Design and Analysis of a Robust Pipelined Memory System

Hao Wang[1] Haiquan (Chuck) Zhao[2] Bill Lin[1] Jun (Jim) Xu[2]

[1] Department of Electrical and Computer Engineering, University of California, San Diego
Email: {wanghao,billlin}@ucsd.edu
[2] College of Computing, Georgia Institute of Technology
Email: {chz,jx}@cc.gatech.edu

***Abstract*—Many network processing applications require wirespeed access to large data structures or a large amount of flow-level data, but the capacity of SRAMs is woefully inadequate in many cases. 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. 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 even under adversarial memory access patterns, which we demonstrate through a rigorous worst-case theoretical analysis using a combination of convex ordering and large deviation theory.**

I. 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 tradeoffs between the speed and cost of SRAM and DRAM. While fitting all such data into SRAM (access latency typically between 5 to 15ns) 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 (access latency typically between 50 and 100ns) is too slow for providing wirespeed updates. For example, on an 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 need to be completed within this time frame. In this work, we do away with this unfortunate tradeoff 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.

A. Motivation

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

Network flow state: Maintaining precise flow state information 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 address 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.

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 have 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 [1], [2], [3], [4]. Although these designs are both effective and practical, these solutions are problem-specific. Ideally, we would like a general memory architecture that is applicable to both this packet buffering problem as well as the various network flow state implementation problems.

B. 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 results will be available at output exactly at time $t+\Delta$, where Δ is a fixed pipeline delay. 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 a read data is 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 *timestamp* 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 queueing [5], 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, all flow state information needed for scheduling all packets that arrive before or at time t are 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 [6], [7] 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 [8] is a virtually-pipelined memory architecture that aims to mimic the behavior of a single SRAM bank using multiple DRAM and SRAM banks. All the operations that can be stored in the buffer 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, even under arbitrary (including adversarial) memory access patterns. 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 queueing delays (to be made precise in Section V). This result is a *worst-case large deviation theorem* in nature [9] because it establishes a bound on the largest (worst case) value among the tail probabilities of having long queueing delays under all admissible (including adversarial) memory access patterns. Our methodology for proving this challenging mathematical result is a novel combination of convex ordering and (traditional) large deviation theory.

C. Outline of Paper

The rest of the paper is organized as follows. Section II defines the notion of SRAM emulation. Section III describes the basic memory architecture. Section IV extends our proposed memory architecture providing robustness against adversarial access patterns. Section V provides a rigorous analysis on the performance of our architecture in the worst-case. Section VI presents an evaluation of our proposed architecture. Finally, Section VII concludes the paper.

II. 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 the SRAM-to-DRAM random access latency ratio is μ (e.g. μ = 4ns/40ns = 1/10, where the access latencies of SRAM and DRAM are assumed to be 4 ns and 40 ns respectively). 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 $\frac{1}{B\mu} < 1$. For simplicity, we assume that an ideal SRAM can complete a read or write operation in the same cycle that the operation is issued. On the other hand, it takes a DRAM bank $1/\mu$ cycles to finish a read or write operation. A 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 is 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, ..., 5 respectively. If the memory is SRAM, then the read accesses at times 0, 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 Δ, $4 + \Delta$, $5 + \Delta$ respectively.

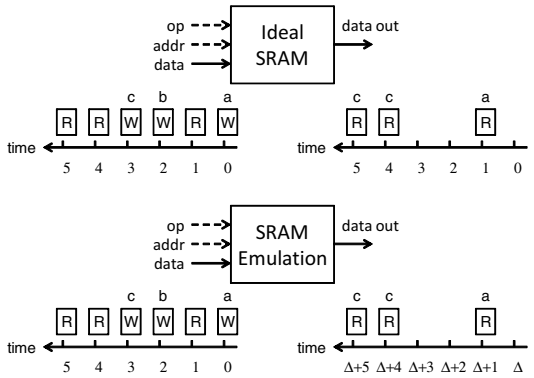

Fig. 1. SRAM Emulation

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 bank 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 is required to update the memory to provide coherency, whereas all write operations correspond to actual memory updates in an ideal SRAM.

