Robust Pipelined Memory System with Worst Case Performance Guarantee for Network Processing

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

Abstract—Many network processing applications require wirespeed access to large data structures or a large amount of packet and flow-level data. Therefore, it is essential for the memory system of a router to be able to support both read and write accesses to such data at link speeds. As link speeds continue to increase, router designers are constantly grappling with the unfortunate trade-offs between the speed and cost of SRAM and DRAM. The capacity of SRAMs is woefully inadequate in many cases and it proves too costly to store large data structures entirely in SRAM, while DRAM is viewed as too slow for providing wirespeed updates at such high speed. In this paper, we analyze a robust pipelined memory architecture that can emulate an ideal SRAM by guaranteeing with very high probability that the output sequence produced by the pipelined memory architecture is the same as the one produced by an ideal SRAM under the same sequence of memory read and write operations, except time shifted by a fixed pipeline delay of Δ. Given a fixed pipeline delay abstraction, no interrupt mechanism is required to indicate when read data are ready or a write operation has completed, which greatly simplifies the use of the proposed solution. The design is based on the interleaving of DRAM banks together with the use of a reservation table that serves in part as a data cache. In contrast to prior interleaved memory solutions, our design is robust under all memory access patterns, including adversarial ones, which we demonstrate through a rigorous worst case theoretical analysis using a combination of convex ordering and large deviation theory.

Index Terms—Robust memory system, network processing, large deviation theory, convex ordering.

1 Introduction

Modern Internet routers often need to manage and move a large amount of packet and flow-level data. Therefore, it is essential for the memory system of a router to be able to support both read and write accesses to such data at link speeds. As link speeds continue to increase, router designers are constantly grappling with the unfortunate trade-offs between the speed and cost of SRAM and DRAM. While fitting all such data into SRAM with access latency typically between 4 to 8 ns is fast enough for the highest link speeds, the huge amount of SRAM needed renders such an implementation prohibitively expensive, as hundreds of megabytes or even gigabytes of storage may be needed. On the other hand, although DRAM provides inexpensive bulk storage, the prevailing view is that DRAM with access latency typically between 50 and 100 ns is too slow for providing wirespeed updates. For example, on a 40 Gb/s OC-768 link, a new packet can arrive every 8 ns, and the corresponding read or write operation to the data structure needs to be completed within this time frame. In this work, we do away with this unfortunate trade-off entirely, by proposing a memory system that is almost as fast as SRAM, *for the purpose of managing and moving packet and flow data*, and almost as affordable as DRAM.

1.1 Motivation

To motivate our problem, we list here three massive data structures that need to be maintained and/or moved inside high-speed network routers.

Network flow state. Maintaining precise flow state information [2] at an Internet router is essential for many network security applications, such as stateful firewalls, intrusion and virus detection through deep packet inspection, and network traffic anomaly analysis. For example, a stateful firewall must keep track of the current connection state of each flow. For signature-based deep packet inspection applications such as intrusion detection and virus detection, per-flow state information is kept to encode partial signatures (e.g., current state of the finite state automata) that have been recognized. For network traffic anomaly analysis, various statistics and flow information, such as packet and byte counts, source and destination addresses and port information, flow connection setup and termination times, routing and peering information, etc., are recorded for each flow during a monitoring period for later offline analysis. Indeed, the design of very large flow tables that can support fast lookups and updates has been a fundamental research problem in network security area.

Precise flow state also needs to be maintained for many other network applications unrelated to security. For

- *H. Wang and B. Lin are with the Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, CA 92093. E-mail: {wanghao, billlin}@ucsd.edu.*
- *H. Zhao and J. Xu are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {chz, jx}@cc.gatech.edu.*

Manuscript received 17 Dec. 2010; revised 19 Aug. 2011; accepted 22 Aug. 2011; published online 31 Aug. 2011.
Recommended for acceptance by L. Wang.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-12-0695.
Digital Object Identifier no. 10.1109/TC.2011.171.

example, in packet scheduling algorithms (for provisioning network Quality of Service (QoS)), for each *flow*, the flow table needs to maintain a *time stamp* that indicates whether the flow is sending faster or slower than its fair share, which in turn determines the priority of the packets currently waiting in the queue that belongs to this flow.

The management of precise per-flow state information is made challenging by at least two factors. First, the number of flows can be extremely large. Studies have shown that millions of concurrent flows are possible at an Internet router in a backbone network [3]. Second, the lookup and updating of flow records must be performed at very high speeds. As mentioned, although SRAM implementations can support wirespeed access, they are inherently not scalable to support millions of flow records (potentially hundreds of millions in the future). On the other hand, although DRAM provides huge amounts of inexpensive, naïve usage of DRAMs would lead to worst case access times that are well beyond the packet arrival rates on a high-speed link. As a result, most network security applications have typically been deployed in the access network close to the customers to reduce both the number of flows that need to be managed as well as the wirespeed. However, with the proliferation of mobile devices, the number of concurrent flows is expected to increase rapidly even at the edge of the network. In addition, there are a number of reasons for deploying some network security functions deeper into the backbone network where the number of concurrent flows is substantially higher. For example, the backbone network carries a far more diverse traffic mix that can potentially expose a wider range of security problems. Other solutions for mitigating the need to maintain large amounts of precise state information at wirespeed include data streaming [4] and sampling methods, such as Cisco's sampled NetFlow [5], Juniper's filter-based accounting [6], and the sample-and-hold solution [7]. However, these methods cannot keep track of accurate flow states, which can impose limitations on their effectiveness.

High-speed packet buffers. The implementation of packet buffers at an Internet router is another essential application that requires wirespeed access to large amounts of storage. Historically, the size of packet buffers has been increasing with link speeds. To tackle this problem, several designs of packet buffers based on hybrid SRAM/DRAM architectures or memory interleaved DRAM architectures have been proposed [8], [9], [10], [11], [12], [13]. Although the designs are both effective and practical, these solutions are problem specific. Ideally, we would like a general memory architecture that is applicable to both the packet buffering problem as well as the various network flow state implementation problems.

Statistics counter arrays. Network measurement is essential for the monitoring and control of large networks. For implementing 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. Therefore, the design of memory systems specialized in statistics counting has received considerable research attention in the recent years. The problem of maintaining statistics counters is simpler in nature compared to the design of a general memory system since the memory only needs to support wirespeed updates (write operations), whereas a general memory system supports arbitrary memory read and write operations at wirespeed.

1.2 Our Approach

In this work, we design a DRAM-based memory architecture that can emulate a fast massive SRAM module by exploiting memory interleaving. In this architecture, multiple DRAM banks are running in parallel to achieve the throughput of a single SRAM. Our architecture provides fixed delay for all the read requests while ensuring the correctness of the output results of the read requests. More specifically, if a memory read operation is issued at time t, its result will be available at output exactly at time $t + \Delta$, where Δ is a fixed pipeline delay. In this way, a processor can issue a new memory operation every cycle, with deterministic completion time Δ cycles later. No interrupt mechanism is required to indicate when read data are ready or a write operation has completed for the processor that issued the requests. With a fixed pipeline delay, a processor can statically schedule other instructions during this time. Although this Δ is much larger than the SRAM access latency, router tasks can often be properly implemented to work around this drawback thanks to the deterministic nature of this delay.

For example, in packet scheduling algorithms for provisioning Quality of Service, the flow table needs to maintain for each *flow* a *time stamp* that indicates whether the flow is sending faster or slower than its fair share, which in turn determines the priorities of the packets belonging to this flow that are currently waiting in the queue. Using weighted fair queuing [14], when a new packet pkt arrives at time t, a read request will be issued to obtain its flow state such as the virtual finish time of the last packet in the same flow. At time $t + \Delta$, when the result comes back, the virtual finish time of pkt (a per-flow variable) and the system virtual time (a global variable) can both be precisely computed/updated because they depend only on packet arrivals happening before or at time t, all of which are retrieved at time $t + \Delta$. In other words, given any time t, the flow state information needed for scheduling the packets that arrive before or at time t is available for computation at time $t + \Delta$. Although all packets are delayed by Δ using our design, we will show that this Δ is usually in the order of tens of microseconds, which is three orders of magnitude shorter than end-to-end propagation delays (tens of milliseconds across the US).

While multiple DRAM banks can potentially provide the necessary raw bandwidth that high-speed routers need, traditional memory interleaving solutions [15], [16] do not provide consistent throughput reliably because certain memory access patterns, especially those that can conceivably be generated by an adversary, can easily get around the load balancing capabilities of interleaving schemes and overload a single DRAM bank. Another solution proposed in [17] is a virtually pipelined memory architecture that aims to mimic the behavior of a single SRAM bank using multiple DRAM and SRAM banks under average arrival traffic with no adversaries. All the pending operations to the memory system are provided with fixed delay using a

cyclic buffer. However, this architecture assumes perfect randomization in the arrival requests to the memory banks. Under adversarial arrivals, the buffer will overflow and start dropping requests, which greatly degrades the system performance.

