Robust Statistics Counter Arrays with Interleaved Memories

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

Abstract—Statistics counters are essential in network measurement on tracking various network statistics and implementing various network counting sketches. For such applications it is crucial to maintain a large number of statistics counters at very high speeds. On the Internet with millions of flows, potentially millions of counters are required to be updated at wirespeed of 40 Gb/s and beyond. It is widely accepted that SRAM is too costly to store such large counter arrays entirely, and DRAM is too slow to catch up with the line rate. In this paper, we propose a DRAM-based architecture that takes advantage of the performance of modern commodity DRAM by interleaving counter updates to multiple memory banks. Our architecture is based on the observation that most flows on the Internet consist of multiple packets that are transmitted during a relatively short period of time, which are referred to as traffic bursts. Our proposed architecture makes use of a simple randomization scheme and a set of small fully associative request queues to statistically guarantee a near-perfect load balancing of counter updates to the memory banks. The architecture explores the benefit of traffic bursts to greatly reduce the maximum size of the request queues while providing a diminishing overflow probability guarantee. We also develop queuing models to show that as long as the flow sizes are heavy-tailed distributed due to traffic bursts, the maximum request queue length is always bounded by a small constant. The simulation results confirm the effectiveness of our queuing models. The proposed statistics counter arrays can effectively maintain line rate updates to a large number of counters while guaranteeing a diminishing overflow probability in the system.

Index Terms—Interleaved memories, statistics counter array, random access memories, queuing theory

1 Introduction

For the monitoring and control of large networks, it is crucial to maintain a large number of statistics counters at high link rate. Several millions of counters are required in applications such as per-flow measurements [1], [2] for tracking various network statistics, implementing various network measurement, router management, intrusion detection, traffic engineering, and data streaming. For example, by counting the total size of the packets with the same flow record, it is possible to study the network traffic pattern, distinguish heavy users, and quickly identify network abnormalities such as a virus exploit. The quality-of-service (QoS) provided by the traffic manager also relies heavily on the statistical information provided by the counters. For example, counters are essential in providing committed access rate (CAR) support, such as committed information rate (CIR) and peak information rate (PIR) statistics, so that a traffic manager can offer differentiated services to different flows based on their service level agreements (SLA) of their fair-share bandwidth and the amount of bandwidth they are actually utilizing. Statistics counter arrays are also crucial for database, data mining, and other related applications, where statistical information needs to be collected and managed.

Various types of statistical information can be stored using counters. For example, each counter can be used to track the total packet size from a single flow, which is identified by the same flow record, such as the 5-tuple for an IP flow. In such applications, each arriving packet will trigger at least one counter update. On a 40-Gb/s OC-768 link with minimum packet size of 64 bytes, a new packet can arrive every 12.8 ns, and its corresponding counter updates need to be completed within this time frame. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows as smaller counters can quickly overflow. For other applications, such as to provide CAR support, counters can be utilized to keep track of the leftover bandwidth for a specific type of traffic, which can be implemented at flow, user group, virtual queue, port, or other service levels. For the packets belonging to the same traffic type, the corresponding counter values will be decremented according to the packet sizes to reflect the leftover bandwidth. On the other hand, the counter values need to be incremented periodically to reflect the fair-share bandwidth allocated to the traffic type for the next service period. It is easy to see that the bandwidth can be more accurately managed if the service period is made arbitrarily small and the amount of increments to the counters in each service period are reduced accordingly. Thus, it is highly desirable to building counters that support both increment and decrement operations at high speed. Moreover, a practical

• H. Wang is with Oracle Corporation, 400 Oracle Pkwy 4OP947, Redwood City, CA 94065. E-mail: wanghao.ucsd@gmail.com.
• B. Lin is with the University of California, San Diego, 9500 Gilman Drive, Mail Code 0407, La Jolla, CA 92093. E-mail: billlin@ece.ucsd.edu.
• J. Xu is with the Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332-0280. E-mail: jx@cc.gatech.edu.

Manuscript received 2 Nov. 2011; revised 20 Jan. 2012; accepted 26 Sept. 2012; published online 3 Oct. 2012.
Recommended for acceptance by S. Ranka.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-11-0813.
Digital Object Identifier no. 10.1109/TPDS.2012.287.

counter array solution needs to be able to deal with any arbitrary incoming sequence of counter update requests, especially for security applications such as intrusion detection where an adversary has incentives to compromise any performance guarantees provided by the statistics counter array system.

Even though static random access memories (SRAMs) can satisfy the performance needs for counter arrays, the amount of SRAM required is often both infeasible and impractical. There can be millions of different flows during a measuring period for real-world Internet traffic traces [3]. 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 just for the counter storage, which is prohibitively expensive. The state-of-the-art commercial SRAM holds up to 72 Mb [4], with a random access latency of 3.3 ns, power consumption of 1.8 W, and price of over $200. To build such a statistics counter array, it would require 12-SRAM devices and consume approximately 21.6 W of power. Moreover, the required number of SRAM devices and the power consumption grow linearly as the number of flows grows, which makes it less scalable. On the other hand, dynamic random access memories (DRAMs) provide much larger capacity and consume significantly less power. The state-of-the-art commercial DRAM chip provides 4 Gb of storage [5] and only consumes 1.76 W of power. As of this writing, 4 GB of DRAM consisting of eight chips costs under $20, over 200 MB/$. However, it has a random access latency of 36 ns, which is too slow to catch up with the line rate. Therefore, alternative ways to implement large arrays of statistics counters at high speed have received considerable research attention.

Hybrid SRAM/DRAM counter architectures have been proposed [1], [2], [3], [6]. The basic idea of these schemes is to divide the storage of counters into two parts: the lower order bits and the whole counter. The lower order bits are stored in SRAM while the whole counters are stored in DRAM. Before the lower order bits of a counter overflow, the value is sent to DRAM to update the whole counter value. The hybrid SRAM/DRAM architectures substantially reduce the SRAM requirements; however, they still require at least 10 MB of SRAM, which is significant more than the amount of SRAM required using the scheme developed in this paper. If the counters are updated by larger counter value increments, more SRAM will be required in the hybrid SRAM/DRAM schemes to store more lower order bits of the counters in SRAM to maintain a low-overflow probability. Simple calculation shows that if the average counter value increment is S instead of 1, then $\log_2 S$ extra bits are needed for each counter to maintain the same overflow probability, which can be a substantial increase in the overall SRAM requirement if S is large. In general, such approaches need large SRAM to support arbitrary increments. Moreover, they are based on an integer number representation, which makes them unsuitable for applications that require floating point number representations [7], [8].

Randomized DRAM architecture has been introduced [9] to harness the performance of modern commodity DRAM offerings by interleaving counter updates to multiple memory banks. A random permutation function distributes the arriving counter updates across memory banks so that memory accesses to different banks can be near-perfectly balanced without the presence of adversarial traffic. For each memory bank, there is a small request queue to buffer pending updates. In case there are adversarial traffic generating repetitive memory accesses, a cache module is introduced, which is located before the request queues. An arriving counter update request will first be compared with the other requests stored in the cache. If there is a match, the matched entry in the cache will be updated to reflect the new counter update. In case there is no match, an existing entry in the cache will be evicted to store the new counter update based on a certain cache update policy. The evicted entry will then be inserted into the corresponding request queue based on the DRAM bank it is mapped to using a random permutation function. The randomized DRAM architecture in [9] is designed to accommodate the worse case memory access patterns so that the overflow probability in the request queues is bounded by a small value even under the worst case arriving traffic. It is worth noting that such a design is not optimized for traffic from real-world Internet traces, so that it yields a rather small request queue size and a large cache, which is inefficient considering that overflows can only occur in the request queues but not in the cache.

In this paper, we propose a DRAM-based architecture that combines the functionalities of both the cache and request queues to enhance the system performance and reduce the overflow probability for real-world Internet traffic. Our architecture is based on the observation that most flows on the Internet consist of multiple packets that are transmitted during a relatively short period of time. We refer to the packets from the same flows received within the short period time as traffic bursts. In a statistics counter tracking accurate flow statistics, each arriving packet triggers an update to the corresponding counter. Thus, traffic bursts would trigger repetitive updates to the same counters within a short time, which can drastically increase the traffic loads to the memory banks storing the counter values and further cause system overflow. In our new architecture, we turn such traffic bursts to our advantage and effectively reduce the traffic loads to the memory banks by merging the repetitive counter updates in the request queues. Our proposed architecture makes use of a simple randomization scheme and small fully associative request queues to statistically guarantee a near-perfect load balancing of counter updates to the memory banks. To mitigate the congestion caused by traffic bursts, an incoming counter update is merged if there is a matched counter update in the request queues to the same memory address. In case there is no matched update in the request queues, the new update is inserted into a request queue based on the random permutation function. The counter update requests in the request queues are constantly sent to DRAM banks to update the corresponding counter values. In choosing a counter update to be sent to the memory banks from a request queue, we present two request queue update policies: first-in-first-out (FIFO) and least-recently-used (LRU). We show that with a simple merging operation supported by the request queues and a request queue update policy, we can effectively resolve the adversarial access patterns of receiving bursts of counter updates to the same memory locations. We show that for the worst case traffic patterns, such as the ones can be potentially triggered by external threats, our new architecture can perform at least as good as the random counter scheme in [9]. For real-world Internet traffic, we develop queuing models to show that as long as there is some traffic bursts in the arrival, the maximum request queue length is always bounded by a finite number. We also provide simulations results using Internet traffic traces to confirm the effectiveness of our queuing models. In summary, our proposed statistics counter architecture can effectively maintain line rate updates to large counter arrays while providing a diminishing overflow probability.