III. 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.

A. 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 $(t + \Delta) \mod \Delta$. Each reservation table entry consists of three parts: a 1-bit field to specify the operation (read or write), the memory address, and the data field. The data field stores the data to be written to or to be retrieved from the memory.

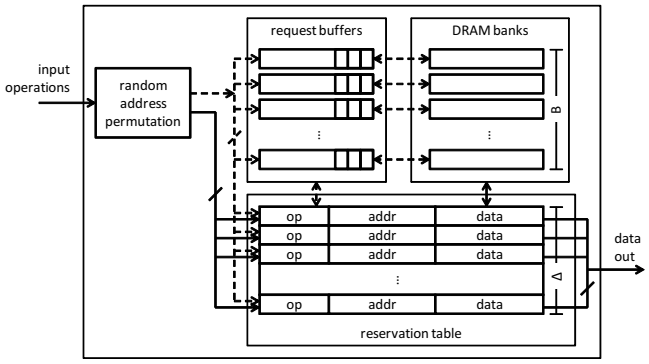

Fig. 2. Basic Memory Architecture

- *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 V 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 $1/B$. Unless otherwise noted, when referring to a memory address, we will be referring to the address after random permutation. Note that if the incoming operations access the same memory location, then they will still generate entries to the same request buffer.

B. Operations

For a read operation issued at time t, its completion time is $h = t + \Delta$. A reservation table entry is created at location $h \mod \Delta$. 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 gets copied to the output. The reservation table entry is removed after Δ cycles.

For a write operation issued at time t, a reservation table entry is created at location $h \mod \Delta$, 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 is 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.

C. 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. 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 IV effectively utilizes the reservation table in SRAM as a cache to eliminate unnecessary DRAM operations queued in the request buffers.

IV. 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 details 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

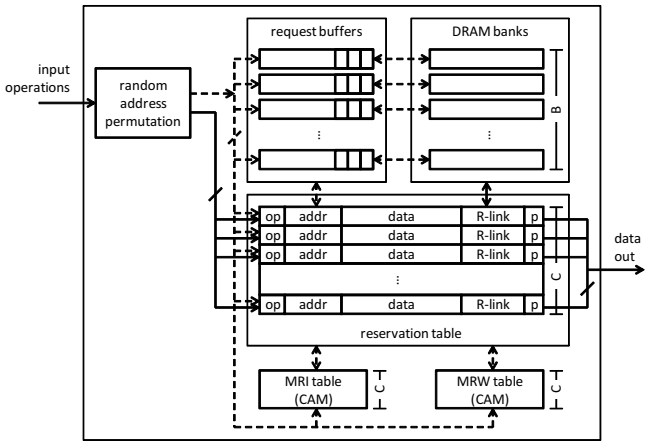

Fig. 3. Extended Memory Architecture

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 non-pending 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 additional 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 are implemented using Content-Addressable-Memories (CAM) 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.

A. 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 a MRI (or MRW) lookup, if it returns a reservation table entry with different address (*addr*), then we also remove this MRI (or MRW) entry.

B. 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:
  1) We create a 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.
  2) 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.

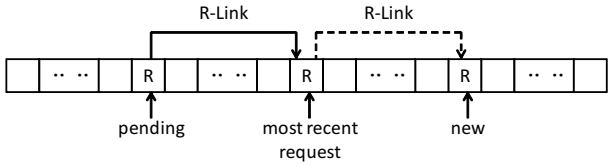

Fig. 4. R-link for read operation

- *Case 3*: The most recent operation to the same address returned by the MRI lookup is a write operation or a non-pending read operation. In this case, we copy the data from this most recent write or non-pending read operation to the new R location. In this way, R is guaranteed to return the most recent write or non-pending 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 is 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 three 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 we create an entry in the reservation table for W. Also, an MRW lookup is 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 there is such a write operation (i.e. the MRW lookup returns a different memory address), then no request buffer is created for W. Otherwise, there is no other write operation in the reservation table, and the MRW lookup returns the address of W operation. Then a new entry in the request buffer is generated for operation W to update the corresponding DRAM bank location. Also, we delete the entry in MRW pointing to the operation W.