To guard against adversarial access patterns, memory locations are randomly distributed across multiple memory banks so that a near-perfect balancing of memory access loads can be provably achieved. This random distribution is achieved by means of a random address permutation function. Note that an adversary can conceivably overload a memory bank by sending traffic that would trigger the access of the same memory location, because they will necessarily be mapped to the same memory bank. However, this case can be easily handled with the help of a reservation table (which serves in part as a data cache) in our memory architecture. With a reservation table of C entries, we can ensure that repetitive memory requests to the same memory address within a time window C will only result in at most two memory accesses in the worst case, with one read request followed by one write request. All the other requests will only be stored in the reservation table to provide a fixed delay.

Another key contribution of this paper is a mathematical one: we prove that index randomization combined with a reasonably sized reservation table can handle with overwhelming probability arbitrary including adversarial memory request patterns without having overload situations as reflected by long queuing delays, which is to be made precise in Section 6. This result is a *worst case large deviation theorem* [18] in nature because it establishes a bound on the largest (worst case) value among the tail probabilities of having long queuing delays under all admissible including adversarial memory access patterns. Our methodology for proving this challenging mathematical result is a novel combination of convex ordering and traditional large deviation theory.

1.3 Outline of Paper

The rest of the paper is organized as follows: Section 2 presents the related work on the memory system design. Section 3 defines the notion of SRAM emulation. Section 4 describes the basic memory architecture. Section 5 extends our proposed memory architecture by providing robustness against adversarial access patterns. Section 6 provides a rigorous analysis on the performance of our architecture in the worst case. Section 7 presents an evaluation of our proposed architecture. Finally, Section 8 concludes the paper.

2 Related Work

The problem of designing memory systems for different network applications has been addressed in many past research efforts. We will first present the development of specialized memory systems for statistics counter arrays. Then, we will show memory systems designed for allowing only restricted memory access patterns, and also the memory systems with techniques to accommodate unrestricted memory access patterns.

2.1 Statistics Counter Arrays

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 [19], 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 University of North Carolina (UNC) has 13.5 million flows. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows if the measurements take place at high-speed links as smaller counters can quickly overflow. Thus, a total of 108 MB of SRAM would already be needed for just the counter storage, which is prohibitively expensive. Researchers have been actively seeking alternative ways to realize large arrays of statistics counters at wirespeed [19], [20], [21], [22], [23], [24], [25], [26], [27], [28].

Several designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed. Their basic idea is to store some lower order bits (e.g., 9 bits) of each counter in SRAM, and all its bits (e.g., 64 bits) in DRAM. The increments are made only to these SRAM counters, and when the values of SRAM counters come close to overflowing, they are scheduled to be "flushed" back to the corresponding DRAM counters. These schemes all significantly reduce the SRAM cost. However, such designs are not suitable for arbitrary increments to the counters which would cause the counters in SRAM to be flushed back to DRAM arbitrarily more frequently. In addition, they do not support arbitrary decrements and are based on an integer number representation. Another statistics counter array design proposed in [28] utilizes several DRAM banks to mimic a single SRAM. There is a request queue for each DRAM bank to buffer the pending counter update requests to the bank. The counter updates are distributed into different memory banks with the help of a random permutation function. Also a cache module is implemented to reduce the memory accesses caused by repetitive updates to the same counters. Such a design supports both counter increments and decrements, as well as floating point number representation and other formats. However, it is suitable only for the maintenance of statistics counters where memory write operations to update the counters occur much more frequently than read operations to retrieve the counter values, and such counter retrieval read operations are sparsely distributed in time or even processed offline. Special purpose memory architectures are also developed for IP lookup [29], [30], [31], which is a fundamental application supported by all routers. In general, specialized memory systems for maintaining statistics counters cannot be easily adapted to support arbitrary data accesses and updates.

2.2 Memory Systems with Restricted Accesses

To design memory systems with bulk storage working at wirespeed, multiple DRAM are often implemented in parallel to achieve SRAM throughput. Since the access latency of a DRAM bank is much larger (several order of magnitudes) than that of SRAM, different approaches are developed to hide such a latency. Memory interleaving solutions are proposed in [15], [16] where prefetching and other bank-specific techniques are developed. In these

schemes, the memory locations are carefully distributed into several memory banks in some pseudorandom fashion. If the memory accesses are evenly distributed across memory banks, then the number of bank conflicts can be reduced. Therefore, the supported memory access patterns are greatly restricted which limit the applications of such designs. For example, in the application of a packet buffer for one traffic flow, the arriving packets can simply be stripped sequentially across memory banks in the FIFO order using the memory interleaving solution. However, consider a high-speed packet buffer supporting millions of flows. The order in which packets are retrieved from the memory is typically determined by a packet scheduler implementing a specific queuing policy. The packets across the memory banks may be retrieved in an unbalance fashion, which can cause the system performance to degrade drastically. Other applications such as lookups of global variables will trigger accesses to the same memory locations repeatedly and all the accesses are mapped to the same memory banks for sure, since the memory locations in memory banks are fixed after the mapping. Such events can potentially cause unbounded delay for pending memory accesses to the memory banks. Moreover, in any memory system, the buffer size is always limited; therefore, eventually memory access requests will be dropped when a buffer overflows. For network routers, dropped TCP packets may cause frequent retransmissions by the senders, which would further reduce the available network bandwidth and drain the available buffers. In network security applications, buffer overflows may introduce unaccounted network traffic which would cause failures in tracking down suspicious flows, thus leading to the increase of miss rate in identifying malicious traffic patterns. In summary, memory systems with restricted access patterns are not robust to unrestricted traffic patterns and have very limited applications.

2.3 General Memory Systems with Unrestricted Accesses

Extensive research efforts have taken place in the design of high-speed packet buffers to support unrestricted packet retrievals. For example, in order to implement packet buffers at an Internet router, the buffers need to be able to work at wirespeed and provide bulk storage. As the link speed increases, the size of packet buffers has been increasing also. To tackle this problem, several designs of packet buffers based on hybrid SRAM/DRAM architectures or memory interleaved DRAM architectures have been proposed [8], [9], [10], [11], [12], [13], [32]. The basic idea of these architectures is to use multiple DRAM banks to achieve SRAM throughput, while reducing [10], [11], [32] or eliminating [8], [9], [12], [13] the potential bank conflicts. One of the designs, the prefetching-based solutions described in [8], [9], supports linespeed queue operations by aggregating and prefetching packets for parallel DRAM transfers using fast SRAM caches. These architectures require complex memory management algorithms for real-time sorting and a substantial amount of SRAM for caching the head and tail portions of logically separated queues to handle worst case access patterns. Another type of the packet buffer design, randomization-based architectures [10], [11], is based on a random placement of packets into different memory banks so that the memory loads across the DRAM banks are balanced. While effective, these architectures only provide statistical guarantees as there are deterministic packet sequences that will cause system overflow. The third type of the packet buffer design is the reservation-based architectures [12], [13] where a reservation table serves as a memory management module to select one bank out of all available banks to store an arriving packet such that it will not cause any conflicts at time of packet arrival or departure. An arrival conflict occurs when the arrival packet tries to access a busy bank. A departure conflict occurs when a packet in a busy bank needs to depart. In order to avoid such conflicts, these architectures need to use about three times the number of DRAM banks and deploy complicated algorithm in the memory selection step. In general, while the above approaches are effective in the design of packet buffers, the substantial amount of SRAM requirement, the vulnerability toward certain arrival patterns, or the lack of providing fixed delay for memory read or write accesses render them unsuitable for many other network applications.

Ideally, we would like a general memory architecture that is applicable to both the packet buffering problem as well as various other network applications. The idea of a pipelined memory system with a fixed pipelined delay was proposed by Agrawal and Sherwood in [17], where a virtually pipelined network memory (VPNM) architecture is presented. In this system, a universal hashing function is applied on all arriving memory access requests to distribute them into multiple DRAM banks. Each memory access request is delayed by a fixed D cycles. The VPNM works quite effectively in the average case, as verified by both their analytical and empirical results. However, network operators are often concerned about worst case performance, particularly in the case of network security applications. Indeed, in the literature, most proposed algorithms [33] for packet classification, regular expression matching, and weighted fair queuing, just to name a few, are all designed for the worst case, even though worst case situations may not be common. For example, it is well known that most IP longest prefix matches can in practice be resolved with the first 16 to 24 bits, but yet the IP lookup algorithms that have been adopted commercially have all been designed to operate at wirespeed to match all 32 bits of an IPv4 address in the worst case. Similarly, it has also been shown that most regular expression matches fail after a small number of symbols, but in the worst case, monitoring very long sequences of partial matches may be necessary. In general, network operators would like solutions that are robust under a wide variety of (often unforeseen) operating conditions, which are often dependent on uncertainties that are beyond the control of the operator.

