BRICK: A Novel Exact Active Statistics Counter Architecture

Nan Hua[†] Bill Lin[‡] Jun (Jim) Xu[†] Haiquan (Chuck) Zhao[†]
[†] College of Computing, Georgia Tech [‡] Dept of ECE, UCSD

ABSTRACT

In this paper, we present an exact active statistics counter architecture called BRICK (Bucketized Rank Indexed Counters) that can efficiently store per-flow variable-width statistics counters entirely in SRAM while supporting both fast updates and lookups (e.g., 40 Gb/s line rates). BRICK exploits statistical multiplexing by randomly bundling counters into small fixed-size buckets and supports dynamic sizing of counters by employing an innovative indexing scheme called rank-indexing. Experiments with Internet traces show that our solution can indeed maintain large arrays of exact active statistics counters with moderate amounts of SRAM.

Categories and Subject Descriptors

C.2.3 [COMPUTER-COMMUNICATION NETWORKS]: Network Operations - Network Monitoring

General Terms

Algorithms, Measurement, Performance

Keywords

Statistics Counter, Router

1. INTRODUCTION

It is widely accepted that network measurement is essential for the monitoring and control of large networks. For implementing various network measurement, router management, and data streaming algorithms, there is often a need to maintain very large arrays of statistics counters at wirespeeds (e.g., million counters for per-flow measurements). For example, on a 40 Gb/s OC-768 link, a new packet can arrive every 8 ns and the corresponding counter updates need to be completed within this time. While implementing large counter arrays in SRAM can satisfy performance needs, the amount of SRAM required for worst-case counter sizes is often both infeasible and impractical. Therefore, researchers have actively sought alternative ways to realize large arrays of statistics counters at wirespeeds [21, 19, 20, 26].

In particular, several SRAM-efficient designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed. Their baseline idea is to store some lower order bits (e.g., 9 bits) of each counter in SRAM, and all its bits (e.g., 64 bits) in DRAM. The increments are made only to these SRAM counters, and when the values of SRAM counters become close to overflow, they will be scheduled to be "committed" back to the corresponding DRAM counter. These schemes all significantly reduce the SRAM cost. For example, the scheme by Zhao et al. [26] achieves the theoretically minimum SRAM cost of between 4 to 6 bits per counter, when the speed difference between SRAM and DRAM ranges between 10 (50ns/5ns) and 50 (100ns/2ns). However, in these schemes, while writes can be done as fast as on-chip SRAM latencies (2 to 5ns), read accesses can only be done as slowly as DRAM latencies (e.g., 60 to 100ns). Therefore, such schemes only solve the problem of so-called *passive counters* in which full counter values in general do not need to be read out frequently (not until the end of a measurement epoch). Besides the problem of slow reads, hybrid architectures also suffer from the problem of significantly increasing the amount of traffic between SRAM (usually on-chip) and DRAM (usually off-chip) across the system bus. This may become a serious concern in today's network processors, where system bus and DRAM bandwidth are already heavily utilized for other packet processing functions [26].

While passive counters are good enough for many network monitoring applications, a number of other applications require the maintenance of *active counters*, in which the values of counters may need to be read out as frequently as they are incremented, typically on a per packet basis. In many network data streaming algorithms [4, 7, 12, 13, 24, 25], upon the arrival of each packet, values need to be read out from some counters to decide on actions that need to be taken. For example, if Count-Min sketch [4] is used for elephant detection, we need to read the counter values on a per packet basis because such readings will decide whether a flow needs to be inserted into a priority queue (implemented as a heap) that stores "candidate elephants". A prior work on approximate active counters [22] identifies several other data streaming algorithms that need to maintain active counters, including multistage filters for elephant detection [7] and online hierarchical heavy hitter identification [24]. Currently, all existing algorithms that use active counters implement them as full-size SRAM counters. An efficient solution for exact active counters clearly will save memory cost for all such applications.

1.1 Our approach and contributions

In this paper, we propose the first solution to the open problem of how to efficiently maintain exact active counters. Our objective is to design an exact counter array scheme that allows for extremely fast read *and* write accesses (at on-chip SRAM speeds). However, these goals will clearly push us back to the origins of using an array of full-size counters in SRAM if we do not impose any additional constraint on the counter values. Fast read access demands that the counters reside entirely in SRAM and we can make the values of each counter large enough (and random enough) so that each of them needs the worst-case (full-size) counter size. Therefore we will solve our problem under a very natural and reasonable constraint. We assume that the total number of increments, which is exactly the sum of counter values in the array, is bounded by a constant M during the measurement interval.

This constraint is a very reasonable constraint for several reasons. First, this constraint is natural since the number of increments is bounded by the maximum packet arrival rate times the length of the measurement epoch. We can easily enforce an overall count sum limit by limiting the length of the measurement epoch. Moreover, this constraint has been assumed in designing other memory-efficient data structures such as Spectral Bloom Filters [3]. Furthermore, our scheme will work for arbitrarily large M values, although its relative memory savings compared to full-size counters gets gradually lower with larger M values.

Let N be the total number of counters in the array. Then the ratio $\frac{M}{N}$ corresponds to the (worst-case) average value of a counter, which is indeed a more relevant parameter than M for evaluation purposes, as it corresponds to the "per-counter workload". We observe that small $\frac{M}{N}$ ratio is dictated by many real-world applications. For example, if we use a Count-Min [4] sketch with $\ln \frac{1}{\delta}$ arrays of $\frac{e}{\epsilon}$ ($e \approx 2.718$) counters each, for estimating the sizes of TCP/UDP flows, then with probability at least $1 - \delta$, the CM-sketch overcounts (it never under-counts) by at most $M\epsilon$. Suppose we set δ to 0.1 and ϵ to 10^{-5} so that we use a total of $\ln(\frac{1}{0.1}) \times \frac{e}{10^{-5}} \approx 6.259 \times 10^5$ counters. When the total number of increments M is set to 10^8 and correspondingly the average counts per counter $\frac{M}{N}$ is approximately 160, we can guarantee that the error is no more than 1,000 ($= 10^8 \times 10^{-5}$) with probability at least 0.9. However, 1,000 are considered very large errors and hence for practice we always want $\frac{M}{N}$ to be much smaller.

We emphasize that even when the ratio $\frac{M}{N}$ is small, it is still important to figure out ways to save memory, as naive implementations can be grossly wasteful. For example, let the total counts be $M = 16$ million and the number of counters be $N = 1$ million. In other words, the average counter value $\frac{M}{N}$ is 16. Since all increments can go to the same counter, fixed-counter-size design would require a conservative counter size of $\lg(16 \times 10^6) = 24$ bits[1]. However, as we will show, our scheme can significantly reduce the SRAM requirement, which is very important for ASIC implementations where SRAM cost is among the primary costs.

In this paper, we present an exact active counter architecture called Bucketized (B) Rank (R) Indexed (I) Counter (CK), or BRICK. It is built entirely in SRAM so that both read and increment accesses can be processed at tens to hundreds of millions of packets per second. In addition, since it is stored entirely in SRAM, it will not introduce traffic between SRAM and DRAM. This makes it also a very attractive solution for passive counting applications in which the aforementioned problem of increased traffic over system bus caused by the hybrid SRAM/DRAM architecture becomes a serious concern.

The basic idea of our scheme is intuitive and is based on a very familiar networking concept: statistical multiplexing. Our idea is to bundle groups of a fixed number (let it be 64 in this case) of counters, which is randomly selected from the array, into buckets. We allocate just enough bits to each counter in the sense that if its

[1]Throughout this paper, $\lg(x) \equiv \log_2(x)$.

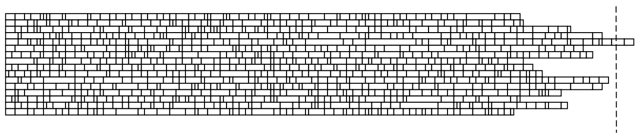


Figure 1: BRICK wall (conceptual baseline scheme)

current value is C_i, we allocate $\lfloor \lg C_i \rfloor + 1$ bits to it. Therefore, counters inside a bucket have variable widths. Suppose the mean width of a counter averaged over the entire array is γ. By the law of large numbers, the total widths of counters in most of the buckets will be fairly close to γ multiplied by the number of counters per bucket. Depicting each counter as a "brick", as shown in Figure 1, a section of the "brick wall" illustrates the effect of statistical multiplexing, where each horizontal layer of bricks (consisting of 64 of them) corresponds to a bucket and the length of bricks corresponds to the real counter widths encoding flow sizes in a real-world Internet packet trace (the USC trace in Section 4.2).