1.1 Summary of Contributions

We make several contributions in this work:

- We propose a DRAM-based statistics counter architecture that can take the advantage of traffic bursts to reduce the work load to DRAM banks and reduce the chance of system overflow. Traffic bursts which would have caused more accesses to DRAM banks can actually reduce the effective traffic loads to our counter system.
- In our counter architecture, we use request queues with merging capability to temporarily buffer pending counter updates. Unlike previous work in [9], there is no separate cache. We show that by designing our request queues as large as the size of the cache in [9], we can provide the same worst case performance guarantee. Since overflow occurs only in the request queues, the overflow probability in our new architecture is lower than that in [9] due to the larger request queue sizes.
- We provide queuing models to show that as long as there are traffic bursts in the arriving traffic, the request queue size to avoid system overflow is always bounded by a finite number.
- We provide simulations using real-world Internet traces to analyze the request queue size requirement. The results of the experiments closely match the bounds predicted by our queuing models.

The rest of the paper is organized as follows: Section 2 introduces the related work. Section 3 defines the notion of an SRAM emulation. Section 4 describes our proposed randomized counter architecture in details. Section 5 analyzes the performance of our architecture with different traffic models. Section 6 compares our design with previous proposed robust counter design to show that ours outperforms the previous one while using less SRAM. Section 7 presents simulation results using Internet traffic traces. The results predicted by our models closely match the simulation results. Finally, Section 8 concludes the paper.

2 Related Work

The hybrid SRAM/DRAM [1], [2], [3], [6] and randomized DRAM architecture [9] shown in Section 1 belong to the passive counter category, where a DRAM access is required to retrieve the value of a counter. They have the advantage that the counter values are accurately stored in DRAM banks. However, they suffer from the drawback that the retrieval of counters stored in the same DRAM bank would experience DRAM access latency, even though counters stored in different DRAM banks can be accessed independently.

The other type of counter design is the active counter category. There are several schemes [10], [11], [12], [13], [14], [15], [16], [17], [18] belonging to this category, where counter values are only approximated to a certain extent. They share in their approaches that large counter arrays are compressed in certain ways to be stored using only SRAM with the help of efficient counter representations, while the accuracy is sacrificed to keep the cost low. They only differ in the way to represent and compress the counter values to recover and approximate the true counter values at the time of counter retrieval.

One such approach is approximate counting [10], [11], [12], [13], [14]. The idea is to probabilistically increment a counter based on the current counter value. Such designs in general suffer the problem of having large error margins when the possible estimation values are sparsely distributed, which renders them only applicable to those network measurement and data streaming applications that can tolerate large inaccuracies. It is worth noting that many approximate counting designs have been built into products due to their significantly lower cost, such as the Netflow from Cisco [15] and the filter-based accounting from Juniper [16]. A second approach is counter braids [17], which was inspired by the construction of low-density parity-check codes. At each packet arrival, counter increments can be performed quickly by hashing the flow label to several counters and incrementing them. Flow counts can be decoded through an iterative decoding process to recover the actual counter values. A counter value can be recovered with a certain probability during the decoding process. It is worth noting that the decoding process incurs significantly longer delay than a DRAM access, thus it is preferably processed offline. A third approach is based on an efficient variable-length counter representation called BRICK [18]. It uses a simple operator called rank-indexing to link together counter segments rather than using expensive memory pointers.

In all SRAM-based active counter designs above, significant amounts of SRAM are still necessary for very large counter arrays (say for tens of millions of counters). In contrast, the designs in the passive counter category 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 [19], [20] are also applicable for the maintenance of large counter arrays. However, they are not as efficient and incur longer delays compared to the memory systems specialized for this purpose.

The idea of using interleaved DRAM banks to achieve higher throughput is also explored in several packet buffer designs [21], [22], [23]. In packet buffers, packets are written to multiple DRAM banks following a certain order and later retrieved according to their departure times. In each cycle, only one packet can be written to or read from a DRAM bank to avoid bank conflicts. The problems of managing counters in a statistics counter array and managing packets in a packet buffer are fundamentally different. For a packet buffer design, the goal is to find one memory bank 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 and predetermined memory locations. It has to be guaranteed that any counter update requests can be successfully processed by a predetermined memory bank.

3 Ideal SRAM Emulation for Statistics Counter

Memory interleaving has been successfully developed for boosting the performance of computer systems [24], [25], [26], for speeding-up graphics or video intensive applications [27], and for implementing routing functions such as high-performance packet buffers [21], [22], [23]. In this section, we introduce a statistics counter architecture that can emulate a fast SRAM by exploiting memory interleaving using multiple DRAM banks. The definition of an SRAM emulation was first introduced in [19] for general memory systems, where both memory read and write operations are supported. For statistics counters, an SRAM emulation only need to support frequent memory update (write) operations and infrequent read operations to retrieve data from the counters. Memory updates should be operating at line rate, while data retrievals occur less frequently by several orders of magnitudes. For certain applications, data can even be processed offline for further analysis, such as in the Internet traffic size distribution analysis. We assume that time is slotted. A time slot is defined as time for a minimum sized packet to finish arriving at line rate. For example, on a 40-Gb/s OC-768 link with minimum packet size of 64 bytes, it takes 12.8 ns for a minimum sized packet to be received in whole. So a new packet can arrive at most once every 12.8 ns. We define this period of time as a time slot. Thus, at most one packet can arrive to our system in a time slot, which would trigger one counter update request to the statistics counter array. Suppose it takes b cycles for a DRAM bank to process one counter update request, while it takes only one cycle for SRAM to process the same request. An SRAM emulation that mimics the behavior of an ideal SRAM for statistics counter array is defined as follows:

Definition 1 (Emulation). *A statistics counter architecture using slower memories is said to* emulate *an ideal SRAM counter array if it can admit with high probability the arriving counter update patterns that are admissible to an ideal SRAM counter array. All the admitted counter updates are processed within a bounded delay.*

It is easy to see that an SRAM emulation guarantees the following semantics:

- *High throughput*. An emulated SRAM counter array can work at the same line rate and approximate the throughput of an ideal SRAM counter array.
- *Small drop rate (outage)*. In an emulated SRAM counter array, the counter update requests are dropped with diminishing probability. Preferably, the drop rate is zero for all incoming counter update patterns just as in an ideal SRAM counter array system.
- *Bounded delay*. The counter updates are processed within a bounded period of time once they are admitted into the system.

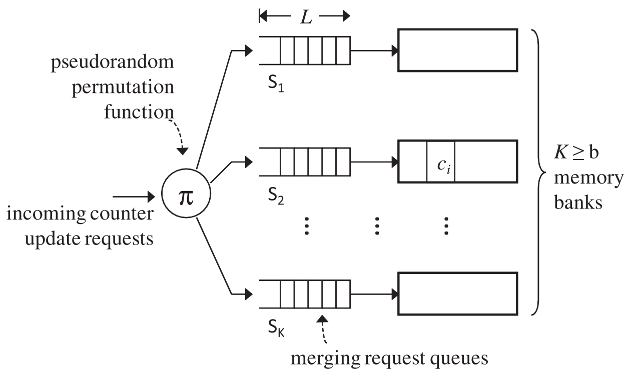


Fig. 1. Statistics counter array architecture.

4 Our Design

Our proposed statistics counter architecture is shown in Fig. 1. There are K interleaving DRAM banks with $K > b$ to match the SRAM throughput. There is one request queue for each memory bank to temporarily store the counter update requests waiting to be processed by the memory banks. The request queues are implemented as fully associative caches that employ certain *cache update algorithm*. We implement a random permutation function to distribute incoming counter updates into the K request queues. In the following, we present the detailed descriptions for each module in the figure.

4.1 Architecture

- *Pseudorandom Permutation Function* π. A pseudorandom permutation function is implemented to randomly distribute memory locations so that counter update requests to different counter locations are spread into K memory banks uniformly and at random (u.a.r) with equal probability $1/K$. Unless otherwise noted, when referring to a counter address, we will be referring to the address after the pseudorandom permutation. Note that incoming counter updates accessing the same memory address will be mapped to the same request queue even with the presence of the pseudorandom permutation function, because each counter is only stored in a fixed location in the DRAM banks.

