DRAM-Based Statistics Counter Array Architecture With Performance Guarantee

Hao Wang, *Student Member, IEEE*, Haiquan (Chuck) Zhao, Bill Lin, *Member, IEEE*, and Jun (Jim) Xu, *Senior Member, IEEE*

***Abstract*—The problem of efficiently maintaining 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 line rates. In particular, 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, such as 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 randomized 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 they can support different number representations, including both integer and floating point number representations.**

***Index Terms*—Convex ordering, large deviation theory, majorization, network measurement, statistics counter.**

I. 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 [2], [3]). 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 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 frame. Large counters, such as 64 b 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, such as intrusion detection, and 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 [4], 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 b for each flow counter, 108 MB of SRAM would already be needed for just the counter storage, which is prohibitively expensive. Therefore, researchers have been actively seeking alternative ways to realize large arrays of statistics counters at wirespeed [2]–[11].

Several designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed [2]–[5]. Their basic idea is to store some lower-order bits (e.g., 9 b) of each counter in SRAM, and all its bits (e.g., 64 b) in DRAM. The increments are made only to these SRAM counters. When the values of SRAM counters come close to overflowing, they are scheduled to be "flushed" back to the corresponding DRAM counters. These schemes significantly reduce the SRAM cost. In particular, the scheme by Zhao *et al.* [4] achieves the theoretically minimum SRAM cost of 4–5 b per counters when the SRAM-to-DRAM access latency ratio is between 1/10 (4 ns/40 ns) and 1/20 (3 ns/60 ns). While this is a substantial reduction over a straightforward SRAM implementation, storing say 4 b per counter in SRAM for 13.5 million flows would still require nearly 7 MB of SRAM, which is a significant 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 a counter to overflow could

Manuscript received December 15, 2010; revised July 20, 2011; accepted October 02, 2011; approved by IEEE/ACM Transactions on Netwokring Editor I. Keslassy. Date of publication October 20, 2011; date of current version August 14, 2012. This work was presented in part at the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Princeton, NJ, October 19–20, 2009.

H. Wang and B. Lin are with the Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, CA 92093 USA (e-mail: wanghao@ucsd.edu; billlin@ucsd.edu).

H. Zhao was with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA. He is now with Microsoft, Redmond, WA 98052 USA (e-mail: chz@cc.gatech.edu).

J. Xu is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: jx@cc.gatech.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2011.2171360

occur much earlier, and more SRAM counter bits would be needed to compensate, resulting in yet larger SRAM requirements. On the other hand, in our counter array scheme, we only need to increase the storage for several thousand entries. To maintain 16 million active counters, the size increase in our scheme is less than the traditional SRAM/DRAM schemes by more than three orders of magnitude. 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 [12], [13] to maintain values such as the entropies of data streams.

A. 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 accesses 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 [14]–[17] that can be exploited to make a DRAM solution practical. We then propose DRAM-based counter architectures that allow for wirespeed updates to large counter arrays.

Motivated 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 continuously 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 [18] uses two 32-b channels of XDR memories [19] with an aggregated memory bandwidth of 25.6 GB/s. Using an approach called micro-threading [17], 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 [20] are expected to achieve a data rate upwards of 16 GB/s on a single 16-b channel, 64 GB/s on an equivalent dual 32-b 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 [21] 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 [16].

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 [19], 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.

B. Our Approach

Our main idea is to randomly distribute the memory addresses to which consecutive counter indices are mapped 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. Suppose the SRAM-to-DRAM random access latency ratio is μ, e.g., $\mu = 4$ ns/64 ns $= 1/16$. 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 scheme that is in use.

The randomized scheme does not require extra DRAM. It works by ensuring that the memory load is evenly distributed when different counters are updated over time, *under arbitrary counter update sequences*. However, 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. 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, a 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 IV). This result is a *worst-case large deviation theorem* in nature [22] 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 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 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. Since the exponent function is a convex function, we are able to dominate the moment generating function (MGF) of queueing delays, which is 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, to say on the order of $K = 45$ entries per request queue, are required to ensure a negligible overflow probability (say under 10^{-14}).

C. *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 it 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 randomized counter architecture in the worst-case as one of its applications.
- We present concrete evaluations of our proposed scheme using the XDR memory architecture [17], [19], [20], which has 16 internal independent memory banks in each memory chip, 256 memory banks across 16 memory chips in a single memory module, and we show that wirespeed performance can be achieved without the expense of multiple memory channels.
- Compared to existing hybrid SRAM/DRAM counter architectures [2]–[5], our randomized counter solution offers three clear advantages. First, our solution can achieve the same update speeds to counters, without the need for a nontrivial 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 [12], [13]), while hybrid SRAM/DRAM counter architectures typically can only accommodate "increment by 1" efficiently. Finally, as we shall show in Section V, 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 the low cost of DRAM.[1]

[1]As of this writing, 2 GB of DRAM costs under \$20, over 100 MB/dollar.

D. *Outline of Paper*

The rest of the paper is organized as follows. Section II outlines additional related work. Section III describes our proposed randomized counter architecture in details. Section IV provides a rigorous analysis on the performance of our randomized counter architecture under the worst-case memory access pattern. Section V presents evaluations of our proposed architecture. Finally, Section VI concludes the paper.

II. Related Work

In this section, we outline prior work related to our problem. As already discussed in Section I, the naive approach of storing full counters in SRAM is prohibitively expensive. Although a hybrid SRAM/DRAM architecture [2]–[5] 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 [6]–[8], which are all based on the basic idea invented by Morris [6]. Approximate counting can keep the cost low by sacrificing counter accuracy. The idea is to probabilistically increment a counter based on the current counter value. However, approximate counting in general has very large error margin when the number of bits per counter used is small, and possible estimation values are very sparsely distributed in the range of possible counts. Therefore, when the counter values are small, the estimation can cause very high relative errors (well over 100%). Thus, this approach is only applicable to those network measurement and data streaming applications that can tolerate the inaccuracies. It may not be acceptable for those network accounting and data streaming applications where small counter values are important for the overall measurement accuracy. Other such systems include the NetFlow from Cisco [23], the filter-based accounting from Juniper [24], and the sample-and-hold solution [25].

A second approach is a counter architecture called counter braids [9], 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. In a way, counter braids are "more passive" than SRAM/DRAM hybrid architectures. Flow counts can be decoded through an iterative decoding process at the end of the measurement epoch. The decoding process incurs significantly longer delay than a DRAM access, and it can be processed offline.

A third approach 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 the advantage that it can support "active"

[2]Counter braids consider a more general problem that also addresses flow association.