As we see in this figure, when we set the bucket size to be slightly longer than 64γ (the vertical dashed line), the probability of the total widths of the bricks overflowing this line is quite small; among the 20 buckets shown, only 1 of them has an overflow. Although overflowed buckets need to be handled separately and will cost more memory, we can make this probability small and the overall overflow cost is small and bounded. Therefore, our memory consumption only needs to be slightly larger than 64γ per bucket.

This baseline approach is hard to implement in hardware in practice for two reasons. First, we need to be able to randomly access (i.e., jump to) any counter with ease. Since counters are of variable sizes, we still need to spend several bits per counter for the indexing within the bucket. Note being able to randomly access is difference from being able to delimit all these counters. The latter can be solved with much less overhead using prefix-free coding (e.g., Huffman coding) of the counter values. However in this case, to access the i^{th} counter in a bucket, one has to scan through the first $i - 1$ counters (and hence very slow). Second, when the i^{th} counter (brick) in a bucket grows, counters $i + 1, i + 2, ..., 64$ will have to be shifted.

BRICK addresses these two difficulties with a little more overall SRAM cost. It allows for very efficient read and expansion (for increments that increase the width of a counter such as from 15 to 16). A key technique in our data structure is an indexing scheme called *rank indexing*, borrowed from the compression techniques in [11, 6, 23, 10]. The operations involved in reading and updating this data structure are not only simple for ASIC implementations, but are also supported in modern processors through built-in instructions such as "shift" and "popcount" so that software implementation is efficient (as the involved basic operations such as shift and popcount are supported by modern processors [1, 2]). Therefore our scheme can be implemented efficiently both in hardware or software.

1.2 Background and related work

In this section, we compare and contrast our work with previous approaches. One category of approaches is based on the idea of a SRAM/DRAM hybrid architecture [21, 19, 20, 26]. The state of art scheme [26] only requires $\lg \mu$ bits per counter where μ is the speed different between SRAM and DRAM. This translates into between 4 to 6 SRAM bits per SRAM counter. However, the read can take quite long (say at least 100ns). Therefore, these approaches only solve the passive counting problem.

Another category of approaches is existing active counter solutions [16, 5, 22], which are all based on the approximate counting idea invented by Morris [16]. The idea is to probabilistically increment a counter based on the current counter value. However, approximate counting in general has a very large error margin when the number of bits used is small because the possible estimation values are very sparsely distributed in the range of possible counts. Therefore, when the counter values are small (say 5), its estimation can have a very high relative error (well over 100%). This is not acceptable in network accounting and data streaming applications where small counter values can be important for overall measurement accuracy. In fact, when the (worst-case) average counter value $\frac{M}{N}$ is no more than 128, the SRAM cost of our BRICK scheme (about 12 bits) is no more than that of [22], which is approximate.

Recently, another counter architecture called counter braids [14] has been proposed, which is inspired by the construction of LDPC codes [8] and can keep track of exact counts of all flows without remembering the association between flows and counters. At each packet arrival, counter increments can be performed quickly by hashing the flow label to several counters and incrementing them. The counter values can be viewed as a linear transformation of flow counts, where the transformation matrix is the result of hashing *all* flow labels during a measurement epoch. However, counter braids are not active and are in fact "more passive" than the SRAM/DRAM hybrid architectures. To find out the size of a single flow, one needs to decode all the flow counts through a fairly long iterative decoding procedure. [2].

Finally, Spectral Bloom Filter [3] has been proposed, which provides an internal data structure for storing variable width counters. It uses a hierarchical indexing structure to locate counters that are packed next to each other, which allows for fast random accesses (reads). However, an update that causes the width of the counter i to grow will cause a shift to counters $i+1$, $i+2$, ..., which can have a global cascading effect even with some slack bits provided in between, making it prohibitively expensive when there can be millions of counters. As acknowledged in [3], although the expected amortized cost per update remains constant, and the global cascading effect is small in the average case, the worst-case cannot be tightly bounded. Therefore, SBF with variable width encoding is not an active counter solution as it cannot ensure fast per-packet write accesses at every packet arrival, forcing it to become a mostly-read-only data structure in the sense that updates should be orders of magnitude less frequent than queries.

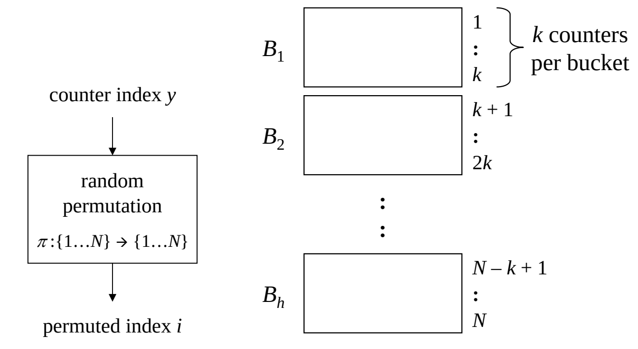
The rest of the paper is organized as follows. Section 2 describes the design of our scheme in detail. Section 3 establishes the tail probabilities that allow us to bound and optimize the SRAM requirement. Section 4 evaluates our scheme by presenting numerical results on memory costs and tail probabilities under various parameter settings, including those extracted from real-world traffic traces.

2. DESIGN OF BRICK

In this section, we describe the proposed BRICK counter architecture. The objective of BRICK is to efficiently encode a set of N *exact active* counters $C_1, C_2, \ldots, C_N$, under the constraint that throughout a network measurement epoch the total counts[3] across all counters $\sum_{i=1}^{N} C_i$ is no more than a pre-determined threshold M, which is carefully justified in Section 1. As we explained earlier, since all increments can go to the same counter, the value of a counter can be as large as M, and hence the worst-case counter width is $L = \lfloor \lg M \rfloor + 1$. However, it is unnecessarily expensive to allocate L bits to every counter since only a tiny number of them will have counts large enough to require this worst-case width while most others need significantly fewer bits. Therefore, BRICK adopts a sophisticated *variable width encoding* of counters and can statistically multiplex these variable width counters through a bucketing scheme to achieve a much more compact representation. However, unlike the aforementioned baseline bucketing scheme, BRICK is extremely SRAM-efficient yet allows for very fast counter lookup and increment operations.

[2]In [14], they need 25 seconds on a 2.6GHz computer to decode the flow counts inside a 6-minute-long traffic trace.

[3]Here with an abuse of notation, we will use C_i to denote both the counter and its current count (value).



(a) Index permutation (b) Bucketization

Figure 2: Randomly bundling counters into buckets.

In the following, we will first present an overview of our proposed design in Section 2.1, followed by how it handles lookups, increments, and bucket overflows in Sections 2.2 to 2.4, respectively.

2.1 Overview

The basic idea of BRICK is to *randomly* bundle N counters into h buckets, B_1, B_2, ..., B_h, where each bucket holds k counters (e.g. $k = 64$ in practice) and $N = hk$. In each bucket, some counters will be long (possibly L bits in the worst-case) and some will be short, depending on the values they contain. As discussed earlier, the objective of bundling is to "statistically multiplex" the variable counter widths in a bucket so that each bucket only needs to be allocated memory space that is slightly larger than k times the average counter width (across N counters). Note that since we do not know the actual average width of a counter in advance, we need to instead use the *average width* in the following adversarial context. Imagine that an adversary chooses $C_1, C_2, \ldots C_N$ values under the constraint $\sum_{i=1}^{N} C_i \le M$ that maximizes the metrics (e.g., average counter width). We emphasize that such an adversary is defined entirely in the well-established context of randomized online algorithm design [17] and has nothing to do with its connotation in security and cryptography.