  The pseudorandom permutation function can be any linear permutation function that randomly "shuffles" counter locations inside the DRAM banks. However, once chosen, the function needs to be kept fixed for an extensive period of time, because counters have to be stored at deterministic locations in the memory banks and the cost of moving them from one memory location to another is both expensive and time consuming. However, the pseudorandom permutation function can be changed infrequently, to say once in several minutes, to provide robustness against probe-response attempts by the adversaries trying to figure out the function.

Also, the pseudorandom permutation function has to be kept unknown to the incoming traffic, so that an adversary cannot purposely unbalance the loads to different banks by exploring the counters that are stored in any memory bank. However, it is still possible for an adversary to purposely trigger updates to the same memory bank by sending repetitive counter update requests to the same counter locations, because such requests will be processed by the same memory bank eventually.

- *DRAM Banks*. There are K DRAM banks in the counter array to match the SRAM throughput, where $K > b$ and b is the number of time slots it takes for a DRAM bank to finish processing one counter update request. We assume that a bank can process only one counter update at a time. Therefore, for any counter update arriving to a memory bank, it will keep the bank *busy* for the next b cycles before another update can be processed by the same bank.
- *Request Queues*. There are K request queues, one for each memory bank. The request queues are denoted from S_1 to S_K. They temporarily store counter update requests waiting to be processed in the DRAM banks. Each entry in the request queues contains information of the counter index and the accumulated counter update. For a counter array supporting 16 million counters, the unique counter index for each counter can be set to be of size 3 bytes. The request at the head of a request queue will be sent to the corresponding memory bank when the bank is not busy.

  To make our counter architecture robust against adversarial counter update patterns, we require each request queue to be implemented as a fully associative cache that supports fast query and update operation. Once a new counter update enters a request queue, a query is initiated in the queue using the address of incoming update as the key. If there is a match in the request queue, the new counter update will not generate new entry in the request queue. Instead, the matched counter update request in the request queue is updated with the new request using simple arithmetic operations. We call this step a *merge* operation. We assume that the request queues can finish three operations in one time slot: a query operation, an insert operation (merging or creating a new entry), and a removal operation to send a request already in the request queue to the corresponding memory bank when the bank is not busy. On a 40-Gb/s OC-768 link with minimum packet size of 64 bytes, a time slot equals 12.8 ns. Given that the SRAM random access latency is about 3.3 ns, the three memory operations can be fit into one time slot in the SRAM. So the request queues are capable of processing arriving counter update requests at the line rate. Therefore, in each time slot, one new request may enter a request queue and another request in the queue may depart to update the corresponding bank. It is worth noting that even though by pipelining the three operations the system can achieve higher throughput, pipelined design is not required for implementing our statistics counter array to operate at line rate. Several cache update algorithms can be implemented in the request queues. A simple FIFO rule always chooses the earliest request in a request queue. LRU cache update policy can also be adopted to remove the request with the oldest update time in a request queue. In Section 7, we will show that LRU rule is slightly more efficient at merging incoming counter updates and reducing the loads in the request queues, even though it is more complex than the FIFO rule.

We will provide rigorous analysis on the performance of our statistics counter architecture in Section 5 to show that even with diminishing merging probability in the request queues, the total occupied queue size is always bounded, as long as there are traffic bursts in the arrival requests, and the flow sizes follow a certain heavy-tailed distribution. Such a result is in contrary to the general M/D/1 queuing model, where the queue length can grow to any size with nonzero probability no matter what the arrival rate is. We also provide simulation results using real-world Internet traces for our architectures implementing FIFO or LRU request queue update policy. Note that the new design is different from a general purpose CPU design even though as shown later we have significantly reduced the total SRAM requirement to make the request queues small enough for CPU cache. In our design, different request queues process counter updates concurrently, which require independent memory controller, to achieve line rate operations. Such a design is not supported by a general purpose single CPU.

4.2 Why Merging Request Queues Are Necessary?

With the help of a pseudorandom permutation function, we can safely assume that the counter update requests are distributed into the memory banks uniformly at random when the arriving requests are in the absence of adversaries. So, we expect that each memory bank receives roughly an equal share of the total requests in the long run when there is no adversaries. However, in a short period of time, the counter update requests to the DRAM banks may not be balanced due to the Birthday Paradox [28]. Therefore, there is no guarantee that the arriving counter updates are distributed into the memory banks truly uniformly at any moment, such as in a round-robin pattern. So request queues are necessary to temporarily store counter update requests pending to access the memory banks. Moreover, certain adversarial counter update patterns may further overload a memory bank with bursty arrivals. For example, although the random permutation function is unknown to attackers so that they cannot figure out how the counter addresses are mapped to DRAM banks, an attacker can still trigger repeated updates to the same DRAM bank by sending packets with the same flow record, which will generate counter updates to the same counter stored in the same memory location. Therefore, the number of pending updates in a request queue may grow indefinitely. Flows containing large files being transferred over the network may also cause sudden increase in the arriving packets with the same flow record in a short period of time, which may overwhelm a fixed sized request queue it is mapped to. To mitigate such situations, in our design it is necessary to have the request queues merging incoming counter update requests with the ones in the request queue with the same flow record. We will show that our statistics counter outperforms the recent proposed robust counter design in [9] under both adversarial and nonadversarial arrivals.

5 Performance Analysis

In this section, we provide a queuing model for our statistics counter with merging request queues. We will show that with a reasonable merging probability assumption, the maximum queue size is always bounded.

5.1 Preliminaries

We assume in each cycle at most one counter update request arrives. Even though in a multicore multiport network processor (NP) systems more counter updates may arrive, which would complicate the analysis, it can be easily handled by using an input queue to temporarily buffer the arrival traffic and provide an admissible arrival to the system. With $K \geq b$, we can ensure that the counter array is rate-stable, which means that the arrival rate to the counter array is no larger than the total throughput. More specifically, the maximum arrival rate to the counter array is at most one request per cycle, while the rate of requests leaving all the request queues, i.e., the service rate can be as large as $K/b \geq 1$ when all of the K queues are nonempty. We adopt the terminologies in [21] to first define the arrival and departure processes for a request queue. Then, we will provide detailed analysis of our queuing model with merging request queues.

Let us assume that time is slotted. We have defined the smallest unit of time as a slot or a *cycle*. Thus, the following analysis will be carried out in the discrete time domain. We assume that there are a total of N distinct counters in the DRAM banks, so counter update requests are mapped to at most N different memory locations. We denote by $A[t]$ the cumulative number of requests arriving at the statistics counter during cycles $[0, t]$, and $A_j[t]$ the cumulative number of requests for counter j during the same time $[0, t]$. The arrival processes of counter update requests $A_j[t]$ can be shown to be stationary and ergodic [21], [29]. So, we have

$$A[t] = \sum_{j=1}^{N} A_j[t]. \tag{1}$$

We denote the arrival rate to counter j as λ_j such that the overall arrival rate is not larger than 1, so

$$\sum_{j=1}^{N} \lambda_j \leq 1. \tag{2}$$

We also assume $A_j[t]$ to be independent of each other. This assumption is based on the fact that in the statistics counter array each counter keeps track of a specific flow (or a specific group of flows). On the Internet, traffic consists of millions of flows generated by thousands of independent sources. The counter update requests generated by these flows can be assumed to be independent processes.

We denote by $RA^i[t]$ the cumulative number of requests arriving at request queue S_i during cycles $[0, t]$. Each incoming counter update request is processed by the pseudorandom permutation function and assigned to one of the K request queues uniformly at random. We denote by π the pseudorandom permutation function. So the following should hold:

$$RA^i[t] = \sum_{\pi(j)=i} A_j[t]. \tag{3}$$

Each counter update request is assigned to one of the request queues. Each request queue contains distinct counter update requests that are not in any other request queues. Since $A_j[t]$ are independent of each other, $RA^i[t]$ are independent processes. For simplicity, we assume that $A_j[t]$ are independent and identically distributed (i.i.d.) random processes. Later in the section, we will deal with the cases when the assumption does not hold.

When $A_j[t]$ are i.i.d. and distributed into K request queues uniformly at random, the arrival process to a request queue S_i is also i.i.d. For a request queue S_i, its arrival rate λ^i is

$$\lambda^i = \frac{\sum_{j=1}^{N} \lambda_j}{K} \leq \frac{1}{K}, \tag{4}$$

which states that due to the pseudorandom permutation function, each request queue receives about $1/K$ fraction of the total counter updates.

Similar to the arrival processes, we denote by $D[t]$ the cumulative number of departures from the request queues during cycles $[0, t]$. Each one of the request queues is being serviced deterministically at a rate of $\mu^i = 1/b$, because it takes a DRAM bank b cycles to finished updating one counter. Combining all the K request queues, therefore, the overall maximum departure rate for the statistics counter array is K/b.