C. 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 operations. 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 are shown on the right-hand-side of the arrows.

1) RW $\rightarrow$ 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) WW $\rightarrow$ W: The earlier (right) W operation does not generate new entry in the request buffers.
3) RR $\rightarrow$ 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) WR → WR: Both the R operation and the W operation have to access the DRAM banks. They can not be merged. Entries in the request buffer needs to be created for the W and R operations.

For the last rule, WR → WR, even though the operations cannot be merged, the latter incoming operation to the same memory address can be merged. Consider the following two cases (input sequence from right to left):

- (RW)R → (W)R: The last two (RW) merged to (W) using the RW → W rule.
- (WW)R → (W)R: The last two (WW) merged to (W) using the WW → 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 are 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. 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 operations 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 entry 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 with a fixed delay C, and therefore the system is data coherent.

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

- There can be only one write operation in the 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 are more than one arriving read operations to the same address within C cycles, we are guaranteed that all except the first read are merged using R-link.
- 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. Therefore, no new entry will be generated in the request buffer for the read operation.

V. 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 (including any adversarial) sequences of read/write operations. As explained earlier, this worst-case large deviation result is proved using a novel combination of convex ordering and (traditional) large deviation theory.

The main idea of our proof is as follows. Given any sequence of read/write operations (to the DRAM banks) over a time period $[s, t]$ (viewed as a parameter setting), we are able to obtain a tight stochastic bound of the number of arrivals of read/write operations to the request buffer of a DRAM bank during $[s, t]$ 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 them. However, the space of all such sequences is so large that enumeration over all of them is computationally prohibitive and low complexity optimization procedures in finding the worst case does not seem to exist. Fortunately, we discover that the aforementioned number of arrivals under all these sequences are dominated by that under a particular (i.e., worst-case) sequence, in the convex order (not in the stochastic order). Since $e^{\theta x}$ is a convex function, we are able to upper-bound the moment generating functions (MGF) 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 Chernoff technique to it.

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

A. 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 operations 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, μ is the SRAM-to-DRAM random access latency ratio, C is the size of the cache. In the following, we shall fix n and will therefore shorten $\tilde{D}_{0,n}$ to $\tilde{D}$. Note that $\Pr[\tilde{D}]$ is the overflow probability for just one out of B such 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 letting $X_{s,t}$ denote the number of read/write operations (to the DRAM bank) generated during time interval $[s, t]$, then we have

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

Here we will say a few words about the implicit probability space Ω, which is the set of all permutations on $\{1, ..., N\}$, 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

$$\tau' = \tau + \min(\tau, C) \tag{2}$$

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 can not repeat within C cycles, and write requests to the same address can not 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. Given an arbitrary sequence of operations satisfying the above restriction, then 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:* $\tilde{D} = \bigcup_{0 \le s \le t \le n} D_{s,t}$

We omit its proof here as it is straightforward from elementary queueing theory and is identical to that of [10, Lemma 1].

B. Bounding Individual $\Pr[D_{s,t}]$

In this subsection we find the worst-case read/write operation sequence for deriving tail bounds for individual $\Pr[D_{s,t}]$ terms. 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., sequences) to find the worst-case $\Pr[D_{s,t}]$ bound. Fortunately, convex ordering comes to our rescue by allowing us to analytically bound the MGF of $X_{s,t}$ under all parameter settings by that under a worst-case setting. For simplicity, in this section we will drop the subscripts of $X_{s,t}$ and use X instead.