Fig. 2 depicts these ideas of randomization and bucketization. In particular, as depicted in Fig. 2(a), to access the y^{th} counter, a pseudorandom permutation function $\pi : \{1\ldots N\} \to \{1\ldots N\}$ is first applied to the index y to obtain a permuted index i. This pseudorandom permutation function in practice can be as simple[4] as reversing the bits of y. The corresponding counter C_i can then be found in the ℓ^{th} bucket B_ℓ, where $\ell = \lceil \frac{i}{k} \rceil$. The bucket structure is depicted in Fig. 2(b). Unless otherwise noted, when we refer to the i^{th} counter C_i, we will assume i is already the result of a random permutation.

[4]Since the adversary is defined in the online algorithm context discussed above, we do not believe cryptographically strong pseudo-random permutations, which may increase our cost and slow down our operations, are needed here.

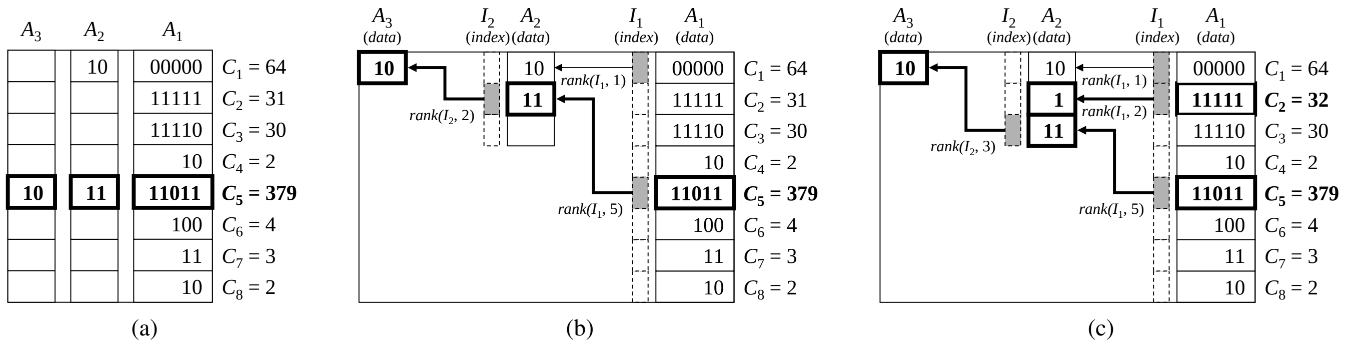


Figure 3: (a) Within a bucket, segmentation of variable-width counters into sub-counter arrays. (b) Compact representation of variable-width counters. (c) Updated data structure after incrementing C_2.

As we explained before, the baseline bucketing scheme does not allow for efficient read and write (increment) accesses. In BRICK, a multi-level partitioning scheme is designed to address this problem as follows. The worst-case counter width L is divided into p parts, which we refer to as "sub-counters". The j^{th} sub-counter, $j \in [1, p]$ (from the least significant bits to most significant bits) has w_j bits, such that $0 < w_j \leq L$ and $\sum_{j=1}^{p} w_j = L$. To save space, for each counter, BRICK maintains just enough of its sub-counters to hold its current value. In other words, counters with values no more than $2^{w_1+w_2+\cdots+w_i}$ will not have its $(i+1)^{th}, \ldots, p^{th}$ sub-counters stored in BRICK. For example, if $w_1 = 5$, any counter with value less than $2^5 = 32$ will only be allocated a memory entry for its 1^{st} sub-counter. Consider the example shown in Fig. 3(a) with $k = 8$ counters in a bucket. Only C_1 and C_5 require more than their first sub-counters. Such an on-demand allocation requires us to link together all sub-counters of a counter, which we achieve using a simple and memory-efficient bitmap indexing scheme called *rank indexing*. Rank indexing enables efficient lookup as well as efficient expansion (when counter values exceed certain thresholds after increments), which will be discussed in detail in Section 2.2.

Each bucket contains p sub-counter arrays A_1, A_2, ..., A_p to store the 1^{st}, 2^{nd}, ..., p^{th} sub-counters (as needed) of all k counters in the bucket. How many entries should be allocated for each array A_i, denoted as k_i, turns out to be a non-trivial statistical optimization problem. On the one hand, to save memory, we would like to make k_2, k_3, ..., k_p (k_1 is fixed as k) as small as possible. On the other hand, when we encounter the unlucky situation that we need to exceed any of these limits (say for a certain d, we have more than k_d counters in a bucket that have values larger than or equal to $2^{w_1+w_2+\cdots+w_{i-1}}$), then we will have a "bucket overflow" that would require that all counters inside the bucket be relocated to an additional array of full-size buckets with fixed worst-case width L for each counter, as we will show in Section 2.4. Given the high cost of storing a duplicated bucket in the full-size array, we would like to choose larger $k_2, \ldots, k_p$ to make this probability as small as possible. In Section 3, we develop extremely tight tail bounds on the overflow probability that allows us to choose parameters $\{k_i\}_{2 \leq i \leq p}$ and $\{w_i\}_{1 \leq i \leq p-1}$ to achieve near-optimal tradeoffs between these two conflicting issues and minimize the overall memory consumption.

2.2 Rank Indexing

A key technique in our data structure is an indexing scheme that allows us to efficiently identify the locations of the sub-counters across the different sub-counter arrays for some counter C_i. In particular, for C_i, its d sub-counters $C_{i,1}, \ldots, C_{i,d}$ are spread across $A_1, \ldots, A_d$ at locations $a_{i,1}, \ldots, a_{i,d}$, respectively (i.e., $C_{i,j} = A_j[a_{i,j}]$). For example, as shown in Fig. 3(b), C_5 is spread across $A_3[1] = 10$, $A_2[2] = 11$, and $A_1[5] = 11011$.

For each bucket, we maintain an index bitmap I. I is divided into $p-1$ parts, $I_1, \ldots, I_{p-1}$, with an one-to-one correspondence to the sub-counter arrays $A_1, \ldots, A_{p-1}$, respectively. Each part I_j is a bitmap with k_j bits, $I_j[1], \ldots, I_j[k_j]$, one bit $I_j[a]$ for each entry $A_j[a]$ in A_j. Each $I_j[a]$ is used to determine if the counter stored in $A_j[a]$ has expanded beyond the j^{th} sub-counter array. I_j is also used to compute the index location of C_i in the next sub-counter array A_{j+1}. Because a counter cannot expand beyond the last sub-counter array, there is no need for an index bitmap component for the most significant sub-counter array A_p. For example, consider the entries $A_1[1]$ and $A_1[5]$ where the corresponding counter has expanded beyond A_1. This is indicated by having the corresponding bit positions $I_1[1]$ and $I_1[5]$ set to 1, as shown in shaded boxes in Fig. 3(b). All remaining bit positions in I_1 are set to 0, as shown in clear boxes.

For each counter that has expanded beyond A_1, an arrow is shown in Fig. 3(b) that links a sub-counter in A_1 with the corresponding sub-counter entry in A_2. For example, for C_5, its sub-counter entry $A_1[5]$ in A_1 is linked to the sub-counter entry $A_2[2]$ in A_2. Rather than expending memory to store these links explicitly, which could vanish savings gained by reduced counter widths, we *dynamically* compute the location of a sub-counter in the next sub-counter array A_{j+1} based on the current bitmap I_j. This way, no memory space is needed to store link pointers. This dynamic computation can be readily determined using an operation called $\texttt{rank}(s, j)$, which returns the number of ones only in the range $s[1] \ldots s[j]$ in the bit-string s. This operation is similar to the `rank` operator defined in [11].