In this work, we design a provably robust pipelined memory system that performs well under all operating conditions, including adversarial ones. We provide a novel mathematical model to analyze the overflow probability of such a system using the combination of convex order and large deviation theory. We establish a bound on the overflow probability and identify the worst case memory access patterns. To our best knowledge, it is the only proven bound

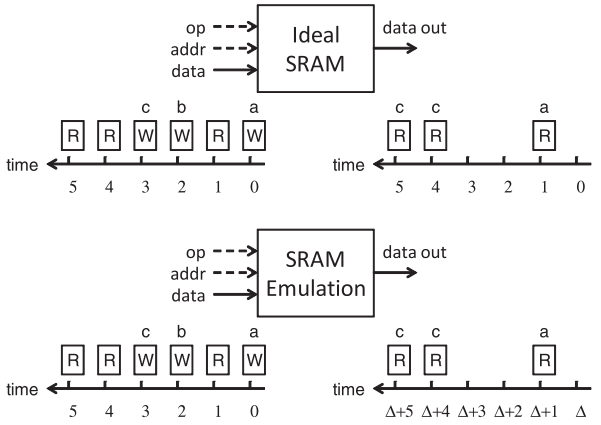

Fig. 1. SRAM emulation.

on a memory system for any admissible and even adversarial memory access patterns. The overflow probability for any real-world Internet traffic is always upper bounded by the bounds predicted in our model.

3 Ideal SRAM Emulation

In this paper, we aim to design a DRAM-based memory architecture that can emulate a fast SRAM by exploiting memory interleaving. Suppose that a new memory access request is generated by each arriving packet. We define a cycle to be the minimum time it takes for a new packet to arrive at the router linecard. For example, at line rate 40 Gb/s (OC-768) with minimum packet of size 40 bytes, a cycle is 8 ns. A larger packet takes multiple cycles to arrive and thus generates a memory access request at a lower rate. We assume that it takes a DRAM bank $1/\mu$ cycles to finish one memory transaction. We use $B > 1/\mu$ DRAM banks and randomly distribute the memory requests across these DRAM banks so that when the loads of these memory banks are perfectly balanced, the load factor of any DRAM bank is $\frac{1}{B\mu} < 1$. We also assume that an ideal SRAM can complete a read or write operation in the same cycle that the operation is issued. An SRAM emulation that mimics the behavior of an ideal SRAM is defined as follows:

Definition 1 (Emulation). *A memory system is said to* emulate *an ideal SRAM if under the same input sequence of reads and writes, it produces exactly the same output sequences of* read *results, except time shifted by some fixed delay* Δ.

More specifically, our emulation guarantees the following semantics:

- *Fixed pipeline delay*. If a read operation is issued at time t to an emulated SRAM, the data are available from the memory controller at exactly time $h = t + \Delta$ (instead of the same cycle), where h is the *completion time*.
- *Coherency*. The read operations output the same results as an ideal SRAM system, except for a fixed time shift.

Fig. 1 illustrates the concept of SRAM emulation. In Fig. 1, a series of six read and write accesses to the same memory location are initiated at times $0, 1, 2, \ldots, 5$, respectively. If the memory is SRAM, then the read accesses at times 1, 4, 5 should return values a, c, c, respectively. With our SRAM emulation, exactly the same values a, c, c will be returned, albeit at times $1 + \Delta$, $4 + \Delta$, $5 + \Delta$, respectively.

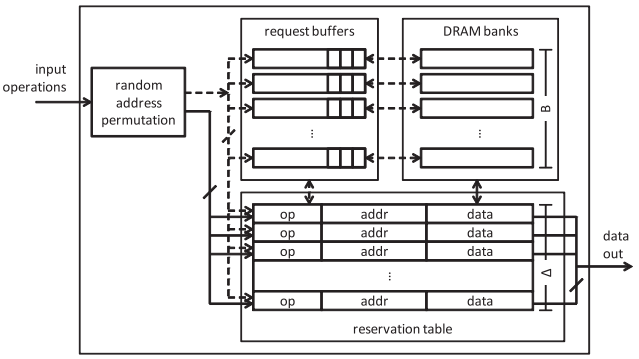

Fig. 2. Basic memory architecture.

Note the definition of emulation does not require a specific completion time for a write operation, because nothing is returned from the memory system to the issuing processor. As to the read operations, the emulation definition only requires the output sequence (i.e., sequence of read results) to be the same, except time shifted by a fixed Δ cycles. The definition does not require the *snapshot* of the memory contents to be the same. Therefore, the contents in the DRAM banks do not necessarily have to be the same as in an ideal SRAM at all times. For example, as we shall see later in our extended memory architecture, with the two back-to-back write operations in Fig. 1, only the data of the second write operation are required to update the memory to provide coherency, whereas both write operations correspond to actual memory updates in an ideal SRAM.

4 The Basic Memory Architecture

The basic memory architecture is shown in Fig. 2. It consists of a reservation table, a set of DRAM banks with a corresponding set of request buffers, and a random address permutation function.

4.1 Architecture

- *Reservation table*. The reservation table with Δ entries is implemented in SRAM to provide for a fixed delay of Δ for all incoming operations. For each operation arriving at time t, an entry (a row in Fig. 2) is created in the reservation table at location $(t + \Delta) \bmod \Delta$. Each reservation table entry consists of three parts: a 1-bit field to specify the operation (read or write), a memory address field, and a data field. The data field stores the data to be written to or to be retrieved from the memory.
- *DRAM banks and request buffers*. There are $B > 1/\mu$ DRAM banks in the system to match the SRAM throughput. Also, there are B request buffers, one for each DRAM bank. Memory operations to the same DRAM bank are queued at the same request buffer. Each request buffer entry is just a *pointer* to the corresponding reservation table entry, which contains the information about the memory operation. Given the random permutation of memory addresses,

we can statistically size the request buffers to ensure an extremely low probability of overflow. We defer to Section 6 to discuss about this analysis. Let K be the size of a request buffer. Then, the B request buffers can be implemented using multiple memory channels. In each channel, the request buffers are serviced in a round-robin order and different channels are serviced concurrently. For each channel, there is a dedicated link to the reservation table. For example, with $\mu = 1/16$ and $B = 32$, the request buffers can be implemented on two channels of 16 banks on each channel. Therefore, each request buffer takes $1/\mu$ cycles to be serviced. A memory operation queued at a request buffer would take at most $\Delta = K/\mu$ cycles to complete. We set the fixed pipeline delay in cycles for the external memory interface to this Δ.

- *Random address permutation function*. The goal of the random address permutation function is to randomly distribute memory locations so that memory operations to different memory locations are uniformly spread over B DRAM banks with equal probability $1/B$. The function is a one-to-one repeatable mapping. Therefore, the function is pseudorandom and deterministic. Unless otherwise noted, when referring to a memory address, we will be referring to the address after the random permutation. Note that if incoming operations access the same memory location, they will still generate entries to the same request buffer.

4.2 Operations

For a read operation issued at time t, its completion time is $h = t + \Delta$. A reservation table entry is created at location $h \bmod \Delta$. The data field is initially pending. A DRAM read operation gets inserted into the corresponding request buffer. When a read operation at the head of a request buffer is serviced, which may be earlier than its completion time h, the corresponding data field of its reservation table entry gets updated. In this way, the reservation table effectively serves as a *reorder* buffer. At the completion time h of the read operation, data from corresponding reservation table entry get copied to the output. A reservation table entry is removed after Δ cycles.

For a write operation issued at time t, a reservation table entry is created at location $h \bmod \Delta$, where $h = t + \Delta$. Also, a DRAM write operation is inserted into the corresponding request buffer. When the write operation at the head of a request buffer is serviced, which may be earlier than its completion time h, the write data are updated to the DRAM address. Its reservation table entry is removed Δ cycles after its arrival time t.

By sizing the reservation table to have Δ entries, the lifetime of a read or write reservation entry is guaranteed to be longer than the time it takes for the DRAM read or write to occur, respectively.

4.3 Adversarial Access Patterns

Even though a random address permutation is applied, the memory loads to the DRAM banks may not be balanced due to some adversarial access patterns as follows: first, many applications require the lookup of global variables.

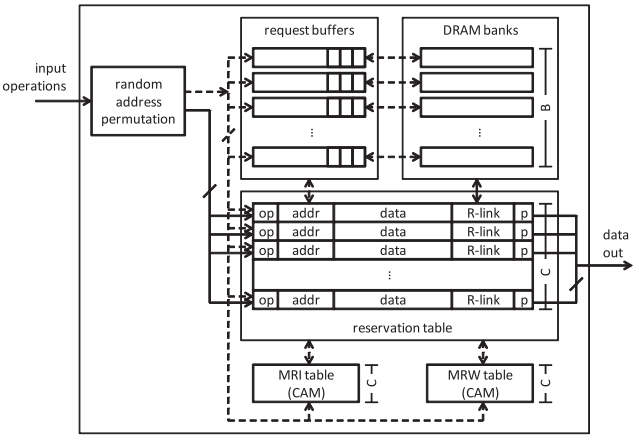

Fig. 3. Extended memory architecture.

Repeated lookups to a global variable will trigger repeated operations to the same DRAM bank location regardless of the random address permutation implemented. Second, although attackers cannot know how memory addresses are mapped to DRAM banks, they can still trigger repeated operations to the same DRAM bank by issuing memory operations to the same memory locations. Due to these adversarial access patterns, the number of pending operations in a request buffer may grow indefinitely. To mitigate these situations, our extended memory architecture in Section 5 effectively utilizes the reservation table as a cache to eliminate unnecessary DRAM operations queued in the request buffers.