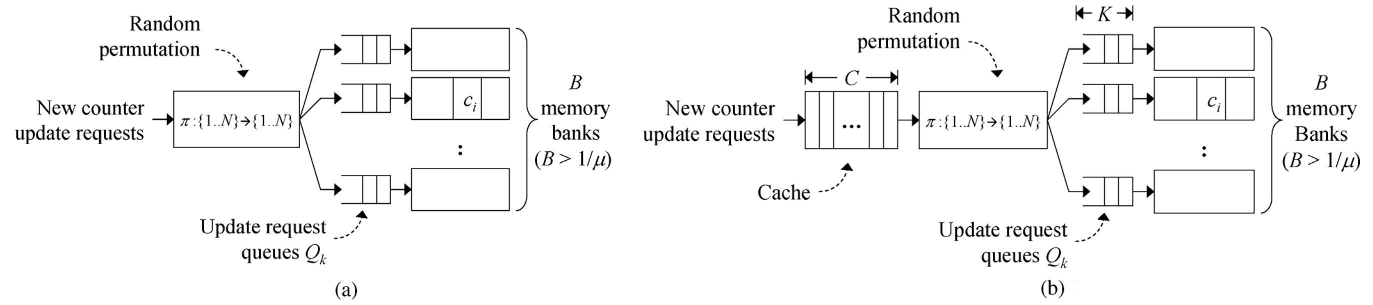

Random permutation
New counter update requests
$\pi:\{1..N\}\to\{1..N\}$
c_i
B memory banks ($B > 1/\mu$)
Update request queues Q_k
(a)
K
C
Cache
B memory Banks ($B > 1/\mu$)
(b)

Fig. 1. Memory architecture for randomized DRAM-based counter schemes. (a) Basic architecture. (b) Extended architecture to handle adversaries.

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 mentioned 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. It is worth noting that general memory systems supporting arbitrary memory read and write accesses [26], [27] are applicable for the maintenance of large counter arrays. However, they are not as efficient and incur large delays compared to the memory systems specialized for this purpose.

Several packet buffer designs [28], [29] use interleaved DRAM banks, so that packets can be written to multiple DRAM banks following a certain order and later retrieved according to their departure times. Only one bank can be addressed in each cycle to avoid bank conflicts. Even though we also explore DRAM interleaving in our counter array, the problems of managing statistics counters and buffering packets are fundamentally different. For packet buffer design, the goal is to find one DRAM bank entry among all banks to store an arriving packet so that it can be retrieved in time for departure. While for statistics counters, the counters are stored at fixed location. It has to be guaranteed that any counter update requests can be successfully processed.

Finally, the idea of using DRAM interleaving to implement large counter arrays was first proposed in [30]. 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 IV. We also introduced a cache module in the architecture to combat adversarial counter update patterns, which adds considerably to the difficulty of our analysis.

III. Randomized Counter Architecture

Memory interleaving has been successfully used in the past for improving the performance of computer systems [15], [31], [32], for graphics or video intensive applications [17], and for implementing routing functions like high-performance packet buffers [33]. In this section, we describe how this technique can be employed for statistics counting.

In this section, we describe our randomized counter architecture with statistical service guarantee. Fig. 1(a) depicts a simplified basic version of our randomized counter architecture. Given an 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. If the input counter update request generated by an arriving packet is a counter index, a permuted index can be achieved by applying a pseudorandom permutation function $\pi : \{1, \ldots, N\} \to \{1, \ldots, N\}$ to the counter index. We then use a simple location scheme where counter c_i will be stored in the kth memory bank, where $k = \pi(i) \bmod B$, at address location $a = \lfloor \pi(i)/B \rfloor$. Our experiments show that simple modulo function mapping the counter indexes to different DRAM banks is sufficient to guarantee fairly uniform load distribution to different banks, in which case there is no need to store the pseudorandom permutation function explicitly. In fact, if the input packet label (such as its 5-tuple) needs to be hashed to a counter index, we can view the hashing as random, and a separate permutation is no longer necessary. Hash functions on 5-tuple in the packet header are commonly applied in network processors, thus no extra cost will be added.

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 kth 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 $s + 2/\mu$. 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. It is worth noting that I/O is not the bottleneck in our system, since in each memory channel, DRAM banks are always accessed sequentially and different banks are never accessed in the same cycle. It takes the I/O only one cycle to finish a read or write operation to a DRAM bank on the chip, even though the actual operation takes several cycles to finish inside a bank. The DRAM bank manager will handle the actual writing of data into a bank after receiving commands and data from the I/O.

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 *unknown* 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 Fig. 1(b). This cache employs a first-in–first-out (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 [34, Ch. 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 look up 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., to change the request from "$+1$" to "$+2$." Since these two updates will result in only one, instead of two, eventual DRAM access that is either read or 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, we present a detailed theoretical analysis for the architecture depicted in Fig. 1(b) under all counter update sequences.

IV. Performance Analysis

In this section, we analyze the performance of our randomized counter architecture. 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 kth 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 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 arrival 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 do 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., the worst-case counter update sequence, in the convex order but 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 Section III 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 [35].

The rest of this section is organized as follows. In Section IV-A, 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 three sections, we show how to bound each individual event $D_{s,t}$ using the aforementioned convex ordering technique. In Section IV-C, we establish the worst-case number of update requests $X_{s,t}$ during time interval $[s, t]$, in terms of convex order. In Section IV-D, we bound $X_{s,t}$ with the sum of i.i.d. random variable that can be easily computed.

A. 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, and 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 $\mathtt{P_{overall}}$ can be bounded by,

$$\mathtt{P_{overall}} \le B \times \Pr[\tilde{D}] \tag{1}$$

which is the 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

$$\Pr[\tilde{D}] \le \sum_{0 \le s \le t \le n} \Pr[D_{s,t}]. \tag{2}$$

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 $[s, t]$, we have

$$D_{s,t} \equiv \{\omega \in \Omega : X_{s,t} - \mu(t - s) > K\}. \tag{3}$$

Here, we will say a few words about the implicit probability space Ω, which is the set of all permutations on $\{1,\ldots,N\}$. 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 are 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, each instance $\omega \in \Omega$ gives us an arrival sequence to the queues of the DRAM banks.

The inequality (2) 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 that 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, obviously the queue will overflow at time t or earlier, so $\omega \in \tilde{D}$. ∎

Remark: A similar lemma is proved in [4, Lemma 1]. Here, we point out the stronger relationship of equivalence.

B. Mathematical Preliminaries

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 that can be 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 IV-D. However, it is not possible to enumerate over all possible parameter settings, i.e., counter update 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 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.

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

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

where $\theta > 0$ is any constant. The last step is due to the Markov inequality. Here, τ is defined as $t-s$. Then, the overflow probability for one request queue during cycles $[0,n]$ is calculated according to (2). The system overall overflow probability can be calculated as (1).

Since this is true for all θ, we have

$$\Pr[D_{s,t}] \le \min_{\theta>0} \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}}. \tag{4}$$

Then, we aim to bound the MGF $E[e^{X\theta}]$ by identifying the worst-case counter update sequences. 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 [36, Definition 1.A.1]): For any n-dimensional vectors a and b, let $a_{[1]} \ge \cdots \ge a_{[n]}$ denote the components of a in decreasing order, and $b_{[1]} \ge \cdots \ge b_{[n]}$ denote the components of b in decreasing order. We say a is *majorized* by b, denoted $a \le_M b$, if

$$\begin{cases} \sum_{i=1}^{k} a_{[i]} \le \sum_{i=1}^{k} b_{[i]}, & \text{for } k=1,\ldots,n-1 \\ \sum_{i=1}^{n} a_{[i]} = \sum_{i=1}^{n} b_{[i]}. \end{cases} \tag{5}$$

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 $0 < \alpha < 1$.

The following lemma about convex functions will be used later.

Lemma 2: Let $b_1 \le a_1 \le a_2 \le b_2$ be such that $a_1+a_2 = b_1+b_2$, and let $f(x)$ be a convex function. Then, $f(a_1)+f(a_2) \le f(b_1)+f(b_2)$.