1) Mathematical Preliminaries: 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].

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

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

Since this is true for all θ, we have

$$\Pr[D_{s,t}] \le \min_{\theta > 0} \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}}. \tag{3}$$

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 are needed to establish the bound, we present the definition of convex function and convex ordering here:

***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.

***Definition** 3 (Convex order [11, 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 $X \leq_{cx} Y$), if $E[f(X)] \leq E[f(Y)]$ holds for all real convex functions f such that the expectations exist.

2) Worst-Case Parameter Setting: In this section, we specify the worst-case parameter setting (in the sense of convex ordering) and prove it is indeed the worst-case.

Let $m_i, 1 \leq i \leq N$ be the total number of read and write operations generated to the request buffers for the i^{th} address during time interval $[s,t]$. So $\sum_{i=1}^{N} m_i = \tau'$, where τ' is defined in (2). We have $X = \sum_{i=1}^{N} m_i X_i$, where X_i is the indicator random variable for whether the i^{th} address is mapped to the DRAM bank. We have $E[X_i] = \frac{1}{B}$. But the X_i's are not independent since we are doing permutation on the addresses.

As explained earlier, among operations generated to request buffers, read requests to the same address can not repeat within C cycles, and write requests to the same address can not repeat within C cycles. Therefore none of the counts $m_1, ..., m_N$ can exceed $2T$, where $T = \lceil \frac{\tau}{C} \rceil$. One can achieve $m_i = 2T$ by issuing pairs of a read operation followed by a write operation for the i^{th} address every C cycles.[2] Moreover, let $q_1 = \tau - (T-1)C$, then at most q_1 addresses could have count $2T$.

We call any vector $m = \{m_1, ..., m_N\}$ a valid splitting pattern of τ' if the following are satisfied: $0 \leq m_i \leq 2T$, $\sum_{i=1}^{N} m_i = \tau'$, $|\{i : m_i = 2T\}| \leq q_1$. Let $\mathcal{M}$ be the set of all valid splitting patterns. [3] Let $X_m = \sum_{i=1}^{N} m_i X_i$.

Let $q_2 = \lfloor (\tau' - 2Tq_1)/(2T-1) \rfloor$, $r = \tau' - 2Tq_1 - (2T-1)q_2$. Let m^* be such a splitting pattern: $m_1 = \ldots = m_{q_1} = 2T$, $m_{q_1+1} = ... = m_{q_1+q_2} = 2T-1$, $m_{q_1+q_2+1} = r$, and the rest of m_i are 0. We omit the proof of the following theorem since it is similar to that of [10, Theorem 2].

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

Remark: Note that stochastic order does not hold here, since $\mathrm{E}[X_m] = \mathrm{E}[X_{m^*}] = \tau'/B$. For stochastic order to hold between two random variables of different distributions, their expectations must differ [11, Theorem 1.2.9].

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 lemma by Hoeffding [12, 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. We omit the proof here since it is similar to those of [10, Theorem 3] and [10, Theorem 4].

[2]Strictly speaking it is every $C+1$ cycles.

[3]Not all valid splitting pattern may have a plausible read/write sequence matching it, but this does not affect out bound.

***Theorem** 2:*

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

For $\tau > C$,

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{e^{(q_1 e^{2T\theta} + q_2 e^{(2T-1)\theta} + e^{r\theta} - q_1 - q_2 - 1)/B}}{e^{(K+\mu\tau)\theta}}.$$

We can see that the bound is translation invariant, i.e. it only depends on $\tau = t - s$. Therefore, the computation cost of (1) is $O(n)$ instead of $O(n^2)$, where n is the 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 though $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 2.

VI. Evaluation

In this section, we present evaluation results for our proposed extended memory architecture described in Section IV. In particular, we used parameters derived from two real-world Internet traffic traces for our evaluations. 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 data storage for both traces, we set the number of addresses in the DRAM banks to be $N = 16$ million.

