a shared-node fast hash table); see [57] for details.

Kirsch and Mitzenmacher [32] give a different approach to this problem. First, they propose the use of an MHT as the underlying hash table. This gives a worst case bound on the number of items in a bucket, although it introduces the possibility of a crisis, which must be very unlikely in order for the MHT to give good performance. Furthermore, since the distribution of items over the sub-tables of an MHT decays doubly exponentially with high probability, it suffices to design summaries that perform well under the assumption that most of the items in the MHT are in the first sub-table, most of the rest are in the second, etc.

Kirsch and Mitzenmacher [32] propose two summary design techniques based on this observation, both based on Bloom filter techniques. We review the one that is easier to describe and motivate here. As before, for simplicity we start by assuming that we are only interested in building a hash table and corresponding summary for n items by inserting the n items sequentially, and then the data structures are fixed for all time. If the MHT consists of d sub-tables $T_1,\ldots,T_d$, then the summary consists of d Bloom filters $F_1,\ldots,F_d$. Each filter F_i represents the set of all items stored in $T_i,\ldots,T_d$. To perform a query for an item x, we first check whether F_1 yields a positive for x; if not, then x cannot be in the MHT. Otherwise, we find the largest i where F_i returns a positive for x, and declare that x is in T_i.

The first important observation here is that F_1 is simply a standard Bloom filter for the set of items stored in the MHT. Thus, false positives for F_1 merely yield false positives for the summary. As before, such false positives are acceptable as long as they are sufficiently infrequent (e.g., the false positive probability of F_1 is 0.5%). However, if an item x is in T_i, then it will be inserted into $F_1,\ldots,F_i$ but not F_{i+1}. If F_{i+1} gives a false positive for x, then querying the summary for x yields an incorrect answer, which is unacceptable to us because x is actually in the MHT. Thus, $F_2,\ldots,F_d$ must have extremely small false positive probabilities.

The second important observation is the effect of the distribution of the items over the sub-tables of the MHT on the quality of the summary. Recall that for a typical MHT, with high probability, most of the items are stored in T_1, most of the rest are stored in T_2, etc. In fact, the distribution of items over the sub-tables decays doubly exponentially. In particular, if S_i is the set of items stored in $T_i, \ldots, T_d$, then S_i is almost surely small for $i \geq 2$. Thus, for $i \geq 2$, since F_i is just a standard Bloom filter for S_i, we can achieve an extremely small false probability for F_i without using too much space. Since we only need a moderate false positive probability for F_1 (as described above), we can adequately construct it in a reasonable amount of space.

A further advantage of the approach of [32] is that one can obtain fairly precise estimates of the probabilities that the relevant hash-based data structures fail using numerical techniques. This property is very useful, as purely theoretical techniques often obscure constant factors and simulation results can be computationally expensive to obtain (especially when the probabilities being estimated are very small, as they should be for the sorts of failure probabilities that we are interested in bounding). Numerical techniques offer the potential for very accurate and predictive results for only a moderate computational cost.

For the summary approach described above, it is fairly easy to see that if we can obtain numerical information about the distributions of the sizes of the S_i's, then the standard Bloom filter analysis can be directly applied to estimate the failure probability of the summary. The primary issue then becomes obtaining this numerical information (note that the probability of a crisis is just $\mathbf{Pr}(|S_1| < n)$, and so it too follows from this information). Now, if $m_1, \ldots, m_d$ are the sizes of $T_1, \ldots, T_d$, then it is fairly easy to see that the distribution of the number of items that are stored in T_1 is the same as the number of bins that are occupied if we throw n balls at random into m_1 buckets. Furthermore, for $i > 1$, the conditional distribution of the number of items in T_i given that N_{i-1} items are stored in $T_1, \ldots, T_{i-1}$ is the same as the number of bins that are occupied if we throw $n - N_{i-1}$ balls into m_i buckets. These balls-and-bins probabilities are easy to compute and manipulate to obtain the desired information about the distribution of the S_i's; for details, see [32]. As an example of the improvement, for 10,000 items these techniques allow for a hash table that uses 50% of the space as the hash table in [57] and an accompanying summary that uses about 66% of the space of the corresponding summary from [57], for a comparable false positive probability and a failure probability of about 5×10^{-12}; of course, different tradeoffs are possible (e.g., a larger hash table to get more skew in the distribution of the items, allowing for a smaller summary).

As is usually the case for hash-based data structures, it is more difficult to measure the effectiveness of these techniques when deletion operations are allowed. However, we do note that the summary construction above can be modified to handle deletions by replacing the Bloom filters with counting Bloom filters. Unfortunately, while the MHT supports deletions in the natural way (an item can simply be removed from the table), intermixing deletion and insertion operations can have a major impact on the distribution of the S_i's, which is critical for the failure probability guarantees. Thus, while the approach of [32] seems adaptable to deletions, understanding and mitigating the impact of deletions on these sorts of data structures remains an important open problem.

In more practical work, Kumar et al. [37, 38] use similar ideas to construct alternative high-performance hash tables. The paper [37] presents a variant of a multiple choice hash table. A hash table is broken into multiple segments, each of which can be thought of as a separate hash table; each item chooses a bucket in each segment. If an item cannot be placed without a collision, a standard collision-resolving technique, such as double hashing, is applied to the segment where the item is placed. To avoid searching over all segments when finding an item, a Bloom filter is used for each segment to record the set of items placed in that segment. When an item is placed, the first priority is to minimize the length of the collision chain, or the search time, for that item. Often, however, there will be ties in the search length, in which case priority is given to a segment where the new item will introduce the fewest new 1's into the Bloom filter; this reduces the chances of false positives. (Some related theory is presented in [43]).

The paper [38] introduces *peacock hashing*, which takes advantage of the skewed construction developed for MHT's. The main innovation of [38] appears to arise from using more limited hash functions in order to improve rebalancing efforts in case of deletion. In sub-tables beyond the first, the possible locations of an item depend on its location in the previous table. Because of this, when an item is deleted, there are only a very small number of possible locations in the subsequent sub-tables that need to be checked to see if an item from a later, smaller table can be moved to the now empty position in the larger, earlier table, potentially reducing the probability of a crisis. In exchange, however, one gives up some of the power of hashing each item independently to multiple locations. Also, currently peacock hashing lacks a complete mathematical analysis, making it hard to compare to other schemes except by experiment.