Proof: Let $\alpha = \frac{a_1-b_1}{b_2-b_1} \in (0,1)$. It is not hard to verify that $a_1 = (1-\alpha)b_1+\alpha b_2$ and $a_2 = \alpha b_1 + (1-\alpha)b_2$. Hence, $f(a_1) \le (1-\alpha)f(b_1)+\alpha f(b_2)$ and $f(a_2) \le \alpha f(b_1) + (1-\alpha) f(b_2)$, which combine to prove the lemma. ∎

Definition 4 (Convex Order [37, Section 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 result from Marshall [36] relates majorization, exchangeable random variables, and convex order together. It is restated here in the language

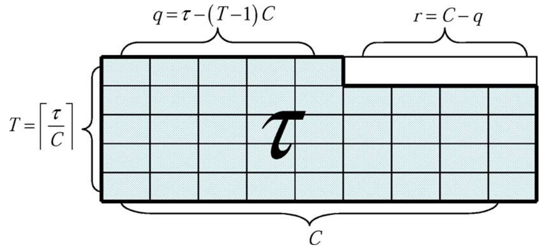


Fig. 2. Relationship of q, r, T, and τ.

of convex ordering. It is a special case of a more general theorem [36, Proposition 11.B.2].

Lemma 3 [36, Proposition 11.B.2.c]: If $X_1, \ldots, X_n$ are exchangeable random variables, a and b are n-dimensional vectors, then $a \leq_M b$ implies $\sum_{i=1}^n a_i X_i \leq_{cx} \sum_{i=1}^n b_i X_i$.

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

Definition 5 (Stochastic Order [37, Section 1.2.1]): The random variable X is said to be smaller than the random variable Y in stochastic order (written $X \leq_{st} Y$), if $\Pr[X > t] \leq \Pr[Y > t]$ for all real t.

C. 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 $X_i, 1 \leq i \leq N$, be the indicator random variable for whether the ith address is mapped to the DRAM bank under consideration, and N is the total number of addresses/indices in the DRAM banks. We have

$$E[X_i] = \frac{1}{B}. \tag{6}$$

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 $m_i, 1 \leq i \leq N$ be the count of the number of appearances of the ith 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 $m_1, \ldots, m_N$ can exceed T, where

$$T = \left\lceil \frac{\tau}{C} \right\rceil. \tag{7}$$

Moreover, let $q = \tau - (T-1)C$ and $r = C - q$. Then, only the first q requests could repeat with count T. Fig. 2 will help readers understand the relationship of q, r, T as functions of τ.

We call any vector $m = \{m_1, \ldots, m_N\}$ a valid splitting pattern of τ, if it satisfies the following:

$$\begin{cases} 0 \leq m_i \leq T \\ \sum_{i=1}^N m_i = \tau \\ |\{i : m_i = T\}| \leq q. \end{cases} \tag{8}$$

Let $\mathcal{M}$ be the set of all valid splitting patterns.[3] For simplicity, we set

$$X_m = \sum_{i=1}^N m_i X_i. \tag{9}$$

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 $a_1, a_2, \ldots, a_{q+r}$, and then they repeat for $T-1$ times in total, finally followed by q update requests for counter indices $a_1, a_2, \ldots, a_q$. In other words, inside this window of τ cycles, counters $a_1, a_2, \ldots, a_q$ (q of them in total) are accessed T times and counters $a_{q+1}, \ldots, a_{q+r}$ (r of them in total) are accessed $T-1$ times. In the following Theorem 1, we show that this arrival sequence is indeed the worst-case arrival counter updates in the sense of convex ordering.

Let m^* be the aforementioned pattern for one of such counter update sequences, i.e., let $m_1^* = \cdots = m_q^* = T, m_{q+1}^* = \cdots = m_{q+r}^* = T-1, m_{q+r+1}^* = \cdots = m_N^* = 0$. We have the following theorem.

Theorem 1: m^* is the worst-case splitting pattern in terms of convex ordering, i.e., $X_m \leq_{cx} X_{m^*}, \forall m \in \mathcal{M}$.

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

$$\begin{cases} m_{[i]} \leq T = m_i^*, & \text{for } 1 \leq i \leq q \\ m_{[i]} \leq T - 1 = m_i^*, & \text{for } q+1 \leq i \leq q+r. \end{cases} \tag{10}$$

Therefore, majorization condition (5) is true for $1 \leq k \leq q+r$. Since

$$\sum_{i=1}^{q+r} m_i^* = \tau = \sum_{i=1}^N m_i. \tag{11}$$

The condition is true for $i > q+r$ as well. Therefore, by definition, $m \leq_M m^*$.

The theorem follows from Lemma 3 because $X_1, \ldots, X_n$ are also 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 [37, Theorem 1.2.9].

Unfortunately, it is in general not possible to apply the Chernoff bound directly to the MGF of X_{m^*}. For $\tau \leq C$, X_{m^*} is a hypergeometric random variable whose MGF $\mathrm{E}[e^{X_{m^*}\theta}]$ is a hypergeometric series [38] 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.

D. Relaxations of X_{m^*} for Computational Purposes

We first state the main theorem of the paper.

[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 in our analysis.

Theorem 2:
For $\tau \leq C$

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{\theta} + \left(1 - \frac{1}{B}\right)\right)^{\tau}}{e^{(K+\mu\tau)\theta}}. \tag{12}$$

For $\tau > C$

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{T\theta}+\left(1 - \frac{1}{B}\right)\right)^{q} \left(\frac{1}{B}e^{(T-1)\theta}+\left(1 - \frac{1}{B}\right)\right)^{r}}{e^{(K+\mu\tau)\theta}}. \tag{13}$$

Proof: To prove the theorem, 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 Chernoff technique can be applied. The following result 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 for $\tau \leq C$.

Lemma 4 [39, Theorem 4]: Let the population S consist of N values $c_1, \ldots, c_N$. Let $X_1, \ldots, X_n$ denote a random sample without replacement from S, and let $Y_1, \ldots, Y_n$ denote a random sample with replacement from S. Let $X = X_1 + \cdots + X_n$, $Y = Y_1 + \cdots + Y_n$. Then, $X \leq_{cx} Y$.

In our algorithms, we need to use weighted sum of random variables, therefore we extend Hoeffding's result to the following theorem.

Theorem 3: Same notations as in Lemma 4. Let $a_1, \ldots, a_n$ be constants, $a_i > 0, \forall i$. Then, $\sum_{i=1}^{n} a_i X_i$ is dominated by $\sum_{i=1}^{n} a_i Y_i$ in the convex order. Thus, for any convex function f, we have

$$\mathrm{E}f\left(\sum_{i=1}^{n} a_i X_i\right) \leq \mathrm{E}f\left(\sum_{i=1}^{n} a_i Y_i\right). \tag{14}$$

Proof: The proof of Theorem 3 is in the Appendix. ■

We consider the two cases: $\tau \leq C$, which is the scenario that the measurement window τ in number of cycles is no larger than the cache size in number of entries, and the case $\tau > C$ separately.