5 The Extended Memory Architecture

Fig. 3 depicts our proposed extended memory architecture. As with the basic architecture, there is a reservation table, a set of DRAM banks, and a corresponding set of request buffers. For each memory operation, its completion time is still exactly Δ cycles away from the perspective of the issuing processor. However, in contrast to the basic memory architecture, we do not necessarily generate a new DRAM operation to the corresponding request buffer for every incoming memory operation. In particular, we can avoid generating new DRAM operations in many cases by using the reservation table effectively as a data cache. We will describe in more detail in our *operation merging rules* later in this section.

At the high level, our goal is to *merge* operations that are issued to the same memory location within a window of C cycles. This is achieved by extending the basic architecture in the following ways: first, the size of the reservation table can be set to any $C \geq \Delta$ entries to catch mergeable operations that are issued at most C cycles apart. Second, each reservation table entry is expanded to contain the following information:

- *Pending status p*. A memory read operation is said to be pending with $p = 1$ if there is a corresponding entry in the request buffer, or if it is linked to an operation that is still pending. Otherwise, the read operation is nonpending with $p = 0$. The memory write operations in the reservation table are set as pending with $p = 1$.

- *R-link*. Pending read operations to the same memory address are linked together into a linked list using the R-link field, with the earliest/latest read operations at the head/tail of the list, respectively.

In addition to the extra fields in the reservation table, two lookup tables are added to the extended architecture: a Most Recently Issued (MRI) lookup table and a Most Recent Write (MRW) lookup table. Both of these tables can be implemented using Content-Addressable-Memories (CAMs) to enable direct lookup. The MRI table keeps track of the most recently issued operation (either read or write) to a memory address. When a new read operation arrives, an MRI lookup is performed to find the most recent operation in the reservation table. The MRW table keeps track of the most recent write operation to a memory address. When a write operation in the reservation table is removed, an MRW lookup is performed to check if there is another more recent write operation to the same memory address. For an MRI lookup, if there is a matched entry, it returns a pointer to the row in the reservation table that contains the most recent operation to the same memory address. Similarly, an MRW lookup returns a pointer to the row in the reservation table corresponding to the most recent write operation for a given memory address.

In the extended architecture, each entry in the request buffers needs a data field for write operations. For read operations in the request buffers, a request buffer entry serves as a pointer to the corresponding reservation table entry. But for write operations in the request buffers, a request buffer entry stores the actual data to be written to a DRAM bank.

5.1 Reservation Table Management

A reservation table entry gets freed when the operation in the entry stays in the reservation table for C cycles. For an operation arrived at time t, its reservation table entry will be freed at time $g = t + C$. To achieve this, we maintain two pointers: pointer h as a completion time pointer, and pointer g as a garbage collection time pointer. When the current time is g, then the corresponding reservation table entry is freed (meaning the entry is cleared). On an MRI (or MRW) lookup, if it returns a reservation table entry with different address (*addr*), then we also remove this MRI (or MRW) entry.

5.2 Memory Operations

When a read operation (R) arrives, a new entry in the reservation table is created for R. Then, we have the following three cases:

- *Case 1*. If the MRI lookup fails, which means there is no previous operation to the same memory address in the reservation table, we then generate a new read operation to the corresponding request buffer. A new MRI entry is created for this operation also, and its pending status p is set to 1. When the actual DRAM read operation finishes, the data field of R in the reservation table entry is updated, and p is reset to 0.
- *Case 2*. The most recent operation to the same memory address returned by the MRI lookup is a pending read operation. In this case, we follow these steps:

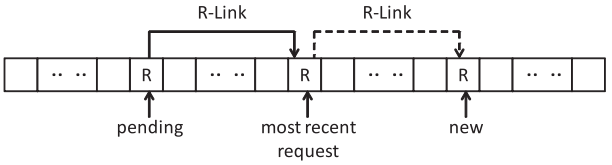

Fig. 4. R-link for read operations.

  - We create an R-link (shown as the dashed line in Fig. 4) from this most recent read operation to R and set its pending status p to 1. We do not create a new read operation in the request buffers.
  - The MRI entry is updated with the location of R in the reservation table.

  Essentially, a linked list of read operations to the same address is formed, with head/tail corresponding to the earliest/latest read, respectively. At the completion time of a read operation, we copy the read data to the next read operation by following its R-link, and we reset p to 0. Therefore, only the first read operation in the linked list generates an actual read operation in the request buffer. The remaining read operations simply copy the data from the earlier read operation in the linked list order. An example of the management of R-links for read operations is shown in Fig. 4.
- *Case 3*. The most recent operation to the same address returned by the MRI lookup is a write operation or a nonpending read operation. In this case, we copy the data from this most recent write or nonpending read operation to the new R location. In this way, R is guaranteed to return the most recent write or nonpending read data. The MRI entry is updated by pointing to this new operation.

When a read operation reaches its fixed delay Δ and departs from the system, its data are sent to the external data out. The corresponding reservation table entry is removed after C cycles. Also, if there is an entry in MRI that points to the reservation table entry, then this MRI entry is removed also.

When a new write operation (W) arrives, a new entry in the reservation table is created for W. Also, we have the following two cases:

- *Case 1*. If the MRI lookup fails, we create a new write operation into the corresponding request buffer, and we create a new MRW and MRI entry for this new operation.
- *Case 2*. If the MRI lookup returns an entry in the reservation table, then an MRW lookup is also performed. If the MRW lookup returns an entry in the reservation table, then the corresponding entries in the MRI and MRW are updated by pointing to the new operation W in the reservation table. Otherwise, if the MRW lookup does not find an entry, then the corresponding entry in MRI is updated to W and a new entry is created in MRW for W.

A write operation W is removed from the reservation table after C cycles. An MRW lookup is performed to check if there is a more recent write operation in the reservation table. If the MRW lookup returns a different entry in the

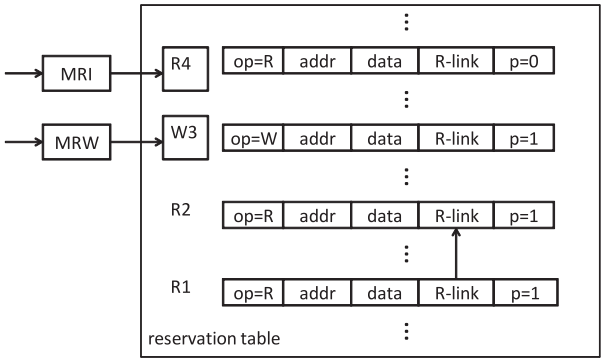

Fig. 5. Reservation table snapshot.

reservation table for the same address, then no request buffer entry will be created for W. Otherwise, there is no other write operation to the same address in the reservation table, and the MRW lookup only returns the address of W. Then, a new entry in the request buffer is generated for W to update the corresponding DRAM bank location. Also, we delete the entry in MRW pointing to W.

To guarantee that the reservation table can provide a cache of size C instead of $C-1$ as will be shown in Section 5.3, a new operation arriving at the reservation table will first check against the address of the expiring reservation table entry before rewriting it. For a new read operation, if the reservation table entry it shall occupy contains the same address field, then the data field is preserved and reused by the new read operation. For a new write operation, if the reservation table entry it shall occupy is a write operation of the same address, then the expiring write operation will simply be discarded without being inserted into the request queues. Both processes described above can be easily supported by requiring the reservation table to read its expiring entry first and then updating it with the new memory operation.

A snapshot of the reservation table entries accessing the same DRAM address is shown in Fig. 5. The memory operations are read (R1), read (R2), write (W3), and read (R4), with R4 being the most recent operation. The R-link of the pending read operation R1 points to the second read R2. Since the most up-to-date data are available in W3, the data are copied to the read operation R4 directly. Therefore, only R1 and W3 generate entries in the request buffers. Operations R2 and R4 are never inserted into the request buffers.

5.3 Operation Merging Rules

Based on the descriptions of the read and write operations above effectively, we are performing the following merging rules to avoid introducing unnecessary DRAM accesses. The operations arrive in the order from right to left on the left-hand side of the arrows. The actual memory operation(s) generated in the request buffers is shown on the right-hand side of the arrows.

1. $\mathrm{RW} \rightarrow \mathrm{W}$. The R operation copies data directly from the W operation stored in the reservation table; thus, it is not inserted into the request buffers.
2. $\mathrm{WW} \rightarrow \mathrm{W}$. The earlier (right) W operation does not generate new entry in the request buffers.
3. $\mathrm{RR} \rightarrow \mathrm{R}$. The latter (left) R operation is linked to the earlier R operation by its R-link. No entry is created in the request buffer for the latter R operation.
4. $\mathrm{WR} \rightarrow \mathrm{WR}$. Both the R operation and the W operation have to access the DRAM banks. They cannot be merged. Entries in the request buffer need to be created for both W and R operations.

