Design and Performance Analysis of a DRAM-based Statistics Counter Array Architecture[*]

Haiquan (Chuck) Zhao[†] Hao Wang[§] Bill Lin[§] Jun (Jim) Xu[†]

[†] College of Computing
Georgia Institute of Technology

[§] Department of Electrical and Computer Engineering
University of California, San Diego

ABSTRACT

The problem of maintaining efficiently a large number (say millions) of statistics counters that need to be updated at very high speeds (e.g. 40 Gb/s) has received considerable research attention in recent years. This problem arises in a variety of router management and data streaming applications where large arrays of counters are used to track various network statistics and implement various counting sketches. It proves too costly to store such large counter arrays entirely in SRAM while DRAM is viewed as too slow for providing wirespeed updates at such high speeds.

In this paper, we propose a DRAM-based counter architecture that can effectively maintain wirespeed updates to large counter arrays. The proposed approach is based on the observation that modern commodity DRAM architectures, driven by aggressive performance roadmaps for consumer applications (e.g. video games), have advanced architecture features that can be exploited to make a DRAM-based solution practical. In particular, we propose a randomized DRAM architecture that can harness the performance of modern commodity DRAM offerings by interleaving counter updates to multiple memory banks. The proposed architecture makes use of a simple randomization scheme, a small cache, and small request queues to statistically guarantee a near-perfect load-balancing of counter updates to the DRAM banks. The statistical guarantee of the proposed scheme is proven using a novel combination of convex ordering and large deviation theory. Our proposed counter scheme can support arbitrary increments and decrements at wirespeed, and it can support different number representations, including both integer and floating point number representations.

[*]This work is supported by collaborative NSF grants CNS-0905169 and CNS-0904743, funded under the American Recovery and Reinvestment Act of 2009 (Public Law 111-5)

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations – Network Monitoring

General Terms

Algorithms, Measurement, Theory, Performance

Keywords

Convex Ordering, Majorization, Large Deviation Theory, Statistics Counter, Network Measurement

1. INTRODUCTION

It is widely accepted that network measurement is essential for the monitoring and control of large networks. For tracking various network statistics (e.g. performing SNMP link counts) and for implementing various network measurement, router management, intrusion detection, traffic engineering, and data streaming applications, there is often the need to maintain very large arrays of statistics counters at wirespeed (e.g. many millions of counters for per-flow measurements [25, 21]). In general, each packet arrival may trigger the updates of multiple per-flow statistics counters, resulting in possibly tens of millions of updates per second. For example, on an 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. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows if the measurements take place at high-speed links as smaller counters can quickly overflow. Additionally, a practical counter array solution has to be able to deal with any arbitrary (including adversarial) incoming sequence of counter addresses (i.e., indices) to be incremented because statistics counter arrays may often be used in security applications (e.g., intrusion detection) and/or in settings where an adversary has incentives to compromise their performance guarantees.

While implementing large counter arrays in SRAM can satisfy the performance needs, the amount of SRAM required is often both infeasible and impractical. As reported in [31], real-world Internet traffic traces show that a very large number of flows can occur during a measurement period. For example, an Internet traffic trace from UNC has 13.5 million flows. Assuming 64 bits for each flow counter, 108 MB of SRAM would already be needed for just the counter storage, which is prohibitively expensive. Therefore, researchers have actively sought alternative ways to realize large arrays of statistics counters at wirespeed [25, 21, 23, 31].

Several designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed. Their basic 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 come close to overflowing, they will be scheduled to be "flushed" back to the corresponding DRAM counter. These schemes all significantly reduce the SRAM cost. In particular, the scheme by Zhao et al. [31] achieves the theoretically minimum SRAM cost of 4 to 5 bits per counters when the SRAM-to-DRAM speed ratio is between 1/10 (4ns/40ns) and 1/20 (3ns/60ns). While this is a substantial reduction over a straightforward SRAM implementation, storing say 4 bits per counter in SRAM for 13.5 million flows would still require nearly 7 MB of SRAM, which is a substantial amount and difficult to implement on-chip. Moreover, since the bounds on SRAM requirements for the hybrid SRAM/DRAM approaches are based on preventing SRAM counter overflows, the SRAM requirements are also dependent on the *size* of the increments. If a wide range of increments is needed, and *large* increments are possible, then the possibility for overflows could occur earlier and more SRAM counter bits would be needed to compensate, resulting in yet larger SRAM requirements. In addition, the existing hybrid SRAM/DRAM approaches do not support arbitrary decrements and are based on an integer number representation, whereas a *floating point number* representation may be needed in some applications [11, 30].

1.1 DRAM Can Be Plenty Fast

In this paper, we challenge the main premise behind previous hybrid SRAM/DRAM architecture proposals. Their main premise is that DRAM access latencies are too slow for wirespeed updates, though DRAMs provide plenty of storage capacity for maintaining exact counts for large arrays of counters. However, our main observation is that modern DRAM architectures have advanced architecture features [9, 13, 28, 29] that can be exploited to make a DRAM solution practical. We then propose a DRAM-based counter architecture that allows for wirespeed updates to large counter arrays.

Driven by a seemingly insatiable appetite for extremely aggressive memory data rates in graphics, multimedia, video game, and high-definition television applications, the memory semiconductor industry has continually been driving very aggressive roadmaps in terms of ever increasing memory bandwidths that can be provided at commodity pricing (about $0.01/MB as of this writing). For example, the Cell processor from IBM/Sony/Toshiba [7] uses two 32-bit channels of XDR memories [2] with an aggregated memory bandwidth of 25.6 GB/s. Using an approach called micro-threading [29], the XDR memory architecture provides internally 16 independent banks inside just a *single* DRAM chip, 256 memory banks across 16 DRAM chips that are typically packaged into a single memory module. Next generation memory architectures [3] are expected to achieve a data rate upwards of 16 GB/s on a single 16-bit channel, 64 GB/s on an equivalent dual 32-bit channel interface used by the Cell processor. This enormous amount of memory bandwidth can be shared or time-multiplexed by multiple network functions. The Intel IXP network processor [4] is another example of a state-of-the-art network processor that has multiple high-bandwidth memory channels. Besides XDR, other memory consortia have similar capabilities and advanced architecture features on their roadmaps as well, since they are driven by the same demanding consumer applications. For example, extremely high data efficiency can be achieved using DDR3 memories as well [28].

Although these modern high-speed DRAM offerings provide extraordinary memory bandwidths, the peak access bandwidths are only achievable when memory locations are accessed in a *memory interleaving mode* (to ensure that *internal memory bank conflicts* are avoided). Unlike graphics and video applications with mostly sequential memory access patterns, which are known to be friendly to memory interleaving, the conventional wisdom is that the random (or even adversarial) access nature of network measurement applications would *render interleaved access modes unusable.* For example, for XDR memories [2], a new memory operation could be initiated every 4 ns when the internal memory banks are interleaved, but a worst-case access latency of 40 ns is required for a read or a write operation if memory bank accesses are unrestricted.