5.2 Union Bound—System Overflow Probability

Our statistics counter array overflows when any one of the K queues overflows. Since the arrivals to request queues are i.i.d., it is reasonable for us to set the request queues to be of equal length L. We denote the counter array system overflow probability as $P_{overflow}$ and the overflow probability for a request queue S_i of length L as $P^i_{overflow}(L)$. Thus, we have

$$P_{overflow} \leq \sum_{i=1}^{K} P^i_{overflow}(L), \tag{5}$$

which is the union bound.

Therefore, it suffices to analyze the overflow probability of one request queue. We start by obtaining the steady-state probability of the occupancy for a request queue of infinite size exceeding L as a surrogate for the overflow probability of a finite request queue of size L. The reason is that this surrogate is shown to be an upper bound to the actual overflow probability [21], [29]. Let ℓ be the number of pending requests in the request queue S_i, then

$$P^i_{overflow}(L) \leq P(\ell^i > L). \tag{6}$$

Combining (5) and (6), we have

$$P_{overflow} \leq \sum_{i=1}^{K} P(\ell^i > L). \tag{7}$$

Since the arrival processes to the request queues are i.i.d., and the request queues are serviced deterministically at the same rate $1/b$, we expect the occupancies $\ell^i(t)$ of any request queue at cycle t to be i.i.d. Therefore, we can drop the superscript i and rewrite (7) as

$$P_{overflow} \leq K \times P(\ell > L). \tag{8}$$

Therefore, in the following we will analyze the probability that the number of requests ℓ in a request queue of infinite size exceeds L, and use (8) to bound the counter array system overflow probability.

5.3 Request Merging Probability AMP

The merging request queues in the statistics counter are implemented as fully associative caches. In case of merging, the incoming request does not generate any new entry in the request queues. We define the *average merging probability* (AMP) function as $\Theta_{\mathrm{amp}}(\ell)$, which is the probability that an incoming counter update request is merged into a request queue of length ℓ upon its arrival. The function $\Theta_{\mathrm{amp}}(\ell)$ has the following properties:

1. $\Theta_{\mathrm{amp}}(0) = 0$;
2. $\Theta_{\mathrm{amp}}(\ell) > 0$, for $\ell > 0$;
3. $\Theta_{\mathrm{amp}}(\ell_i) \leq \Theta_{\mathrm{amp}}(\ell_j)$, for $0 \leq \ell_i \leq \ell_j$.

Property 1 states that the merging probability is zero when the request queue is empty. Property 2 states that the merging probability is positive when the request queue is nonempty. Property 3 is from the fact that as the occupancy of a request queue increases, on average it is more likely to find a request in the request queue that has the same address as the incoming request. We will analyze the behavior of a request queue given a specific AMP function. Our model can be regarded as a general birth-death process with variable birth rate and a fixed death rate. Here, a birth is the event of a counter update request creating a new entry in a request queue, and a death is the event that an entry in a request queue is removed to update the corresponding memory bank.

Now, let us derive the steady-state probability of the occupancy of a request queue. For simplicity, we denote by λ the arrival rate and μ the service/departure rate of a request queue. For each request queue, we have the following:

- *arrival rate*: $\lambda \leq 1/K$.
- *service rate*: $\mu = 1/b$.
- *AMP*: $\Theta_{\mathrm{amp}}(\ell)$, where ℓ is the queue length observed by an arrival request.

To develop the worst case overflow probability of a request queue, we assume that the arrival rate is at its maximum value with $\lambda = 1/K$. We denote by P_ω the steady-state probability of a request queue with ω occupied entries (denoted as state ω). The balance equations for our model are as follows:

$$\lambda P_0 = \mu P_1; \tag{9}$$

$$\begin{aligned}(\lambda - \Theta_{\mathrm{amp}}(\omega) + \mu)P_\omega = (\lambda - \Theta_{\mathrm{amp}}(\omega-1))P_{\omega-1} \\ + \mu P_{\omega+1}, \quad \omega > 0.\end{aligned} \tag{10}$$

The above equations are due to the fact that at the steady state, the rate at which a request queue leaves a state ω should be the same as the rate it enters the state. In (9), the request queue can leave state 0 only by an arrival, because clearly there cannot be a departure when the queue is empty. Likewise, the request queue enters state 0 only by departure of one entry from state 1. On the left of (10), the request queue leaves state $\omega > 0$ by one arrival or one departure, minus the cases when it is merged upon arrival. On the right side of (10), the request queue enters state ω by one arrival when it is at state $\omega - 1$ if it is not merged, or by a departure when it is at state $\omega + 1$.

By solving (9) and (10), we get the steady-state probabilities of the system as

$$P_\omega = \frac{\prod_{i=0}^{\omega-1}(\lambda - \Theta_{\mathrm{amp}}(i))}{\mu^\omega} P_0, \quad \omega > 0. \tag{11}$$

To understand how request merging helps reducing the request queue length, let us compare the steady-state probabilities of a system with request merging with a system without request merging. Let us denote the steady-state probabilities of a system without merging as Q_ω, where ω is its queue length at state ω. For the fairness of comparison, we assume the same arrival rate λ and service rate μ, so

$$\lambda Q_0 = \mu Q_1 \tag{12}$$

$$(\lambda + \mu)Q_\omega = \lambda Q_{\omega-1} + \mu Q_{\omega+1}, \quad \text{for } \omega > 0. \tag{13}$$

This is an M/D/1 queuing model with steady-state probability $Q_\omega = (\frac{\lambda}{\mu})^\omega Q_0$ [30].

Theorem 1. *$\exists\, \beta > 0$, such that $P_\omega < Q_\omega$, $\forall \omega > \beta$.*

Before proving Theorem 1, we present the following lemmas.

Lemma 1. *$P_{\omega+1} \leq \frac{\lambda}{\mu} P_\omega$, $\forall \omega \geq 0$.*

Proof. We know that

$$P_{\omega+1} = \frac{\lambda - \Theta_{\mathrm{amp}(\omega)}}{\mu} P_\omega, \quad \text{for } \omega \geq 0 \tag{14}$$

$$\Theta_{\mathrm{amp}} \geq 0. \tag{15}$$

Therefore, $P_{\omega+1} \leq \frac{\lambda}{\mu} P_\omega$. □

Lemma 2. *$P_0 \geq Q_0$.*

Proof. From Lemma 1, we have

$$P_\omega \leq \left(\frac{\lambda}{\mu}\right)^\omega P_0, \quad \forall \omega \geq 0. \tag{16}$$

So,

$$1 = \sum_{\omega=0}^{\infty} P_\omega \le \sum_{\omega=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^{\omega} P_0. \tag{17}$$

Thus,

$$P_0 \ge \frac{1}{\sum_{\omega=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^{\omega}} = Q_0. \tag{18}$$

□

Now, we shall prove Theorem 1.

Proof. From Lemma 2, we have $P_0 \ge Q_0$. We claim that there exists β such that $P_\beta \le Q_\beta$, for $\beta \ge 0$. It clearly holds when $P_0 = Q_0$. Let us consider when $P_0 > Q_0$. If $\forall\, i$, $P_i > Q_i$, then

$$\sum_{\omega=0}^{\infty} P_\omega > \sum_{\omega=0}^{\infty} Q_\omega.$$

However, we know $\sum_{\omega=0}^{\infty} P_\omega = 1 = \sum_{\omega=0}^{\infty} Q_\omega$, which is a contradiction. So there is β such that $P_\beta \le Q_\beta$. Let us consider the smallest β. So we have that $P_k > Q_k$, where $0 \le k < \beta$. We need to show that starting from β we have $P_j \le Q_j$ for $\forall j \ge \beta$, which can be proved as follows:

$$P_{\beta+1} = \frac{\lambda - \Theta_{\mathrm{amp}}(\beta)}{\mu} P_\beta. \tag{19}$$

Thus,

$$\begin{aligned} P_{\beta+1} &\le \frac{\lambda - \Theta_{\mathrm{amp}}(\beta)}{\mu} Q_\beta, \quad \text{since } P_\beta \le Q_\beta, \\ &< \frac{\lambda}{\mu} Q_\beta = Q_{\beta+1}. \end{aligned} \tag{20}$$

Using the same argument, we have $P_{\beta+2} < Q_{\beta+2}$, and so on. Therefore, $P_j < Q_j$ for $\forall j > \beta$, which proves Theorem 1. □

Theorem 2. *If $\Theta_{\mathrm{amp}}(l)$ is an increasing function of l and upper bounded by 1, then there exits γ such that $\forall \omega > \gamma$, $P_\omega = 0$.*

Proof. The AMP function $\Theta_{\mathrm{amp}}(l) \le 1$. From Lemma 1, we know if $\lambda - \Theta_{\mathrm{amp}}(\omega) \le 0$, then $P_{\omega+1} \le 0$. Since $\Theta_{\mathrm{amp}}(l)$ is a nonnegative increasing function and $\lambda \le 1$, then for any fixed λ, there exits γ such that $1 \ge \Theta_{\mathrm{amp}}(\gamma) \ge \lambda$. Thus $P_{\gamma+1} \le 0$. However, since $P_{\gamma+1}$ is the steady-state probability of request queue of length $\gamma + 1$, $P_{\gamma+1} \ge 0$. Therefore, $P_{\gamma+1} = 0$, which means the request queue will never grow to a length $\gamma + 1$. As a result, the request queue will never grow beyond length $\gamma + 1$, which leads to $P_\omega = 0$, $\forall \omega > \gamma$. □

For a statistics counter array, it is intuitive to see that as the number of requests in a request queue increases, the probability for an arrival request accessing the same memory address as one of the request already buffered in the request queue increases. Therefore, we expect the lengths of the request queues to be bounded when request merging occurs with request merging probabilities following properties 1, 2, and 3.