For the last rule, $\mathrm{WR} \rightarrow \mathrm{WR}$, even though the operations cannot be merged, the future incoming operation to the same memory address can be merged. Consider the following two cases (inputs are sequenced from right to left):

- $(\mathrm{RW})\mathrm{R} \rightarrow (\mathrm{W})\mathrm{R}$. The last two (RW) are merged to (W) using the $\mathrm{RW} \rightarrow \mathrm{W}$ rule.
- $(\mathrm{WW})\mathrm{R} \rightarrow (\mathrm{W})\mathrm{R}$. The last two (WW) are merged to (W) using the $\mathrm{WW} \rightarrow \mathrm{W}$ rule.

With the above merging rules, data coherency is maintained. Since the reservation table has already provided fixed delay Δ for all the read operations, to show data coherency, we just need to prove that all the read operations will output the correct data. We will prove this by showing in the following that the relative ordering of the read and write operations is preserved in our architecture, which means that the read operations output the same data as in an SRAM system. In particular, we will focus on a write operation and show that operation merging rules will not affect the read operations before or after the write operation. Let's consider the following cases:

1. A newly arrived write operation W will not affect the data retrieved by the read operations before it, since the write operation W will not be inserted into the request buffer until C cycles later, by which time all the read operations before W have already finished.
2. For read operations arriving after a write operation W in the reservation table, with no other writes in between, they will read the correct data directly from W.
3. For read operations arriving after a write operation W in the reservation table, with other writes in between, they will read the correct data from the most recent write operation W'.
4. A write operation W is removed from the reservation table C cycles after it arrives. Upon removal, if there is another most recent write operation W' in the reservation table, the newly arrived read operations will read the correct data from W'. Therefore, the removal of W will not affect the data of the future read operations.
5. Upon the removal of a write operation W in the reservation table, if there is no other more recent write operation, but there is a most recent read operation R, then an entry is generated in the request buffer for W to update the data in the corresponding DRAM bank. According to the memory operations described in Section 5.2, R copies the data directly from W, since R arrives later than W. All future incoming read operations will read the correct data from R directly or from the DRAM bank. Since W is

already in the request buffer, future incoming read operations will generate entries in the request buffer only after W, and thus they will read the correct data from the memory banks.

6. Upon the removal of a write operation W in the reservation table if there is no other operation accessing the same DRAM location in the reservation table, then an entry is generated in the request buffer for W to update the memory banks. All the future read operations will only generate entries in the request buffer after W; thus, they will be able to retrieve the correct data.

In summary, the read operations always return the correct data except for a fixed delay Δ; therefore, the system is data coherent.

Further, using the proposed merging rules, we can ensure the following properties:

- There can be only one write operation in a request buffer every C cycles to a particular memory address. A write operation is generated in the request buffer only when there is a write operation in the reservation table for C cycles and there is no more recent write operation in the reservation table during this period of time. So, a write operation is generated in the request buffer at most once every C cycles.
- There can be at most one read operation in a request buffer every C cycles to a particular address. When there is more than one arriving read operation to the same address within C cycles, we are guaranteed that all except the first read are merged using R-links.
- There can be at most one read operation followed by one write operation in a request buffer every C cycles to a particular address. If there is a read operation following a write operation, then the read operation can get data directly from the write operation. In this case, no new entry will be generated in the request buffer for the read operation.

5.4 Arrivals to the Request Buffers

In this section, we further analyze the arrival patterns to one request buffer. In particular, we will present the worst case arrival pattern to a request buffer in one cycle.

Essentially, all the write operations are delayed in our memory system by C cycles to be inserted in to a request buffer if they are not merged with other write operations. This doesn't violate our fixed delay Δ, since we only provide the fixed delay Δ for read but not write operations. For a read operation, if it is not merged in the reservation table, a new entry in the request buffer is created immediately upon its arrival. On the other hand, the return data of the read operation will only leave the system after Δ cycles. Therefore, even though in our memory system, there is at most one arriving read or write operation every cycle, it may happen in the request buffer that more than one entry is created in a cycle, or no entry is created at all in a cycle. Consider that the reservation table entries are occupied by pending write operations to different addresses in the same memory bank, and the arrivals are read operations to a new set of different addresses in the same memory bank. In this case, in each cycle, a new read operation entry needs to be created in the request buffer. Also, a write operation which has been in the reservation table for C cycles will be removed and thus it generates a new entry in the same request buffer. Therefore, in one cycle, at most two new entries can be created in the request buffer, with one entry for the read operation and the other for the write operation. Since the read and write operations are to different memory locations, they are mapped to the same memory bank with probability $1/B$ due to the random address permutation function, where B is the total number of request buffers. In the worst case, a specific request buffer experiences bursty arrival of two requests per cycle with probability of $1/B$. Such bursty arrival pattern lasts at most C cycles, which is the maximum number of write requests stored in the reservation table.

It is worth noting that since there is at most one arrival read or write operation to our memory system every cycle, on average, there is strictly less than one entry created in the request buffers every cycle thanks to our operation merging rules. Also, in the system, we use extra memory banks to match the SRAM throughput, i.e., $B > 1/\mu$. So in the long run, our system is stable, since we have more departures than arrivals in the request buffers. However, in a short period of time, a request buffer may still overflow due to certain bursty arrivals. In the next section, we will present probability bounds on such overflow events under the worst case memory access patterns.

6 Performance Analysis

In this section, we prove the main theoretical result of this paper, which bounds the probability that any of the request buffers will overflow for all arrival sequences, including adversarial sequences of read/write operations. 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 sequence of arrival read/write operations to the DRAM banks over a time period $[s,t]$ which is viewed as a parameter setting, we are able to obtain a tight stochastic bound of the number of arrivals of read/write operations to the request buffer of a specific DRAM bank during time period $[s,t]$ using various tail bound techniques. Since our scheme has to work with all possible sequences, our bound clearly has to be the worst case (i.e., the maximum) stochastic bound over all of arrival sequences. However, the space of all such sequences is so large that enumeration over all of them is computationally prohibitive. Moreover, low complexity optimization procedures in finding the worst case bound does not seem to exist. Fortunately, we discover that the number of arrival operations for all of the possible arrival sequences are dominated by a particular sequence, which is also the worst case 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 moment generating functions (MGFs) of the number of arrivals under all other sequences by that under the worst case sequence. However, even this worst case MGF is prohibitively expensive to compute. We solve this problem through upper bounding this MGF by a computationally friendly formula, then applying the Chernoff technique to it.

The rest of this section is organized as follows: in Section 6.1, we describe the overall structure of the tail bound problem, which shows that the overall overflow event $\tilde{D}$ over a time period $[0, n]$ is the union of a set of the overflow events $D_{s,t}$, with $0 \le s < t \le n$, which leads to a union bound. In Section 6.2, we present the mathematical preliminaries. In Section 6.3, we show how to bound the probability of each individual overflow event $D_{s,t}$ using the aforementioned technique of combining convex ordering with large deviation theory.

6.1 Union Bound—The First Step

In this section, we bound the probability of overflowing request buffer Q of a particular DRAM bank. 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 the request buffer, B is the number of DRAM banks, $1/\mu$ is the number of cycles it takes for a DRAM bank to finish one memory transaction, and C is the size of the cache. In the following, we shall fix n and 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 request buffers. The overall overflow probability can be bounded by $B \times \Pr[\tilde{D}]$ (union bound).

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

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

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

$$D_{s,t} \equiv \{\omega \in \Omega : X_{s,t} - \mu(t-s) > K\}. \tag{2}$$

Here, we will say a few words about the implicit probability space Ω, which is the set of all permutations on $\{1, \ldots, N\}$, where N is the number of distinct memory addresses. Since we are considering the worst case bound, we assume the maximum number of read/write operations that can be generated to request buffers during time interval $[s, t]$. Let $\tau = t - s$, and let the maximum number be τ'. When $\tau \le C$, since each request can result in at most one read and one write back, we have $\tau' = 2\tau$. When $\tau > C$, there can be at most C write backs that cannot be accounted for by one of the writes among the τ requests, so $\tau' = \tau + C$. Combining both, we get

$$\tau' = \tau + \min(\tau, C). \tag{3}$$

We assume that the operations to the request buffers are generated following arbitrary pattern, with the only restriction that read requests to the same address cannot repeat within C cycles, and write requests to the same address cannot repeat within C cycles. This is due to the "smoothing" effect of the reservation table, i.e., repetitions within C cycles would be absorbed by the reservation table. With an arbitrary sequence of operations satisfying the above restriction, each instance $\omega \in \Omega$ gives us an arrival sequence to the DRAM request buffers.

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

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 arrived at time z finds the queue of size K full, $X_{y,z}$, the total number of arrivals during time $[y, z]$ is at least $K + 1 + \lfloor \mu(z-y) \rfloor \ge K + \mu(z-y)$. In other words, $D_{y,z}$ happens and $\omega \in D_{y,z}$. This means for any outcome ω in the probability space, if $\omega \in \tilde{D}$, then $\omega \in D_{s,t}$ for some $0 \le s < t \le n$.

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

6.2 Mathematical Preliminaries

The probability $\Pr[D_{s,t}]$ is clearly a random function of the sequence of read/write operations viewed as parameters during the interval $[s, t]$. As mentioned before, it is not possible to enumerate over all possible parameter settings, i.e., memory access patterns, 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 technique for obtaining sharp tail bounds from the MGF of a random variable (in this case X).[1]

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

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

Since this is true for all θ, we have

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

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

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

Definition 2 (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 $0 < \alpha < 1$.*

1. This technique was apparently first used by Bernstein.

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 3 (Convex Order [34], 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.