1.2 Our Approach

Our main idea is to randomly distribute the memory addresses to which consecutive counter indices map across the memory banks so that a near-perfect balancing of memory access loads can be provably achieved, under arbitrary (including adversarial) counter update patterns. In particular, we propose a novel scheme called a *randomized counter architecture* that works as follows. Suppose the SRAM-to-DRAM random access latency ratio is μ (e.g. μ = 4ns/64ns = 1/16). The randomized counter scheme works by using $B > 1/\mu$ DRAM banks and randomly distributing the array of counters across these DRAM banks so that when the loads of these memory banks are perfectly balanced, the worst-case load factor of any DRAM bank is $1/B\mu < 1$. In particular, we apply a random permutation function to the counter index to obtain a randomly permuted counter index, which is in turn mapped to a memory bank (according to the traditional memory interleaving/addressing scheme that is in use).

The purpose of this simple randomization scheme is to ensure that the memory load is evenly distributed when different counters are updated over time, *under arbitrary counter update sequences.* Note that an adversary can conceivably overload a memory bank by sending traffic that would trigger the update of the same counter because these counter updates will necessarily be mapped to the same memory bank. However, this case can be easily handled through caching. By caching pending counter update requests, we can ensure that repeated updates to the same counter within a certain time window will not result in any new memory operations. Instead, the pending counter update request is simply modified to reflect the new counter update request.

While this architecture of randomization plus caching sounds simple and straightforward, the key contribution of this paper is a mathematical one: we prove that index randomization combined with a reasonably sized cache can handle with overwhelming probability arbitrary (including adversarial) counter update patterns without having overload situations as reflected by long queuing delays (to be made precise in Section 4). This result is a *worst-case large deviation theorem* in nature [19] because it establishes a bound on the largest (worst case) value among the tail probabilities of having long queueing delays under all admissible (including adversarial) counter update patterns. In the course of proving this result, we also establish a novel (to mathematicians) general methodology for establishing worst-case large deviation bounds. Worst-case large deviation bounds such as ours are very hard to obtain because the parameter space (in our case all admissible counter update patterns) underlying the large deviation (tail bound) problem is gigantic. Although given a particular parameter setting (i.e., a particular counter update sequence), establishing the tail bound in our problem is straightforward through the Chernoff technique, enumerating this procedure over the entire parameter space is computationally impossible, and finding the maximum of such bounds appears to be analytically impossible as well (through tractable optimization techniques).

Our methodology to overcome this difficulty is a novel combination of convex ordering and (traditional) large deviation theory. To our surprise, we found that we are able to find a parameter configuration (i.e., counter update sequence) that dominates all other configurations by the convex order (explained later). Since the exponent function is a convex function, we are able to dominate the moment generating function (MGF) of queueing delay (a random variable) under all other parameter settings by the MGF under the worst-case parameter setting. We can then apply the Chernoff technique to this worst-case MGF to obtain very sharp tail bounds. Using this theoretical framework, we show that only very small queues (say on the order of $K = 45$ entries per request queue) are required to ensure a negligible overflow probability (say under 10^{-14}).

1.3 Summary of contributions

We make several contributions in this work:

- We propose a DRAM-based counter architecture that can effectively maintain wirespeed updates to large counter arrays. As we shall see, our proposed counter scheme can leverage the internal independent memory banks already available inside a modern DRAM chip without the expense of multiple parallel memory channels, thus making both schemes very cost effective.

- We develop a novel mathematical methodology for establishing worst-case large deviation bounds, and present a rigorous theoretical analysis on the performance of our proposed architecture in the worst-case as one of its applications.

- Compared to existing hybrid SRAM/DRAM counter architectures [25, 21, 23, 31], our DRAM-bases solution offers three clear advantages. First, our solution can achieve the same update speeds to counters, without the need for a non-trivial amount of SRAMs for storing partial counts. Second, our solution can easily accommodate increments/decrements of any arbitrary integer (needed for counting bytes) or floating point values (needed in certain data streaming applications [11, 30]), while hybrid SRAM/DRAM counter architectures typically can only accommodate "increment by 1" efficiently. Finally, as we shall show in Section 4, our DRAM-based solution requires only a small amount of "control" SRAM, the size of which is *independent* of the number of counters being maintained. Therefore, our approach is scalable to future application scenarios in which much larger counter arrays are conceivable (possibly hundreds of millions of counters). This is in contrast to hybrid SRAM/DRAM architectures where the SRAM requirement grows linearly with the number of counters being maintained. Our solution only grows linearly in the DRAM requirement with respect to the number of counters, which is practical given low cost of DRAM[1].

1.4 Outline of Paper

The rest of the paper is organized as follows. Section 2 outlines additional related work. Section 3 describes our proposed randomized counter architecture in details. Section 4 provides a rigorous analysis on the performance of our architecture in the worst-case. Section 5 presents an evaluation of our proposed architecture. Finally, Section 6 concludes the paper.

2. RELATED WORK

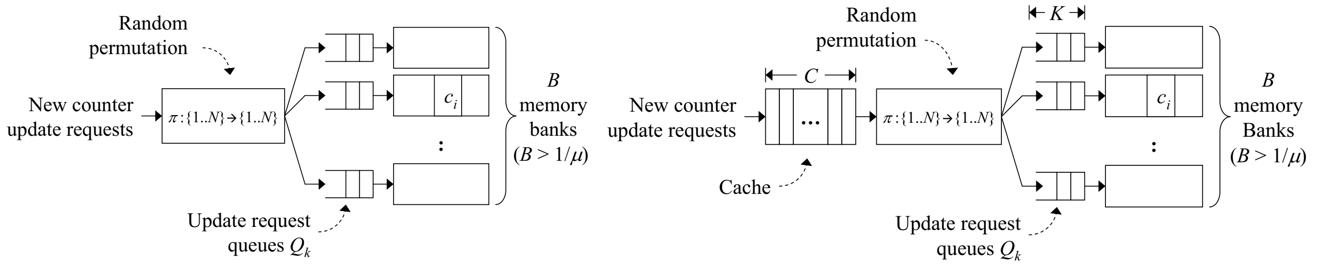
In this section, we outline prior work related to our problem. As already discussed in Section 1, the naive approach of storing full counters in SRAM is prohibitively expensive. Although a hybrid SRAM/DRAM architecture [25, 21, 23, 31] significantly reduces the SRAM requirement, the amount of SRAM required for tracking a large number of counters (say in the tens of millions) is still substantial and difficult to implement on-chip.