1) Case $\tau \leq C$: When $\tau \leq C$, $X_{m^*} = X_1 + \cdots + X_\tau$. Let the population S consist of $c_1, \ldots c_N$ 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 4 has the same distribution as our X_{m^*}, and Y_i's 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 $X_m \leq_{cx} X_{m^*} \leq_{cx} Y$ we get

$$\begin{aligned}\mathrm{E}[e^{X_m\theta}] \leq \mathrm{E}[e^{X_{m^*}\theta}] &\leq \mathrm{E}[e^{Y\theta}] \\ &= \mathrm{E}\left[e^{(Y_1+\cdots+Y_\tau)\theta}\right] \\ &= \mathrm{E}[e^{Y_1\theta}]^{\tau} \\ &= \left(\frac{1}{B}e^{\theta} + \left(1 - \frac{1}{B}\right)\right)^{\tau}.\end{aligned}$$

By (4), we now have the following bound:

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

2) Case $\tau > C$: For $\tau > C$, we have $X_{m^*} = T(X_1 + \cdots + X_q)+(T-1)(X_{s+1}+\cdots+X_{q+r})$, from Section IV-C. Lemma 4 cannot be applied due to the coefficients T and $(T-1)$. Instead, we apply Theorem 3 and get

$$\begin{aligned}\mathrm{E}[e^{X_m\theta}] \leq \mathrm{E}[e^{X_{m^*}\theta}] &\leq \mathrm{E}[e^{Y\theta}] \\ &= \mathrm{E}\left[e^{(T(Y_1+\cdots+Y_q)+(T-1)(Y_{q+1}+\cdots+Y_{q+r}))\theta}\right] \\ &= \mathrm{E}[e^{TY_1\theta}]^{q}\mathrm{E}\left[e^{(T-1)Y_1\theta}\right]^{r} \\ &= \left(\frac{1}{B}e^{T\theta}+\left(1-\frac{1}{B}\right)\right)^{q}\left(\frac{1}{B}e^{(T-1)\theta}+\left(1-\frac{1}{B}\right)\right)^{r}.\end{aligned}$$

Together with (4), we have

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{T\theta}+\left(1 - \frac{1}{B}\right)\right)^{q} \left(\frac{1}{B}e^{(T-1)\theta}+\left(1 - \frac{1}{B}\right)\right)^{r}}{e^{(K+\mu\tau)\theta}}.$$

Thus, we have proved Theorem 2. ■

Remark: For $\tau > C$, we can get away with Lemma 4 alone by taking a different viewpoint. For the worst-case counter update sequence, instead of selecting $C = q + r$ permutation destinations for the counter addresses and checking which ones are mapped to the $\frac{N}{B}$ addresses in a DRAM bank, we can treat it as selecting $\frac{N}{B}$ permutation sources and checking which ones are among the addresses that contain the counter updates. Hence, let the population S' be exactly the components of m^*, i.e., let $c_i = m*_i$. Thus, 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. Therefore

$$X_{m^*} = X'_1 + \cdots + X'_{\frac{N}{B}}, \tag{15}$$

where X'_i's are a random sample without replacement from S'. By Lemma 4, we have

$$X_{m^*} \leq_{cx} Y' = \sum_{i=1}^{N/B} Y'_i \tag{16}$$

where Y'_i's are a random sample with replacement from S'. Thus, the Y'_i's are i.i.d. random variable with

$$\Pr[Y'_i = y] = \begin{cases} \frac{q}{N}, & \text{if } y = T \\ \frac{r}{N}, & \text{if } y = T - 1 \\ \frac{N-q-r}{N}, & \text{if } y = 0. \end{cases} \tag{17}$$

Hence, similar to the $\tau \leq C$ case, we can derive the following bound:

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

For the last inequality, we used the inequality $(1+\frac{a}{n})^n < e^a$. By (4), we now have the following bound (it is not hard to verify that it also applies to $\tau \le C$):

$$\Pr[D_{s,t}] \le \min_{\theta>0} e^{(qe^{T\theta}+re^{(T-1)\theta}-q-r)/B-(K+\mu\tau)\theta}. \tag{18}$$

However, the bounds in Theorem 2 are better numerically in our setting.

We also see that the bounds in Theorem 2 are translation invariant, i.e., they only depend on $\tau = t - s$. Therefore, the computation cost of (2) 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 bounds for the overflow probability under the worst-case counter update request sequences. The bounds can be computed by $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 2.

V. Performance Evaluation

In this section, we present evaluation results for our proposed DRAM-based counter array architecture. The outline of this section is as follows. Section V-A describes the instantiation of our proposed solution. Section V-B outlines the parameters of two real-world Internet traffic traces for our evaluations. Section V-C presents numerical results derived using the analytical models presented in Section IV for our randomized counter architecture. Finally, Section V-D provides a comparison of our new approach with the state-of-the-art hybrid SRAM/DRAM approach [4].

A. Implementation Details

To provide a general formulation, we have assumed in Section III 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. Therefore, a single memory channel can be used to pipeline memory transactions across them at peak rates when operating in an interleaving manner.

To provide a concrete analysis of our proposed solution, we use the specifications of an actual commercial high-bandwidth memory part, namely the XDR memory from Rambus [17], [19]. As depicted in Fig. 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 packet (40 B) 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.

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 the 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

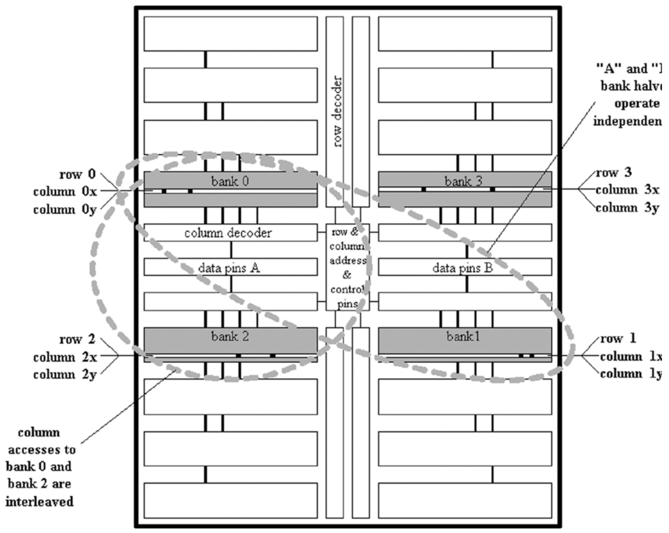


Fig. 3. Example high-bandwidth DRAM architecture [17]. Each XDR memory IC has 16 internal memory banks that can be interleaved to achieve high-bandwidth memory access.

memory channels are becoming increasingly common. For example, both the Cell processor from IBM/Sony/Toshiba [18] and the latest mainstream Intel x86 multicore processor [40] 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 access latency ratio that is no smaller than 1/16. It is worth noting that we use an ASIC implementation where we can directly interact with the external DRAMs. Such a design is different from using a general-purpose CPU to implement the algorithm since modern CPUs always use intermediate caches to access external DRAMS.

B. Traffic Traces

For our evaluations, we use 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-Gb/s 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 flows.

C. Tail Bounds for Randomized Counter Architecture

The randomized counter scheme avoids the need to replicate counters, and thus requires the same amount of DRAM as the hybrid SRAM/DRAM schemes.

1) Overflow Probabilities With $\mu = 1/16$: In this section, we present the numerical results computed from the formulae derived in Section IV using MATLAB 7.11. We use $n = 10^{10}$ for all the following examples, where n is the total number of cycles for the measurement period. The overall overflow probability can be calculated from (1) and (2).