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

Definition 4 (Stochastic Order [34], 1.2.1). *The random variable X is said to be smaller than the random variable Y in stochastic order (written $X \le_{st} Y$), if $\Pr[X > t] \le \Pr[Y > t]$ for all real t.*

6.3 Worst Case Parameter Setting

In this section, we find the worst case read/write operation sequence for deriving tail bounds for individual $\Pr[D_{s,t}]$ terms. Suppose ℓ distinct memory addresses are read or written back during time interval $[s, t]$, and the counts of their appearances are $m_1, \ldots, m_\ell$, with $\sum_{i=1}^{\ell} m_i = \tau'$, where τ' is defined earlier as $\tau' = \tau + \min(\tau, C)$. We have

$$X = \sum_{i=1}^{\ell} m_i X_i, \tag{5}$$

where X_i is the indicator random variable for whether the ith address is mapped to the DRAM bank. We have

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

But the X_i's are not independent since we are doing selection without replacement here.

With the help of the reservation table, the read operations in the request buffers to the same DRAM address should not repeat within any sliding window of C cycles, and the write back operations in the request buffers to the same DRAM address should not repeat within any sliding window of C cycles. Therefore, none of the counts $m_1, \ldots, m_l$ can exceed $2T$, where $T = \lceil \frac{\tau'}{C} \rceil$. Moreover, let $q_1 = \tau' - (T-1)C$, then only q_1 addresses could have count $2T$. Fig. 6 will help readers understand the relationship of q, r, T as functions of τ'.

We call any vector $m = \{m_1, \ldots, m_\ell\}$ a valid splitting pattern of τ' if the following conditions are satisfied:

$$\begin{cases} 0 \le m_i \le 2T, \\ \sum_{i=1}^{\ell} m_i = \tau', \\ |\{i : m_i = 2T\}| \le q_1. \end{cases} \tag{7}$$

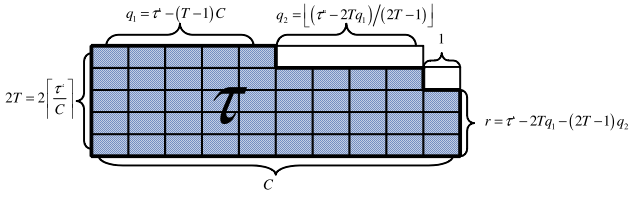

Fig. 6. Relationship of q_1, q_2, r, T, and τ'.

Let $\mathcal{M}$ be the set of all valid splitting patterns.[2] Let $X_m = \sum_{i=1}^{\ell} m_i X_i$. We note that if we reorder the coordinates in m, X_m still maintains the same distribution due to symmetry between the X_i's.

First, we establish a partial order among $\mathcal{M}$.

Lemma 3. *Let $m = \{m_1, \ldots, m_\ell\}$ and $m' = \{m'_1, m'_2, m_3 \ldots, m_\ell\}$ differ only in the first two coordinates, and $0 \le m'_1 \le m_1 \le m_2 \le m'_2$ be such that $m_1 + m_2 = m'_1 + m'_2$. Then, $X_m \le_{cx} X_{m'}$.*

Proof. Let's consider the distribution of (X_1, X_2) conditioned on $X_3, \ldots, X_\ell$. Let $p_{x_1 x_2} = \Pr[X_1 = x_1, X_2 = x_2 | X_3, \ldots, X_\ell]$, and $c = m_3 X_3 + \cdots + m_l X_l$. We have $p_{01} = p_{10}$ due to symmetry of X_1 and X_2. So for any convex function $f(x)$,

$$\begin{aligned} &\mathrm{E}[f(X_m)|X_3, \ldots, X_l] \\ &= \mathrm{E}[f(m_1 X_1 + m_2 X_2 + c)|X_3, \ldots, X_l] \\ &= p_{00} f(c) + p_{10} f(c + m_1) + p_{01} f(c + m_2) \\ &\quad + p_{11} f(c + m_1 + m_2) \\ &\le p_{00} f(c) + p_{10} f(c + m'_1) + p_{01} f(c + m'_2) \\ &\quad + p_{11} f(c + m'_1 + m'_2) \\ &= \mathrm{E}[f(X_{m'})|X_3, \ldots, X_l]. \end{aligned}$$

The inequality is due to Lemma 2 applied to the sequence $c + m'_1, c + m_1, c + m_2, c + m'_2$, and the facts that $p_{01} = p_{10}$ and $m'_1 + m'_2 = m_1 + m_2$.

Now applying the law of total expectation,

$$\begin{aligned} \mathrm{E}[f(X_m)] &= \mathrm{E}[\mathrm{E}[f(X_m|X_3, \ldots, X_l)]] \\ &\le \mathrm{E}[\mathrm{E}[f(X_{m'}|X_3, \ldots, X_l)]] \\ &= \mathrm{E}[f(X_{m'})], \end{aligned}$$

which proves the lemma. □

Remark. Note that stochastic order does not hold here, since $\mathrm{E}[X_m] = \mathrm{E}[X_{m'}] = \sum_{i=1}^{l} m_i / B = \tau'/B, \forall m \in \mathcal{M}$. For stochastic order to hold between two random variables of different distributions, their expectations must differ [34], Theorem 1.2.9.

Let $q_1 = (\tau' - (T-1)C$, $q_2 = \lfloor (\tau' - 2Tq_1)/(2T-1) \rfloor$, $r = \tau' - 2Tq_1 - (2T-1)q_2$. Let m^* be such a splitting pattern: q_1 memory addresses are accessed $2T$ times, q_2 addresses are accessed $2T-1$ times, and one address is accessed r times. We have the following theorem:

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

2. Not all valid splitting pattern may have a plausible read/write sequence matching it, but this does not affect our bound.

Proof. We will first show that if m cannot be reordered to become m^*, there exists $m' \in \mathcal{M}$ such that $X_m \leq_{cx} X_{m'}$.

Suppose $m = \{m_1, \ldots, m_l\}$. If m has less than q_1 items with count $2T$, then it has at least two items with count less than $2T$, say m_1 and m_2. Let $m_1' = m_1 - 1$, $m_2' = m_2 + 1$, $m' = \{m_1', m_2', m_3, \ldots, m_l\}$. It's easy to see that m' has at most q_1 items with count $2T$, so $m' \in \mathcal{M}$. By Lemma 3, $X_m \leq_{cx} X_{m'}$. (If $m_1' = 0$, we can now drop it from m'.).

If m has exactly q_1 items with count $2T$, it must have less than q_2 items with count $2T-1$, so it has at least two items with count less than $2T-1$. Using similar arguments as above, we show that there exists $m' \in \mathcal{M}$ such that $X_m \leq_{cx} X_{m'}$.

As we pointed out earlier, if we reorder the coordinates, X_m will maintain the same distribution. By repeatedly applying the above adjustment and reordering, we see that we can finally reach m^*. Since convex ordering is transitive, $X_m \leq_{cx} X_{m^*}$, $\forall m \in \mathcal{M}$. □

Unfortunately, it is in general not possible to apply the Chernoff bound directly to the MGF of X_{m^*}, as X_{m^*} is the sum of dependent random variables and is very expensive to compute. Our solution is to find a way to upper bound $\mathrm{E}[e^{X_{m^*}\theta}]$ by a more computationally friendly formula. For this purpose, we use a theorem by Hoeffding [35], Theorem 4, which bounds the outcome of sampling without replacement by that of sampling with replacement in the convex order. Then, we apply the Chernoff technique to obtain the following bound.

6.4 Relaxations of X_{m^*} for Computational Purposes

We first state the main theorem of the paper.

Theorem 2.

$$\text{For } \tau \leq C, \quad \Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{2\theta} + (1-\frac{1}{B})\right)^{\tau}}{e^{(K+\mu\tau)\theta}}. \tag{8}$$

$$\text{For } \tau > C, \quad \Pr[D_{s,t}] \leq \min_{\theta>0} \frac{1}{e^{(K+\mu\tau)\theta}} \cdot \left[\frac{1}{B}e^{2T\theta} + \left(1-\frac{1}{B}\right)\right]^{q_1} \times \left[\frac{1}{B}e^{(2T-1)\theta} + \left(1-\frac{1}{B}\right)\right]^{q_2} \cdot \left[\frac{1}{B}e^{r\theta} + \left(1-\frac{1}{B}\right)\right]. \tag{9}$$

The values of q_1, q_2, and r are as defined in Section 6.3.

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 aforementioned 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.

