DRAM is Plenty Fast for Wirespeed Statistics Counting

Bill Lin
Electrical and Computing Engineering
University of California, San Diego
billlin@ece.ucsd.edu

Jun (Jim) Xu
College of Computing
Georgia Institute of Technology
jx@cc.gatech.edu

ABSTRACT

Per-flow network measurement at Internet backbone links requires the efficient maintanence of large arrays of statistics counters at very high speeds (e.g. 40 Gb/s). The prevailing view is that SRAM is too expensive for implementing large counter arrays, but DRAM is too slow for providing wirespeed updates. This view is the main premise of a number of hybrid SRAM/DRAM architectural proposals [2, 3, 4, 5] that still require substantial amounts of SRAM for large arrays. In this paper, we present a contrarian view that modern commodity DRAM architectures, driven by aggressive performance roadmaps for consumer applications (e.g. video games), have advanced architecture features that can be exploited to make DRAM solutions practical. We describe two such schemes that can harness the performance of these DRAM offerings by enabling the interleaving of counter updates to multiple memory banks. These counter schemes are the first to support arbitrary increments and decrements for either integer or floating point number representations at wirespeed. We believe our preliminary success with the use of DRAM schemes for wirespeed statistics counting opens the possibilities for broader opportunities to generalize the proposed ideas for other network measurement functions.

1. INTRODUCTION

It is widely accepted that network measurement is essential for the monitoring and control of large networks. For tracking various network statistics and for implementing various network measurement, router management, and data streaming algorithms, there is often the need to maintain very large arrays of statistics counters at wirespeeds (e.g. many million counters for per-flow measurements). In general, each packet arrival may trigger the updates of multiple per-flow statistics counters, resulting in possibly tens of millions of updates per second. For example, on an 40 Gb/s OC-768 link, a new packet can arrive every 8 ns and the corresponding counter updates need to be completed within this time. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows if the measurements take place at high-speed links as smaller counters can quickly overflow. 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 [5], real-world Internet traffic traces show that a very large number of flows can occur during a measurement period. For example, an Internet traffic trace from UNC has 13.5 million flows. Assuming 64 bits for each flow counter, 108 MB of SRAM would already be needed for just the counter storage, which is prohibitively expensive. Therefore, researchers have actively sought alternative ways to realize large arrays of statistics counters at wirespeed [2, 3, 4, 5].

In particular, several designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed. Their baseline 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 become close to overflow, they will be scheduled to be "flushed" back to the corresponding DRAM counter. These schemes all significantly reduce the SRAM cost. In particular, the scheme by Zhao et al. [5] achieves the theoretically minimum SRAM cost of say 4 to 5 bits per counters when the speed difference between SRAM and DRAM ranges between 10 (40ns/4ns) and 20 (80ns/4ns). While this is a huge reduction over a straightforward SRAM implementation, storing say 4 bits per counter in SRAM for 13.5 million flows would still require nearly 7 MB of SRAM, which is a substantial amount and difficult to implement on-chip. Moreover, since the bounds on SRAM requirements for the hybrid SRAM/DRAM approaches are based on preventing SRAM counter overflows, the SRAM requirements are also dependent on the *size* of the increments. If a wide range of increments is needed, and *large* increments are possible, then the possibility for overflows could occur earlier and more SRAM counter bits would be needed to compensate, resulting in yet larger SRAM requirements. In addition, the existing hybrid SRAM/DRAM approaches do not support arbitrary decrements and are based on an integer number representation, whereas a *floating point number* representation may be needed in some applications [16, 17].

1.1 DRAM Solutions Can Be Plenty Fast

In this paper, we take a *contrarian* view that challenges the main premise of the hybrid SRAM/DRAM architecture proposals. Their main premise is that DRAM access laten-

ACM HotMetrics'08, June 6, 2008, Annapolis, MD

cies are too slow for wirespeed updates, though DRAMs provide plenty of storage capacity for maintaining exact counts for large arrays of counters. However, our main observation is that modern DRAM architectures have advanced architecture features [8, 9, 10, 11] that can be exploited to make a DRAM solution practical.