Besides hybrid SRAM/DRAM architectures, several complementary SRAM-based approaches have also been proposed that aim to make feasible the storage of large counter arrays in SRAM through efficient representations. One category of approaches is approximate counting [16, 5, 27], which are all based on the basic idea invented by Morris [16]. The idea is to probabilistically increment a counter based on the current counter value. This approach is applicable to those network measurement and data streaming applications that can tolerate the inaccuracies.

A second approach, which was recently proposed, is a counter architecture called counter braids [14], which was inspired by the construction of low-density parity-check codes and can keep track of exact counts of all flows without remembering the association between flows and counters[2]. 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. Flow counts can be decoded through an iterative decoding process at the end of the measurement epoch.

A third approach, which was also recently proposed, is based on an efficient variable-length counter representation called BRICK [10]. It uses a simple operator called rank-indexing to link together counter segments rather than using expensive memory pointers. This counter architecture has

[1]As of this writing, 2GB of DRAM costs under \$20, over 100MB/\$.
[2]Counter braids consider a more general problem that also addresses flow association.


Random permutation
New counter update requests
$\pi:\{1..N\}\to\{1..N\}$
c_i
:
B memory banks $(B > 1/\mu)$
Update request queues Q_k
Random permutation
K
C
...
New counter update requests
Cache
$\pi:\{1..N\}\to\{1..N\}$
c_i
:
B memory Banks $(B > 1/\mu)$
Update request queues Q_k

(a) Basic Architecture.

(b) Extended architecture to handle adversaries.

Figure 1: Memory architecture for a randomized DRAM-based counter scheme.

the advantage that it can support "active" counter applications in which individual counter values need to be retrieved at wirespeed. For such applications, this approach provides a much more efficient representation than a naive SRAM implementation.

In all three above SRAM-based approaches, significant amounts of SRAM are still necessary for very large counter arrays (say for tens of millions of counters). In contrast, our proposed solution stores all counters only in DRAM. We believe these approaches are complementary as they have different design tradeoffs.

Finally, the idea of using DRAM interleaving to implement large counter arrays was first proposed in [12]. We extend that work considerably in this paper by presenting a new mathematical framework for analyzing the behavior of such architectures under general practical conditions, as detailed in Section 4. We also introduced a cache module in the architecture to combat adversarial counter update patterns, which adds considerably to the difficulty of our analysis.

3. OUR SCHEME

Memory interleaving has been successfully used in the past for improving the performance of computer systems [13, 20, 22], for graphics or video intensive applications [29], and for implementing routing functions like high-performance packet buffers [26]. In this section, we describe how this technique can be employed for statistics counting.

Figure 1(a) depicts a simplified basic version of our randomized counter architecture. Given a SRAM-to-DRAM random access latency ratio of μ (e.g. $\mu = 4\text{ns}/64\text{ns} = 1/16$), we use $B > 1/\mu$ memory banks to store the counters. The basic idea is to randomly distribute the counters evenly across the B memory banks so that with high probability each memory bank will receive about one out of B counter updates to it on average. This is achieved by applying a pseudorandom permutation function $\pi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$ to a counter index to obtain a permuted index. We then use a simple location scheme where counter c_i will be stored in the k^{th} memory bank, where $k = \pi(i) \mod B$, at address location $a = \lfloor \pi(i)/B \rfloor$.

At each memory bank k, we maintain a small update request queue Q_k of pending update requests. To update counter c_i that is stored in the k^{th} memory bank, an update request is *inserted* into Q_k. These request queues are then conceptually serviced concurrently. The actual *read* and *write* operations are serviced alternately at the time slot level. In particular, at each memory bank k, suppose a read operation is initiated at time s for the counter request at the head of Q_k. The data will be available $1/\mu$ time slots later. Then a write operation for the incremented counter value can be initiated at time $s + 1/\mu$, which will be finished by time $s + 2/\mu$. Like in [31], we define a *cycle* as two time slots, the equivalent time for reading from SRAM (one time slot) and for writing back to SRAM (another time slot). Although an update would actually take $1/\mu$ cycles ($2/\mu$ time slots) to complete for performing both a DRAM read as well as a DRAM write, our design can effectively keep count of new packet arrivals as long as $B > 1/\mu$, which enables wirespeed throughput. That is, by randomly load-balancing incoming counter updates to B request queues, the arrival rate of new requests to each request queue is just once every B cycles, but the request queues are serviced at the faster rate of once every $1/\mu$ cycles.

Counter index permutation in our scheme makes it difficult for an adversary to purposely trigger a large number of consecutive counter updates to the same memory bank with updates to distinct counters since the pseudorandom permutation function (or the key it uses) is *not* known to the outside world. An adversary can only try to trigger consecutive counter updates to the same counter, which would result in consecutive accesses to the same memory bank. To safeguard our scheme against this adversarial situation, we add a fully associate cache module (say containing C cache entries) to our architecture to absorb such repetitions, as shown in Figure 1(b). This cache employs a FIFO replacement policy because (1) only FIFO allows us to study the worst-case performance of this system analytically and (2) it has long been proved in the adversarial paging literature that fancy policies such as LRU will not outperform FIFO under adversarial conditions [17, Chapter 13]. With the addition of the cache module, we can catch repeated updates to the same counter within a sliding window of C cycles. That is, if a new counter update request arrives for counter c_i, we can lookup the cache to see if there is already a pending update request to this counter. If there is, then we can just simply modify that request rather than creating a new one (e.g. change the request from "+1" to "+2"). Since these two updates will result in only one (instead of two) eventual DRAM access (read and write), there is no incentive (toward degrading the performance of our system) for an adversary to access the same counter repeatedly within a sliding window of C cycles.

In the next section (Section 4), we present a detailed theoretical analysis for all counter update sequences for the architecture depicted in Figure 1(b).

4. ANALYSIS

In this section, we prove the main theoretical result of this paper, which bounds the probability of having a long queueing delay at any aforementioned update request queue Q_k (associated with the k_{th} DRAM bank) for all (including any adversarial) counter update sequences. As explained earlier, this worst-case large deviation result is proven using a novel combination of convex ordering and (traditional) large deviation theory.

The main idea of our proof is as follows. Given any arbitrary counter update sequence over a time period $[s,t]$ (viewed as a parameter setting), we are able to obtain a tight stochastic bound of the number of arrivals of counter update requests to a DRAM bank during $[s,t]$ using various tail bound techniques. Since our scheme has to work with all possible counter update sequences, our bound clearly has to be the worst case (i.e. the maximum) stochastic bound over all of them. However, the space of all such sequences is so large that enumeration over all of them is computationally prohibitive and low complexity optimization procedures in finding the worst case does not seem to exist. Fortunately, we discover that the aforementioned number of arrivals under all these counter update sequences are dominated by that under a particular (i.e., worst-case) counter update sequence, in the convex order (not in the stochastic order). Since $e^{\theta x}$ is a convex function, we are able to upper-bound the MGFs of the number of arrivals under all other counter update sequences by that under the worst-case update sequence. The final tail bound is obtained by simply applying the Chernoff bound to this worst-case MGF. However we will show this worst-case MGF is prohibitively expensive to compute, let alone applying Chernoff technique to it. We solve this problem through upper-bounding this MGF by a computationally friendly formula.