5.4 AMP for Internet Flows

Mergings in the request queues occur when an arriving update and an update in the request queues are accessing the same counters with the same memory addresses. Traffic bursts caused by multiple packets from the same flows being sent across the network from senders to receivers during a short period time contribute greatly to the mergings in the request queues. On the Internet, the flow sizes (in the number of packets) of the traffic have been known to follow certain heavy-tailed (or long-tailed, power-law tailed) distribution [31], [32]. Unlike Gaussian distribution, a heavy-tailed distribution has no "typical" scale and is, hence, frequently called "scale-free." Such a feature implies that for any fix-sized time window, the flow size distribution of all active flows still follows a certain heavy-tailed distribution. For a counter array maintaining flow statistics, such type of distribution would lead to frequent updates to the same counters, which would necessarily be mapped to the same memory banks. More update requests may be generated by traffic bursts than the throughput of a counter array during even a relatively short period of time, due to the "scale-free" nature of the heavy-tailed distribution, which would cause a system with fix-sized request queues to overflow.

To solve the problem above, in our statistics counter array with merging request queues, traffic bursts within a short time window can be successfully combined in the request queues without generating extra accesses to memory banks. In this section, we will analyze the AMP in a request queue under the assumption that the flow sizes follow a certain heavy-tailed distribution.

Let X be variable representing the flow size. The flow size distribution on the Internet is heavy-tailed and it can be expressed as follows [31], [32]:

$$\mathtt{Prob}(X > x) \sim x^{-\alpha}, \quad \text{as } x \to \infty, \tag{21}$$

where X is a random variable representing the number of packets in a flow, and α is a shape parameter. For heavy-tailed distribution, we have that $0 < \alpha < 2$ and the distribution has infinite variance. The shaper parameter α determines how frequently large flows appear. A smaller α means that the distribution has larger tail and large flows appear more frequently, which would result in smaller request queue size requirement. As α becomes larger, large flows appear less frequently, which would result in larger request queue size requirement for the same amount of arriving traffic because request merging becomes less likely.

If the flow sizes on the network follow the heavy-tailed distribution in (21), then the probability mass function for a flow to be of size $X = x$ packets can be expressed as

$$\mathtt{p}(X = x) = c\alpha x^{-\alpha-1}, \tag{22}$$

where c is a constant such that the sum of all the probabilities over X equals 1.

Let us assume that the size of the largest flow is F_{max}. The average number of active flows within a time window T is defined as N_{active}. For a randomly chosen packet arrived within the same time window T, the probability that it belongs to a flow of size x is denoted as $\mathtt{Prob}_x$, which is as follows:

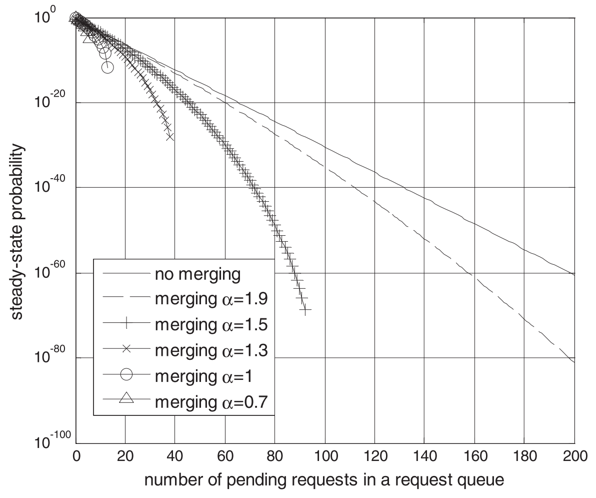


Fig. 2. Steady-state probability for merging request queue under heavy-tailed traffic with $b = 16$ and $K = 32$ banks.

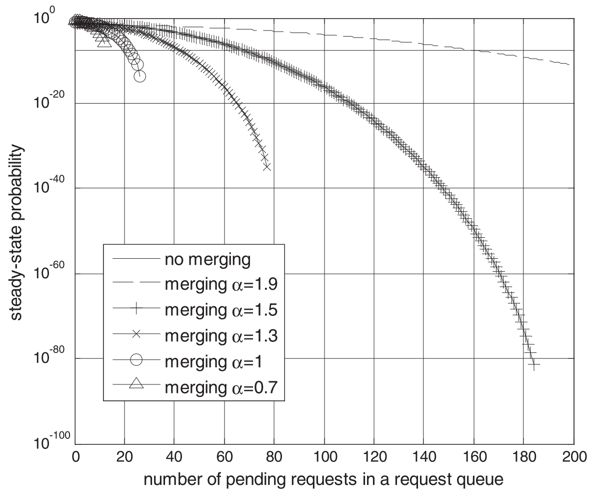


Fig. 3. Steady-state probability for merging request queues under heavy-tailed traffic with $b = 16$ and $K = 16$ banks.

$$\texttt{Prob}_x = \frac{x}{\sum_{y=1}^{F_{max}} y \cdot \texttt{p}(X = y) \cdot N_{active}}, \tag{23}$$

where $\texttt{p}(X = y) \cdot N_{active}$ is the total number of flows of size y within the time window T. The denominator in (23) is the total number of packets on the network during the time period T. Within the same time window T, the maximum number of total packets arrived is denoted as S. Then, the number of active flows can be expressed as

$$N_{active} = \frac{S}{\sum_{y=1}^{F_{max}} y \cdot \texttt{p}(X = y)}. \tag{24}$$

For a request queue currently containing ℓ pending requests, the probability that a newly arriving counter update request is merged with one of the ℓ requests in the request queue can be expressed as

$$\texttt{Prob}_{\text{merge}}(\ell) = \sum_{x=2}^{F_{max}} \ell \cdot \texttt{Prob}_x^2 \cdot \texttt{p}(X = x) \cdot N_{active}. \tag{25}$$

Equation (25) can be explained as follows: A new counter update request is merged with the one already stored in the request queue only if they are from the same flow and, thus, sharing the same counter address. For an arriving packet from a flow of size x, each pending request in the request queue is generated by a packet from the same flow with probability $\texttt{Prob}_x$. There are ℓ pending requests in the request queue. So the merging probability for a request generated by a packet from a flow of size x is denoted as $\ell \cdot \texttt{Prob}_x^2$. Among all the active flows, there are $\texttt{p}(X = x) \cdot N_{active}$ flows of size x. Merging is only possible for flows with more than two packets. Thus, to calculate the overall merging probability, the summation is from the flows of size two packets to F_{max} packets. Equation (25) is also the AMP as a function of the number of requests ℓ in a request queue.

The steady-state probability for a merging request queue under heavy-tailed traffic with DRAM-to-SRAM access latency ratio $b = 16$ and a total of $K = 32$ DRAM banks is shown in Fig. 2. It is easy to see from this figure that for request queues with no merging capability, the maximum queue length is unbounded, because the steady-state probability for the request queue to reach any queue length is positive. However, for our merging request queues under heavy-tailed traffic with $\alpha = 1.5$, the maximum queue length is bounded by about 100 entries. With $\alpha = 1.3$, the maximum queue length becomes 40. With $\alpha = 1$, the maximum queue length is bounded by less than 20 entries.

Fig. 3 shows the steady-state probability for a merging request queue under heavy-tailed traffic with parameters $b = 16$ and $K = 16$. For request queues with no merging capability, the maximum queue length is unbounded. The request queue can never reach steady state, because the arrival rate is the same as departure rate. On the other hand, for merging request queues under heavy-tailed traffic with $\alpha = 1.5$, the maximum queue length is bounded by about 185 entries. When $\alpha = 1.3$, the maximum queue length becomes 80. For heavy-tailed traffic with parameter $\alpha = 1$, the maximum queue length becomes only about 25.

6 Robustness against Adversaries

For real-world Internet traffic, the arriving processes $A_j[t]$ to different counters may not be i.i.d. Certain flows may trigger the corresponding counters to be updated more frequently than the others, which means that in our system, counter update requests to certain counters may arrive at higher probability than average. However, higher arrival flow rate generates more incoming requests in a short time; therefore, the requests are actually more likely to be merged to a request already in the request queue. The merging request queues effectively smooth the traffic bursts in the arrival process.