We apply the `rank` operator on a bitmap I_j by interpreting it as a bit-string. As we shall see in Sections 3 and 4, our approach is designed to work with small buckets of counters (e.g. $k = 64$). Therefore, the corresponding bit-strings I_j are also relatively short since all sub-counter arrays satisfy $k_j \leq k$. Moreover, each successive k_j in the higher sub-counter arrays is substantially smaller than the previous sub-counter array, with the corresponding reduction in the length of the bit-string I_j. In turn, the `rank` operator can be efficiently implemented by combining a bitwise-AND instruction with another operation called $\texttt{popcount}(s)$, which returns the number of ones in the bit-string s. Fortunately, the `popcount` operator is becoming an increasingly available hardware-optimized instruction in modern microprocessors and network processors. For example, current generations of 64-bit x86 processors have this instruction built-in [1, 2]. Using this `popcount` instruction, the `rank` operation for bit-strings with lengths up to $|s| = 64$ bits can be readily computed in as few as two instructions. As shown with numerical examples and trace simulations in Section 4, very good results can be achieved with a bucket size fixed at 64.

The pseudo-code for the `lookup` operation is shown in Algorithm 1. The retrieval of the sub-counters using rank indexing is shown in Lines 3-6, with the final count returned at the end of the procedure. For a hardware implementation, the iterative procedure can be readily pipelined. As we shall see in Section 4, we only need a small number of levels (e.g. three) in practice to achieve efficient results.

Algorithm 1: Pseudo-code

```
1  lookup($i$)
2    $C_i = 0$; $a = i \mod k$;
3    for $j = 1$ to $p$
4      $C_{i,j} = A_j[a]$;
5      if ($j == p$ or $I_j[a] == 0$) break;
6      $a = \mathtt{rank}(I_j, a)$;
7    return $C_i$;

8  increment($i$)
9    $a = i \mod k$;
10   for $j = 1$ to $p$
11     $A_j[a] = A_j[a] + 1$;
12     if ($j == p$ or $A_j[a] \neq 0$) break; /* last array or no carry */
13     if ($I_j[a] == 1$) /* next level already allocated */
14       $a = \mathtt{rank}(I_j, a)$;
15     else /* expand */
16       $I_j[a] = 1$;
17       $a = \mathtt{rank}(I_j, a)$;
18       $b = (a-1)w_{j+1} + 1$;
19       $A_{j+1} = \mathtt{varshift}(A_{j+1}, b, w_{j+1})$;
20       $I_{j+1} = \mathtt{varshift}(I_{j+1}, a, 1)$;
21       $A_{j+1}[a] = 1$;
22       break;
```

2.3 Handling Increments

The `increment` operation is also based on the traversal of sub-counters using rank indexing. We will first describe the basic idea by means of an example. Consider the counter C_2 in Fig. 3(b). Its count is 31, which can be encoded in just the sub-counter array A_1 with $C_{2,1} = 11111$. Suppose we want to increment C_2. We first increment its first sub-counter component $C_{2,1} = 11111$, which results in $C_{2,1} = 00000$ with a *carry propagation* to the next level. This is depicted in Fig. 3(c).

This carry propagation triggers the increment of the next sub-counter component $C_{2,2}$. The location of $C_{2,2}$ can be determined using rank indexing (i.e. $\mathtt{rank}(I_1, 2) = 2$). However, the location of $A_2[2]$ was previously occupied by the counter C_5. To maintain *rank ordering*, we have to *shift* the entries in A_2 down by one to free up the location $A_2[2]$. This is achieved by applying an operation called $\mathtt{varshift}(s, j, c)$, which performs a right shift on the sub-string starting at bit-position j by c bits (with vacant bits filled by zeros). The `varshift` operator can be readily implemented in most processors by means of shift and bitwise-logical instructions.

In particular, we can view a sub-counter array A_j as a *bit-string* formed by the concatenation of its entries, namely $A_j = A_j[1]A_j[2]\dots A_j[k_j]$. The starting bit-position for an entry $A_j[a]$ in the bit-string can be computed as $b = (a-1)w_j + 1$, where w_j is the bit-width of the sub-counter array A_j. Consider C_5 in Fig. 3(c). After the shifting operation has been applied, the location of its sub-count in A_2 will be shifted down by one entry. Therefore, its corresponding expansion status in I_2 must be shifted down by one position as well. The carry propagation of C_2 into A_2 is achieved by setting $A_2[2] = 1$.

As with the `rank` operator, BRICK has been designed to work with small fixed size buckets so that `varshift` can be directly implemented using hardware-optimized instructions. In particular, `varshift` only has to operate on A_2 or higher. Since the size of each level decreases exponentially, the bit-strings formed by each sub-counter array A_2 and above are also very short. As the results show in Section 4, with a bucket size of 64, all sub-counter arrays A_2 and above have a string length at most 64 bits, much less for the higher levels. Therefore, `varshift` can be directly implemented using 64 bit instructions.

The pseudo-code for the `increment` operation is shown in the latter part of Algorithm 1. Again, the iterative procedure shown in Algorithm 1 for `increment` is readily amenable to pipelining in hardware. In general, the lookup or update of each successive level of sub-counter arrays can be pipelined such that at each packet arrival, a lookup or update can operate on A_1 while a previous operation operates on A_2, and so forth.

2.4 Handling Overflows

Thus far, we have assumed in our basic data structure that we are guaranteed that each sub-counter array has been dimensioned to always provide sufficient entries to store all sub-counters in a bucket. To achieve greater memory efficiency, the number of entries in the sub-counter arrays can be reduced so that there is only a very small probability that a bucket will not have sufficient sub-counter array entries. As rigorously analyzed in Section 3 and numerically evaluated in Section 4, this bucket overflow probability can be made arbitrarily small while achieving significant reduction in storage for each bucket.

To facilitate this overflow handling, we extend the basic data structure described in Section 2.1 with a small number of *full-size* buckets $F_1, F_2, \dots, F_J$. Each full-size bucket F_t is organized as k full-size counters (i.e., all counters with a worst-case width of L bits). When a bucket overflow occurs for some B_ℓ, the next available full-size bucket F_t is allocated to store its k counters, where t is just +1 of the last allocated full-size bucket. An overflow status flag f_ℓ is set to indicate the bucket has overflowed. The index of the full-size bucket F_t is stored in a field labeled t_ℓ, which is associated with B_ℓ. In practice, we only need a small number of full-size buckets. As shown in Section 4, for real Internet traces with over a million counters, only about $J \approx 100$ full-size buckets are enough to handle the overflow cases. Therefore, the index field only requires a small number of extra bits per bucket (e.g. 7 bits).

Rather than migrating *all* k counters from B_ℓ to F_{t_ℓ} *at once*, a counter is only migrated *on-demand* upon the next increment operation ("migrate-on-write"). This way, the migration of an overflow counter to a full-size counter does not disrupt other counter updates. The location of counter C_i in F_{t_ℓ} is simply $a = i \mod k$, as before. To indicate if counter C_i has been migrated, a migration status flag $g_{t_\ell}[a]$ is associated with each counter entry $F_{t_\ell}[a]$ (i.e. $g_{t_\ell}[a] = 1$ indicates that the corresponding counter has been migrated).

The modified lookup operation simply first checks if a counter from an overflowed bucket has already been migrated, in which case the full-size count is simply retrieved from corresponding full-size bucket entry. Otherwise, the counter is retrieved as before. The modified increment operation is extended in a similar manner. It first checks if a counter from an overflowed bucket has already been migrated, in which case the full-size counter in the corresponding full-size bucket is incremented. If the counter is from a previously overflowed bucket B_ℓ, but it has not been migrated yet, then it is read from B_ℓ, incremented, and migrated-on-write to the corresponding location in the full-size bucket. Otherwise, the counter in B_ℓ is incremented as before. Finally, before propagating a carry to the next level, we first check if all entries in the next sub-counter array are already being used. If so, the next full-size bucket is allocated and the incremented count is migrated-on-write to the corresponding location.

3. ANALYSIS

3.1 Analytical Guarantees

In this section, we bound the failure probability P_f that the number of overflowed buckets, each of which carries the hefty penalty of having to be allocated an additional bucket of full-size counters (as discussed in Section 2.4), will exceed any given threshold J. We will establish a rigorous relationship between P_f and parameters k_2, k_3, ..., k_p the number of entries BRICK allocates to sub-counter arrays $A_2, ..., A_p$ (The size of A_1 is already fixed to k) and w_1, w_2, ..., w_p, the widths of an entry in $A_2, ..., A_p$. The ultimate objective of this analysis is to find the optimal tradeoff between k_2, k_3, ..., k_p and J that allows us to minimize the amount of overall memory consumption ($h = N/k$ regular buckets + J full-size buckets) while keeping the failure probability P_f under an acceptable threshold (say 10^{-10} or even smaller). Surprisingly, the theory of stochastic ordering [18], which seems unrelated to the context of this work, plays a major role in these derivations.