We introduced in the previous section the concept of a *cycle*, which consists of an SRAM read time slot and an SRAM write time slot. Throughout the following analysis, we will use cycle as our basic unit of time. As explained in the previous section, we assume the system is continuously 100% loaded – that is, there is one incoming counter update every cycle. We refer to this worst-case workload as an arrival rate of 1. It is intuitive that this assumption indeed represents the worst-case, in the sense that the probability bounds derived for this case will be no better than allowing certain cycles to be idle. We omit the proof of this fact here since it would be a tedious application of the elementary stochastic ordering theory [24].

The rest of this section is organized as follows. In Section 4.1, we describe the overall structure of the tail bound problem, which shows that the overall overflow event $\tilde{D}$ over time period $[0,n]$ is the union of a set of the overflow events $D_{s,t}$, $0 \le s < t \le n$, which leads to a union bound. In the next two sections we show how to bound each individual event $D_{s,t}$ using the aforementioned convex ordering technique. In Section 4.2.2, we establish the worst-case number of update requests $X_{s,t}$ during time interval $[s,t]$, in terms of convex order. In Section 4.2.3 we bound $X_{s,t}$ with the sum of i.i.d. random variable which can be easily computed.

4.1 Union bound – the first step

In this section we bound the probability of overflowing a request queue Q. Let $\tilde{D}_{0,n}$ be the event that one or more requests are dropped because Q is full during time interval $[0,n]$ (in units of cycles). This bound will be established as a function of system parameters K, B, μ, and C. Recall that K is the size of each request queue, B is the number of DRAM banks, μ is the SRAM-to-DRAM random access latency ratio, C is the size of the cache.

In the following, we shall fix n and will therefore shorten $\tilde{D}_{0,n}$ to $\tilde{D}$. Note that $\Pr[\tilde{D}]$ is the overflow probability for just one out of B such queues. The overall overflow probability can be bounded by $B \times \Pr[\tilde{D}]$ (union bound).

We first show that $\Pr[\tilde{D}]$ is bounded by the summation of probabilities $\Pr[D_{s,t}]$, $0 \le s \le t \le n$, that is,

$$\Pr[\tilde{D}] \le \sum_{0 \le s \le t \le n} \Pr[D_{s,t}]. \qquad (1)$$

Here $D_{s,t}$, $0 \le s < t \le n$, represents the event that the number of arrivals during the time interval $[s,t]$ is larger than the maximum possible number of departures in the queue, by more than the queue size K. Formally letting $X_{s,t}$ denote the number of update requests (to the DRAM bank) generated during time interval $[s,t)$, then we have

$$D_{s,t} \equiv \{\omega \in \Omega : X_{s,t} - \mu(t-s) > K\}.$$

Here we will say a few words about the implicit probability space Ω, which is the set of all permutations on $\{1,...,N\}$. Since we are considering the worst case bound, we assume that for each cycle there is a request dequeued from the cache, thus creating an arrival for one of the request queues for DRAM banks. We assume that the requests dequeued follow arbitrary pattern, with the only restriction that the same requested address can not repeat within C cycles. This is due to the "smoothing" effect of the cache, i.e. repetitions within C cycles would be absorbed by the cache. Given an arbitrary dequeued pattern satisfying the above restriction, then each instance $\omega \in \Omega$ gives us an arrival sequence to the queues of the DRAM bank.

The inequality (1) is a direct consequence (through the union bound) of the following lemma, which states that if the event $\tilde{D}$ happens, at least one of the events $\{D_{s,t}\}_{0 \le s < t \le n}$ must happen, and vice versa.

LEMMA 1. $\tilde{D} = \bigcup_{0 \le s \le t \le n} D_{s,t}$

PROOF. Given an outcome $\omega \in \tilde{D}$, suppose an overflow happens at time z. The queue is clearly in the middle of a busy period at time z. Now suppose this busy period starts at y. Then the number of departures from y to z is equal to $\lfloor \mu(z-y) \rfloor$. Since an update request happens at time z to find the queue of size K full, $X_{y,z}$, the total number of arrivals during time $[y,z]$ is at least $K+1+\lfloor \mu(z-y) \rfloor \ge K+\mu(z-y)$. In other words, $D_{y,z}$ happens and $\omega \in D_{y,z}$. This means for any outcome ω in the probability space, if $\omega \in \tilde{D}$, then $\omega \in D_{s,t}$ for some $0 \le s < t \le n$.

On the other hand, given an outcome $\omega \in D_{s,t}$ for some s,t, then obviously the queue will overflow at t or earlier, so $\omega \in \tilde{D}$. $\square$

Remark: A similar lemma is proved in [31, Lemma 1]. Here we point out the stronger relationship of equivalence.

4.2 Bounding individual $\Pr[D_{s,t}]$

In this subsection we find the worst-case update request sequence for deriving tail bounds for individual $\Pr[D_{s,t}]$ terms. Recall that $D_{s,t}$ is the event that the number of arrivals during the time interval $[s,t]$, denoted as $X_{s,t}$, is larger than the maximum possible number of departures in the queue, by more than the queue size K. The probability $\Pr[D_{s,t}]$ is clearly a (random) function of the sequence of update requests (viewed as parameters) during the interval $[s,t]$. Fixing any arbitrary sequence, it is not hard to bound $\Pr[D_{s,t}]$ using Chernoff type of techniques, as $X_{s,t}$ can be bound by the sum of independent random variables, in the convex order, using the techniques in Section 4.2.3. However, as mentioned before, it is not possible to enumerate over all possible parameter settings (i.e., sequences) to find the worst-case $\Pr[D_{s,t}]$ bound. Fortunately, convex ordering comes to our rescue by allowing us to analytically bound the moment generating function (MGF) of $X_{s,t}$ under all parameter settings by that under a worst-case setting. For simplicity, in this section we will drop the subscripts of $X_{s,t}$ and use X instead.

4.2.1 Mathematical preliminaries

In the following, we first describe the standard Chernoff method for obtaining sharp tail bounds from the MGF of a random variable (in this case X).

$$\begin{aligned}\Pr[D_{s,t}] &= \Pr[X > K + \mu\tau] \\ &= \Pr[e^{X\theta} > e^{(K+\mu\tau)\theta}] \\ &\le \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}}.\end{aligned}$$

where $\theta > 0$ is any constant, and the last step is due to Markov inequality. Here τ is defined as $t-s$.