4 Lookups in a Uniform Memory Model: Hardware Hash Tables with Moves

We now turn our attention to the setting where it is feasible to examine all of an item's hash locations in a multiple choice hash table in parallel. In this case, our goal becomes to increase the space utilization of our hash table constructions, while ensuring that they are amenable to hardware implementations for high-speed applications. For instance, one can think of these techniques as potentially enabling us to take an off-chip hash table and decrease its space overhead enough so that it can be effectively implemented on-chip, thus eliminating the chip I/O problem from Section 3 and replacing it with a very restrictive memory constraint.

As discussed in Section 2.1.3, there are a number of hashing schemes in the theory literature that allow items to be moved in the table during the insertion of a new item in order to increase the space utilization of the table. Unfortunately, while these schemes have excellent amortized performance, they occasionally allow for insertion operations that take a significant amount of time. For high-speed packet processing applications, such delays may be unacceptable.

We discuss two approaches to this problem. First, we consider new hashing schemes that are designed to exploit the potential of moves while ensuring a reasonable worst case time bound for a hash table operation. Second, we consider a more direct adaptation of an existing scheme from the theory literature (specifically, cuckoo hashing) to this setting, striving for more de-amortized performance guarantees.

4.1 The First Approach: The Power of One Move

The first approach is taken by Kirsch and Mitzenmacher in [31]. That work proposes a number of modifications to the standard MHT insertion scheme that allow at most one move during an insertion operation. The opportunity for a single move demonstrates that the space utilization of the standard MHT can be significantly increased without a drastic increase in the worst case time of a hashing operation. The proposed schemes are all very similar in spirit; they differ primarily in the tradeoffs between the amount of time spent examining potential moves during an insertion and the resulting increases in space utilization.

The core idea behind these schemes is best illustrated by the following procedure, called the *second chance scheme*. Essentially, the idea is that as we insert items into a standard MHT with sub-tables $T_1, \dots, T_d$, the sub-tables fill up from left to right, with items cascading from T_i to T_{i+1} with increasing frequency as T_i fills up. Thus, a natural way to increase the space utilization of the table is to slow down this cascade at every step.

This idea is implemented in the second chance scheme in the following way. We mimic the insertion of an item x using the standard MHT insertion procedure, except that if we are attempting to insert x into T_i, if the buckets $T_i[h_i(x)]$ and $T_{i+1}[h_{i+1}(x)]$ are occupied, rather than simply moving on to T_{i+2} as in the standard scheme, we check whether the item y in $T_i[h_i(x)]$ can be moved to $T_{i+1}[h_{i+1}(y)]$. If this move is possible (i.e., the bucket $T_{i+1}[h_{i+1}(y)]$ is unoccupied), then we perform the move and place x at $T_i[h_i(x)]$. Thus, we effectively get a *second chance* at preventing a cascade from T_{i+1} to T_{i+2}.

Just as in the standard MHT insertion scheme, there may be items that cannot be placed in the MHT during the insertion procedure. Previously, we considered this to be an extremely bad event and strived to bound its probability. Here we take a different perspective and say that if an item x is not successfully placed in the MHT during its insertion, then it is placed on an overflow list L, which, in practice, would be implemented with a CAM. To perform a lookup, we simply check L in parallel with the MHT.

It turns out that since the second chance scheme only allows moves from left to right, it is analyzable by a *fluid limit* or *mean-field* technique, which is essentially a way of approximating stochastic phenomena by a deterministic system of differential equations. The technique also applies to the standard MHT insertion procedure, as well as a wide variety of extensions to the basic second chance scheme. This approach makes it possible to perform very accurate numerical analyses of these systems, and in particular it allows for some interesting optimizations. We refer to [31] for details, but as an example, we note that when both the standard MHT insertion scheme and second chance insertion scheme are optimized to use as little space as possible with four hash functions so that no more than 0.2% of the items are expected to overflow from the table under either scheme, then the second chance scheme requires 72% of the space of the standard scheme, with about 13% of insertion operations requiring a move.

The second chance scheme is also much more amenable to a hardware implementation than it may a first seem. To insert an item x, we simply read all of the items $y_1 = T_1[h_1(x)], \ldots, y_d = T_d[h_d(x)]$ in parallel. Then we compute the hashes $h_2(y_1), \ldots, h_d(y_{d-1})$ in parallel. (Here, for notational simplicity, we are assuming that all of $T_1[h_1(x)], \ldots, T_d[h_d(x)]$ are occupied, so that the y_i's are well-defined; it should be clear how to handle the general case.) At this point, we now have all of the information needed to execute the insertion procedure without accessing the hash table (assuming that we maintain a bit vector indicating which buckets of the table are occupied).

The second chance scheme and its relatives also support deletions in the natural way: an item can simply be removed from the table. However, as in Section 3, the intermixing of insertions and deletions fundamentally changes the behavior of the system. In this case, the differential equation approximations become much more difficult and heuristic, but still useful. For details, see [31].

4.2 The Second Approach: De-amortizing Cuckoo Hashing

We now discuss some possible adaptations of the standard cuckoo hashing scheme, proposed by Kirsch and Mitzenmacher [33], to obtain better de-amortized performance. Recall that in standard cuckoo hashing, the insertion of an item x corresponds to a number of *sub-operations* in the following way. First, we attempt to insert x into one of its hash locations. If that is unsuccessful, then we choose one of x's hash locations at random, evict the item y in that place, and replace it with x. We then attempt to place y into one its hash locations, and, failing that, we choose one of y's hash locations other than the one from which it was just evicted at random, evict the item z in that location, and replace it with y. We then attempt to place z similarly.

We think of each of these attempts to place an item in its hash locations as a sub-operation. In a hardware implementation of cuckoo hashing, it is natural to consider an *insertion queue*, implemented in a CAM, which stores sub-operations to be processed. To process a sub-operation, we simply remove it from the queue and execute it. If the sub-operation gives rise to another sub-operation, we insert the new sub-operation into the queue. When we wish to insert a new item into the hash table, the sub-operation we insert the corresponding initial attempt to place the item into the queue. Generally speaking, the queue is implemented with some policy for determining the order in which sub-operations should be processed. For the standard cuckoo hashing algorithm, this policy would be for sub-operations coming from newly inserted items to be inserted at the back of the queue, and a sub-operation arising from a sub-operation that was just executed to be inserted at the front of the queue.