Recall that the maximum counter width L is partitioned into sub-counter widths $w_1, w_2, ..., w_p$. Only counters whose value is larger than or equal to 2^{L_d}, where L_d is defined as $\sum_{j=1}^{d-1} w_j$, will need an entry in the sub-counter array A_d of a bucket. Since the aggregate count of all counters is no more than M, we know that there will be at most m_d of such counters in the whole counter array, where m_d is defined as $M2^{-L_d}$.

Now imagine at most m_d such counters are uniformly randomly distributed into N array locations through the aforementioned index permutation scheme. We hope that they are very evenly distributed among these buckets so that very few buckets will have more than k_d of them falling into it (i.e., overflow of A_d). Suppose we dimension J_d full-size buckets to handle bucket overflows caused by these counters. We would like to bound the probability that more than J_d buckets will have their A_d arrays overflowed.

We will consider the worst case scenario that there are exactly m_d counters needing entries in A_d. If there are less such counters, the overflow probability will only be smaller, and our tail bound still applies. For convenience, we denote the percentage of them in the counter array $\frac{m_d}{N}$ as α_d.

Let random variables $X_{1,d}, X_{2,d}, ..., X_{h,d}$ be the number of used entries in the sub-counter array A_d among the buckets B_1, B_2, ..., B_h. Each array location has a probability α_d of being assigned one of the m_d counters, and there are k array locations in each bucket, so $X_{j,d}$ is roughly distributed as $Binomial(k, \alpha_d)$ for any j. Here $Binomial(\mathcal{N}, \mathcal{P})$ is the Binomial distribution with $\mathcal{N}$ trials and $\mathcal{P}$ as the success probability of each trial. Therefore, the overflow probability of level d from any bucket B_j is roughly

$$\epsilon_d = \mathcal{B}inotail_{k,\alpha_d}(k_d)$$

where $\mathcal{B}inotail_{\mathcal{N},\mathcal{P}}(\mathcal{K}) \equiv \sum_{z=\mathcal{K}+1}^{\mathcal{N}} \binom{\mathcal{N}}{z} \mathcal{P}^z (1-\mathcal{P})^{(\mathcal{N}-z)}$ denotes the tail probability $Pr[Z > \mathcal{K}]$, where Z has distribution $Binomial(\mathcal{N}, \mathcal{P})$.

Intuitively, these random variables are *almost independent*, as the only dependence among them seems to be that their total is m_d. If we do assume that they are independent, then the probability that the number of total overflows be larger than J_d entries is roughly

$$\delta_d = \mathcal{B}inotail_{h,\epsilon_d}(J_d)$$

Readers understandably will immediately protest this voodoo tail bound result since the $X_{j,d}$'s are not exactly Binomial, and they are not actually independent. Interestingly, we are able to establish a rigorous tail bound of $2\delta_d$, which is only two times the voodoo tail bound δ_d. A similar bound has been established by Mitzenmacher and Upfal in their book [15] which used independent Poisson distributions to bound multinomial distributions (In our case we use independent binomial distributions to bound $X_{1,d}$, $X_{2,d}$, ..., $X_{h,d}$), using techniques from stochastic ordering theory [18] implicitly (i.e., without introducing such concepts).

Based on this rigorous tail bound to be proven in Section 3.2 and summarizing the overflow events from all the subarrays, we arrive at the following corollary.

COROLLARY 1. *Let parameters δ_2, $\cdots$, δ_p be defined as above. The failure probability of insufficient full-size buckets, i.e. that the total number of overflows that need to be moved to the additional full-size buckets from all subarrays exceeds $J = J_2 + \cdots + J_p$, is no more than $2(\delta_2 + \cdots + \delta_p)$.*

If given a target worst-case failure probability P_f of insufficient full-size buckets, e.g. 10^{-10} or even smaller, an optimization procedure remains to configure parameters from w_1 to w_{p-1}, and k_2 to k_p, so that we can achieve the best tradeoff for the overall memory space, which takes into consideration the storage of all sub-counter arrays, index bitmaps, and all full-size buckets, and even the $\lfloor \lg(J) \rfloor + 2$ bits for f_ℓ and t_ℓ in each bucket, which indicate the migration to full-size buckets.

Given the messy nature of the Binomial distribution, "clean" analytical solutions (e.g., based on Lagrange multipliers) do not exist. We designed a quick search strategy that can generate near-optimal configurations. Our evaluation results in Sec 4 are obtained based on the near-optimal parameter configurations generated by this procedure. We omit the detail of this procedure in the interest of space.

3.2 Our Main Tail Bound

In this section, we state formally the aforementioned tail bound theorem (two times the voodoo bound). We would like to state this theorem using generic parameters that have the same symbol as before but without the subscript d, since they can be replaced by the corresponding parameters with subscript d to obtain the tail bound on the number of overflows from every subarray A_d. In particular, we will replace m_d (the number of counters that will have an entry in sub-counter array A_d) by m, and k_d (the number of entries in sub-counter array A_d) by c, as k has been used to denote the number of counters in each bucket in the original counter array. Furthermore, to highlight the general nature of our theorem, we further detach ourselves from the application semantics by stating the theorem as follows.

THEOREM 1. *m balls are uniformly randomly thrown into h buckets that has k entries each, with at most one ball in each entry. Let $N = hk$. Let $X_1^{(m)}$, ..., $X_h^{(m)}$ be the number of balls that falls into each bucket. Let $\alpha = \frac{m}{hk}$ and assume $\alpha \leq \frac{1}{2}$. Let $Y_1^{(\alpha)}$, ..., $Y_h^{(\alpha)}$ be independent random variables distributed as $Binomial(k, \alpha)$. Let $f(x_1, ..., x_h)$ be an increasing function in each argument. Then*

$$E[f(X_1^{(m)}, ..., X_h^{(m)})] \leq 2E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)})]$$

Before we prove this theorem, we need to formally characterize the underlying probability model and in particular specify precisely what we mean by throwing m balls "uniformly randomly" into N entries as follows. Among all $\binom{N}{m}$ ways of injective mapping from m balls into N entries, every way happens with equal probability $\frac{1}{\binom{N}{m}}$, when these balls are considered indistinguishable. We refer to this characterization of the underlying probability model as "throwing m balls into N entries in one shot". It is not hard to verify that the following process of "throwing m balls into N entries one by one" results in the same probability model. In this process, at first a ball is thrown into an entry chosen uniformly from these N entries. Then another ball is thrown into an entry uniformly picked from the remaining $N-1$ entries, and so on. This equivalent characterization of the underlying probability model makes it easier for us to establish the stochastic ordering relationship among vectors of random variables in Section 3.3, an essential step for the proof of Theorem 1. Now we are ready to prove Theorem 1.