Since this is true for all θ, we have

$$\Pr[D_{s,t}] \le \min_{\theta>0} \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}}. \tag{2}$$

Then, we aim to bound the moment generating function (MGF) $E[e^{X\theta}]$ by finding the worst-case sequence. Note that we resort to convex ordering, because *stochastic order*, which is the conventional technique to establish ordering between random variables and is stronger than *convex order*, does not hold here, as we will show shortly.

Since convex ordering techniques and related concepts are needed to establish the bound, we first present the definition of majorization, exchangeable random variables, convex function, and convex ordering as follows:

Definition 1 (Majorization [15, 1.A.1]). *For any n-dimensional vectors a and b, let $a_{[1]} \ge \ldots \ge a_{[n]}$ denote the components of a in decreasing order, and $b_{[1]} \ge \ldots \ge b_{[n]}$ denote the components of b in decreasing order. We say a is* majorized *by b, denoted $a \le_M b$, if*

$$\begin{cases}\sum_{i=1}^{k} a_{[i]} \le \sum_{i=1}^{k} b_{[i]}, & \text{for } k = 1,\ldots,n-1 \\ \sum_{i=1}^{n} a_{[i]} = \sum_{i=1}^{n} b_{[i]} \end{cases} \tag{3}$$

Definition 2 (Exchangeable random variables). *A sequence of random variables $X_1,\ldots,X_n$ is called* exchangeable*, if for any permutation $\sigma : [1,\ldots,n] \to [1,\ldots,n]$, the joint probability distribution of the permuted sequence $X_{\sigma(1)},\ldots,X_{\sigma(n)}$ is the same as the joint probability distribution of the original sequence.*

For example, a sequence of independent and identically distributed random variables are exchangeable. Another example is a sequence of random variables resulting from sampling without replacement.

Definition 3 (Convex function). *A real function f is called* convex*, if $f(\alpha x + (1-\alpha)y) \le \alpha f(x) + (1-\alpha) f(y)$ for all x and y and all $0 < \alpha < 1$.*

Definition 4 (Convex order [18, 1.5.1]). *Let X and Y be random variables with finite means. Then we say that X is less than Y in convex order (written $X \le_{cx} Y$), if $E[f(X)] \le E[f(Y)]$ holds for all real convex functions f such that the expectations exist.*

Since the MGF ($\mathrm{E}[e^{X\theta}]$) is expectation of a convex function ($e^{x\theta}$) of X, establishing convex order will help to bound the MGF.

The following theorem from Marshall [15] relates majorization, exchangeable random variables and convex order together. It is restated here in the language of convex ordering. It is a special case of a more general theorem [15, 11.B.2].

Theorem 1 ([15, 11.B.2.c]). *If $X_1,\ldots,X_n$ are exchangeable random variables, a and b are n-dimensional vectors, then $a \le_M b$ implies $\sum_{i=1}^{n} a_i X_i \le_{cx} \sum_{i=1}^{n} b_i X_i$.*

Finally, we define *stochastic order*, the lack of which in our case leads us to resort to the convex ordering.

Definition 5 (Stochastic order [18, 1.2.1]). *The random variable X is said to be smaller than the random variable Y in stochastic order (written $X \le_{st} Y$), if $\Pr[X > t] \le \Pr[Y > t]$ for all real t.*

4.2.2 Worst-case update request sequence

In this section, we specify the worst-case update request sequence (in the aforementioned sense of convex ordering) and prove it is indeed the worst-case.

Let $X_i, 1 \le i \le N$ be the indicator random variable for whether the i^{th} address is mapped to the DRAM bank under consideration. We have $E[X_i] = \frac{1}{B}$. Thanks to the random counter index permutation scheme, we can view X_i's as a result of sampling without replacement from N values, $\frac{N}{B}$ of which are 1 and the rest are 0. Therefore the X_i's are exchangeable random variables, though they are not independent.

Let $m_i, 1 \le i \le N$ be the count of the number of appearances of the i^{th} address during time interval $[s,t]$. Then $X = \sum_{i=1}^{N} m_i X_i$.

Due to caching, the dequeued requests to the same DRAM address should not repeat within any sliding window of C cycles. Therefore none of the counts $m_1,...,m_N$ can exceed T, where $T = \lceil \frac{\tau}{C} \rceil$. Moreover, let $q = \tau - (T-1)C$ and $r = C - q$. Then only the first q requests could repeat with count T. Figure 2 will help readers understand the relationship among q, r, T as functions of τ.

We call any vector $m = \{m_1,...,m_N\}$ a valid splitting pattern of τ, if $0 \le m_i \le T$, $\sum_{i=1}^{N} m_i = \tau$, and $|\{i : m_i = T\}| \le q$. Let $\mathcal{M}$ be the set of all valid splitting patterns.[3] Let X_m denote $\sum_{i=1}^{N} m_i X_i$.

[3]It is possible to prove that for every valid splitting pattern, there is a possible counter update sequence matching the pattern. However this is not essential for our analysis.

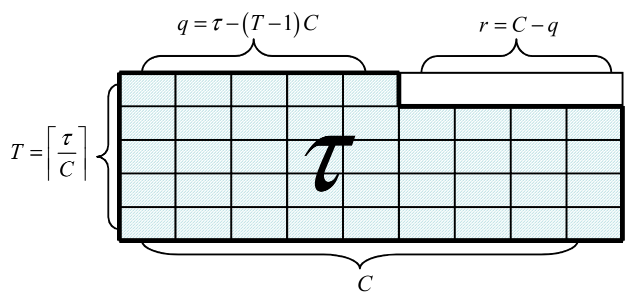


Figure 2: Relationship among q, r, T and τ

We are ready to specify the family of worst-case counter update sequences. A worst-case counter update sequence takes the following form: first come $q+r(=C)$ update requests for distinct counter indices $a_1, a_2, ..., a_{q+r}$, and they then repeat for $T-1$ times in total, followed finally by q update requests for counter indices $a_1, a_2, ..., a_q$. In other words, inside this window of τ cycles, counters $a_1, a_2, ..., a_q$ (q of them) are accessed T times and counters $a_{q+1}, ..., a_{q+r}$ (r of them) are accessed $T-1$ times. In the following Theorem 2, we will see that this arrival sequence is indeed the worst-case in the sense of convex ordering.

Let m^* be the aforementioned pattern for one of such counter update sequences, i.e. let $m^*_1 = \ldots = m^*_q = T, m^*_{q+1} = \ldots = m^*_{q+r} = T-1, m^*_{q+r+1} = \ldots = m^*_N = 0$.