Driven by a seemingly insatiable appetite for extremely aggressive memory data rates in graphics, multimedia, video game, and high-definition television applications, the memory semiconductor industry has continually been driving very aggressive roadmaps in terms of ever increasing memory bandwidths that can be provided at commodity pricing (about $0.01/MB as of this writing). For example, the Cell processor from IBM/Sony/Toshiba [6] uses two 32-bit channels of XDR memories [12] with an aggregated memory bandwidth of 25.6 GB/s. Using an approach called microthreading [11], the XDR memory architecture provides internally 16 independent banks inside just a *single* DRAM chip. Next generation memory architectures [13] are expected to achieve a data rate upwards of 16 GB/s on a single 16-bit channel, 64 GB/s on an equivalent dual 32-bit channel interface used by the Cell processor. This enormous amount of memory bandwidth can be shared or time-multiplexed by multiple network functions. The Intel IXP network processor [7] is another example of a state-of-the-art network processor that has multiple high-bandwidth memory channels. Besides XDR, other memory consortia have similar capabilities and advanced architecture features on their roadmaps as well as they are driven by the same demanding consumer applications. For example, extremely high data efficiency can be achieved using DDR3 memories as well [10].

Although these modern high-speed DRAM offerings provide extraordinary memory bandwidths, the peak access bandwidths are only achievable when memory locations are accessed in a *memory interleaving mode* (to ensure that *internal memory bank conflicts* are avoided). Conventional wisdom is that the random access nature of network measurement applications would *render such access modes unusable*. For example, for XDR memories [12], a new memory operation could be initiated every 4 ns when the internal memory banks are interleaved, but a worst-case access latency of 38 ns is required for a read or a write operation if memory bank accesses are unrestricted.

Our main observation is that memory interleaving *can* be effectively used to maintain wirespeed updates to large counter arrays by employing new counter management schemes. In particular, we describe two such schemes in this paper – one based on the *replication* of counters across memory banks, and the other based on the *randomized distribution* of counters across memory banks.

The first scheme, called the *replicated counter scheme*, works as follows. The main idea in the replicated counter scheme is to maintain *multiple copies of each counter*. Suppose the DRAM access latency is b times slower than the corresponding SRAM (e.g. $b = \lceil 38/4 \rceil = 10$). The replicated counter scheme works by using b DRAM banks and keeping b copies of the same counter C_i as $c_i[0], c_i[1], \ldots, c_i[b-1]$, one in each DRAM bank. Then, for each update to C_i, we read the k^{th} copy of C_i, namely $c_i[k]$, in the k^{th} memory bank, where $k = t \mod b$ rotates in a round-robin manner with respect to the cycle t. As in [5], each *cycle* is defined as two time slots, one for reading from SRAM, and one for writing back to SRAM. With a DRAM-to-SRAM latency ratio of b, the update would take b cycles to complete for performing both a DRAM read as well as a DRAM write. However, since we are *interleaving* across b DRAM banks, *we can initiate a new counter update every cycle*, which enables wirespeed throughput.

The second scheme, called the *randomized counter scheme*, works as follows. To randomly distributed counters across b memory banks, we apply a random permutation function to the counter index to obtain a randomly permuted index and a memory bank location. At each memory bank k, we maintain a small update request queue Q_k of pending update requests. If an update to counter C_i is required and C_i is store in memory bank k, then an update request is inserted into Q_k. Using stochastic and queuing analysis, we show that only a very small queue (on the order of 80 entries) is required to ensure a negligible overflow probability (say under 10^{-9}). Then the requests at the *head* of the request queues can be serviced in an *interleaving order*.

As we will see, for both schemes, we can leverage the internal independent memory banks already available inside a modern DRAM chip without the expense of multiple parallel memory channels, thus making both schemes very cost effective. Moreover, since the counters are only stored in DRAM, and there is no dependence on a notion of "flushing a partial counter before it overflows," both schemes can support arbitrary increments and decrements, and both schemes can support different number representations, including unsigned integers, signed integers (if decrements are needed), or floating point numbers.