Lemma 4 ([35], 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 the weighted sum of random variables; therefore, we extend Hoeffding's theorem to the following result:

Lemma 5 ([36], Theorem 5). *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. That is, for any convex function f, we have*

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

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

1. *The 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 2 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 there are i.i.d Bernoulli random variables with probability $\frac{1}{B}$. Because $f(x) = e^{x\theta}$ is a convex function of x, from $X_m \leq_{cx} X_{m^*} \leq_{cx} Y$ we get

$$\begin{aligned}\mathrm{E}[e^{X_m\theta}] &\leq \mathrm{E}[e^{X_{m^*}\theta}] \leq \mathrm{E}[e^{Y\theta}] \\ &= \mathrm{E}[e^{(Y_1+\cdots+Y_\tau)\theta}] \\ &= \mathrm{E}[e^{Y_1\theta}]^\tau \\ &= \left(\frac{1}{B}e^{2\theta} + \left(1-\frac{1}{B}\right)\right)^\tau.\end{aligned}$$

2. *The case $\tau > C$.* For $\tau > C$, we have

$$\begin{aligned}X_{m^*} = {} & 2T(X_1 + \cdots + X_{q_1}) \\ & + (2T-1)(X_{q_1+1} + \cdots + X_{q_1+q_2}) + rX_{q_1+q_2+1}.\end{aligned}$$

We can similarly apply Lemma 5 and get

$$\begin{aligned}\mathrm{E}[e^{X_m\theta}] &\leq \mathrm{E}[e^{X_{m^*}\theta}] \leq \mathrm{E}[e^{Y\theta}] \\ &= \mathrm{E}[e^{(2T(Y_1+\cdots+Y_{q_1})+(2T-1)(Y_{q_1+1}+\cdots+Y_{q_1+q_2})+rY_{q_1+q_2+1})\theta}] \\ &= \mathrm{E}[e^{2TY_1\theta}]^{q_1}\mathrm{E}[e^{(2T-1)Y_1\theta}]^{q_2}\mathrm{E}[e^{r\theta}] \\ &= \left[\frac{1}{B}e^{2T\theta} + \left(1-\frac{1}{B}\right)\right]^{q_1} \cdot \left[\frac{1}{B}e^{(2T-1)\theta} + \left(1-\frac{1}{B}\right)\right]^{q_2} \\ &\quad \times \left[\frac{1}{B}e^{r\theta} + \left(1-\frac{1}{B}\right)\right].\end{aligned}$$

Together with (4), we have Theorem 2. □

We can see that the bound in Theorem 2 is translation invariant, i.e., it only depends on $\tau = t - s$. Therefore, the computation cost of (1) is $O(n)$ instead of $O(n^2)$, where n is the length of the total time interval.

In conclusion, we have established the bound for the overflow probability under any read or write sequences. It can be computed through $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 2.

7 Performance Evaluation

In this section, we present evaluation results for our proposed robust memory system in Section 5 using Matlab with parameters set according to the current DRAM and

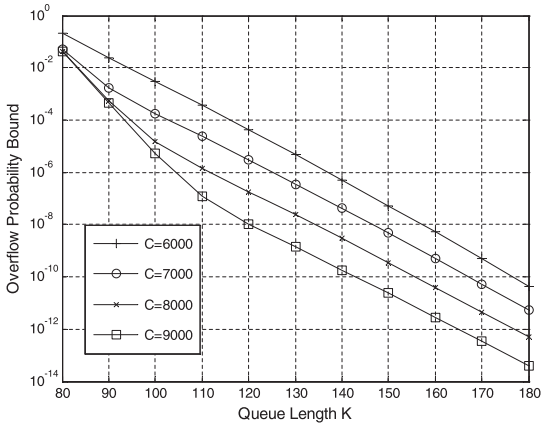

Fig. 7. Overflow probability bound as a function of request buffer size K with $\mu = 1/10$ and $B = 32$.

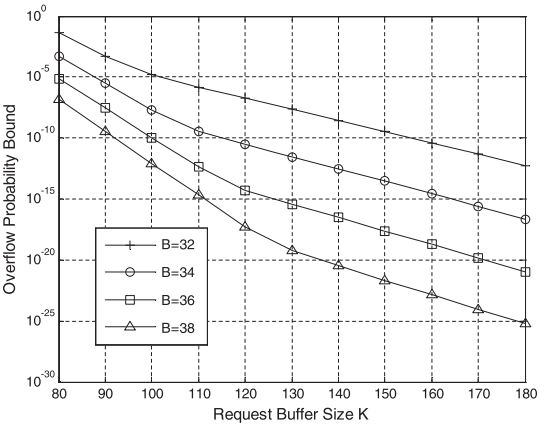

Fig. 9. Overflow probability bound as a function of number of memory banks B with $\mu = 1/10$ and $C = 8{,}000$.

SRAM technology under the worst case memory access patterns that are identified in Section 6. In particular, we use parameters derived from two real-world Internet traffic traces, with one from University of Southern California (USC) and the other from University of North Carolina. 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 data storage for both traces, we set the number of addresses in the DRAM banks to be $N = 16$ million. The performance of our robust memory system on real traffic traces, including the two above, is guaranteed to be better than the bounds shown in this section.

7.1 Numerical Examples of the Tail Bounds

In Fig. 7, the overflow probability bounds with different reservation table sizes C as a function of request buffer sizes K are presented, where $\mu = 1/10$ and $B = 32$. As K increases, the overflow probability bound decreases. With $C \geq 8{,}000$, the overflow probability bound of 10^{-12} is achieved starting from a request buffer of size $K = 180$.

In Fig. 8, the overflow probability bounds with different reservation table sizes C as a function of request buffer sizes K are presented, where $\mu = 1/10$ and $B = 64$. As C approaches ∞, the overflow probability as a function of queue length K becomes the result of the overflow probability of a Geom/D/1 process with arrival probability $1/B = 1/64$ to a queue of length K. By increasing the number of memory banks, with a moderate $C = 3{,}000$, we can achieve an overflow probability bound of 10^{-30} with a request buffer of size only $K = 120$.

In Fig. 9, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/10$ and $C = 8{,}000$. It shows in this figure that by fixing K, the overflow probability bound decreases as B increases.

In Fig. 10, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/10$ and $C = 4{,}000$. As expected, the overflow probability bound is greatly reduced by using more memory banks or larger request buffers.

Now let's consider the size of the reservation table. In each of its entry, one bit is used for *op* to distinguish read and write operations. With $N = 16$ million entries in the DRAM banks, *addr* of size $\log_2 N = 24$ bits is sufficient to address every memory location. The size of the *R-link* is

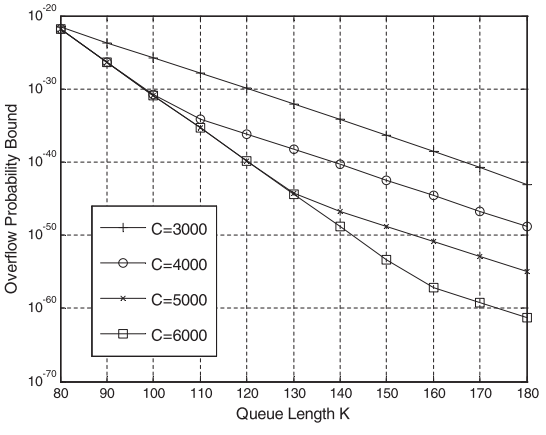

Fig. 8. Overflow probability bound as a function of request buffer size K with $\mu = 1/10$ and $B = 64$.

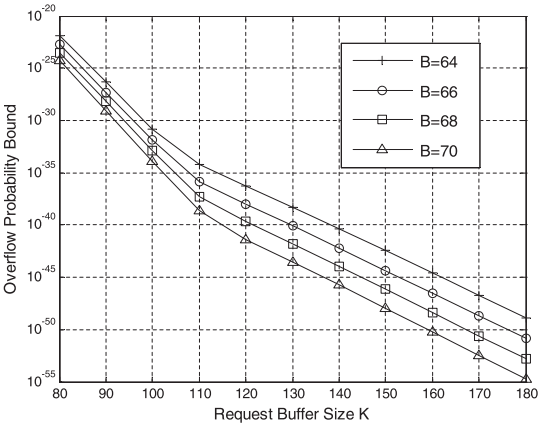

Fig. 10. Overflow probability bound as a function of number of memory banks B with $\mu = 1/10$ and $C = 4{,}000$.

TABLE 1
Comparison of Different Schemes for a Reference Configuration with 16 Million Addresses of 40 Bytes Data, $\mu = 1/10$, $B = 32$, and $C = 8{,}000$

	naïve	VPNM [17]	ours
DRAM	none	640 MB	640 MB
SRAM	640 MB	200 KB	451 KB
CAM	none	176 KB	48 KB
worst case overflow probability bound	0	no guarantee	$< 10^{-12}$

$\log_2 C = 13$ bits, with $C = 8{,}000$. The pending status p takes 1 bit. Let the data size be 40 bytes or 320 bits. Altogether the total size of an entry in the reservation table is 359 bits. With $C = 8{,}000$ entries, its total size is about 351 KB, which can easily fit in SRAM.

For the MRI and MRW tables, only pointers are stored to enable fast searching on the reservation table entries. Each entry in the MRI or MRW table is of size $\log_2 N = 24$ bits, where N is the number of addresses in the DRAM banks. There can be at most C entries in the MRI or MRW, where C is the size of the reservation table. With $C = 8{,}000$, the total size of MRI or MRW is about 24 KB, which can be easily implement in CAMs.