An adversary can only attack the system by sending the same counter update request as infrequently as possible, thereby effectively reducing the AMP in the request queues. In the robust statistics counter architecture in Fig. 4 [9], the cache module is separated from the request queues. The request queues only buffers the pending requests to memory banks. The cache module merges the repetitive requests to the same memory location. In [9], a worst case performance evaluation is provided. To achieve an overflow probability of 10^{-14}, the cache is of size about

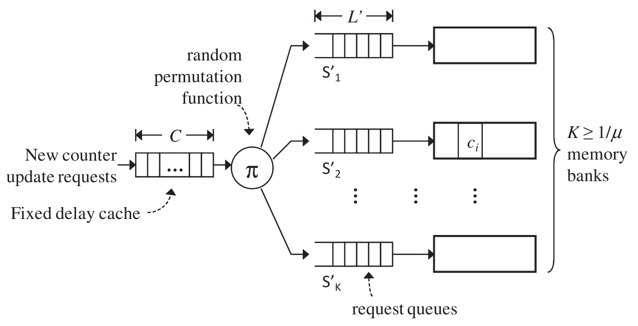


Fig. 4. Cache scheme in [9].

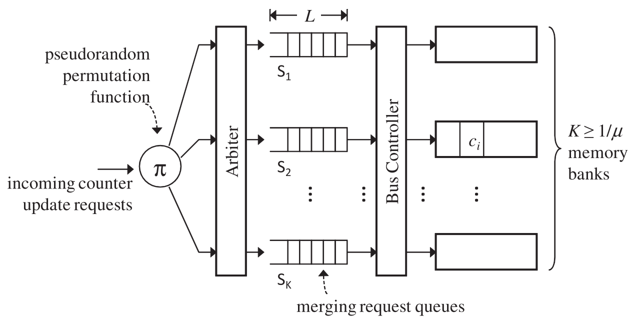


Fig. 5. Statistics counter array with shared request queues.

7,000 entries, each request queue has 50 entries, and a total of $K = 32$ memory banks. The key problem with such a design is that to lower the overflow probability for the worst case arrival traffic pattern (with adversaries), the performance of the system under regular traffic is not optimized. To compare our statistics counter array with merging request queues to the one in [9], let us consider the design where the total size of the request queues is the same as the size of the cache in [9] of 7,000 entries. Therefore, there are about 218 entries in each request queue. From Section 5.4, we can achieve zero overflow probability for arriving traffic with heavy-tailed distribution, while in [9] only achieves an overflow probability of about 10^{-14}, which is due to the much smaller request queue size in [9].

In the presence of adversaries sending repetitive counter updates trying to overflow the system, our statistics counter array can be modified to be more robust by allowing different DRAM banks to share their request queues as shown in Fig. 5. When one request queue is full and the arriving counter update cannot be merged to any entry in the request queue, the arbiter will pick another request queue to store the arriving update. Since a request queue can finish three operations in one cycle as stated in Section 4.1, including a query, an insertion, and a removal, query operations can be initiated instantaneously in different request queues to look for a match for the arriving counter update request. If there is a match in any request queue, the newly arrived request will be merged with the one in the request queue and no new entry will be generated in the request queues. The arbiter effectively snoops all the request queues for a match upon a request arrival. When a counter update request departs from a request queue, the bus controller decides which bank the request should be sent to based on its counter index. Therefore, the repetitive counter updates separated by up to $K \times L$ cycles can be merged by the request queues when the queues are almost full. When the request queues have empty entries, repetitive updates will not cause an overflow anyway. So the statistics counter with shared request queues performs at least as good as the scheme in [9] with the presence of adversaries. To see this, let us first consider that the best strategy for an adversary to attack is by sending same counter update requests every C cycles, which is proved in [9], where C is the size of the cache. All the repeated requests within C cycles will be merged to another request already in the cache. In Fig. 5, when the request queues are full, we effectively have a cache of size KL. All the repeated requests within KL cycles are merged. When the request queues are not full, then the arbiter will always be able to find an empty entry to store the request. By setting $KL = C$, we can achieve the same overflow probability as in [9] for the worst case traffic pattern. Moreover, the size of each merged request queue L is typically much larger than L' in Fig. 4; therefore, we also achieve lower overflow probability for incoming traffic in absence of adversaries, because system overflows only occur in the request queues.

7 Performance Evaluation

7.1 Traffic Traces

In this section, we present simulation results of our merging statistics counter architecture with 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. To test the performance of our statistics counter array under heavy traffic loads, we speed up the traffic traces so that in each time slot one counter update request arrives at the system.

7.2 Experimental Results

We are interested in the maximum queue length (occupancy) in the request queues for different queue update policies. In a statistics counter design, if the maximum queue length is bounded by a constant B for a specific traffic trace, then request queues with each queue of length B guarantee no overflow for the traffic trace. It is predicted by our queuing model that for request queues without merging operation, the maximum queue length is unbounded because the steady-state probability for any queue length is nonzero, no matter what the arrival rate of the incoming traffic is. On the other hand, for our merging request queue design, the maximum queue length is bounded, if the incoming traffic follows a certain heavy-tailed distribution.

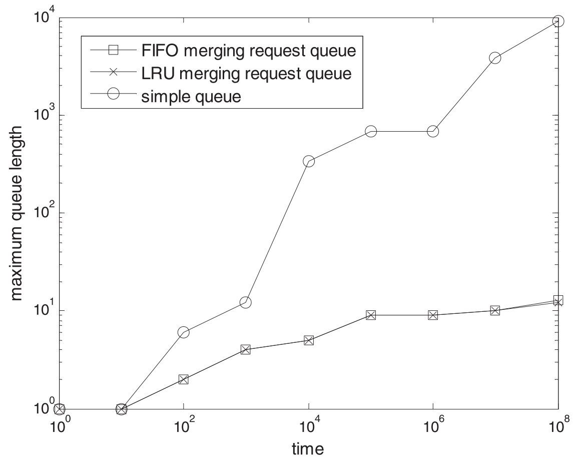


Fig. 6. Maximum queue length, $K = 32$, USC trace.

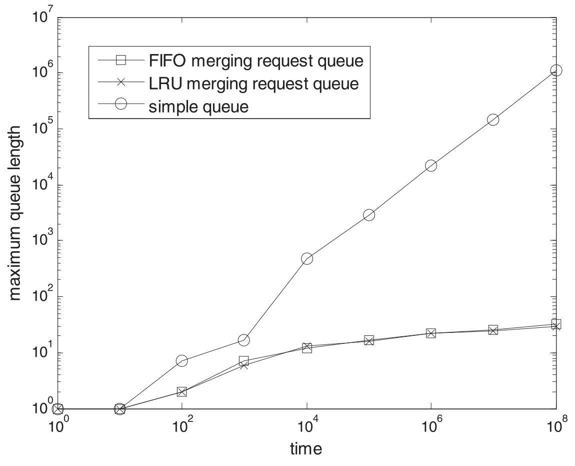


Fig. 7. Maximum queue length, $K = 16$, USC trace.

Typically, DRAM access latency is 10 to 20 times larger than the SRAM access latency. In our experiment, we assume that the SRAM is working at line rate, so that one counter update takes one cycle to process. On the other hand, it takes a DRAM bank b cycles to process a counter update. In the following results, we use $b = 16$.

To provide a concrete analysis of our proposed solution, we use the specifications of the off-the-shelf XDR memory from Rambus [27], [33]. Each XDR memory chip contains 16 internal memory banks. By using two chips, we can provide $B = 32$ memory banks and two memory channels, with 16 banks on each memory channel. 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, because modern CPUs always use intermediate caches to access external DRAMs. I/O is not the bottleneck in our system. 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.

With K DRAM banks, each bank needs to store $16/K$ millions counters. We assume that each DRAM access takes $b = 16$ cycles. The maximum request queue lengths for $K = 32$ memory banks using the USC trace is shown in Fig. 6. The results for both FIFO and LRU update rules are presented together with the maximum queue length of a simple request queue with no request merging for comparison. The maximum queue lengths with the FIFO update rule and LRU update rule are about the same while the LRU scheme achieves slightly smaller queue size. After 10^8 packets arrivals, the maximum queue length with FIFO update rule is 13 entries, while it is only 11 entries for request queues with LRU update rule. The maximum queue length of the simple request queue with no merging operation is in the order of 10^4, which is significantly larger than that of the FIFO or LRU scheme. Also, the maximum queue length of the simple request queue increases as more updates arrive, because during a larger period of time there are more traffic bursts generating updates to the same memory locations, which increase the maximum queue length considerably. Also, over time the occupancies in different request queues will drift apart [34]. It is worth noting that in this experiment, we use enough DRAM banks to provide a potential throughput that is twice the line rate by using 32 memory banks while a single bank is only 16 times slower. Such a scenario can also be translated to a reduced arriving traffic, where the arrival rate is only 50 percent of the service rate. However, even with the pseudorandom permutation function, there is no guarantee that the loads to the memory banks are uniformly distributed at all time due to the existence of traffic bursts. So by building a counter array with more DRAM banks to provide higher equivalent throughput or by reducing the arrival rate, there is still no guarantee of a bounded request queue size. However, with the merging operations, most such traffic bursts are merged in the request queues. The maximum queue length stays bounded in our statistics counter array as predicted by our models in Section 5.4.