We now have the main theorem of this section:

THEOREM 2. *m^* is the worst case splitting pattern in terms of convex ordering, i.e. $X_m \le_{cx} X_{m^*}, \forall m \in \mathcal{M}$.*

PROOF. Let $m_{[1]}, \ldots, m_{[N]}$ denote the components of m in decreasing order. m^* is already in decreasing order. Because m is a valid splitting pattern, we have

$$\begin{array}{lcl} m_{[i]} \le T = m^*_i & , & \text{for } 1 \le i \le q \\ m_{[i]} \le T-1 = m^*_i & , & \text{for } q+1 \le i \le q+r. \end{array}$$

Therefore (3) is true for $1 \le k \le q+r$. Since $\sum_{i=1}^{q+r} m^*_i = \tau = \sum_{i=1}^{N} m_i$, (3) is true for $i > q+r$ as well. Therefore by definition $m \le_M m^*$.

The theorem follows from Theorem 1 because $X_1, \ldots, X_n$ are exchangeable. □

Remark: Note that stochastic order does not hold here in general, since $\mathrm{E}[X_m] = \mathrm{E}[X_{m^*}] = \tau/B, \forall m \in \mathcal{M}$. For stochastic order to hold between two random variables of different distributions, their expectations must differ [18, Theorem 1.2.9].

Unfortunately, it is in general not possible to apply the Chernoff bound directly to the MGF of X_{m^*}. For $\tau \le C$, X_{m^*} is a hypergeometric random variable whose MGF $\mathrm{E}[e^{X_{m^*}\theta}]$ is a hypergeometric series [6] that is expensive to compute. For $\tau > C$, X_{m^*} is a weighted sum of two hypergeometric random variables that are not independent of each other, so its MGF is prohibitively expensive to compute. The next section is devoted to dealing with this problem.

4.2.3 *Relaxations of X_{m^*} for computational purposes*

Our remaining task is to find a way to upper-bound $\mathrm{E}[e^{X_{m^*}\theta}]$ by a more computationally friendly formula, to which the aforementioned Chernoff technique can be applied. The following lemma by Hoeffding, which bounds the outcome of sampling without replacement by that with replacement in the convex order, will be used to accomplish this task.

LEMMA 2 ([8, THEOREM 4]). *Let the population S consist of N values $c_1, ..., c_N$. Let $X_1, ..., X_n$ denote a random sample without replacement from S and let $Y_1, ..., Y_n$ denote a random sample with replacement from S. Let $X = X_1 + ... + X_n$, $Y = Y_1 + ... + Y_n$. Then $X \le_{cx} Y$.*

We consider the two cases $\tau \le C$ (i.e., the measurement window τ, in number of cycles, is no larger than the cache size, in number of entries) and $\tau > C$ separately.

The case $\tau \le C$

When $\tau \le C$, $X_{m^*} = X_1 + ... + X_\tau$. Let the population S consist of $c_1, ...c_N$ such that $\frac{N}{B}$ of c_i's are of value 1 and the rest of them are of value 0. If we let $n = \tau$, then X in Lemma 2 has the same distribution as our X_{m^*}, and Y_i's there are i.i.d Bernoulli random variables with probability $\frac{1}{B}$. Because $f(x) = e^{x\theta}$ is a convex function of x, from $X_m \le_{cx} X_{m^*} \le_{cx} Y$ we get

$$\begin{array}{rcl} \mathrm{E}[e^{X_m\theta}] & \le & \mathrm{E}[e^{X_{m^*}\theta}] \le \mathrm{E}[e^{Y\theta}] \\ & = & \mathrm{E}[e^{(Y_1+\ldots+Y_\tau)\theta}] \\ & = & \mathrm{E}[e^{Y_1\theta}]^\tau \\ & = & (\frac{1}{B}e^\theta + (1-\frac{1}{B}))^\tau. \end{array}$$

By (2), we now have the following bound:

THEOREM 3.

$$\Pr[D_{s,t}] \le \min_{\theta>0} \frac{(\frac{1}{B}e^\theta + (1-\frac{1}{B}))^\tau}{e^{(K+\mu\tau)\theta}}, \quad \tau \le C.$$

The case $\tau > C$

For $\tau > C$, we have $X_{m^*} = T(X_1 + ... + X_q) + (T-1)(X_{s+1} + ... + X_{q+r})$. We cannot apply Lemma 2 in the same way as for the case $\tau \le C$. However we could take a different viewpoint. Instead of selecting $C = q+r$ permutation destinations for the addresses and see which ones are among the $\frac{N}{B}$ locations in the DRAM bank, we can treat it as selecting $\frac{N}{B}$ permutation sources, and see which ones are among the addresses that have requested update. So let the population S' be exactly the components of m^*, i.e. let $c_i = m{*}_i$. So there are q of c_i's of value T, r of c_i's of value $T-1$, and the rest of them of value 0. Thus $X_{m^*} = X'_1 + ... + X'_{\frac{N}{B}}$, where X'_i's are a random sample without replacement from S'. By Lemma 2 we have $X_{m^*} \le_{cx} Y' = \sum_{i=1}^{N/B} Y'_i$, where Y'_i's are a random sample with replacement from S'. Thus the Y'_i's are i.i.d. random variable with

$$\Pr[Y'_i = y] = \begin{cases} \frac{q}{N} & y = T \\ \frac{r}{N} & y = T-1 \\ \frac{N-q-r}{N} & y = 0 \end{cases}.$$

So similar to the $\tau \le C$ case, we have

$$
\begin{aligned}
\mathrm{E}[e^{X_m\theta}] &\le \mathrm{E}[e^{X_{m*}\theta}] \le \mathrm{E}[e^{Y'\theta}] \\
&= \mathrm{E}[e^{(Y'_1+\ldots+Y'_{\frac{N}{B}})\theta}] \\
&= \mathrm{E}[e^{Y'_1\theta}]^{\frac{N}{B}} \\
&= \left(\frac{q}{N}e^{T\theta} + \frac{r}{N}e^{(T-1)\theta} + \frac{N-q-r}{N}\right)^{\frac{N}{B}} \\
&= \left(1 + \frac{qe^{T\theta} + re^{(T-1)\theta} - q - r}{N}\right)^{\frac{N}{B}} \\
&\le e^{(qe^{T\theta} + re^{(T-1)\theta} - q - r)/B}.
\end{aligned}
$$

For the last inequality we used the inequality $(1+\frac{a}{n})^n < e^a$. By (2), we now have the following bound (it's not hard to verify that it also applies to $\tau \le C$) :

THEOREM 4.

$$\Pr[D_{s,t}] \quad \le \min_{\theta > 0} e^{(qe^{T\theta} + re^{(T-1)\theta} - q - r)/B - (K + \mu\tau)\theta}.$$

Remark: Although Theorem 4 also applies to the case $\tau \le C$, Theorem 3 gives better bound in our problem setting.