The key feature of this approach is that we can efficiently perform insertions, lookups, and deletions even if we reorder sub-operations. Indeed, an insertion can be performed by inserting a single sub-operation into the queue, and a lookup can be performed by examining all of the items' hash locations in the table and the entire queue in parallel (since the queue is implemented with a CAM). To perform a deletion, we check whether the item is in the table, and if so we mark the corresponding bucket as deleted so that the item is overwritten by a future sub-operation. If the item is in the queue, we remove the corresponding sub-operation from the queue.

Since the queue must actually fit into a CAM of modest size (at least under ordinary operating conditions), the main performance issue is the size of the queue when it is equipped with a particular policy. For instance, the problem with standard cuckoo hashing policy is that it can become "stuck" attempting to process an unusually large number of sub-operations arising from a particularly troublesome insertion operation, allowing new insertion operations to queue up in the meantime. A natural first step towards fixing this problem is to insert the sub-operations corresponding to newly inserted items on the front of queue, rather than on the back. In particular, this modification exploits the fact that a newly inserted item has a chance of being placed in any of its d hash locations, whereas an item that was just evicted from the table has at most $d-1$ unoccupied hash locations (assuming that the item responsible for the eviction has not been deleted).

Another useful observation comes from introducing the following notion of the *age* of a sub-operation. If a sub-operation corresponds to the initial attempt to insert an item, then that sub-operation has age 0. Otherwise, the sub-operation results from the processing of another sub-operation with some age a, and we say that the new sub-operation has age $a+1$. The previous queuing policy can then be thought of as a modification of the standard policy to give priority to sub-operations with age 0. More generally, we can introduce a policy in which insertion operations are prioritized by their ages. Intuitively, this modification makes sense because the older a sub-operation is, the more likely the original insertion operation that gave rise to it is somehow troublesome, which in turn makes it more likely that this sub-operation will not give a successful placement in the table, resulting in a new sub-operation.

While it may not be practical to implement the insertion queue as a priority queue in this way, since sub-operations with large ages are fairly rare, we should be able to approximate the performance of the priority queue with following approach. As before, sub-operations corresponding to an initial insertion of an item are placed on the front of the queue. Furthermore, whenever the processing of a sub-operation yields a new sub-operation, the new sub-operation is placed on the back of the queue.

All of these policies are evaluated and compared empirically in [33]. (It does not seem possible to conduct a numerical evaluation here, due to the complexity of mathematically analyzing cuckoo hashing.) Overall, the results indicate that all of the intuition described above is accurate. In particular, the last queuing policy is extremely practical and performs substantially better than the standard policy over long periods of time. More specifically, the size of the queue under the standard policy is much more susceptible to occasional spikes than the last policy. In practice, this observation means that when the insertion queue is implemented with a CAM that should hold the entire queue almost all of the time, the last policy is likely to perform much better than the original one.

5 Bloom Filter Techniques

This section describes some additional improvements and applications of Bloom filters for high-speed packet processing that have been proposed in recent work. We start by describing some improvements that can be made to the standard Bloom filter and counting Bloom filter data structures that are particularly well-suited to hardware-based networking applications. Then we describe the *approximate concurrent state machine*, which is a Bloom filter variant that makes use of these ideas to efficiently represent state information for a set of items, as opposed to membership, as in a standard Bloom filter. Finally, we review a number of additional applications of Bloom filters to a wide variety of high-speed networking problems.

5.1 Improved (Counting) Bloom Filters

Standard counting Bloom filters, by their very nature, are not particularly space-efficient. Using the standard optimization for false positives for a Bloom filter from Section 2.1.2, the value for a particular counter in a counting Bloom filter is 0 with probability approximately $1/2$. Using multiple bits (e.g., 4, which is the usual case) to represent counters that take value 0 roughly half the time is an inefficient use of space. Some space gains can be made in practice by introducing additional lookups; counters can be kept to two bits and a secondary table can be used for counters that overflow. More sophisticated approaches exist, but their suitability for hardware implementation remains untested (see, e.g., [11, 49]).

To address this issue, Bonomi et al. [5] develop new constructions with the same functionality as a Bloom filter and counting Bloom filter, based on d-left hashing. These schemes are designed particularly for hardware implementation. In particular, they generally reduce the number of hashes and memory accesses required by the standard data structures. The idea behind these constructions actually first appears in another work by Bonomi et al. [3], where an extension to the Bloomier filter dubbed *approximate concurrent state machines*, or ACSMs, are developed. Here we describe the Bloom filter and counting Bloom filter variants, and discuss ACSMs in Section 5.2.

The starting point for these constructions is the folklore result that one can obtain the same functionality as a Bloom filter for a static set S with near-optimal performance using a perfect hash function. (A *perfect* hash function is an easily computable bijection from S to an array of $|S|$ hash buckets.) One finds a perfect hash function P, and then stores at each hash location an $f = \lceil \log 1/\varepsilon \rceil$ bit fingerprint, computed according to some other hash function H. A query on z requires computing $P(z)$ and $H(z)$, and checking whether the fingerprint stored at $P(z)$ matches $H(z)$. When $z \in S$ a correct response is given, and when $z \notin S$ a false positive occurs with probability at most ε; this uses $n\lceil \log 1/\varepsilon \rceil$ bits for a set S of n items.

The problem with this approach is that it does not cope with changes in the set S — either insertions or deletions — and perfect hash functions are generally too expensive to compute in many settings. To deal with this, we make use of the fact, recognized by Broder and Mitzenmacher [8] in the context of designing hash-based approaches to IP lookup (along the lines of the work by Waldvogel et al. [66] discussed in Section 2.3), that using d-left hashing provides a natural way to obtain an "almost perfect" hash function. The resulting hash function is only almost perfect in that instead of having one set item in each bucket, there can be several, and space is not perfectly utilized.

An example demonstrates the idea behind the approach; details are presented in [5]. Suppose we wish to handle sets of n items. We utilize a d-left hash table with $d = 3$ choices per item, so that on insertion each item chooses one bucket from each of three sub-tables uniformly at random, and the fingerprint for the item is then stored in the least loaded of the three choices. Each sub-table will have $n/12$ buckets, for $n/4$ buckets in total, giving an average of 4 items per bucket. With high probability, the maximum number of items in a bucket will be six (for large n, the probability converges to a value less than 10^{-30}). Hence we can implement the hash table as a simple array, with space for 6 fingerprints per bucket, and be guaranteed a very small probability of a failure due to bucket overflow. As with the folklore perfect hashing result, to check if an item is in the set, one checks for the fingerprint, but here one now has to check all three sub-tables.