In Fig. 4, the overflow probability bounds with different cache size C as a function of queue length K are presented,

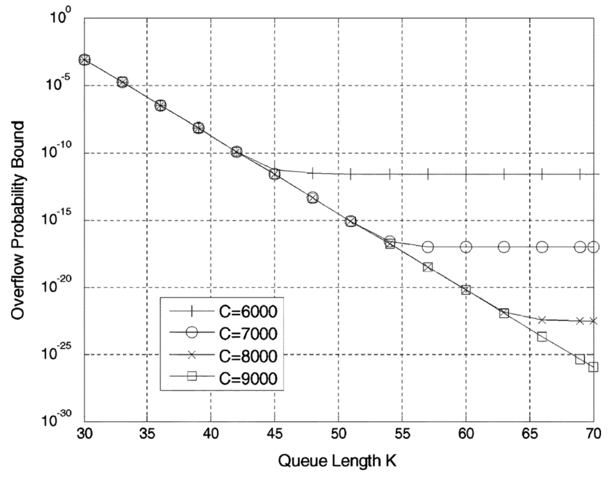


Fig. 4. Overflow probability bounds as a function of request queue size K with $\mu = 1/16$ and $B = 32$.

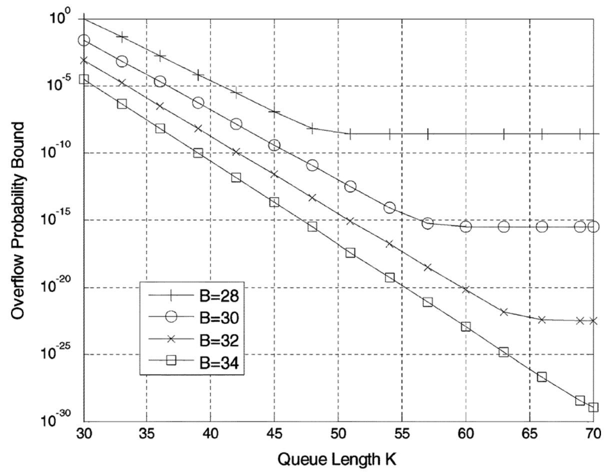


Fig. 5. Overflow probability bounds as a function of the number of memory banks B with $\mu = 1/16$ and $C = 8000$.

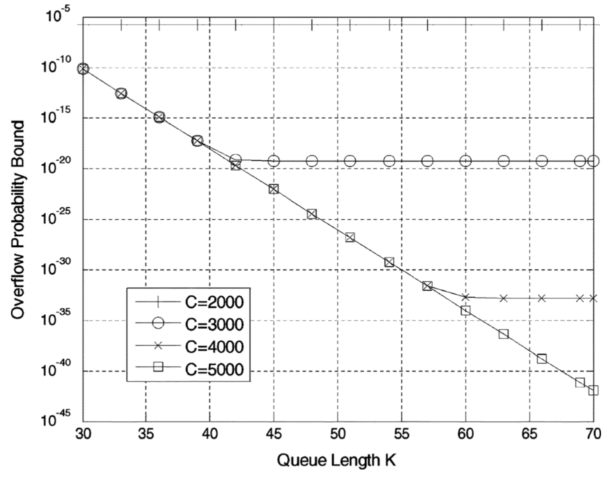


Fig. 6. Overflow probability bounds as a function of queue size K with $\mu = 1/12$ and $B = 32$.

where $\mu = 1/16$ and $B = 32$. It is easy to see from this graph that as the request queue length K increases, the overflow probability bound decreases. However, after K reaches certain thresholds, the overflow probability bound stays practically flat. The flat level depends on the cache size C, which confirms our prediction that given a fixed-size cache of size C, in the worst case an adversary can keep sending repetitive requests that is C cycles apart, thereby causing system to overflow with nonzero probability no matter how large the request queue size is. Actually, as cache size C approaches infinity, the overflow probability as a function of queue length K becomes the overflow probability of a discrete-time single server FIFO queue with Bernoulli arrivals (Geo/D/1 queue) with arrival probability $1/B = 1/32$ to a queue of limited size K. As shown in Fig. 4, when $C \geq 8000$, the overflow probability bound has only negligible decreases as cache size C increases for the practical setting of overflow probabilities. In other words, by increasing the size of the cache, the performance of the system will not improve significantly. Therefore, we consider that $C = 8000$ is enough for practical purposes in achieving an overflow probability bound of 10^{-14} with request queue larger than 45 entries.

In Fig. 5, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/16$ and $C = 8000$. Cache size $C = 8000$ is used here because by using a larger cache, the system performance will not achieve significant improvements. It can be seen from this figure that given the same K, as the number of memory banks B increases, the overflow probability bound decreases, which confirms our analysis that by using more memory banks, the load of arrival requests to each bank is reduced. Let us define the total queue size as M, where $M = B \cdot K$. From Fig. 5, the optimal configuration for total queue size that can achieve a certain overflow probability bound can be calculated by choosing the curve with the smallest M that achieves this overflow probability bound.

2) Overflow Probabilities With $\mu = 1/12$: In this section, the overflow probability for the system with $\mu = 1/12$ is presented. Compared to a system with $\mu = 1/16$, we expect the system with $\mu = 1/12$ to achieve better/lower overflow bounds with the same setting since the increase of value μ corresponds to a smaller gap between the DRAM and SRAM access latencies.

In Fig. 6, the overflow probabilities as a function of cache size C and queue length K are presented, where $\mu = 1/12$ and $B = 32$, i.e., there are 32 memory banks in the system. It is easy to see from this graph that as K increases, the overflow probability decreases. For $C \geq 4000$, the overflow probability $\Pr[D_{s,t}]$ decreases exponentially. In Fig. 6, it is also shown that as the cache size C grows, the overflow probability decreases. When $C \geq 4000$, the overflow probability has only negligible decreases as cache size C increases. I.e., by increasing the cache size used in the system, the performance will not be improved much for the practical setting of overflow probabilities. Therefore, $C = 4000$ is enough for the practical purposes. Compared to Fig. 4, with a larger μ and the same request queue size, the cache size C can be significantly reduced to achieve the same overflow bounds.

In Fig. 7, the system overflow probability with different number of memory banks B as a function of K is presented, where $\mu = 1/12$ and $C = 4000$. It can be seen from this figure that given the same K, as B increases, the overflow probability

TABLE I
COMPARISON OF DIFFERENT SCHEMES FOR A REFERENCE CONFIGURATION WITH 16 MILLION 64-b COUNTERS. FOR OUR METHOD, $K = 50$, $B = 32$, AND $C = 7000$

	Naïve	Hybrid SRAM/DRAM [4]	Randomized Counter
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

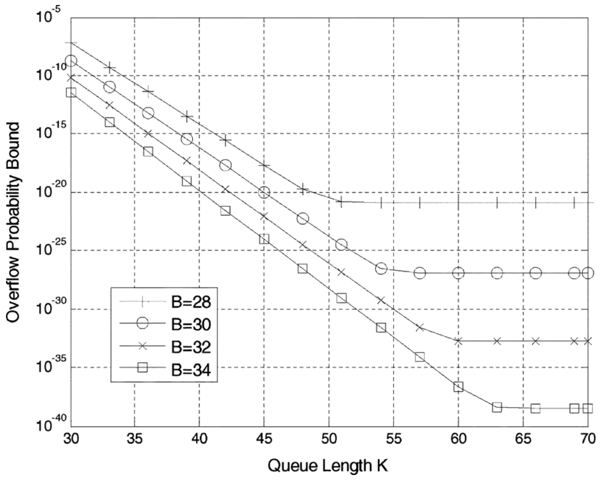


Fig. 7. Overflow probability bounds as a function of number of memory banks B with $\mu = 1/12$ and $C = 4000$.

decreases. Compared to Fig. 5, we can see that with a small increase in the SRAM-to-DRAM access latency ratio μ and the same setting of K, even with half the cache size, the overflow probability bounds are significantly reduced.