1.2 Summary of Main Points

The main points of this paper are as follows:

- Contrary to the prevailing view that DRAM is too slow for wirespeed maintanence of large arrays of statistics counters, we observe that modern commodity high-bandwidth memories provide advanced architecture features that can be exploited to make DRAM solutions practical, and that consumer applications like video games and high-definition television drive extremely aggressive roadmaps in terms ever increasing memory bandwidths at commodity pricing.
- We propose two DRAM-based schemes that can harness the performance of these DRAM offerings by enabling the interleaving of updates to multiple memory banks.
- We present concrete evaluations of our proposed schemes using the XDR memory architecture [11, 12, 13], which has 16 internal independent memory banks in each memory chip, and we show that wirespeed performance can be achieved without the need for (or expense of) multiple memory channels.
- We further note that modern broadband engines such as the cell processor [6] and modern network processors such as the Intel IXP 2855 [7] already incorporate multiple high-speed memory channels to support their intended applications, and such aggregate memory bandwidths can be shared by multiple network functions. Also, higher rates of counter updates can be achieved by leveraging these additional memory channels.
- In comparisons with existing hybrid SRAM/DRAM counter architectures [2, 3, 4, 5], we can achieve the same update rates to counters, but without the need for a non-trivial amount of SRAMs for storing partial counts.

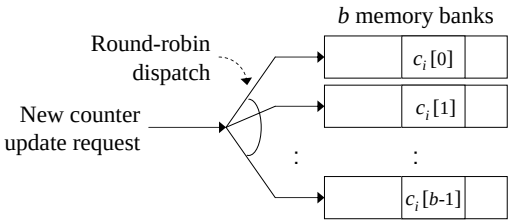

Figure 1: Memory architecture for replicated counter scheme.

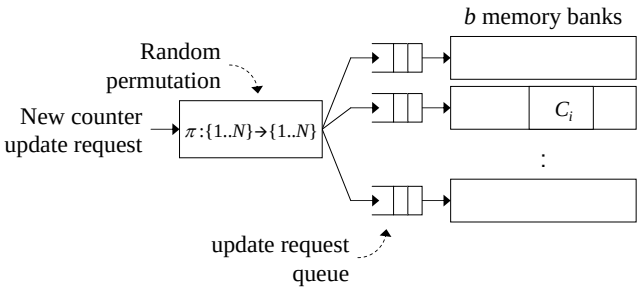

Figure 2: Memory architecture for randomized counter scheme.

- Since our DRAM solutions are *not dependent* on the rate of counter increments, as in hybrid SRAM/DRAM architectures, they can naturally handle increments or decrements of *arbitrary* amounts. For example, flow counters may need to be incremented by different byte amounts depending on the size of the packet arrived.
- Further, our schemes are the first to applicable to different number representations at wirespeed, including integers (signed or unsigned) and floating point numbers.
- Finally, we believe our preliminary success with the use of DRAM schemes for the statistics counting problem can pave the way to broader opportunities for new innovative DRAM solutions for other network measurement functions.

2. DRAM-BASED SOLUTIONS

Memory interleaving has in the past been successfully used for improving the performance of computer systems [9, 14], for graphics or video intensive applications [11], and for implementing routing functions like high-performance packet buffers [15]. In this section, we describe how this technique can be employed for statistics counting.

2.1 Replicated Counter Scheme

As introduced in Section 1.1, our replicated counter scheme is based on the simple idea of maintaining b copies of each counter C_i as $c_i[0], c_i[1], \ldots, c_i[b-1]$, one in each DRAM bank, where b is the DRAM-to-SRAM latency ratio. This is depicted in Figure 1. If a counter update to counter C_i occurs at cycle t, then the update is performed to the copy $c_i[k]$ at the k^{th} memory bank. The actual read and write operations are interleaved at the time slot level. In particular, suppose a read operation is initiated at time s for some counter $c_i[k]$ at the k^{th} memory bank. The data will be available b time slots later. Meanwhile, read operations for subsequent memory banks in the interleaving order can be initiated. That is, a read operation is initiated at time $s+1$ for some counter $c_j[k+1]$ at the $(k+1)^{th}$ memory bank. After b time slots later, $s+b$, the counter value $c_i[k]$ is received from the k^{th} memory bank. Then a write operation for the incremented counter value $(c_i[k]+1)$ can be initiated at cycle $s+b+1$. Similarly, a write operation for the updated counter value (e.g. $c_j[k+1]+1$) can be initiated at cycle $s+b+2$, and so on. The update operation can be arbitrary increments or decrements. After b cycles (or $2b$ time slots), we return back to initiating read operations for the next batch of counter updates. This replicated counter scheme is depicted in Figure 2.1.