To further stress the system and examine its performance, we present the maximum queue lengths for different designs using only $K = 16$ memory banks and $b = 16$ in Fig. 7. For simple queue without merging, the maximum queue length quickly explodes as more updates arrive because on average the arrival rate to a request queue is the same as the service rate, any bursts of updates with the same memory addresses will cause the request queues to increase dramatically. However, in our merging queue design, the maximum queue lengths are only increased by roughly a factor of 2 compared to the result in Fig. 6, where 32 banks are used. For FIFO update rule scheme, the maximum queue length is 29. For LRU update rule scheme, the maximum queue length is only 27. As the maximum queue lengths increase, on average there are more counter updates stored in a request queue at any time and the AMP for arriving updates is also increased, which in return effectively reduces the maximum queue lengths.

We observe that FIFO and LRU queue update schemes achieve similar performances with LRU scheme slightly better. Compared with the result for a simple queue with no merging, the maximum queue lengths for FIFO and LRU schemes are greatly reduced. Fig. 8 shows the ratio of merged counter update requests with different configurations for FIFO and LRU schemes. We can see that about

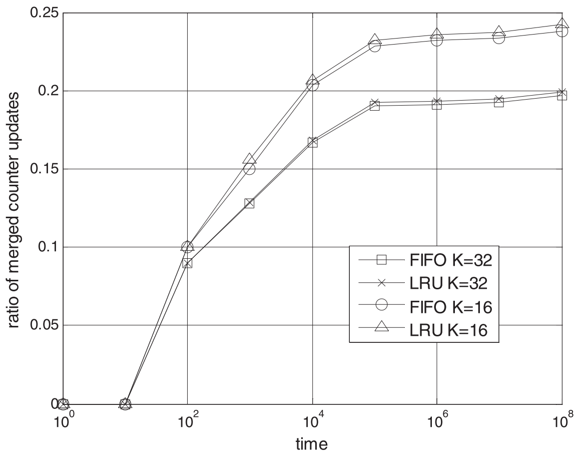


Fig. 8. Ratio of merged counter updates, USC trace.

20 percent of the incoming counter updates are merged with $K = 32$ for both schemes. By reducing the number of memory banks, the ratio of merged incoming counter updates grows to 25 percent, which further increases the AMP Θ_{amp}. The increased ratio of merged updated explains why the maximum queue lengths only increase by a factor of 2 as shown in Fig. 7.

The maximum request queue lengths for $K = 32$ memory banks and each DRAM access takes $b = 16$ cycles using the UNC trace is shown in Fig. 9. The maximum queue length of the simple request queue with no merging operation is significantly larger than the maximum queue length with merging request queues. With merging operation, for both FIFO and LRU schemes the maximum queue lengths stay stable below 12 entries. Fig. 9 also shows that the maximum queue length for the simple queue scheme with UNC traffic is not always increasing, such as from time 10^3 to 10^4 cycles. This is due to the fact that UNC trace has 13.5 million active flows, which is much larger than the 8.6 on active flows from USC trace. During time such as 10^3 to 10^4 cycles, there are more active flows with very few traffic bursts. With the help of the pseudorandom permutation function, the arriving requests are effectively distributed more uniformly without the heavy presence of traffic bursts. Since the average service rate is twice the average arrival rate to any request queue, the maximum request queue length is actually decreasing during this time.

The maximum request queue lengths for $K = 16$ memory banks and $b = 16$ using the UNC trace is shown in Fig. 10. With fewer memory banks, the maximum queue length of the simple request queue with no merging operation grows even faster. The maximum queue length for merging request queues with FIFO update rule stays stable below 52 entries. The maximum queue length for merging request queues with LRU update rule is only 49 entries.

Similar to the results using USC trace, compared with a simple queue with no merging, the maximum queue lengths for both FIFO and LRU schemes are greatly reduced. In Fig. 11, we can see that about 8 percent of the incoming counter updates are merged with $K = 32$. If we reduce the number of memory banks, the maximum queue length increases and the ratio of merged incoming counter updates increases to 12 percent for $K = 16$. It is also shown in Fig. 11 that during the time 10^3 to 10^4 cycles, the ratio of merged counter updates decreases, which is caused by the sudden increase in the number of active flows and the decrease in number of traffic bursts in the UNC trace during that time. Even with a reduced number of traffic bursts, our statistics counter array still outperforms a system with

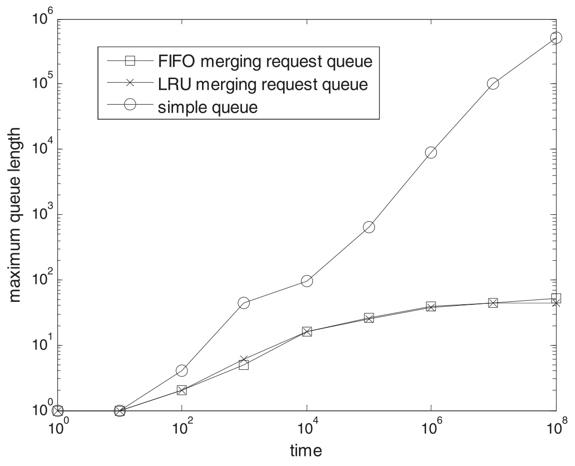


Fig. 10. Maximum queue length, $K = 16$, UNC trace.

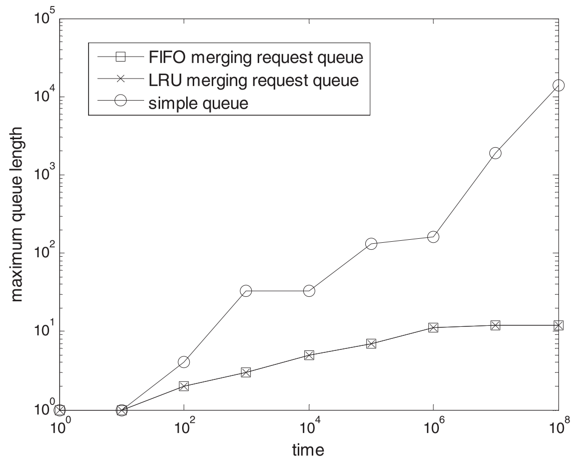


Fig. 9. Maximum queue length, $K = 32$, UNC trace.

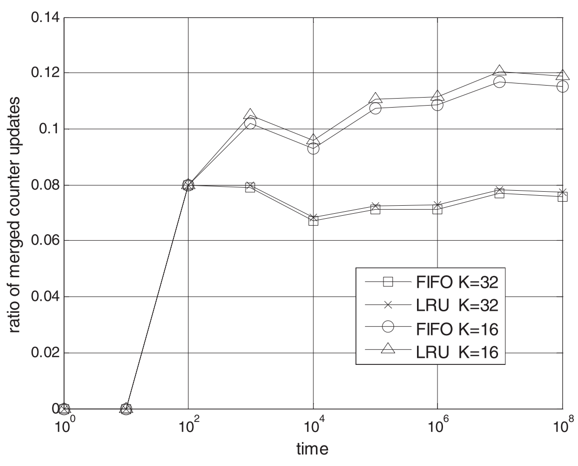


Fig. 11. Ratio of merged counter updates, UNC trace.

TABLE 1
Comparison of Different Schemes for a Reference Configuration with 16 Million 64-Bit Counters

	Naïve	Hybrid SRAM/DRAM [3]	Counter with Cache [9]	Counter with Merging Queues (this paper)
Counter DRAM	None	128 MB DRAM	128 MB DRAM	128 MB DRAM
Counter SRAM	128 MB SRAM	8 MB SRAM	None	None
Control	None	1.5 KB SRAM	25 KB CAM and 5.5 KB SRAM	2.2 KB CAM

For our method, $K = 32$ *and* $L = 20$.

simple queue significantly by providing a bounded maximum request queue size as shown in Figs. 9 and 10.

7.3 Cost-Benefit Comparison with Other Approaches

For our proposed solution, using $K = 32$ DRAM banks and request queues of size $L = 20$ entries each are sufficient for the traffic traces using either FIFO or LRU queue update policy. The fully associative cache needed for request queues is of size $M = K \cdot L$ entries. In the DRAM banks, there are 16 million counters. Each entry in the request queues requires 3 bytes to encode the counter indices of 16 million counters and 4 bits for accumulated counts. So the total cache size is about 2.2 KB. The size of each counter in the request queues is chosen to minimize the overall request queue size. In case a counter overflows if when it is merged with a new update, a new entry in the request queue will be created for the counter to avoid any counter overflow in the request queues. We choose 4 bits for each counter in the request queues as the optimal size for the traffic traces we use. More details on the choice of counter size is included in the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.287. Our experiments show that a simple modulo function mapping the counter indexes to different DRAM banks is sufficient to guarantee fairly uniform load distribution to different banks in absence of traffic bursts, 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 the 5-tuple in the packet header are commonly applied in NPs, thus no extra cost will be added to the system.