Our simulation results above show the overflow probability for the worst-case counter update sequences among all possible ones. For actual traffic pattern from real-world traffic traces, the system performance is much better than the one predicted by the overflow probability bound. We run our simulations using the two Internet traces from USC and UNC for an extensive period of time. By setting the parameters such that C ranges from 2000 to 9000, K ranges from 30 to 70, and B ranges from 28 to 34, we observed no system overflow in either of the traces. This can be explained as that the counter update patterns in real traces are much better than the worst-case update patterns predicted in our paper. Thus, our statistics counter array, which is designed to handle worst-case update patterns, does not overflow at all for the two real traces.

D. Cost-Benefit Comparison

Table I compares our proposed approach with a naive SRAM approach as well as the hybrid SRAM/DRAM counter architecture approaches [2]–[5]. The naive SRAM approach simply implements all counters in SRAM. For the hybrid SRAM/DRAM approach, we specifically compare against the state-of-the-art scheme proposed by Zhao *et al.* [4] that provably achieves the minimum SRAM requirement for this architecture class. As demonstrated in [4], in one example setting their approach requires almost a factor of six times less SRAM than the first hybrid SRAM/DRAM solution proposed in [2] and more than a factor of two times less SRAM than an improved solution proposed in [3]. For an SRAM-to-DRAM latency ratio of $\mu = 1/16$, the architecture in [4] requires $w = \lceil \log 1/\mu \rceil = \lceil \log 16 \rceil = 4$ 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.

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 $\lceil \log 16 \text{ million} \rceil = 24$ b (3 B) to encode the counter index. However, even though the scheme requires just 4 b 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. Other SRAM schemes exist [8], [11], where much smaller-size SRAM is required and no DRAM access is necessary to retrieve a counter value. However, they only approximate the counters to a certain extent, and there is no guarantee that any counter value is exact. On the other hand, our proposed scheme, together with the naive SRAM and hybrid SRAM/DRAM solutions, always return exact counter values if there is no system overflow. For our scheme and hybrid SRAM/DRAM solutions, the exact counters are retrieved at the expense of slow DRAM accesses.

For our proposed randomized counter solution, a small update request queue needs to be maintained at each memory bank. As shown in Fig. 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 B to encode the counter indices of 16 million counters and 4 b 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 Fig. 1(b) in Section III, 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 B encoding of counter indices and 4 b 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.

It is worth noting that the overflow probability bounds are derived for our randomized counter architecture using the

[4]An accumulation counter of 4 b 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.

worst-case counter update assumption. In practice, the system overflow probability for the Internet traffic is much smaller than the overflow probability bound since real-world traffic traces contain more interleaving-friendly counter update patterns than the worst case predicted by our model. While our analysis predicts that request queues with 50 entries are required to achieve an overflow probability bound of 10^{-14} with a cache of size 7000, in our experiments the number of occupied entries in a request queue never exceeds 18.

VI. 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 by exploiting advanced architecture features that are readily available in modern commodity DRAM architectures. In particular, we presented a randomized counter 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 counter architecture under the worst-case counter update sequences using a novel combination of convex ordering and large deviation theory. Our analysis also takes into considerations of the 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 as different number representations, including both integer and floating point number representations.

Appendix
Proof of Theorem 3

Remark: The theorem can be shown to be false if a_i's can have mixed signs. E.g., $n = N = 2, c_1 = 1, c_2 = 2, a_1 = 1, a_2 = -1$, f is strictly convex, then

$$\begin{aligned}&\mathrm{E}\left[f\left(\sum a_iX_i\right)\right] - \mathrm{E}\left[f\left(\sum a_iY_i\right)\right]\\&= \frac{1}{2}[f(-1)+f(1)] - \frac{1}{4}[f(-1)+f(1)+2f(0)]\\&= \frac{1}{2}\left[\frac{1}{2}f(-1)+\frac{1}{2}f(1)-f(0)\right] > 0 \qquad (19)\end{aligned}$$

by Jensen's inequality. On the other hand, the signs of c_i's do not matter since we can do a shift to make all c_i's positive.

Proof: Following Hoeffding's notation, for an arbitrary function g of n variables, we have

$$\mathrm{E}g(X_1,\ldots,X_n) = \frac{1}{N^{(n)}}\sum_{N,n} g(c_{i_1},\ldots,c_{i_n}) \qquad (20)$$

$$\mathrm{E}g(Y_1,\ldots,Y_n) = \frac{1}{N^n}\sum_{i_1=1}^{N}\cdots\sum_{i_n=1}^{N} g(c_{i_1},\ldots,c_{i_n}) \qquad (21)$$

where $N^{(n)} = N(N-1)\ldots(N-n+1)$, and $\sum_{N,n}$ is taken over all n-tuples $i_1,\ldots,i_n$ of distinct positive integers not exceeding N. The goal is to find function $\bar{g}$ such that the N^n terms of g in (21) can be rewritten into $N^{(n)}$ terms of $\bar{g}$, and each term of $\bar{g}$ will dominate each term of g in (20) for

$$g(x_1,\ldots,x_n) = f(a_1x_1+\ldots+a_nx_n). \qquad (22)$$

Hence, we want

$$\begin{aligned}\mathrm{E}g(Y_1,\ldots,Y_n) &= \frac{1}{N^n}\sum_{i_1=1}^{N}\cdots\sum_{i_n=1}^{N} g(c_{i_1},\ldots,c_{i_n})\\&= \frac{1}{N^{(n)}}\sum_{N,n}\bar{g}(c_{i_1},\ldots,c_{i_n})\\&= \mathrm{E}\bar{g}(X_1,\ldots,X_n). \qquad (23)\end{aligned}$$

For $n = 2$, we can use

$$\begin{aligned}\bar{g}(x_1,x_2) &= \frac{N-1}{N}g(x_1,x_2)\\&\quad+\frac{1}{N}\left(\frac{a_1}{a_1+a_2}g(x_1,x_1)+\frac{a_2}{a_1+a_2}g(x_2,x_2)\right). \qquad (24)\end{aligned}$$

In general, let $[n]$ denote the set $\{1,\cdots,n\}$. Let

$$P = \{\rho = \{A_1,\cdots,A_k\} \mid A_i \subset [n], \cup_i A_i = [n]\} \qquad (25)$$

be the set of all partitions of $[n]$. Let

$$H = \{h : [n] \to [n] \mid h \circ h = h\} \qquad (26)$$

be the set of all mappings from $[n]$ to itself, with the restriction that the points in image of h, $Im(h)$, are fixed points of h. We can denote h by the vector $(h(1),\ldots,h(n))$. Any h naturally induces a partition $\rho_h = \{h^{-1}(j) \mid j \in Im(h)\}$. For example, the mappings $(1,1,3,3), (2,2,3,3), (1,1,4,4), (2,2,4,4)$ all induce the partition $\{\{1,2\},\{3,4\}\}$. We will let $\bar{g}$ be linear combinations of $g(x_{h(1)},\ldots,x_{h(n)})$ for all $h \in H$—e.g., in the case of $n = 2$, due to our definition of H, we do not include $g(x_2,x_1)$ in $\bar{g}(x_1,x_2)$. We define $\bar{g}$ as