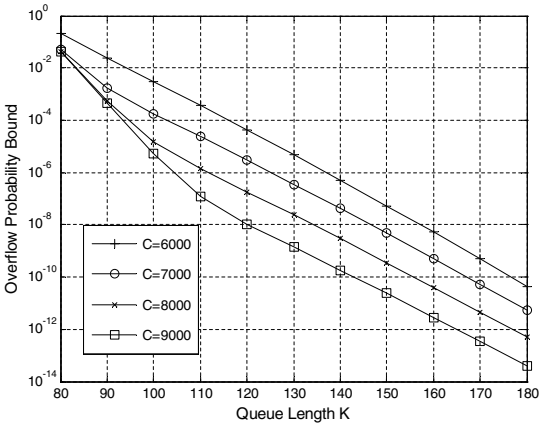

Fig. 5. Overflow probability bound as a function of request buffer size K with $\mu = 1/10$ and $B = 32$.

In Fig. 5, 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

$C \geq 8000$ we can achieve an overflow probability bound of 10^{-12} starting from a request buffer of size $K = 180$.

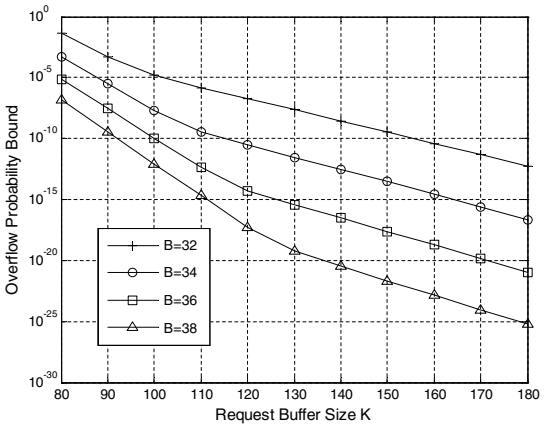

Fig. 6. Overflow probability bound as a function of number of memory banks B with $\mu = 1/10$ and $C = 8000$.

In Fig. 6, 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 $C = 8000$. It can be seen from this figure that given the same K as B increases, the overflow probability bound decreases.

Now let's consider the size of the reservation table. For each entry in a reservation table, one bit is for op to distinguish read and write operations. With $N = 16$ millions entries in the DRAM banks, addr of size $\log_2 N = 24$ bits is sufficient to address every memory location. The size of R-link is $\log_2 C = 13$ bits, with $C = 8000$. Moreover the size of the pending status p is 1 bit. Let the data size be 8 bytes or 64 bits. Altogether the total size of each entry in the reservation table is 103 bits. For a reservation table with $C = 8000$ entries, its total size is about 101 KB, which can be easily implemented 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 $C = 8000$, 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. An entry in the request buffers is of size $\log_2 C$. For $C = 8000$, 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 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 $\Delta = K/\mu$.

VII. 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 location, 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.

Acknowledgement: This work is supported in part by collaborative NSF grants CNS-0905169 and CNS-0904743, funded under the American Recovery and Reinvestment Act of 2009 (Public Law 111-5), and NSF grant CNS-0716423.

References

[1] S. Iyer and N. Mckeown, "Designing buffers for router line cards," Stanford University, Tech. Rep. TR02-HPNG-031001, Mar. 2002.

[2] J. Garca, J. Corbal, L. Cerd, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *International Symposium on Microarchitecture (MICRO)*, 2003.

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

[4] S. Kumar, P. Crowley, and J. Turner, "Design of randomized multichannel packet storage for high performance routers," in *Symposium on High Performance Interconnects (HOTI)*, 2005.

[5] 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, 1993.

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

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

[8] B. Agrawal and T. Sherwood, "Virtually pipelined network memory," in *Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 197–207.

[9] 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.

[10] H. Zhao, H. Wang, B. Lin, and J. Xu, "Design and performance analysis of a dram-based statistics counter array architecture," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

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

[12] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.