Unfortunately, while this approach only uses three hashes and memory accesses for a lookup, analysis shows that as presented it gives a larger false positive probability for a given amount of space than a properly configured standard Bloom filter. Some additional manipulations are necessary to improve performance. The most important idea is to make use of the empty space; we are giving buckets space for six fingerprints, but on average they only hold four, leaving significant empty space throughout the hash table. To better use this space, we utilize variable-length fingerprints and a technique called *dynamic bit reassignment* [4]. That is, suppose we use 64-bit buckets, and give 60 bits to the fingerprints. (The remaining 4 bits are useful for various additional accounting.) This yields 10 bits per fingerprint. But if there are only 3 items in the bucket, we could use as many as 20 bits per fingerprint; if there are 4 items, as many as 15; and if there are 5 items, as many as 12. To improve performance, we dynamically change the fingerprint length in the bucket according to the number of items, shrinking fingerprints as the bucket load increases. This requires reading an entire bucket and processing it to extract the fingerprint, and rewriting an entire bucket when an item is inserted, but this can all be done easily in hardware.

The same idea can be used to construct a counting Bloom filter variant based on a d-left hash table [3, 5]. With a counting Bloom filter, the standard construction is quite wasteful of space, so even without using dynamic bit reassignment, one can gain a factor of two or more in space quite easily. (Indeed, using variable length fingerprints is more problematic in this setting, since one may not be able to increase the fingerprint size naturally when an item is deleted from the set.) Some care needs to be taken, however, to handle deletions properly; if one is not careful, the fingerprint corresponding to an item could appear in more than one bucket, if some other item shares the same fingerprint and one of the buckets. This is not a problem for the Bloom filter setting, where there are no deletions, but is a problem for the counting Bloom filter setting – which copy of the fingerprint do we delete? This problem can be avoided by using a hash function combined with permutations, instead of several hash functions.

Briefly, we can think of the hashing as being done in two phases. First, we hash an item with a hash function $h : U \to [B] \times [R]$ (where for an integer n, the notation $[n]$ denotes the set $\{0, \ldots, n-1\}$). Then, to obtain the d locations, we make use of additional (pseudo)-random permutations $P_1, \ldots, P_d$, so that

$$P_1(h(x)) = (b_1, r_1), P_2(h(x)) = (b_2, r_2), \ldots, P_d(h(x)) = (b_d, r_d),$$

where (b_i, r_i) are the bucket and fingerprint for the ith sub-table. Now, two items x and y will share a fingerprint and bucket if and only if they have the same hash $h(x) = h(y)$, so that a small counter (generally 2 bits) can be used to keep track of collisions for items under the hash function h.

5.2 Approximate Concurrent State Machines

Approximate concurrent state machines (ACSMs), introduced by Bonomi et al. [3], are a generalization of Bloom filters and Bloomier filters designed for router applications. In many such applications, a router may be handling a collection of flows, where a flow is determined by a source-destination pair or the standard IP 5-tuple. Each flow may have an associated state that changes over time, according to transitions over a well-defined finite state machine, usually with a small number of states. We desire a method that allows us to track the state of flows over time, as both the state of flows change and as the set of flows being tracked change over time, using smaller space than an explicit listing. Specifically, in [3], four operations are suggested for an ACSM data structure: insert a new flow with a given state, look up the state of a flow, modify the state of a flow, and delete an existing flow. In the spirit of Bloom filters, it makes sense to consider data structures that may return a state value for flows not currently extant, or that may return an incorrect state value for an extant flow. In practice there are several practical challenges to consider, such as handling situations where flows do not terminate correctly or are not initialized properly.

There are multiple possible ways to construct ACSM data structures, with several suggested in [3]. The best in experiments is based upon the d-left hashing approach. For each flow a fingerprint is stored, along with its associated state, in the d-left hash table. (This technique is intuitively similar to the techniques for improving the space utilization of Bloom filters and counting Bloom filters discussed in Section 5.1.) Interestingly, if fingerprints collide, it may be appropriate in some cases for the data structure to return a "don't know" response; such a response may be significantly less damaging than an incorrect response. The idea of allowing a "don't know" response, found also in [41], appears potentially quite powerful and worthy of further study.

Perhaps the simplest example of a use for an ACSM is to keep a small counter for each flow; the count represents the state of an item. Additional monitoring or other functionality could be required if the count for a flow reached a threshold value. A further example application considered in [3] is for specialized congestion control mechanisms for MPEG-based video streams. By tracking the frame type as the state of an MPEG-video flow, one can implement non-trivial congestion mechanisms based on the frame type, including tail-dropping mechanisms, where all packets are dropped until the next important frame (generally an I-Frame).

5.3 More Applications of Bloom Filters

Bloom filters have also recently been employed for more sophisticated packet processing and packet classification tasks in hardware. A useful example design is given in [14, 16], where the question being tackled is how to find specific substrings, commonly called signatures, in packets at wire speeds. A common current use of signatures is to scan for byte sequences particular to Internet worms, allowing malicious packets to be dropped. However, other natural uses arise in a variety of settings.

If we think of a collection of signatures as being a set of strings, then a natural approach is to represent this set with a Bloom filter. More specifically, it makes sense to separate signatures by length, and use a Bloom filter for each length, allowing the Bloom filters to be considered in parallel as the bytes of the packet are shifted through as a data stream. In order to obtain a suitable hardware implementation, however, there are further details to consider. For example, to handle the possible deletion and insertion of signature strings, one can use an associated counting Bloom filter. As insertions and deletions are likely to be rare, these counting Bloom filters can be kept separately in slower memory [14]. To avoid costly hashing overhead for longer signatures, such strings can be broken into smaller strings, and a small amount of state is kept to track how much of the string has been seen. In such a setting, the Bloom filter can also be used to track the state (in a manner similar to one of the approaches suggested for approximate concurrent state machines [3]).

Using Bloom filters allows a large database of signature strings to be effectively represented with a small number of bits, making use of fast memory in hardware feasible. Thousands and even tens of thousands of string can be effectively dealt with while maintaining wire speed [14].