In the request buffers, each entry is a pointer to an entry in the reservation table plus a data field for write operation. A pointer in the request buffers is of size $\log_2 C$. For $C = 8{,}000$, the size of the pointer in the request buffer is 13 bits. Let the size of the data field be 8 bytes. Let $B = 32$ and $K = 180$ to provide with 10^{-12} overflow probability, the total size of the request buffers is only about 55 KB.

It is worth noting that our evaluations are based on the assumption of the worst case scenarios where the requests to the same memory locations are repeated every C cycles. For real-world traffic, the assumption above is far too pessimistic. We expect that much smaller request buffers (K) and much smaller reservation table size (C) will be sufficient for most real-world Internet traffic, which will result in much smaller delay Δ, where $\Delta = K/\mu$.

7.2 Cost-Benefit Comparison

Table 1 compares our proposed approach with a naïve SRAM approach as well as the VPNM architecture approach [17]. The comparison is for $\mu = 1/10$, which is the case for the XDR memory [37], [38]. We assume that there are 16 million entries in the DRAM banks with each entry 40 bytes wide. The naïve SRAM approach simply implements all entries in SRAM. In the VPNM, the delay storage buffer has to be implemented in CAM in order to eliminate redundant memory accesses. Our new robust pipelined memory uses less CAM than the VPNM and comparable amount of SRAM. However, with the parameter setting, the overflow probability of our robust pipelined memory is bounded by 10^{-12}, while there is no such guarantee by the VPNM.

8 Conclusion

We proposed a memory architecture for high-end Internet routers that can effectively maintain wirespeed read/write operations by exploiting advanced architecture features that are readily available in modern commodity DRAM architectures. In particular, we presented an extended memory architecture that can harness the performance of modern commodity DRAM offerings by interleaving memory operations to multiple memory banks. In contrast to prior interleaved memory solutions, our design is robust to adversarial memory access patterns, such as repetitive read/write operations to the same memory locations, by using only a small amount of SRAM and CAM. We presented a rigorous theoretical analysis on the performance of our proposed architecture in the worst case using a novel combination of convex ordering and large deviation theory. Our architecture supports arbitrary read and write patterns at wirespeed of 40 Gb/s or beyond.

Acknowledgments

This work was presented in part at the 29th Conference on Computer Communications (IEEE INFOCOM 2010), San Diego, CA, March 2010 [1]. This work is supported by collaborative US National Science Foundation grants CNS-0904743, CNS-0905169, and CNS-0910592, funded under the American Recovery and Reinvestment Act of 2009 (Public Law 111-5).

References

[1] H. Wang, H. Zhao, B. Lin, and J. Xu, "Design and Analysis of a Robust Pipelined Memory System," *Proc. IEEE INFOCOM,* Mar. 2010.

[2] K. Claffy, H.-W. Braun, and G. Polyzos, "A Parameterizable Methodology for Internet Traffic Flow Profiling," *IEEE J. Selected Areas in Comm.,* vol. 13, no. 8, pp. 1481-1494, Oct. 1995.

[3] A. Kumar, J. Xu, L. Li, and J. Wang, "Space-Code Bloom Filter for Efficient Traffic Flow Measurement," *Proc. IEEE INFOCOM,* 2004.

[4] S. Muthukrishnan, *Data Streams: Algorithms and Applications at Foundations and Trends in Theoretical Computer Science.* Now Publisher, Inc., 2005.

[5] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow," *Proc. Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '04),* pp. 245-256, 2004.

[6] "Juniper Networks Solutions for Network Accounting,"http://www.juniper.net/techcenter/appnote/350003.html, 2011.

[7] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," *Proc. Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '02),* 2002.

[8] S. Iyer and N. Mckeown, "Designing Buffers for Router Line Cards," Technical Report TR02-HPNG-031001, Stanford Univ., Mar. 2002.

[9] J. García, J. Corbal, L. Cerdà, and M. Valero, "Design and Implementation of High-Performance Memory Systems for Future Packet Buffers," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '03),* 2003.

[10] G. Shrimali and N. McKeown, "Building Packet Buffers Using Interleaved Memories," *Proc. Sixth Workshop High Performance Switching and Routing (HPSR '05),* May 2005.

[11] S. Kumar, P. Crowley, and J. Turner, "Design of Randomized Multichannel Packet Storage for High Performance Routers," *Proc. 13th Ann. Symp. High Performance Interconnects (Hot Interconnects '05),* 2005.

[12] M. Kabra, S. Saha, and B. Lin, "Fast Buffer Memory with Deterministic Packet Departures," *Proc. 14th Ann. Symp. High Performance Interconnects (Hot Interconnects '06),* pp. 67-72, Aug. 2006.

[13] H. Wang and B. Lin, "Block-Based Packet Buffer with Deterministic Packet Departures," *Proc. 11th Workshop High Performance Switching and Routing (HPSR '10),* 2010.

[14] A.K. Parekh and R.G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Trans. Networking,* vol. 1, no. 3, pp. 344-357, June 1993.

[15] B.R. Rau, "Pseudo-Randomly Interleaved Memory," *Proc. 18th Ann. Int'l Symp. Computer Architecture (ISCA '91),* 1991.
[16] W. Lin, S.K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proc. Fifth Int'l Symp. High Performance Computer Architecture (HPCA '01),* 2001.
[17] B. Agrawal and T. Sherwood, "High-Bandwidth Network Memory System through Virtual Pipelines," *Proc. IEEE/ACM Trans. Networking,* vol. 17, no. 4, pp. 1029-1041, Aug. 2009.
[18] C. Pandit and S. Meyn, "Worst-Case Large-Deviation Asymptotics with Application to Queueing and Information Theory," *Stochastic Processes and Their Applications,* vol. 116, no. 5, pp. 724-756, 2006.
[19] 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.
[20] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown, "Maintaining Statistics Counters in Router Line Cards," *Proc. 35th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '02),* vol. 22, no. 1, pp. 76-81, Jan. 2002.
[21] S. Ramabhadran and G. Varghese, "Efficient Implementation of a Statistics Counter Architecture," *SIGMETRICS Performance Evaluation Rev.,* vol. 31, no. 1, pp. 261-271, 2003.
[22] M. Roeder and B. Lin, "Maintaining Exact Statistics Counters with a Multi-Level Counter Memory," *Proc. IEEE Global Telecomm. Conf. (GLOBECOM '04),* vol. 2, pp. 576-581, Nov. 2004.
[23] R. Morris, "Counting Large Numbers of Events in Small Registers," *Comm. ACM,* vol. 21, no. 10, pp. 840-842, 1978.
[24] A. Cvetkovski, "An Algorithm for Approximate Counting Using Limited Memory Resources," *SIGMETRICS Performance Evaluation Rev.,* vol. 35, pp. 181-190, 2007.
[25] R. Stanojevic, "Small Active Counters," *Proc. IEEE INFOCOM,* 2007.
[26] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *SIGMETRICS Performance Evaluation Rev.,* vol. 36, pp. 121-132, 2008.
[27] N. Hua, B. Lin, J. Xu, and H. Zhao, "BRICK: A Novel Exact Active Statistics Counter Architecture," *Proc. Fourth ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS '08),* 2008.
[28] H. Zhao, H. Wang, B. Lin, and J. Xu, "Design and Performance Analysis of a DRAM-Based Statistics Counter Array Architecture," *Proc. Fifth ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS '09),* pp. 84-93, Oct. 2009.
[29] F. Baboescu, D.M. Tullsen, G. Rosu, and S. Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA '05),* 2005.
[30] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: Fast and Efficient IP Lookup Architecture," *Proc. Fourth ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS '06),* pp. 51-60, 2006.
[31] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMs: An SRAM-Based Parallel Multi-Pipeline Architecture for Terabit IP Lookup," *Proc. IEEE INFOCOM,* pp. 1786-1794, Apr. 2008.
[32] J. Hasan, S. Chandra, and T.N. Vijaykumar, "Efficient Use of Memory Bandwidth to Improve Network Processor Throughput," *SIGARCH Computer Architecture News,* vol. 31, no. 2, pp. 300-313, 2003.
[33] G. Varghese, *Network Algorithmics.* Elsevier, 2005.
[34] A. Muller and D. Stoyan, *Comparison Methods for Stochastic Models and Risks.* Wiley, 2002.
[35] W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *J. Am. Statistical Assoc.,* vol. 58, no. 301, pp. 13-30, 1963.
[36] H. Zhao, "Measurement and Resource Allocation Problems in Data Streaming Systems," PhD dissertation, Georgia Inst. of Technology, 2010.
[37] F.A. Ware and C. Hampel, "Micro-Threaded Row and Column Operations in a DRAM Core," *Rambus White Paper,* Mar. 2005.
[38] "XDR Datasheet," *Rambus, Inc.,* 2005-2006.

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

Haiquan (Chuck) Zhao received the PhD degree in computer science from the Georgia Institute of Technology, Atlanta, in 2010. He is currently working as a software engineer at Microsoft. He is a student member of the IEEE.

Bill Lin 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 at 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, 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 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 member of the IEEE.