We also see that the bound is translation invariant, i.e. it only depends on $\tau = t - s$. Therefore, the computation cost of (1) is $O(n)$ instead of $O(n^2)$, where n is the number of cycles during a network measurement interval.

In conclusion, we have established bound for overflow probability for worst-case counter update request sequences. The bound can be computed though $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 3 and Theorem 4.

5. EVALUATION

In this section, we present evaluation results for our proposed randomized DRAM-based counter array architecture described in Section 3. The outline of this section is as follows: Section 5.1 describes a concrete instantiation of our proposed solution. Section 5.2 outlines the parameters of two real-world Internet traffic traces that we used for our evaluations. Section 5.3 presents numerical results derived using the analytical models presented in Section 4. Finally, Section 5.4 provides a comparison of our new approach with the state-of-the-art hybrid SRAM/DRAM approach [31].

5.1 Implementation details

To provide a general formulation, we had assumed in Section 3 that the request queues are conceptually serviced concurrently. This could be realized by using B separate parallel memory channels to B separate sets of memories. However, this is unnecessarily expensive as modern DRAM architectures already provide a plentiful number of internal memory banks, and a single memory channel can be used to pipeline memory transactions across them at peak rates when performed in an interleaving manner.

To provide a concrete analysis of our proposed solution, we use the specification of an actual commercial high-bandwidth memory part, namely the XDR memory from Rambus [29, 2]. As depicted in Figure 3, each XDR memory chip contains 16 internal memory banks. Although the XDR memory has a worst-case access latency of 40 ns for a read or a write operation, a new read or write transaction could be initiated every 4 ns if it is initiated to a different memory bank. For an OC-768 link at 40 Gb/s, a new minimum size (40 bytes) packet can arrive every 8 ns. To support a counter update on every packet arrival, about 4 ns is available for a memory read or a memory write. Fortunately, the XDR memory can support this rate of new memory operations.

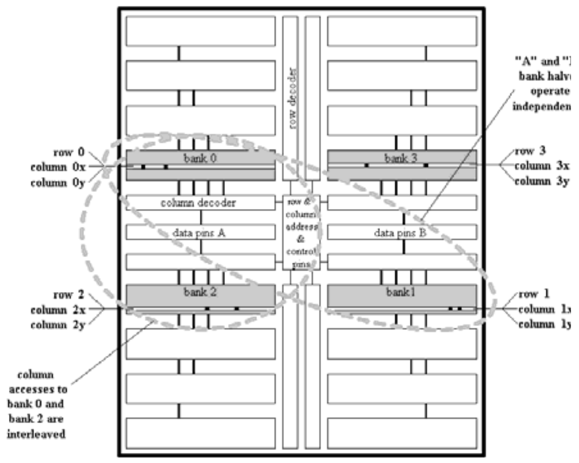


Figure 3: An example high-bandwidth DRAM architecture. Each XDR memory IC has 16 internal memory banks that can be interleaved to achieve high-bandwidth memory access.

In particular, one concrete implementation is to use $B = 32$ memory banks and two memory channels, with 16 banks on each memory channel. For each memory channel, we can service its memory banks in round-robin order. Therefore, a new memory operation can be serviced once every 16 cycles. With both memory channels operating in parallel, each of the $B = 32$ memory banks can indeed be serviced deterministically once every 16 cycles. Fortunately, processors with dual memory channels are becoming increasingly common. For example, both the Cell processor from IBM/Sony/Toshiba [7] and the latest mainstream Intel x86 multi-core processor [1] have built-in dual-channel memory controllers. This configuration corresponds to setting $\mu = 1/16$ in our analysis, and it can handle any SRAM-to-DRAM speed ratio that is no smaller than 1/16. For the remainder of this section (Section 5), we will use the configuration $\mu = 1/16$ and $B = 32$ in our analysis and comparisons.

5.2 Traffic traces

For our evaluations, we used parameters derived from two real-world Internet traffic traces. In particular, 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 the campus to the rest of the Internet on April 24, 2003. The trace from USC has 120.8 million packets and around 8.6 million flows, and the trace segment from UNC has 198.9 million packets and around 13.5 million flows. To support sufficient counters for both traces, we set the counter array configuration to support $N = 16$ million counters.

5.3 Numerical examples of the tail bounds

In this section, we present the numerical results computed from the formulae derived in Section 4 using MATLAB 7.0. We use $n = 10^8$ for all the following examples, where n is the total number of cycles for the measurement period.

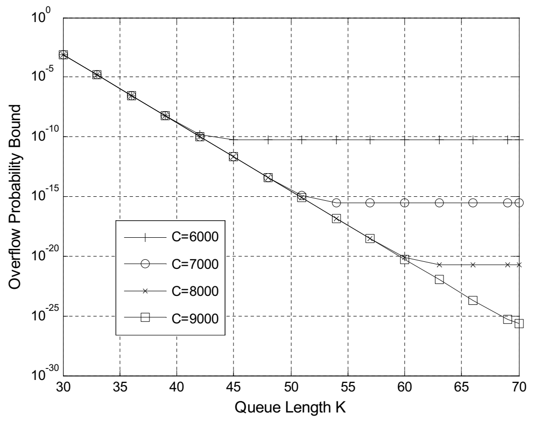


Figure 4: Overflow probability bound as a function of queue size K with $\mu = 1/16$ and $B = 32$.

In Figure 4, the overflow probability bounds with different cache size C as a function of queue length K are presented, where $\mu = 1/16$ and $B = 32$. It is easy to see from this graph that as K increases, the overflow probability bound decreases. However, after K reaches certain thresholds (depending on C), the overflow probability bound stays practically flat. The flat level depends on C. Actually, as C approaches infinity, the overflow probability as a function of queue length K becomes the result of the overflow probability of a Geom/D/1 incoming traffic with arrival probability $1/B = 1/32$ to a queue of length K. As shown in Figure 4, when $C \geq 8000$, the overflow probability bound has only negligible decreases as cache size C increases. In other words, by increasing the size of the cache, the performance of the system will not be improved accordingly. Therefore, $C = 8000$ is enough for practical purposes in achieving an overflow probability bound of 10^{-14}.

5.4 Cost-benefit comparison

Table 1 compares our proposed approach with a naive SRAM approach as well as the hybrid SRAM/DRAM counter architecture approaches [25, 21, 23, 31]. The naive SRAM approach simply implements all counters in SRAM. For the hybrid SRAM/DRAM approach, we specifically compare against the state-of-the scheme proposed by Zhao et al. [31] that provably achieves the minimum SRAM requirement for this architecture class. As demonstrated in [31], in one example setting their approach requires almost a factor of six times less SRAM than the first hybrid SRAM/DRAM solution proposed in [25] and more than a factor of two times less SRAM than an improved solution proposed in [21]. For a SRAM-to-DRAM latency ratio of $\mu = 1/12$, the architecture in [31] requires $w = \lceil \log 1/\mu \rceil = \lceil \log 12 \rceil = 4$ bits SRAM bits per counter. In addition, it needs a very small amount of SRAM to maintain a "flush request queue", on the order of about 500 entries to ensure negligible overflow probabilities. We will compare it with our scheme in Section 5.1, since that scheme can handle $\mu = 1/12 \geq 1/16$.