Bloom filters have also been proposed for use in various longest prefix matching implementations [15, 17]. Many variations of IP lookup algorithms, for example, create hash tables consisting of prefixes of various lengths that have to potentially be matched against a given input IP address, with the goal of finding the longest possible match. The number of accesses to the hash tables can potentially be reduced by using a Bloom filter to record the set of prefixes in each hash table [15]. By checking the Bloom filter, one can avoid an unnecessary lookup into a hash table when the corresponding prefix does not exist in the table. Notice, though, that because of false positives, one cannot simply take the longest match suggested by the Bloom filters themselves; the hash table lookup must be done to check for a true match. The average number of hash table lookups, however, is reduced dramatically, so under the assumption that hash table lookups are dramatically slower or more costly than Bloom filter lookups, there are substantial gains in performance.

Bloom filters can also be used in multi-dimensional longest prefix matching approaches for packet classification, potentially giving a cheaper solution than the standard TCAM approach [17]. The solution builds off of what is known as the cross-product algorithm: find the longest prefix match on each field, and hash the resulting vector of longest matches into a hash table that will provide the packet classification rules associated with that vector. Unfortunately, straightforward implementations of the cross-product rule generally lead to very large hash tables, because the cross product approach leads to a large number of prefix vectors, roughly corresponding to the product of the number of prefixes in each field. This creates a significant overhead. An alternative is to split the rules into subsets and perform the cross-product algorithm on each subset, in order to reduce the overall size for hash tables. To keep the cost of doing longest prefix matchings reasonable, for each field, the longest prefix match is performed just once, over all subsets. The problem with this approach is now that for each subset of rules, for each field, there are multiple prefix lengths that might be possible; conceivably, any sub-prefix of the longest prefix match over all subsets of rules could apply to any specific subset of rules. To avoid hash table lookups for all possible combinations of prefixes over all subsets of rules, a Bloom filter of valid prefix vectors for each subset of rules can be maintained, reducing the number of necessary lookups in the hash table in a spirit similar to [15].

Results based on this approach yield a solution that requires at most $4+r$ memory accesses on average when r rules can match a packet; this can be further reduced with pipelining. Memory costs range on the order of 32 to 45 bytes per rule, allowing reasonably large rule sets to be effectively handled in SRAM.

Another recently proposed approach makes use of multiple choice hashing in order to reduce memory usage for IP lookup and packet classification algorithms, as well as other related algorithms. The setting here revolves around the fact that many algorithms for these types of problems reduce to directed graph traversal problems. Longest matching prefix structures are often be represented by a trie, where one finds the longest prefix match by walking down the trie. Similarly, regular expression matching structures are often represented by finite automata, where one finds the match by a walk on the corresponding automata graph.

Kumar et al. [39] describe an approach for compressing the representation of tries and other directed graphs, by avoiding using pointers to name nodes. Instead, the history of the path used to reach a node is used as a key for that node, and a multiple-choice hash structure stores the graph information associated with the node, such as its neighbors, in a format that allows continued traversal. For this approach to be successful, one needs there to be no collisions in the hash table. This is best accomplished with low space overheads using cuckoo hashing, which works well even in the face of updates to the underlying graph that change the set of keys being stored. By avoiding the use of expensive node identifiers, a factor of two or more in space can be saved over standard representations with minimal additional processing.

Another recent methodology for longest prefix matching problems based on hash-based approaches was described as part of the Chisel architecture [30]. Here the underlying technology used is a Bloomier filter. Prefixes correspond to keys, which are stored in the filter. One issue with this approach is that Bloomier filters do not support updates; insertions, in particular, can require the Bloomier filter to be reconstructed, which can take time linear in the number of keys. The trace analysis in [30] suggests that updates can be done quickly without a reconstruction in most cases, but it is not clear whether this holds more generally, and there is not currently a theoretical justification for this finding. The authors also develop other techniques to handle issues particular to the longest prefix matching problem in this context, such as how to cope with wildcard bits in the keys.

6 Measurement Applications using Hash-Based Algorithms

Today's routers provide per-interface counters that can be read by the management protocol SNMP, and/or a much more expensive solution that involves sampling packets using the NetFlow protocol [21]. The problem with this state of affairs is that SNMP counters are extremely coarse, while NetFlow is extremely expensive. Indeed, for the SNMP approach, there is only a single counter for all packets received and sent on an interface. In particular, there is no way to find how much a particular source is sending over the interface. Sampling packets with NetFlow addresses such issues, but often with prohibitive cost. For instance, even if we only sample one in a thousand packets, the wire speed may be such that terabits of data are collected every minute. Much of this data is lost, and the rest (which is still substantial) must be transferred to a measurement station, where it is logged to disk and eventually post-processed. Furthermore, despite its cost, NetFlow does not provide accurate answers to natural questions, such as the number of distinct source addresses seen in packets [23]. In this section, we describe hash-based measurement algorithms that can be implemented in hardware and can answer questions that SNMP counters cannot in a way that is much more direct and less expensive than NetFlow. The setting for all of these algorithms is a single interface in a router, and the algorithm is implemented by some logic or software associated with the link.

6.1 Finding Heavy-Hitters and Flow Size Distributions

In networking terminology, a *flow* is any collection of packets that pass through a given link and have some common header fields. For example, we could partition packets into flows based upon their destination addresses. In a hardware design, the exact definition of a flow can be left flexible in the interface logic by using simple mask registers on the IP and TCP fields so that network managers can change the definition to ask different questions in a running network.

For some types of flows, such as TCP flows, there are measurements that indicate millions of concurrent flows on a link [25]. Measuring these flows directly is prohibitively expensive (as is measuring them with NetFlow). Fortunately, a network manager may not need direct measurements of all flows on a link, but instead require knowledge of the flows that consume the most bandwidth. This problem is often called finding *heavy-hitters*. More precisely, the heavy-hitters problem can be defined as finding all flows that consume more than some fixed fraction of the bandwidth of the link.

We have already seen that the count-min sketch (or parallel multistage filter) with conservative updating is a natural solution to this problem (for details, see Section 2.1.2 or the original references [12,22]). Indeed, the data structure is essentially designed for this task. Unlike a direct measurement approach like NetFlow, this data structure only requires memory proportional to the number of heavy-hitters and simultaneously provides accurate probabilistic estimates of their bandwidth consumption.