$$\bar{g}(x_1,\ldots,x_n) = \sum_{h\in H} p_h g(x_{h(1)},\ldots,x_{h(n)})$$

where

$$p_h = p_{\rho_h}\Pi_{j\in Im(h)}\frac{a_j}{\sum_{i\in h^{-1}(j)} a_i} \qquad (27)$$

where p_ρ are constants. We note that from the definition, p_h and p_ρ satisfy

$$\sum_{\{h \mid \rho_h=\rho\}} p_h = p_\rho \qquad \forall \rho \in P \qquad (28)$$

$$\sum_{h\in H} p_h = \sum_{\rho\in P} p_\rho. \qquad (29)$$

We need to argue that there exist p_ρ such that (23) is true for any g. Here, we will specify p_ρ exactly. Take $n = 4$ for example. For any distinct $c_1, c_2 \in C$, $g(c_1,c_1,c_2,c_2)$ appears once on the left-hand side (LHS) of (23). It corresponds to $\rho = \{\{1,2\},\{3,4\}\}$. On the right-hand side (RHS), for any distinct $\alpha,\beta \in C$ that are different from c_1, c_2, we have

$$\begin{aligned}\bar{g}(c_1,\alpha,c_2,\beta) &= \frac{\mathbf{p_\rho a_1 a_3 g(c_1,c_1,c_2,c_2)}}{\mathbf{(a_1+a_2)(a_3+a_4)}}\\&\quad+\frac{p_\rho a_2 a_3 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)}\\&\quad+\frac{p_\rho a_1 a_4 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)}\\&\quad+\frac{p_\rho a_2 a_4 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)}+\cdots \qquad (30)\end{aligned}$$

$$\begin{aligned}\bar{g}(\alpha,c_1,c_2,\beta) &= \frac{p_\rho a_1 a_3 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{\mathbf{p_\rho a_2 a_3 g(c_1,c_1,c_2,c_2)}}{\mathbf{(a_1+a_2)(a_3+a_4)}} \\ &+ \frac{p_\rho a_1 a_4 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_2 a_4 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \cdots \qquad (31)\\ \bar{g}(c_1,\alpha,\beta,c_2) &= \frac{p_\rho a_1 a_3 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_2 a_3 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{\mathbf{p_\rho a_1 a_4 g(c_1,c_1,c_2,c_2)}}{\mathbf{(a_1+a_2)(a_3+a_4)}} \\ &+ \frac{p_\rho a_2 a_4 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} + \cdots \qquad (32)\\ \bar{g}(\alpha,c_1,\beta,c_2) &= \frac{p_\rho a_1 a_3 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_2 a_3 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_1 a_4 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{\mathbf{p_\rho a_2 a_4 g(c_1,c_1,c_2,c_2)}}{\mathbf{(a_1+a_2)(a_3+a_4)}} + \cdots. \qquad (33)\end{aligned}$$

Adding up the bold diagonal terms gives us $p_\rho g(c_1,c_1,c_2,c_2)$. Since there are $(N-2)(N-3)$ choices for α,β, we need to have

$$\frac{1}{N^4} = \frac{(N-2)(N-3)p_\rho}{N^{(4)}}$$

so $p_\rho = \frac{N^{(2)}}{N^4}$. In general, we need to set $p_\rho = \frac{N^{(|\rho|)}}{N^n}$, where $|\rho|$ denotes the number of subsets in partition ρ.

Now, we consider the case

$$g(x_1,\ldots,x_n) = f(a_1x_1+\ldots+a_nx_n).$$

If we set $f(x)=1$, we see from (23) and (27) that

$$\sum_{h\in H} p_h = 1. \qquad (34)$$

This can also be shown directly since

$$\sum_{h\in H} p_h = \sum_{\rho\in P} p_\rho = \frac{1}{N^n}\sum_{\rho\in P} N^{(|\rho|)}.$$

We can view $\sum_{\rho\in P} N^{(|\rho|)}$ as one way to count all ordered samples of size n with replacement, by considering all repetition patterns, thus it equals N^n.

If we set $f(x)=x$, then $\bar{g}(x_1,\ldots,x_n)$ is a linear combination of $x_1,\ldots,x_n$. It may be possible to argue that the ratio between the coefficients for x_{i_1} and x_{i_2} will be a_{i_1}/a_{i_2}. Here, we calculate it directly. We have

$$\begin{aligned}&\sum_{\{h\,|\,\rho_h=\rho\}} p_h(a_1x_{h(1)}+\ldots+a_nx_{h(n)}) \\ &\quad = p_\rho(a_1x_1+\ldots+a_nx_n) \qquad \forall\rho\in P.\end{aligned}$$

For example, for $\rho=\{\{1,2\},\{3,4\}\}$, we have

$$\begin{aligned}&\sum_{\{h\,|\,\rho_h=\rho\}} p_h(a_1x_{h(1)}+\ldots+a_nx_{h(n)}) \\ &= \frac{p_\rho a_1a_3(a_1x_1+a_2x_1+a_3x_3+a_4x_3)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_2a_3(a_1x_2+a_2x_2+a_3x_3+a_4x_3)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_1a_4(a_1x_1+a_2x_1+a_3x_4+a_4x_4)}{(a_1+a_2)(a_3+a_4)} \\ &+ \frac{p_\rho a_2a_4(a_1x_2+a_2x_2+a_3x_4+a_4x_4)}{(a_1+a_2)(a_3+a_4)}.\end{aligned}$$

For x_1, the first and third terms yield

$$\frac{p_\rho a_1a_3x_1}{x_3+x_4} + \frac{p_\rho a_1a_4x_1}{x_3+x_4} = p_\rho a_1x_1$$

and similarly for x_2,x_3,x_4. Therefore

$$\begin{aligned}&\sum_h p_h\left(a_1x_{h(1)}+\cdots+a_nx_{h(n)}\right) \\ &\quad = \sum_\rho p_\rho(a_1x_1+\cdots+a_nx_n) \\ &\quad = a_1x_1+\cdots+a_nx_n. \qquad (35)\end{aligned}$$

From (34), (35), and Jensen's inequality, we get

$$\begin{aligned}\bar{g}(x_1,\ldots,x_n) &= \sum_{h\in H} p_h f\left(a_1x_{h(1)}+\ldots+a_nx_{h(n)}\right) \\ &\geq f\left(\sum_{h\in H} p_h(a_1x_{h(1)}+\cdots+a_nx_{h(n)})\right) \\ &= f(a_1x_1+\cdots+a_nx_n). \qquad (36)\end{aligned}$$

Hence, $\mathrm{E}\bar{g}(X_1,\cdots,X_n) \geq \mathrm{E}f(a_1X_1+\cdots+a_nX_n)$, which together with (23) completes the proof. ■