PROOF OF THEOREM 1. We use $X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}$ when there are l balls thrown instead of m. In Proposition 1, we prove that for any l value, $\mu(X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)})$ is equivalent to $\mu(Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)} | \sum_{j=1}^h Y_j^{(\alpha)} = l)$, where $\mu(Z)$ denotes the distribution of a random variable or vector Z. In other words, conditioned upon $\sum_{j=1}^h Y_j^{(\alpha)} = l$, the independent random variables $Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)}$ have the same joint distribution as dependent random variables $X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}$. Then we prove in Proposition 3 that, when $l \le l'$, $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$ is stochastically less than or equal to (defined later) $[X_1^{(l')}, X_2^{(l')}, \ldots, X_h^{(l')}]$. For any increasing function $f(x_1, x_2, ..., x_h)$, we have

$$\begin{aligned}
& E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)})] \\
= & \sum_{l=0}^{N} E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)}) \mid \sum_{j=1}^h Y_j^{(\alpha)} = l] \Pr[\sum_{j=1}^h Y_j^{(\alpha)} = l] \\
\ge & \sum_{l=m}^{N} E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)}) \mid \sum_{j=1}^h Y_j^{(\alpha)} = l] \Pr[\sum_{j=1}^h Y_j^{(\alpha)} = l] \\
= & \sum_{l=m}^{N} E[f(X_1^{(l)}, ..., X_h^{(l)})] \Pr[\sum_{j=1}^h Y_j^{(\alpha)} = l] \quad (1) \\
\ge & \sum_{l=m}^{N} E[f(X_1^{(m)}, ..., X_h^{(m)})] \Pr[\sum_{j=1}^h Y_j^{(\alpha)} = l] \quad (2) \\
= & E[f(X_1^{(m)}, \cdots, X_h^{(m)})] \Pr[\sum_{j=1}^h Y_j^{(\alpha)} \ge m] \\
= & E[f(X_1^{(m)}, \cdots, X_h^{(m)})] \mathcal{B}inotail_{N,\alpha}(m-1) \\
\ge & \frac{1}{2} E[f(X_1^{(m)}, ..., X_h^{(m)})] \quad (3)
\end{aligned}$$

Equality (1) is due to Proposition 1, inequality (2) is due to Proposition 3, and inequality (3) is due to the properties of the 50-percentile point of Binomial distributions proven in [9]. □

COROLLARY 2. *Let the variable be as defined in Theorem 1. Let c and J be some constants. Let* $\epsilon = \mathcal{B}inotail_{k,\alpha}(c)$. *Then*

$$Pr[\sum_{j=1}^h 1_{\{X_j^{(m)} > c\}} > J] \le 2\mathcal{B}inotail_{h,\epsilon}(J)$$

PROOF. Consider function $f(x_1, x_2, \cdots, x_h) \equiv 1_{\{\sum_{j=1}^h 1_{\{x_j > c\}} > J\}}$, which is an increasing function of x_1, ..., x_h. From Theorem 1 we have $\Pr[\sum_{j=1}^h 1_{\{X_j^{(m)} > c\}} > J] \le 2\Pr[\sum_{j=1}^h 1_{\{Y_j^{(\alpha)} > C\}} > J]$. Since $\{1_{\{Y_j^{(\alpha)} > c\}}\}_{1 \le j \le h}$ are independent Bernoulli random variables with probability $\epsilon = \mathcal{B}inotail_{k,\alpha}(c)$, their sum is distributed as $Binomial(h, \epsilon)$. Therefore $\Pr[\sum_{j=1}^h 1_{\{Y_j > c\}} > J]$ is equal to $\mathcal{B}inotail_{h,\epsilon}(J)$. □

3.3 Proofs of propositions 1–3

PROPOSITION 1. $\mu(X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}) = \mu(Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)} | \sum_{j=1}^h Y_j^{(\alpha)} = l)$

PROOF. It suffices to prove that for any nonnegative integers l_1, l_2, ..., l_h that satisfy $\sum_{j=1}^h l_j = l$, $\Pr[X_1^{(l)} = l_1, X_2^{(l)} = l_2, \ldots, X_h^{(l)} = l_h] = \Pr[Y_1^{(\alpha)} = l_1, Y_2^{(\alpha)} = l_2, \ldots, Y_h^{(\alpha)} = l_h \mid \sum_{j=1}^h Y_j^{(\alpha)} = l]$. We show that both the LHS (left hand side) and the RHS (right hand side) are equal to

$$\frac{\binom{k}{l_1}\binom{k}{l_2}\cdots\binom{k}{l_h}}{\binom{N}{l}} \quad (4)$$

Since there are $\binom{N}{l}$ ways of selecting l entries out of a total of N entries, and each way happens with equal probability $\frac{1}{\binom{N}{l}}$, the LHS is equal to (4) because there are $\binom{k}{l_1}\binom{k}{l_2}\cdots\binom{k}{l_h}$ ways among them that result in the event $\{X_1^{(l)} = l_1, X_2^{(l)} = l_2, \ldots, X_h^{(l)=l_h}\}$. Now we prove that the RHS is equal to (4) as well. Since $Y_1^{(\alpha)}$, $Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)}$ are independent random variables with distribution $Binomial(k, \alpha)$, $\sum_{j=1}^h Y_j^{(\alpha)}$ has distribution $Binomial(N, \alpha)$ and therefore

$$\Pr[\sum_{j=1}^h Y_j^{(\alpha)} = l] = \binom{N}{l} \alpha^l (1-\alpha)^{N-l} \quad (5)$$

Additionally, when $\sum_{j=1}^h Y_j^{(\alpha)} = l$, we have

$$\begin{aligned}
& \Pr[Y_1^{(\alpha)} = l_1, Y_2^{(\alpha)} = l_2, \ldots, Y_h^{(\alpha)} = l_h, \sum_{j=1}^h Y_j^{(\alpha)} = l] \\
= & \prod_{j=1}^h \binom{k}{l_j} \alpha^{l_j} (1-\alpha)^{k-l_j} \\
= & \alpha^l (1-\alpha)^{N-l} \prod_{j=1}^h \binom{k}{l_j} \quad (6)
\end{aligned}$$

Combining (5) and (6) we obtain that the RHS is equal to (4) as well. □

Stochastic ordering is a way to compare two random variables. Random variable X is stochastically less than or equal to random variable Y, written $X \le_{st} Y$, iff $E\phi(X) \le E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. An equivalent definition of $X \le_{st} Y$ is that $Pr[X > t] \le Pr[Y > t], -\infty < t < \infty$. The definition involving increasing functions also applies to random vectors $X = (X_1, ..., X_h)$ and $Y = (Y_1, ..., Y_h)$: $X \le_{st} Y$ iff $E\phi(X) \le E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. Here ϕ is increasing means that it is increasing in each argument separately with other arguments being fixed. This is equivalent to $\phi(X) \le_{st} \phi(Y)$. Note this definition is a much stronger condition than $Pr[X_1 > t_1, ..., X_h > t_h] \le Pr[Y_1 > t_1, ..., Y_h > t_h]$ for all $t = (t_1, ..., t_h) \in \mathcal{R}^n$.

Now we state without proof a fact that will be used to prove Proposition 3. Its proof can be found in all books that deal with stochastic ordering [18].

PROPOSITION 2. *Let X and Y be two random variables (or vectors). $X \leq_{st} Y$ iff there exists X' and Y' such that $\mu(X') = \mu(X)$, $\mu(Y') = \mu(Y)$, and $Pr[X' \leq Y'] = 1$.*

Now we are ready to prove the following proposition.

PROPOSITION 3. *For any $0 \leq l < l' \leq N$, we have $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}] \leq_{st} [X_1^{(l')}, X_2^{(l')}, \ldots, X_h^{(l')}]$.*

PROOF. It suffices to prove it for $l' = l + 1$. Our idea is to find random variables Z and W such that Z has the same distribution as $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$, W has the same distribution as $[X_1^{(l+1)}, X_2^{(l+1)}, \ldots, X_h^{(l+1)}]$, and $\Pr[Z \leq W] = 1$. We will use the aforementioned probability model that is generated by "throwing m balls into N entries one-by-one" random process. Now given any outcome ω in the probability space Ω, let $Z(\omega) = [Z_1(\omega), Z_2(\omega), ..., Z_h(\omega)]$, where $Z_j(\omega)$ is the number of balls in the j_{th} bucket after we throw l balls into these N entries one by one. Now with all these l balls there, we throw the $(l+1)_{th}$ ball uniformly randomly into one of the remaining empty entries. We define $W(\omega)$ as $[W_1(\omega), W_2(\omega), ..., W_h(\omega)]$, where $W_j(\omega)$ is the number of balls in the j_{th} bucket after we throw in the $(l+1)_{th}$ ball. Clearly we have $Z(\omega) \leq W(\omega)$ for any $\omega \in \Omega$ and therefore $\Pr[Z \leq W] = 1$. Finally, we know from the property of the "throwing m balls into N entries one-by-one" random process that Z and W have the same distribution as $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$ and $[X_1^{(l+1)}, X_2^{(l+1)}, \ldots, X_h^{(l+1)}]$ respectively. $\Box$

4. PERFORMANCE EVALUATIONS

We first present in Section 4.1 numerical examples of our tail bounds derived in Section 3 under a set of typical parameter configurations. Our results show that BRICK is extremely memory-efficient. Our results show that the number of extra bits needed per counter in addition to the (loose) intuitive lower bound $\lg \frac{M}{N}$ derived under reasonable assumptions (explained later) remains practically constant with increasing number of flows N, and hence the solution is scalable. The results also show that the number of extra bits needed per counter is not far from the information-theoretic lower bound (tighter and 1.5 bits larger than the intuitive lower bound). We then evaluate in Section 4.2 the performance of our architecture using parameters (e.g., M and N) extracted from real-world Internet traffic traces. Finally, we discuss implementation issues in Section 4.3.