Estan and Varghese [22] also describe a sampling-based technique called *sample-and-hold* for heavy-hitters estimation in which each packet is sampled independently with some very low probability and all subsequent occurrences of a sampled packet's flow are counted exactly in a CAM. A problem with this approach is that the CAM may be polluted with small flows. To address this issue, Lu et al. [42] propose *elephant traps* that, in essence, enhance sample-and-hold by periodically removing small flows from the CAM using an LRU-like algorithm that can be efficiently implemented in hardware. Since this chapter concentrates on hash-based algorithms, we do not describe this approach any further.

A general approach to measurement using hash-based counters is described by Kumar et al. [36]. The idea is that, over some time interval, all flows are hashed in parallel to several counters as in the count-min sketch. At the end of the interval, the counters are sent to software to be estimated for the measures of interest. For example, [36] describes how to estimate the flow size distribution (i.e., the number of packets sent by each flow). For simplicity, assume that each counter is incremented by one for each packet and not by the number of bytes. The main idea is to use *expectation maximization* to iteratively estimate the flow size distribution. We first estimate the current vector of flow sizes coarsely using the hardware counters. Then we calculate expectations from the estimates, replace the original estimates with these expectations, and iterate the process until convergence.

The idea of using more complex iterative estimators in software at the end of the time interval is also applied in later work on *counter braids* by Lu et al. [40]. The hardware setup generalizes the earlier work on count-min sketches in that it allows different levels of counters (two seems to suffice), with each level having a smaller number of counters than the previous levels. A flow is hashed in parallel to some number of counters in a table at the first level. If these counters overflow, the flow is then hashed to some number of counters in a table at the next level (again the mapping is done using a hash function), and so on. At the end of the interval an iterative estimate of the counts for each flow is provided, similar to the technique of [36]. However, the authors use tools inspired by coding theory (in particular, turbo-code decoding) rather than expectation maximization.

One problem with both techniques in [36] and [40] is that since estimates are found only at the end of certain time intervals, the techniques lose information about flows. By contrast, a standard count-min sketch uses a simple estimator that can be computed in real time. Indeed, at the instant that a flow's estimator crosses a threshold, the flow ID can be logged. The techniques of [36] and [40], however, lose all information about flow IDs. This is fine in [36] because the measure (flow size distribution) does not require any flow IDs (while heavy-hitters clearly does). This problem is addressed in [40] by assuming that either the flows to be measured are already known (in which case a simple CAM suffices to determine whether a packet should be processed) or that all flow IDs are logged in slow memory. In the latter case, the real gain of [40] is to log all flows in slow and large DRAM, but to *update* flow size information in a much smaller randomized structure in SRAM. Despite this problem, the general technique of using sophisticated iterative estimators, whether directed by expectation maximization or decoding techniques, seems like a promising direction for future work.

6.2 Measuring the Number of Flows on a Link

The number of flows on a link is a useful indicator for a number of security applications. For example, the Snort intrusion detection tool detects port scans by counting all the distinct destinations sent to by a given source, and sounding an alarm if this amount is over a threshold. Similarly, to detect a denial of service attack, one might want to count the number of sources sending to a destination because many such attacks use multiple forged addresses. In both examples, it suffices to count flows, where a flow identifier is a destination (for detecting port scans) or a source (for detecting denial of service attacks).

A naive method to count, say, source-destination pairs would be to keep a counter together with a hash table that stores all of the distinct 64 bit source-destination address pairs seen thus far. When a packet arrives with source-destination address pair (s,d), the algorithm searches the hash table for (s,d); if there is no match, the counter is incremented and (s,d) is added to the hash table. Unfortunately, this solution requires memory proportional to the total number of observed source-destination pairs, which is prohibitively expensive.

An algorithm due to Flajolet and Martin based on *probabilistic counting* [27] can considerably reduce the memory needed by the naive solution at the cost of some accuracy in counting flows. The intuition behind the approach is to compute a metric of how rare a certain pattern is in a random hash of a flow ID, and define the *rarity* $r(f)$ of a flow ID f to be the rarity of its corresponding hash. We then keep track of the largest value X of $r(f)$ ever seen over all flow IDs f that pass across the link. If the algorithm sees a very large value for X, then by our definition of rarity, it stands to reason that there is a large number of flows across the link.

More precisely, for each packet seen the algorithm computes a hash function on the flow ID. It then counts the number of consecutive zeroes starting from the least significant position of the hash result: this is the measure $r(\cdot)$ of rarity used. The tracked value X now corresponds to the largest number of consecutive zeroes seen (starting from the least significant position) in the hashed flow ID values of all packets seen so far.

At the end of some time interval, the algorithm converts X into an estimate 2^X for the number of flows. Intuitively, if the stream contains two distinct flows, on average one flow will have the least significant bit of its hashed value equal to zero; if the stream contains eight flows, on average one flow will have the last three bits of its hashed value equal to zero — and so on. Thus, 2^X is the natural estimate of the number of flows corresponding to the tracked value X.

Hashing is essential for two reasons. First, implementing the algorithm directly on the sequence of flow IDs itself could make the algorithm susceptible to flow ID assignments where the traffic stream contains a flow ID f with many trailing zeroes. If f is in the traffic stream, even if the stream has only a few flows, the algorithm without hashing will wrongly report a large number of flows. Notice that adding multiple copies of the same flow ID to the stream will not change the algorithm's final result, because all copies hash to the same value.

A second reason for hashing is that accuracy can be boosted using multiple independent hash functions. The basic idea with one hash function can guarantee at most 50% accuracy. By using n independent hash functions in parallel to compute n separate estimates $X_1,\ldots,X_n$, we can greatly reduce the error by calculating the mean M of the X_i's and returning 2^M as the estimated number of flows. (Note that M should be represented as a floating point number, not an integer.)

More modular algorithms for flow counting (again hash-based) for networking purposes are described by Estan, Fisk, and Varghese in [23]. Suppose we wish to count up to 64,000 flows. Then it suffices to hash each flow into a single bit in a 64,000 bit map (initially zeroed), and then estimate the number of flows accounting for hash collisions by counting the number of ones.