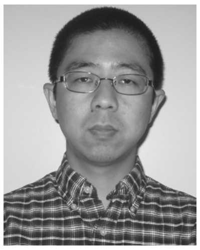
REFERENCES

[1] H. Zhao, H. Wang, B. Lin, and J. Xu, "Design and performance analysis of a DRAM-based statistics counter array architecture," in *Proc. 5th ACM/IEEE ANCS*, Oct. 2009, pp. 84–93.
[2] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown, "Maintaining statistics counters in router line cards," in *Proc. 35th Annu. IEEE/ACM MICRO*, Jan. 2002, vol. 22, no. 1, pp. 76–81.
[3] S. Ramabhadran and G. Varghese, "Efficient implementation of a statistics counter architecture," *Perform. Eval. Rev.*, vol. 31, no. 1, pp. 261–271, 2003.
[4] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," *Perform. Eval. Rev.*, vol. 34, no. 1, pp. 323–334, 2006.
[5] M. Roeder and B. Lin, "Maintaining exact statistics counters with a multi-level counter memory," in *Proc. IEEE GLOBECOM*, Nov. 2004, vol. 2, pp. 576–581.
[6] R. Morris, "Counting large numbers of events in small registers," *Commun. ACM*, vol. 21, no. 10, pp. 840–842, 1978.
[7] A. Cvetkovski, "An algorithm for approximate counting using limited memory resources," *Perform. Eval. Rev.*, vol. 35, no. 1, pp. 181–190, 2007.
[8] R. Stanojevic, "Small active counters," in *Proc. IEEE INFOCOM*, 2007, pp. 2153–2161.
[9] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," *Perform. Eval. Rev.*, vol. 36, no. 1, pp. 121–132, 2008.
[10] N. Hua, B. Lin, J. Xu, and H. Zhao, "BRICK: A novel exact active statistics counter architecture," in *Proc. 4th ACM/IEEE ANCS*, 2008, pp. 89–98.

[11] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, C. Wu, and Y. Cheng, "DISCO: Memory efficient and accurate flow statistics for network measurement," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2010, pp. 665–674.
[12] P. Indyk, "Stable distributions, pseudorandom generators, embeddings, and data stream computation," in *Proc. 41st Annu. IEEE FOCS*, 2000, pp. 189–197.
[13] H. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, "A data streaming algorithm for estimating entropies of OD flows," in *Proc. ACM IMC*, 2007, pp. 279–290.
[14] 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 *Proc. 5th HPCA*, 1999, p. 80.
[15] W. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proc. 5th HPCA*, 2001, p. 301.
[16] F. Ware and C. Hampel, "Improving power and data efficiency with threaded memory modules," in *Proc. IEEE ICCD*, Oct. 2006, pp. 417–424.
[17] F. A. Ware and C. Hampel, "Micro-threaded row and column operations in a DRAM core," Rambus, Inc., Sunnyvale, CA, White Paper, Mar. 2005.
[18] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," in *Proc. 39th Annu. IEEE/ACM MICRO*, 2006, vol. 26, no. 2, pp. 10–24.
[19] "XDR datasheet," Rambus, Inc., Sunnyvale, CA, 2005–2006.
[20] "XDR-2 datasheet," Rambus, Inc., Sunnyvale, CA, 2008–2009.
[21] "Intel IXP 465 network processor product brief," Intel Corporation, Santa Clara, CA, 2006.
[22] C. Pandit and S. Meyn, "Worst-case large-deviation asymptotics with application to queueing and information theory," *Stochastic Process. Appl.*, vol. 116, no. 5, pp. 724–756, 2006.
[23] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *Proc. ACM SIGCOMM*, 2004, pp. 245–256.
[24] C. Semeria and J. Gredler, "Juniper Networks solutions for network accounting," Juniper Networks, Sunnyvale, CA, White Paper no. 20010-001, 2001.
[25] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. ACM SIGCOMM*, 2002, pp. 323–336.
[26] H. Wang, H. Zhao, B. Lin, and J. Xu, "Design and analysis of a robust pipelined memory system," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.
[27] B. Agrawal and T. Sherwood, "High-bandwidth network memory system through virtual pipelines," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1029–1041, Aug. 2009.
[28] S. Iyer and N. Mckeown, "Designing buffers for router line cards," Stanford University, Stanford, CA, Tech. Rep. TR02-HPNG-031001, Mar. 2002.
[29] H. Wang and B. Lin, "Block-based packet buffer with deterministic packet departures," in *Proc. 11th HPSR*, 2010, pp. 38–43.
[30] B. Lin and J. Xu, "DRAM is plenty fast for wirespeed statistics counting," in *Proc. 1st HotMetrics*, Jun. 2008, pp. 45–50.
[31] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1996.
[32] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proc. 18th Annu. ISCA*, 1991, pp. 74–83.
[33] G. Shrimali and N. McKeown, "Building packet buffers using interleaved memories," in *Proc. 6th HPSR*, May 2005, pp. 1–5.
[34] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1995.
[35] S. M. Ross, *Low-Density Parity-Check Codes*, 2nd ed. New York: Wiley, 1995.
[36] A. W. Marshall and I. Olkin, *Inequalities: Theory of Majorization and Its Applications*. New York: Academic, 1979.
[37] A. Muller and D. Stoyan, *Comparison Methods for Stochastic Models and Risks*. New York: Wiley, 2002.
[38] R. Graham, D. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA: Addison-Wesley, 1994.
[39] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *J. Amer. Statist. Assoc.*, vol. 58, no. 301, pp. 13–30, 1963.
[40] "Intel Lynnfield processor," Intel Corporation, Santa Clara, CA [Online]. Available: http://www.intel.com

Hao Wang (S'06) received the B.E. degree in electrical engineering from Tsinghua University, Beijing, China, in 2005, and the M.S. degree in electrical and computer engineering from the University of California, San Diego, in 2008, and is currently pursuing the Ph.D. degree in electrical and computer engineering from the University of California, San Diego.

His current research interests include the architecture and scheduling algorithms for high-speed switching and routing, robust memory system design, network measurement, large deviation principle, and coding and information theory.



Haiquan (Chuck) Zhao received the Ph.D. degree in computer science from the Georgia Institute of Technology, Atlanta, in 2010.

He is currently working as a Software Engineer with Microsoft, Redmond, WA.

Bill Lin (M'97) received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1985, 1988, and 1991, respectively.

He is currently a Professor of electrical and computer engineering and an Adjunct Professor of computer science and engineering with the University of California, San Diego (UCSD). At UCSD, he is actively involved with the Center for Wireless Communications (CWC), the Center for Networked Systems (CNS), and the California Institute for Telecommunications and Information Technology (CAL-IT^2) in industry-sponsored research efforts. Prior to joining the faculty at UCSD, he was the Head of the System Control and Communications Group at IMEC, Leuven, Belgium. IMEC is the largest independent microelectronics and information technology research center in Europe. It is funded by European funding agencies in joint projects with major European telecom and semiconductor companies. His research has led to over 130 journal and conference publications. He also holds four awarded patents.

Prof. Lin has received a number of publication awards, including the 1995 IEEE TRANSACTIONS ON VLSI SYSTEMS Best Paper Award, a Best Paper Award at the 1987 ACM/IEEE Design Automation Conference, Distinguished Paper citations at the 1989 IFIP VLSI Conference and the 1990 IEEE International Conference on Computer-Aided Design, a Best Paper nomination at the 1994 ACM/IEEE Design Automation Conference, and a Best Paper nomination at the 1998 Conference on Design Automation and Test in Europe.

Jun (Jim) Xu (S'98–M'00–SM'10) received the Ph.D. in computer and information science from The Ohio State University, Columbus, in 2000.

He is an Associate Professor with the College of Computing, Georgia Institute of Technology, Atlanta. His current research interests include data streaming algorithms for the measurement and monitoring of computer networks and hardware algorithms and data structures for high-speed routers.

Dr. Xu received the NSF CAREER Award in 2003, the ACM SIGMETRICS Best Student Paper Award in 2004, and IBM faculty awards in 2006 and 2008. He was named an ACM Distinguished Scientist in 2010.