For 16 million counters, a naive implementation would require 128 MB of SRAM, which is clearly far too expensive. For the scheme by Zhao et al., it just requires 1.5 KB of control SRAM to implement a flush request queue with 500 entries. The size of each entry is $\lceil \log 16 \text{ million} \rceil = 24$ bits (3 bytes) to encode the counter index. However, even though the scheme requires just 4 bits per counter to store the partial increments, 8 MB of counter SRAM is required for 16 million counters. This is a substantial amount and difficult to implement on-chip. Moreover, this counter SRAM requirement grows linearly with the number of counters, making it difficult to support faster links or longer measurement periods where more counters would be needed.

For our proposed solution, a small update request queue needs to be maintained at each memory bank. As shown in Figure 4, when we use $B = 32$ memory banks, $K = 50$ entries is sufficient for each update request queue to ensure a queue overflow probability bound of 10^{-14}, for a total of $M = B \cdot K = 1600$ entries. Each entry requires 3 bytes to encode the counter indices of 16 million counters and 4 bits for accumulated counts, resulting in a total of about 5.5 KB of SRAM to implement these update request queues. Since these update request queues are statically sized, they can be simply implemented as an array.

In addition, as shown in Figure 1(b) in Section 3, our randomized counter architecture also maintains a small cache to keep track of pending update requests. This cache can be implemented as a fully-associative cache using a content-addressable memory (CAM) with a FIFO replacement policy. For $C = 7000$, we need a CAM with 25 KB in size to support 3 bytes encoding of counter indices and 4 bits for accumulated counts[4]. Although our comparisons here are for integer counters and increments of one only, we emphasize that our general scheme supports increments and decrements of arbitrary amounts, and other number representations such as floating point numbers.

6. CONCLUSION

In this paper, we have addressed the problem of maintaining a large array of exact statistics counters that needs to be updated at very high speeds. We proposed a DRAM-based counter architecture that can effectively maintain wirespeed updates to large counter arrays exploiting advanced architecture features that are readily available in modern commodity DRAM architectures. In particular, we presented a randomized DRAM architecture that can harness the performance of modern commodity DRAM offerings by interleaving counter updates to multiple memory banks. We presented a rigorous theoretical analysis on the performance of our proposed architecture in the worst-case using a novel combination of convex ordering and large deviation theory. Our analysis provides considerations for adversarial counter update patterns. Among the salient features of our proposed DRAM-based counter architecture is its ability to support arbitrary increments and decrements at wirespeed as well

[4]An accumulation counter of 4 bits can absorb 16 increments to the same counter into one update request, which can be serviced in the time of 16 increments, thus offering an adversary no advantage in repeatedly hitting the same counter within a sliding window of C cycles.

	Naive	Hybrid SRAM/DRAM [31]	Ours
Counter DRAM	None	128 MB DRAM	128 MB DRAM
Counter SRAM	128 MB SRAM	8 MB SRAM	None
Control	None	1.5 KB SRAM	25 KB CAM and 5.5 KB SRAM

Table 1: Comparison of different schemes for a reference configuration with 16 million 64-bit counters. For our method, $K = 50$, $B = 32$, and $C = 7000$.

as support different number representations, including both integer and floating point number representations.

7. REFERENCES

[1] Intel Lynnfield processor.

[2] XDR datasheet. *Rambus, Inc.*, 2002-2003.

[3] XDR-2 datasheet. *Rambus, Inc.*, 2004-2005.

[4] Intel IXP 2855 network processor product brief. *Intel Corporation*, 2005.

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

[6] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 2nd edition, 1994.

[7] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[8] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[9] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *IEEE HPCA*, page 80, 1999.

[10] N. Hua, B. Lin, J. Xu, and H. Zhao. BRICK: A novel exact active statistics counter architecture. In *ACM/IEEE ANCS*, 2008.

[11] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. In *IEEE FOCS*, 2000.

[12] B. Lin and J. Xu. DRAM is plenty fast for wirespeed statistics counting. In *ACM HotMetrics*, June 2008.

[13] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proc. of IEEE HPCA*, page 301, Washington, DC, USA, 2001.

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

[15] A. W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications.* Academic Press, 1979.

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

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

[18] A. Muller and D. Stoyan. *Comparison Methods for Stochastic Models and Risks.* Wiley, 2002.

[19] C. Pandita and S. Meyn. Worst-case large-deviation asymptotics with application to queueing and information theory. *Stochastic Processes and their Applications*, 116(5):724–756, 2006.

[20] D. Patterson and J. Hennessy. *Computer Architecture: A QuantitativeApproach.* Morgan Kaufmann, 2nd edition, 1996.

[21] S. Ramabhadran and G. Varghese. Efficient implementation of a statistics counter architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(1):261–271, 2003.

[22] B. R. Rau. Pseudo-randomly interleaved memory. In *Proc. 18th Annual International Symposium on Computer Architecture*, 1991.

[23] M. Roeder and B. Lin. Maintaining exact statistics counters with a multi-level counter memory. In *IEEE GLOBECOM*, volume 2, pages 576–581, Nov - Dec 2004.

[24] S. M. Ross. *Low-Density Parity-Check Codes.* Wiley, 2nd Edition, 1995.

[25] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown. Maintaining statistics counters in router line cards. *Micro, IEEE*, 22(1):76–81, Jan/Feb 2002.

[26] G. Shrimali and N. McKeown. Building packet buffers using interleaved memories. In *Workshop on High Performance Switching and Routing (HPSR)*, May 2005.

[27] R. Stanojevic. Small active counters. In *IEEE Infocom*, 2007.

[28] F. Ware and C. Hampel. Improving power and data efficiency with threaded memory modules. In *International Conference on Computer Design (ICCD)*, pages 417–424, Oct. 2006.

[29] F. A. Ware and C. Hampel. Micro-threaded row and column operations in a dram core. *Rambus White Paper*, Mar 2005.

[30] H. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu. A data streaming algorithm for estimating entropies of OD flows. In *ACM IMC*, 2007.

[31] Q. Zhao, J. Xu, and Z. Liu. Design of a novel statistics counter architecture with optimal space and time efficiency. *SIGMETRICS Perform. Eval. Rev.*, 34(1):323–334, 2006.