After the measurement period, the full actual value of a counter can be obtained by summing over its b copies. Although we require b time slots (or $b/2$ cycles) to retrieve a component $c_i[k]$ from the k^{th} DRAM bank, we can initiate a read transaction to the next DRAM bank in the next time slot in an interleaving manner, one read transaction initiated every time slot (or every 1/2 cycle). Therefore, the last component $c_i[b-1]$ would be retrieved from the last DRAM bank by the end of the $2b^{th}$ time slot, and all earlier components would have already arrived. The total count, $C_i = \sum_{k=0}^{b-1} c_i[k]$, can be accumulated on the fly as the corresponding components are retrieved. Thus, our *one-of* random retrieval time is $2b$ time slots, which is adequate for the intended applications that primarily need wirespeed update speeds. However, for reporting purposes *after* a measurement period, there is often a need to retrieve count statistics from a large *batch* of counters, or possibly from all counters. In this case, we can start retrieving the next counter from the same memory bank after b time slots later, and thereby achieving an effective retrieval rate of one full counter every b time slots, twice as fast as a one-of retrieval.

2.2 Randomized Counter Scheme

Figure 2 depicts the randomized counter scheme. We again use b memory banks to store the counters, but each counter is only stored in one location. The basic idea is to randomly distribute the counters evenly across the b memory banks so that with high probability each memory bank will receive about one out of b counter updates to it on average. This is achieved by applying a pseudorandom permutation function $\pi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$ to a counter index to obtain a permuted index. We then use a simple location policy where counter C_i will be stored in the k^{th} memory bank, where $k = \pi(i) \mod b$, at address location $a = \lfloor \pi(i)/b \rfloor$.

As discussed in Section 1.1, we maintain a small update request queue Q_k of pending updates for each memory bank. To update counter C_i that is stored in the k^{th} memory bank, an update request is inserted into Q_k. These request queues are then serviced in an interleaving order.

To bound the size of these request queues, we assume that it would be extremely difficult for an adversary to purposely trigger consecutively counter updates to the same memory bank since the pseudorandom permutation function is *not* known to the outside world. An adversary can only try to trigger consecutive counter updates to the same counter, We will defer to later to address how this adversarial situation can be mitigated.

Deferring the aforementioned adversarial situation, and given the explicit randomization, the worst-case workload to an arbitrary DRAM bank can be modeled as a geometric

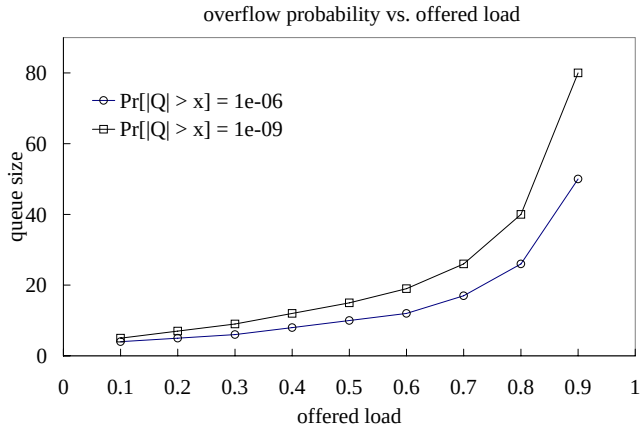

Figure 3: Queue size vs. overflow probabilities.

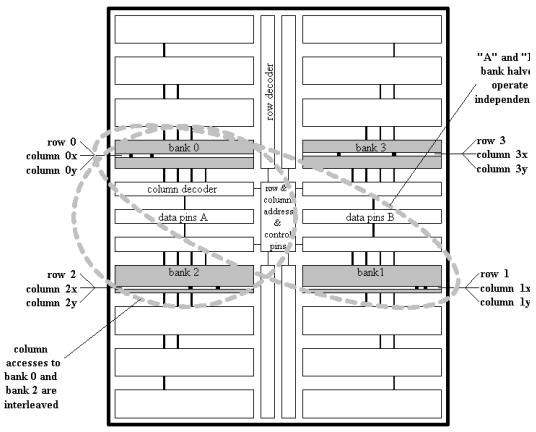

Figure 4: An example high-bandwidth DRAM architecture. Each XDR memory IC has 16 internal memory banks that can be interleaved.

arrival process with an arrival rate of at most once every b time slots, where b is the number of DRAM banks. Correspondingly, the probability that the request queue will overflow some threshold x can be analyzed using the steady state probability of the unfinished work exceeding a certain level x for a geometric source as a surrogate. Given that the interleaving interval for a particular DRAM bank is a constant, and DRAM access is guaranteed within this interval, we can regard the service process as deterministic. Therefore, the overflow probability of a request queue can be derived as the overflow probability of a corresponding Geom/D/1 system, which is known. Figure 3 plots the queue sizes necessary to guarantee steady state overflow probabilities of 10^{-6} and 10^{-9}. For example, to ensure an overflow probability of 10^{-9} for traffic load up to 90%, about 80 entries per queue is sufficient. In practice, operators usually design their networks to operate well below this offered load to ensure low queuing delays and packet drops.