In [9], it requires 25 KB in cache and 5.5 KB in SRAM to achieve overflow probability of 10^{-14}. For our robust statistics counter array, we use only 25 KB in request queues as cache to achieve the same worst case performance guarantee as discussed in Section 6. Additionally, our scheme provide zero overflow probability for normal traffic without adversaries. As a comparison, the hybrid SRAM/DRAM counter architecture approaches [3] would require more than 8 MB of SRAM and the SRAM size grows linearly with the number of counters, while providing no worst case service guarantee. Moreover, repetitive counter updates are merged by our merging request queues while they cause system overflow in hybrid SRAM/DRAM schemes. The detailed comparison is shown in Table 1. Note that flow information stored in the statistics counter arrays may not be removed after the related flows terminates, because certain applications may still need to access the data collected by the counter arrays. Applications can still actively remove counters from DRAM banks by reassigning the memory location to other flows and removing the data.

8 Conclusion

We proposed a robust DRAM-based statistics counter architecture that can effectively maintain exact wirespeed updates to large counter arrays. To take advantage of the fact that most flows on the Internet consist of multiple packets that are being sent over a short time, our proposed architecture makes use of a simple randomization scheme and a set of small fully associative request queues to statistically guarantee a near-perfect load balancing of counter updates to the memory banks, which effective turns the bursty arriving traffic to our favor by reducing the total number of counter update requests to be process by the DRAM banks in the counter array. We also developed queuing models and showed that with traffic bursts in the arriving counter updates, the maximum request queue length is always bounded by a small value. Our simulation results further proved our model by showing that with merging request queues, the maximum queue length can never grow over a fixed size. Therefore, our proposed statistics counter architecture can effectively maintain wirespeed updates to large counter arrays while providing a diminishing overflow probability in the system.

References

[1] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown, "Maintaining Statistics Counters in Router Line Cards," *IEEE Micro*, vol. 22, no. 1, pp. 76-81, Jan./Feb. 2002.

[2] S. Ramabhadran and G. Varghese, "Efficient Implementation of a Statistics Counter Architecture," *SIGMETRICS Performance Evaluation Rev.*, vol. 31, no. 1, pp. 261-271, 2003.

[3] Q. Zhao, J. Xu, and Z. Liu, "Design of a Novel Statistics Counter Architecture with Optimal Space and Time Efficiency," *SIGMETRICS Performance Evaluation Rev.*, vol. 34, no. 1, pp. 323-334, 2006.

[4] Samsung, "Samsung K7S3236U4C QDRII SRAM," http://www.samsung.com/, 2013.

[5] Samsung, "Samsung K4B4G0446A DDR3 SDRAM," http://www.samsung.com/, 2013.

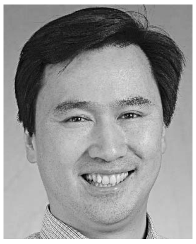
[6] M. Roeder and B. Lin, "Maintaining Exact Statistics Counters with a Multi-Level Counter Memory," *Proc. IEEE GLOBECOM*, vol. 2, pp. 576-581, Nov./Dec. 2004.

[7] P. Indyk, "Stable Distributions, Pseudorandom Generators, Embeddings, and Data Stream Computation," *Proc. IEEE 41st Ann. Symp. Foundations Computer Science (FOCS)*, 2000.

[8] H. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, "A Data Streaming Algorithm for Estimating Entropies of OD Flows," *Proc. Seventh ACM SIGCOMM Conf. Internet Measurement (IMC)*, 2007.

[9] H. Zhao, H. Wang, B. Lin, and J. Xu, "Design and Performance Analysis of a Dram-Based Statistics Counter Array Architecture," *Proc. ACM/IEEE Symp. Architectures for Networking and Comm. Systems (ANCS),* 2009.
[10] R. Morris, "Counting Large Numbers of Events in Small Registers," *Comm. ACM,* vol. 21, no. 10, pp. 840-842, 1978.
[11] A. Cvetkovski, "An Algorithm for Approximate Counting Using Limited Memory Resources," *SIGMETRICS Performance Evaluation Rev.,* vol. 35, pp. 181-190, 2007.
[12] R. Stanojevic, "Small Active Counters," *Proc. IEEE INFOCOM,* 2007.
[13] 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," *Proc. Int'l Conf. Distributed Computing Systems,* pp. 665-674, 2010.
[14] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," *Proc. ACM SIGCOMM,* 2002.
[15] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow," *Proc. ACM SIGCOMM,* pp. 245-256, 2004.
[16] "Juniper Networks Solutions for Network Accounting," http://www.juniper.net/techcenter/appnote/350003.html, 2013.
[17] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 36, pp. 121-132, 2008.
[18] N. Hua, B. Lin, J. Xu, and H. Zhao, "BRICK: A Novel Exact Active Statistics Counter Architecture," *Proc. ACM/IEEE Symp. Architectures for Networking and Comm. Systems (ANCS),* 2008.
[19] H. Wang, H. Zhao, B. Lin, and J. Xu, "Design and Analysis of a Robust Pipelined Memory System," *Proc. IEEE INFOCOM,* 2010.
[20] B. Agrawal and T. Sherwood, "High-Bandwidth Network Memory System through Virtual Pipelines," *IEEE/ACM Trans. Networking,* vol. 17, no. 4, pp. 1029-1041, Aug. 2009.
[21] G. Shrimali and N. McKeown, "Building Packet Buffers Using Interleaved Memories," *Proc. IEEE High Performance Switching Routing Workshop (HPSR),* May 2005.
[22] S. Iyer and N. Mckeown, "Designing Buffers for Router Line Cards," Technical Report TR02-HPNG-031001, Stanford Univ., Mar. 2002.
[23] H. Wang and B. Lin, "Block-Based Packet Buffer with Deterministic Packet Departures," *Proc. IEEE High Performance Switching Routing Workshop (HPSR),* 2010.
[24] W. Lin, S.K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proc. Seventh Int'l Symp. High Performance Computer Architecture,* pp. 301-312, 2001.
[25] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach,* second ed. Morgan Kaufmann, 1996.
[26] B.R. Rau, "Pseudo-Randomly Interleaved Memory," *Proc. IEEE 18th Ann. Int'l Symp. Computer Architecture (ISCA),* 1991.
[27] F.A. Ware and C. Hampel, "Micro-Threaded Row and Column Operations in a Dram Core," Rambus White Paper, Mar. 2005.
[28] E.H. McKinney, "Generalized Birthday Problem," *Am. Math. Monthly,* vol. 4, no. 73, pp. 385-387, Apr. 1966.
[29] J. Cao and K. Ramanan, "A Poisson Limit for Buffer Overflow Probabilities," *Proc. IEEE INFOCOM,* 2002.
[30] S. Ross, *Introduction to Probability Models,* ninth ed. Academic Press, 2006.
[31] A.B. Downey, "Evidence for Long-Tailed Distributions in the Internet," *Proc. ACM SIGCOMM Workshop Internet Measurement,* pp. 229-241, 2001.
[32] M. Garetto and D. Towsley, "Modeling, Simulation and Measurements of Queuing Delay under Long-Tail Internet Traffic," *SIGMETRICS Performance Evaluation Rev.,* vol. 31, pp. 47-57, 2003.
[33] "XDR Datasheet," Rambus, Inc., 2002.
[34] S. Kumar, P. Crowley, and J. Turner, "Design of Randomized Multichannel Packet Storage for High Performance Routers," *Proc. IEEE 13th Symp. Hot Performance Interconnects,* pp. 100-106, 2005.

Hao Wang (S'06) received the BE degree in electrical engineering from Tsinghua University, Beijing, P.R. China, and the MS and PhD degrees in electrical and computer engineering from the University of California, San Diego, in 2005, 2008, and 2011, respectively. He is currently with Oracle Corporation in Redwood Shores, California. His 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. He is a student member of the IEEE.



Bill Lin (M'97) received the BS, MS, and PhD degrees in electrical engineering and computer sciences from the University of California, Berkeley. He is currently a professor of electrical and computer engineering and an adjunct professor of computer science and engineering, both at the University of California, San Diego. 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 ($\mathrm{CAL\text{-}IT}^2$) 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, 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 more than 130 journal and conference publications. He 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. He also holds four awarded patents. He is a member of the IEEE.

Jun (Jim) Xu (S'98-M'00-SM'10) received the PhD degree in computer and information science from The Ohio State University in 2000. He is an associate professor in the College of Computing at Georgia Institute of Technology. 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. He received the US National Science Foundation (NSF) CAREER award in 2003, 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. He is a senior member of the IEEE.