However, just as Hubble estimated the number of stars in the universe by sampling the number of stars in a small region of space, one could reduce memory to, say, 32 bits but still hash flows from 1 to 64,000. In this case, flows that hash to values beyond 32 would not set bits but flows that hash from 0 to 31 would. At the end of some time interval the number of bits set to 1 are counted, an estimate is found by correcting for collisions, and then the estimate is scaled up by $64,000/32 = 2000$ to account for the flows that were lost. Unfortunately, the accuracy of this estimator depends critically on the assumption that the number of flows is within some constant factor of 64,000. If the number of flows is much smaller (e.g., 50), the error is considerable.

The accuracy can be greatly improved in two ways. In a parallel approach, an array of, say, 32 counters are used, each responsible for estimating the number of flows in different ranges (e.g., 1-10, 10-100, 100-1000) using a logarithmic scale. Thus, at least one counter will be accurate. The counter used for the final estimate is the one which has neither too many bits set or too few bits set. The precise algorithm is in [23].

In some cases, even this memory can be too much. An algorithm proposed by Singh et al. [56] addresses this issue by using only one counter and adapting sequentially to reduce memory (at the cost of accuracy). The algorithm has a bit map of say 32 bits. The algorithm initially hashes all flows to between 0 and 31. If all the bits fill up, the number of flows is clearly greater than 32, so the algorithm clears the bitmap, and now hashes future flows (all memory is lost of older flows) from 0 to 64. It keeps doubling the range of the hash function until the number of bits stops overflowing. The range of the hash function is tracked in a small scale factor register. The net result is that flow counting can be done for up to a million flows using a bit map of size 32 and a 16-bit scale factor register at a cost of a factor of 2 loss in accuracy (better tradeoffs are described in [56]).

Singh et al. [56] also describe how both the heavy-hitter and flow size estimators can be used together to extract worm signatures. Any string of some pre-specified fixed size in a packet payload is considered to be a possible worm signature. If the string occurs frequently (measured using a heavy-hitters estimator) and has a large number of associated unique source and destination IP addresses (measured by the sequential flow size estimator in [56]) the string is considered as a possible worm signature and is subjected to further offline tests. The sequential hash-based estimator is crucial to the efficiency of worm detection in [56] as the algorithm often tracks 100,000 potential signatures at the same time. Further, the accuracy of the flow size estimator is less of a concern because a "large" number of sources is used as a rough indicator of worm-like activity.

Finally, the problem with probabilistic counting and the techniques in [23] is that the definition of a flow must be predefined before measurement begins. In other words, the estimator must be programmed in advance to determine whether it is counting, say, source IP addresses, destination IP addresses, or TCP flows. On the other hand, managers may want a more flexible estimator that can be queried for a count of any definition of flows that the analyst might think of after measurements are taken (e.g., the number of TCP flows from Harvard to MIT).

The paper [23] shows how a beautiful algorithm due to Wegman [26] can be used to allow a flow size estimator that can be sliced-and-diced after the fact. The idea is useful in general because it allows a way to compute a random sample from the set of all unique flows. To allow slicing later, the definition of flows used is the most specific definition the analyst can specify in advance of the measurements (e.g., TCP flows).

The idea is to start by recording all unique flows in a hash table with hash function h. If the hash table size exceeds some fixed size s, then all flows f such that $h(f)$ has low order bit 1 are discarded from the existing table and in the future. If the hash table size exceeds S, then only flows f such that $h(f) = 00$ are retained, and so on. Notice that this is a reservoir sampling algorithm, but it does not seem to have received as much attention as the work of Vitter [63]. After the measurement interval is over, the analyst can compute an estimate of the number of flows, for any more specific definition of a flow than was used during the sampling phase (such as TCP flows with some specific source s), by simply counting the number of such flows in the random sample.

7 Conclusion

We have surveyed the recent literature on hash-based approaches to high-speed packet processing in routers, with a particular focus on how theoretical and applied ideas can often interact to yield impressive results. However, we admit that we have only scratched the surface. We could have presented many more results, or given much more detail on the results we have presented. We strongly encourage interested readers to read the original works. We believe that this trend of applying hash-based approaches to these problems will continue, and that there is an enormous potential for designing hash-based approaches that are both theoretically interesting and practically significant for these very demanding and increasingly important applications.

Indeed, we see two important high-level directions for future work. The first is to increase the already existing connections between the theoretical and applied work in this general area. Many new and potentially interesting ideas have been consistently produced in recent years on the theoretical side; finding the right applications and proving the value of these ideas could be extremely beneficial. Also, those building applications need to be able to understand and predict the behavior of schemes they wish as well as evaluate tradeoffs between schemes the wish to implement in advance, based on sound theoretical principles. On the other side, those working at a more theoretical level need to pay attention to the needs and requirements of applications, including possibly some details of hardware implementation.

A second high-level direction, more speculative and far-reaching, is to consider whether a hashing infrastructure could be developed to support hash-based approaches for high-speed packet processing. Hash-based approaches offer great value, including relative simplicity, flexibility, and cost-effectiveness. While not every packet-processing task can naturally be placed in a hashing framework, as this survey shows, a great many can. One could imagine having some standardized, flexible, programmable hashing architecture for Internet devices, designed not for a specific task or algorithm, but capable of being utilized for many hash-based data structures or algorithms. The goal of such an infrastructure would not only be to handle issues that have already arisen in today's network, but also to provide a general framework for handling additional, currently unknown problems that may arise in the future. Additionally, a potential key value in a standardized hashing infrastructure lies in not only its potential use for monitoring or measuring individual routers, links, or other components, but the network as a whole.

8 Acknowledgments

Adam Kirsch and Michael Mitzenmacher received support for this work from NSF grant CNS-0721491 and a research grant from Cisco Systems, Inc. George Varghese received support from NSF grant 0509546 and a grant from Cisco Systems, Inc.

References

1. Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced Allocations. *SIAM Journal on Computing*, 29(1):180-200, 1999.
2. B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422-426, 1970.
3. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proceedings of ACM SIGCOMM*, pp. 315-326, 2006.
4. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Bloom Filters via d-left Hashing and Dynamic Bit Reassignment. In *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2006.
5. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *Proceedings of the 14th Annual European Symposium on Algorithms* (ESA), pp. 684-695, 2006.
6. A. Broder and A. Karlin. Multilevel Adaptive Hashing. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 43-53, 1990.
7. A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485-509, 2004.
8. A. Broder and M. Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. *Proceedings of the 20th IEEE International Conference on Computer Communications* (INFOCOM), pp. 1454-1463, 2001.
9. J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143-154, 1979.
10. B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 30-39, 2004.
11. S. Cohen and Y. Matias. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp. 241-252, 2003.
12. G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58-75, 2005.