We now return to the discussion about adversarial traffic. An adversary can try to trigger consecutive counter updates to the same memory bank by triggering updates to the same counter. This can be mitigated by keeping a small hash table H of pending update requests that are already in the queues. If a new counter update request arrives for counter C_i, we can lookup H to see if there is already a pending update request to this counter. If there is, then we can just simply modify that request rather than creating a new one (e.g. change the request from "+1" to "+2"). The hash table needs to store as many entries as there are entries in the update request queues, which is small since the request queues are small. We can use hash table implementations such as d-left hashing [18, 19, 20] to support insertion, lookups, and deletion in worst-case constant time.

3. PRELIMINARY EVALUATIONS

We present preliminary numerical results on the two DRAM-based counter array architecture schemes described in Sections 2.1 and 2.2. 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 to the campus to the rest of the Internet on April 24, 2003. The trace from USC has 120.8 million packets and around 8.6 million flows, and the trace segment from UNC has 198.9 million packets and around 13.5 million flows. To support sufficient counters for both traces, we set the counter array configuration to support $N = 16$ million counters.

The numerical results are presented in Table 1. In particular, we compare our proposed schemes with a "naive" approach of implementing all counters in SRAM. We also compare our schemes with the hybrid SRAM/DRAM counter architecture approach [2, 3, 4, 5]. Specifically, we compare against the state-of-the scheme proposed by Zhao et al. [5] that provably achieves the minimum SRAM requirement for this architecture class. As demonstrated in [5], their approach requires a factor of *six times* less SRAM than the first hybrid SRAM/DRAM solution proposed in [2] and more than a factor of two times less SRAM than an improved solution proposed in [3].

To provide a concrete analysis of our proposed schemes, we use the specification of an actual commercial high-bandwidth memory part, namely the XDR memory from Rambus [11, 12]. For an OC-768 link at 40 Gb/s, a new minimum size (40 bytes) packet can arrive every 8 ns. To support a counter update on every packet arrival, about 4 ns is available for a memory read or a memory write. SRAMs with 4 ns access latencies are readily available, and we will assume this 4 ns SRAM access time. The XDR memory has a worst-case access latency of 38 ns for a read or a write operation. Therefore, the DRAM-to-SRAM (worst-case) latency ratio is $b = \lceil 38/4 \rceil = 10$. For the hybrid SRAM/DRAM architecture proposed in [5], the number of SRAM bits required for each counter is $\ell = \lceil \log b \rceil = \lceil \log 10 \rceil = 4$ bits. In addition, it needs a very small amount of SRAM to maintain a "flush request queue", on the order of about $K = 500$ entries to ensure negligible overflow probabilities.

For 16 million counters, a naive implementation would require 128 MB of SRAM, which is clearly far too expensive. For the scheme by Zhao et al., it just requires 1.5 KB of control SRAM to implement a flush request queue with $K = 500$ entries. The size of each entry is $\lceil \log 16 \text{ million} \rceil = 24$ bits to encode the counter index. However, even though the

	Naive	Zhao et al. [5]	Replicated (§2.1)	Randomized (§2.2)
Counter DRAM	None	128 MB DRAM	2 GB DRAM	128 MB DRAM
Counter SRAM	128 MB SRAM	8 MB SRAM	None	None
Control SRAM	None	1.5 KB SRAM	None	8 KB SRAM

Table 1: Comparison of different schemes for a reference configuration with 16 million 64-bit counters.

scheme requires just 4 bits per counter to store the partial increments, 8 MB of counter SRAM is required for 16 million counters. This is a substantial amount and difficult to implement on-chip. Moreover, this counter SRAM requirement grows linearly with the number of counters, making it difficult to support faster links or longer measurement periods where more counters would be needed.