4.1 Numerical results of analytical bounds

4.1.1 Memory costs and lower bounds

For a configuration of these parameters, the amount of memory required can be computed as follows:

$$\mathcal{S}_\ell = \left(\left[\sum_{j=1}^{p} k_j (w_j + 1) \right] - k_p \right) + (\lfloor \lg J \rfloor + 2) \quad (7)$$

$$\mathcal{S} = \left(\sum_{\ell=1}^{h} \mathcal{S}_\ell \right) + Jk\,(L+1) \quad (8)$$

Here $\mathcal{S}_\ell$ is the memory cost of each bucket; its first component corresponds to the space required for storing the sub-counter arrays and the index bitmaps, and its second component corresponds to the overflow status flag and the index to the corresponding full-size bucket[5]. Then the total memory cost $\mathcal{S}$ is $h = \lceil \frac{N}{k} \rceil$ buckets of size $\mathcal{S}_\ell$ each plus J full-size buckets of size $k(L+1)$ each. (For each full-size counter of size L, we need 1 bit for indicating the migration status.)

[5]Since there are J full-size buckets, this index can be stored in $\lfloor \lg J \rfloor + 1$ bits.

Table 1: Sub-counter array sizing and per-counter storage for $k = 64$ and $P_f = 10^{-10}$.

(a) Sizing of sub-counter arrays.

p	k_2	k_3	k_4	k_5	w_1	w_2	w_3	w_4	w_5
3	15	3			$\lg \frac{M}{N} + 3$	4	13		
4	25	10	2		$\lg \frac{M}{N} + 2$	2	4	12	
5	25	10	3	1	$\lg \frac{M}{N} + 2$	2	3	4	9

(b) Size of each sub-counter array $= k_j \times w_j$ (in bits).

p	A_2	A_3	A_4	A_5
3	15 × 4 = 60	3 × 13 = 39		
4	25 × 2 = 50	10 × 4 = 40	2 × 12 = 24	
5	25 × 2 = 50	10 × 3 = 30	3 × 4 = 12	1 × 9 = 9

(c) Storage per counter.

$p = 3$	$p = 4$	$p = 5$
$\lg \frac{M}{N} + 6.05$	$\lg \frac{M}{N} + 5.66$	$\lg \frac{M}{N} + 5.50$

For comparison purposes, we derive two lower bounds on the minimum memory requirement per counter under the aforementioned constraint that the sum of counter values C_1, C_2, ..., C_N (i.e., the total number of increments) is no more than M. The first lower bound $\lg \frac{M}{N}$ is intuitive yet loose. It corresponds to only the number of bits we need to store these counters when each counter has the same value $\frac{M}{N}$. This bound is clearly very loose since it does not account for the extra bits needed (1) to delimit these counter values and (2) to allow for fast random accesses, when counter values are not uniformly $\frac{M}{N}$. We will show that we are able to achieve both (1) and (2) in between 5 to 6 bits per counter in the worst case.

The second lower bound is the maximum empirical entropy of all the counter values C_1, C_2, ..., C_N, subject to the constraint $\sum_{i=1}^{N} C_i \leq M$. We omit its detailed (standard information-theoretic) definition, formulation, and solution (through Lagrange multipliers) in the interest of space. Note that this information-theoretic lower bound is in general not achievable in our case for two reasons. First, although prefix-free coding (such as Huffman) of each counter allows us to get very close to this lower bound and settles the aforementioned counter delimiting problem, it does not allow for fast random accesses. Second, no prefix-free coding exists that can reach this bound with probability 1 without knowing the distribution of counter values in advance (even for infinite block size). We will show that we are within 4 bits away from this information theoretic lower bound. In other words, these 4 extra bits per counter allow us to achieve fast random access given arbitrary unknown distributions of counter values.

4.1.2 Numerical results with various configurations

Comparing with the first lower bound, $\lg \frac{M}{N}$, the number of extra bits per counter that our solution adds is $\sigma = (\mathcal{S}/N) - \lg(M/N)$. We use σ as a metric to evaluate the space efficiency of our solution.

Here we only present results for the case with $k = 64$ counters per bucket, which could ensure that all string operations are within 64 bits and allow for direct implementations using 64-bit instructions in modern processors [1, 2]. As we shall see, substantial statistical multiplexing can already be achieved with $k = 64$. For the results presented in Table 1, we used representative parameters with $N = 1$ million counters and $M = 16$ million as the maxi-

Table 2: Information-theoretic lower bound.

$\lg(\frac{M}{N})$=	2	4	6	8
Avg Entropy=$\lg \frac{M}{N}+$	1.61	1.49	1.45	1.45

mum total increments during a measurement period. We also set the failure probability to be $P_f = 10^{-10}$, which is a tiny probability corresponding to an average of one failure (when there are more than J *overflowed buckets*) every ten thousand years. We will later show in Figures 4, 5, and 6 that the additional per-counter storage cost beyond the minimum width of the average count is practically a constant unrelated to the number of flows N, the maximum total increments M, or the failure probability P_f.

In Table 1(a), the number of entries and the width for each sub-counter array are shown for BRICK implementations with varying number of levels p. As can be seen, in each design, the number of entries decreases exponentially as we go to the higher sub-counter arrays. This is the main source of our compression. With $k = 64$, the rank indexing operation described in Section 2 only needs to be performed on bitmaps with $|I_j| \leq 64$ bits (much less than 64 for the higher sub-counter arrays) and can be directly implemented using 64-bit popcount and bitwise-logical instructions that are available in modern processors [1, 2]. Table 1(b) shows the size of each sub-counter array. For all three designs, the space requirement for each sub-counter array other than A_1 is also less than 64 bits. Therefore, the "varshift" operator described in Section 2.3, which only needs to operate on A_2 and higher, can be directly implemented using 64-bit shift and bitwise-logical instructions as well.

In Table 1(c), the per-counter storage cost for the three designs are shown. For three levels, the extra storage cost per counter is 6.05, and the extra storage costs per counter are 5.66 and 5.50 for four and five levels, respectively. The amount of extra storage only decreases slightly with additional levels in the BRICK implementation. For example, as we go from three to five levels, the reduction of $6.05 - 5.50 = 0.55$ extra bits is only about 5.5% in the overall per-counter cost if $\lg \frac{M}{N} = 4$.

In Figures 4, 5, and 6 we evaluate the impact of different N, M, and P_f, where we use $k = 64$ and $p = 4$.

Figure 4 shows that the added per-counter cost remains practically constant as we increase N exponentially by powers of 10. Similarly, Figure 5 shows that the added cost also remains practically constant with different ratios of M and N. These results show that BRICK is scalable to different values of M and N with per-counter storage cost within approximately a constant factor from the minimum width of the average count. Figure 6 shows the impact of decreasing failure probability. We show results for $P_f = 10^{-10}$ down to 10^{-20}. Again, we see that the change in storage cost is negligible with decreasing failure probability, which means BRICK can be optimized to vanishingly small failure probabilities with virtually no impact on storage cost.

Finally, in Table 2, we explore the question "How far is our solution from the theoretically optimum solution (i.e., the second lower bound above)?". We can see that our scheme is only within 4 bits away from that theoretical bound.