13. L. Devroye and P. Morin. Cuckoo Hashing: Further Analysis. *Information Processing Letters*, 86(4):215-219, 2003.
14. S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. *IEEE Micro*, 24(1):52-61, 2004.
15. S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Transactions on Networks*, 14(2):397-409, 2006.
16. S. Dharmapurikar and J. W. Lockwood. Fast and Scalable Pattern Matching for Network Intrusion Detection Systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781-1792, 2006.
17. S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast Packet Classification Using Bloom Filters. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture For Networking and Communications Systems* (ANCS), pp. 61-70, 2006.
18. M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19-51, 1997.
19. M. Dietzfelbinger and C. Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science*, 380:(1-2):47-68, 2007.
20. B. Donnet, B. Baynat, and T. Friedman. Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives. *arxiv, cs.NI/0607038*, 2006.
21. C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *Proceedings of ACM SIGCOMM*, pp. 245-256, 2004.
22. C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270-313, 2003.
23. C. Estan, G. Varghese, and M. E. Fisk. Bitmap Algorithms for Counting Active Flows on High-Speed Links. *IEEE/ACM Transactions on Networks*, 14(5):925-937, 2006.
24. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000.
25. W. Fang and L. Peterson. Inter-AS Traffic Patterns and Their Implications. In *Proceedings of the Global Telecommunications Conference, 1999* (GLOBECOM), 1999.
26. P. Flajolet. On Adaptive Sampling. *Computing*, 43(4):391-400, 1990.
27. P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182-209, 1985.
28. D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space Efficient Hash Tables With Worst Case Constant Access Time. *Theory of Computing Systems*, 38(2):229-248, 2005.
29. G. Gonnet. Expected Length of the Longest Probe Sequence in Hash Code Searching. *Journal of the Association for Computing Machinery*, 28(2):289-304, 1981.
30. J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar. Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture. In *Proceedings of the 33rd International Symposium on Computer Architecture* (ISCA), pp. 203-215, 2006
31. A. Kirsch and M. Mitzenmacher. The Power of One Move: Hashing Schemes for Hardware. In *Proceedings of the 27th IEEE International Conference on Computer Communications* (INFOCOM), 2008.
32. A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Choices. *IEEE/ACM Transactions on Networking*, 16(1):218-231, 2008.
33. A. Kirsch and M. Mitzenmacher. Using a Queue to De-amortize Cuckoo Hashing in Hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, 2007.
34. A. Kirsch, M. Mitzenmacher, and U. Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. To appear in *Proceedings of the 16th Annual European Symposium on Algorithms*, 2008.
35. R. R. Kompella, S. Singh, and G. Varghese. On Scalable Attack Detection in the Network. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pp. 187-200, 2004.

36. A. Kumar, M. Sung, J. Xu, and J. Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS/Performance), pp. 177-188, 2004.
37. S. Kumar and P. Crowley. Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems* (ANCS), pp. 91-103, 2005.
38. S. Kumar, J. Turner, and P. Crowley. Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. In *Proceedings of the 27th IEEE International Conference on Computer Communications* (INFOCOM), 2008.
39. S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. In *Proceedings of the Fifteenth IEEE International Conference on Network Protocols* (ICNP), pp. 246-255, 2007.
40. Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), 2008.
41. Y. Lu, B. Prabhakar, and F. Bonomi. Perfect Hashing for Networking Algorithms. In *Proceedings of the 2006 IEEE International Symposium on Information Theory* (ISIT), pp. 2774-2778, 2006.
42. Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A Low Cost Device for Identifying Large Flows. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects* (HOTI), pp. 99-108, 2007.
43. S. Lumetta and M. Mitzenmacher. Using the Power of Two Choices to Improve Bloom Filters. To appear in *Internet Mathematics*.
44. M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.
45. M. Mitzenmacher, A. Richa, and R. Sitaraman. *The Power of Two Choices: A Survey of Techniques and Results*, edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 255-312.
46. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
47. M. Mitzenmacher and S. Vadhan. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 746-755, 2008.
48. A. Östlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (STOC), pp. 622-628, 2003.
49. A. Pagh, R. Pagh, and S. S. Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 823-829, 2005.
50. R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122-144, 2004.
51. R. Panigrahy. Efficient Hashing with Lookups in Two Memory Accesses. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 830-839, 2005.
52. M. V. Ramakrishna. Hashing Practice: Analysis of Hashing and Universal Hashing. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pp. 191-199, 1988.
53. M. V. Ramakrishna. Practical Performance of Bloom Filters and Parallel Free-Rext Searching. *Communications of the ACM*, 32(10):1237-1239, 1989.
54. M.V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378-1381, 1997.
55. A. Siegel. On Universal Classes of Extremely Random Constant-Time Hash Functions. *Siam Journal on Computing*, 33(3):505-543, 2004.

56. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation* (OSDI), 2004.
57. H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proceedings of ACM SIGCOMM*, pp. 181-192, 2005.
58. D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, 2000. Available at `ftp://ftp.rfc-editor.org/in-notes/rfc2991.txt`.
59. M. Thorup. Even Strongly Universal Hashing is Pretty Fast. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), pp. 496-497, 2000.
60. GIGAswitch System: A High-Performance Packet-Switching Platform. R. J. Souza, P. G. Krishnakumar, C. M. Özveren, R. J. Simcoe, B. A. Spinney, R. E. Thomas, and R. J. Walsh. *Digital Technical Journal*, 6(1):9-22,1994.
61. V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1):1-40, 1999.
62. G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers, 2004.
63. J. S. Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37-57, 1985.
64. S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security* (SNDSS), 149-166, 2005.
65. B. Vöcking. How Asymmetry Helps Load Balancing. *Journal of the ACM*, 50(4):568-589, 2003.
66. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. *ACM SIGCOMM Computer Communication Review*, 27(4):25-36, 1997.
67. M. N. Wegman and J. L. Carter. New Hash Functions and Their Use in Authentication and Set Equality. *Journal of Computer and System Sciences*, 22(3):265-279, 1981.
68. P. Woelfel. Asymmetric Balanced Allocation with Simple Hash Functions. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm* (SODA), pp. 424-433, 2006.

Index