4.2 Results for real Internet traces

In this section, we evaluate our active counter architecture using real-world Internet traffic traces. The traces that we used were collected at different locations in the Internet, namely University of Southern California (USC) and University of North Carolina (UNC), respectively. The trace from USC was collected at their Los Nettos tracing facility on February 2, 2004, and the trace from UNC was collected on a 1 Gbps access link connecting to the campus to the rest of the Internet on April 24, 2003. For each trace, we used a 10-minute segment, corresponding to a measurement epoch. The trace segment from USC has 18.9 million packets and around 1.1 million flows; the trace segment from UNC has 32.6 million packets and around 1.24 million flows.

We use the same parameter settings as the evaluations in Section 4.1 with 64 counters per bucket, four levels, and a failure probability of $P_f = 10^{-10}$. The total storage space required for the USC trace is 1.39 MB, and the total required for the UNC trace is 1.63 MB. In comparison, a naive implementation would require a worst-case counter width for all counters. Both traces require a worst-case width of 25 bits, whereas the BRICK implementations require a per-counter cost of about 10 bits. The total storage required for a naive implementation is 3.85 MB for the USC trace and 4.40 MB for the UNC trace. The BRICK implementations represent a 2.5x improvement in both cases. This is very exciting since with the same amount of memory, we will be able to squeeze in 2.5 times more counters, which is badly needed in future faster and "more crowded" Internet!

Table 3: Percentage of full-size buckets.

Trace	h	J	$\frac{J}{h}$
USC	17.3K	111	0.60%
UNC	19.5K	104	0.57%

Table 3 shows the number of full-size buckets derived using our tail bounds in comparison to the number of buckets for each of the two traces. In practice, only a small number of full-size buckets are needed to guarantee a tiny probability ($P_f = 10^{-10}$) that we will have insufficient number of full-size buckets to handle bucket overflows.

4.3 Implementation issues

In a BRICK implementation, all sub-counter arrays (A_j) and index bitmaps (I_j) are fixed in size, and the number and size of buckets are also fixed. Consider the three level case shown in Table 1 with $k = 64$. Both lookup and increment operations can be performed with 10 memory accesses in total, 5 reads and 5 writes. For the bucket being read or updated, we first retrieve all bitmaps (I_j), bucket overflow status flag f_ℓ, and an index field t_ℓ to a full-size bucket in case a bucket overflow has previously occurred. All this information for a bucket can be retrieved in two memory reads with 64-bit words, the first word corresponds to I_1 with 64-bits, and the second word stores $I_2 = 3$ bits, the overflow status flag, and the t_ℓ (about 7 bits). If f_ℓ is not set, then we need up to three reads and writes to update the three levels of sub-counter arrays. The updated index bitmaps and overflow status flags can be written back in two memory writes. If f_ℓ has been set, then we read directly from the corresponding entry in the full-size bucket indicated by t_ℓ for a lookup operation, avoiding the need to read the sub-counter arrays, hence requiring fewer memory acceses. Similarly, an increment operation for a counter that is already in a full-size bucket takes only one read and one write to update. If a bucket overflow occurs during an increment of a counter in a bucket, there is no need to access the last sub-counter array (otherwise, we wouldn't have an overflow). Therefore, we save two memory accesses at the expense of one write to the full-size bucket. With index bitmaps, overflow status flag, and full-size index field packed into two words, the worst case number of memory accesses is 10 in total, which permits updates in 20ns with a 2ns SRAM time, enabling over 15 million packets per second of updates.

BRICK is also amenable to pipelining in hardware. In general, the lookup or update of each successive level of sub-counter arrays

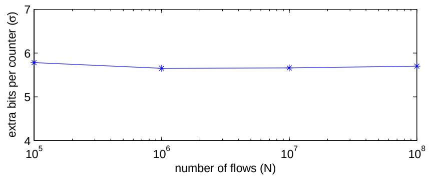


Figure 4: Impact of increasing number of flows N. Extra bits σ in the range of $[5.65, 5.78]$.

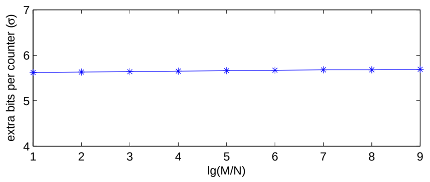


Figure 5: Impact of increasing $\lg \frac{M}{N}$. Extra bits σ in the range of $[5.62, 5.69]$.

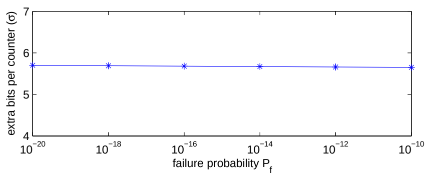


Figure 6: Impact of decreasing failure probability P_f. Extra bits σ in the range of $[5.65, 5.70]$.

can be pipelined such that at each packet arrival, a lookup or update can operate on A_1 while a previous operation operates on A_2, and so forth. This enables the processing of hundreds of millions of packets per second.

5. CONCLUSION

We presented a novel exact active statistics counter architecture called BRICK (Bucketized Rank Indexed Counters) that can very efficiently store large arrays of variable width counters entirely in SRAM while supporting extremely fast increments and lookups. This high memory (SRAM) efficiency is achieved through a statistical multiplexing technique, which by grouping a fixed number of randomly selected counters into a bucket, allows us to tightly bound the amount of memory that needs to be allocated to each bucket. Statistical guarantees of BRICK are proven using a combination of stochastic ordering theory and probabilistic tail bound techniques. We also developed an extremely simple and memory-efficient indexing structure called rank-indexing to allow for fast random access of every counter inside a bucket. Experiments with real-world Internet traffic traces show that our solution can indeed maintain large arrays of exact active statistics counters with moderate amounts of SRAM.

6. REFERENCES

[1] Intel 64 and IA-32 architectures software developer's manual, volume 2B, November 2007. Available at ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf.

[2] Software optimization guide for AMD family 10h processors, December 2007. Available at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf.

[3] S. Cohen and Y. Matias. Spectral bloom filters.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 2004.

[5] A Cvetkovski. An algorithm for approximate counting using limited memory resources. In *Proc of ACM SIGMETRICS*, 2007.

[6] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.

[7] C. Estan and G. Varghese. New directions in traffic measurement and a ccounting. In *Proc. of ACM SIGCOMM*, August 2002.

[8] Robert G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, 1963.

[9] Rainer GöbDoi. Bounds for median and 50 percetage point of binomial and negative binomial distribution. In *Metrika, Volume 41, Number 1, 43-54*, 2003.

[10] Nan Hua, Haiquan Zhao, Bill Lin, and Jun Xu. Rank-indexed hashing: A compact construction of bloom filters and variants. In *Proc. of IEEE ICNP*, October 2008.

[11] G. Jacobson. Space-efficient static trees and graphs. In *30th FOCS*, pages 549–554, 1989.

[12] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. of ACM SIGCOMM IMC*, October 2003.

[13] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. of ACM SIGMETRICS*, 2004.

[14] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: A novel counter architecture for per-flow measurement. In *Prof. of ACM SIGMETRICS*, 2008.

[15] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[16] R. Morris. Counting large numbers of events in small registers. In *Commun. ACM 21(10)*, 1978.

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] Alfred Müller and Dietrich Stoyan. *Comparison Methods for Stochastic Models and Risks*. Wiley, 2002.

[19] S. Ramabhadran and G. Varghese. Efficient implementation of a statistics counter architecture. In *Proc. ACM SIGMETRICS*, June 2003.

[20] M. Roeder and B. Lin. Maintaining exact statistics counters with a multilevel counter memory. In *Proc. of IEEE Globecom, Dallas, USA, 2004*.

[21] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown. Maintaining statistics counters in router line cards. In *IEEE Micro*, 2002.

[22] Rade Stanojevic. Small active counters. In *Proc of IEEE Infocom*, 2007.

[23] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM*, 2004.

[24] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and application. In *Proc. of ACM SIGCOMM IMC*, October 2004.

[25] Q. Zhao, A. Kumar, J. Wang, and J. Xu. Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices. In *Proc. of ACM SIGMETRICS*, June 2005.

[26] Q. Zhao, J. Xu, and Z. Liu. Design of a novel statistics counter architecture with optimal space and time efficiency. In *Proc. of ACM SIGMETRICS*, June 2006.