For the replicated counter scheme and the randomized counter scheme (described in Sections 2.1 and 2.2), we need b memory banks to match the performance requirement of wirespeed updates. This could be realized by using b separate memory channels and performing memory interleaving across these channels. However, this is unnecessarily expensive as modern DRAM architectures already provide a plentiful number of internal memory banks. In particular, using the XDR memory as a concrete example, each XDR memory chip contains 16 internal memory banks (as depicted in Figure 4), and a new read or write transaction could be initiated every 4 ns if it is initiated to a different memory bank. To fully utilize all the memory locations available inside an XDR memory chip, we will interleave across 16 memory banks rather than $b = 10$. In particular, for the replicated counter scheme, the counters are only stored in DRAM. However, the replicated counter scheme requires replicating each counter 16 times, requiring 2 GB of DRAM. Although this may appear to be a substantial amount of DRAM, we note that DRAM is inexpensive, so a higher DRAM requirement does not necessarily lead to a significant cost increase. In particular, we note that 2 GB of DRAM is commercially available today in an inexpensive single memory module form factor (about $20 as of this writing), and only one memory channel interface is needed.

On the other hand, the randomized counter scheme avoids the need to replicate counters, and thus requires the same amount of DRAM as the hybrid SRAM/DRAM schemes. For each memory bank, the randomized counter scheme needs to maintain a small update request queue. As shown in Figure 3 and discussed in Section 2.2, it is sufficient to have about 80 entries in each queue to ensure a queue overflow probability of 10^{-9}, 1280 entries in total for the 16 memory banks, for up to 90% traffic load. In addition, as discussed in Section 2.2, the randomized counter scheme maintains a small hash table to keep track of pending update requests in the update request queues. The hash table needs to support as many entries as the update request queues can store (i.e. 1280 entries). Together, the update request queues and the hash table requires just approximately 8 KB of SRAM, negligible compared to the 8 MB required by the scheme by Zhao et al., but requires the same 128 MB of DRAM. Although our comparisons here are for an integer number representation, we emphasize that our general schemes are applicable to other number representations, such as floating point numbers.

4. CONCLUDING REMARKS

We conclude by suggesting that the main ideas proposed in this paper can be generalized for other wirespeed network measurement functions where SRAM solutions were previously considered necessary. We are currently pursuing such generalizations.

5. REFERENCES

[1] G. Varghese, C. Estan, "The measurement manifesto," HotNets-II, 2003.

[2] D. Shah et al., "Maintaining statistics counters in router line cards," IEEE MICRO, 2002.

[3] S. Ramabhadran, G. Varghese, "Efficient implementation of a statistics counter architecture," ACM SIGMETRICS, 2003.

[4] M. Roeder, B. Lin. "Maintaining exact statistics counters with a multi-level counter memory," IEEE GLOBECOM, 2004.

[5] Q. Zhao, J. Xu, Z. Liu, "Design of a novel statistics counter architecture with optimal space time efficiency," ACM SIGMETRICS, 2006.

[6] M. Gschwin et al., "Synergistic processing in Cell's multicore architecture," IEEE MICRO, 2006.

[7] Intel IXP 2855 network processor product brief. Intel Corporation., Copyright 2005.

[8] S. I. Hong et al., "Access order and effective bandwidth for streams on a direct rambus memory," International Symposium on High-Performance Computer Architecture, 1999.

[9] W. Lin, S. Reinhardt, D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," International Symposium on High-Performance Computer Architecture, 2001.

[10] F. A. Ware, C. Hampel, "Improving power and data efficiency with threaded memory modules," International Conference on Computer Design, 2006.

[11] F. A. Ware, C. Hampel, "Micro-threaded row and column operations in a DRAM core," Rambus White Paper, 2005.

[12] XDR datasheet. Rambus, Inc., Copyright 2002-2003.

[13] XDR-2 datasheet. Rambus, Inc., Copyright 2004-2005.

[14] D. Patterson, J. Hennessy, Computer Architecture: A Quantitative Approach, 2nd. ed., San Francisco: Morgan Kaufmann Publishers, 1996.

[15] G. Shrimali, N. McKeown, "Building packet buffers using interleaved memories," IEEE HPSR, 2005.

[16] P. Indyk, "Stable distributions, pseudorandom generators, embeddings, and data stream computation," IEEE-FOCS, 2000.

[17] H. Zhao et al., "A data streaming algorithm for estimating entropies of OD flows," ACM Internet Measurement Conference, 2007.

[18] B. Vocking, "How asymmetry helps load balancing," IEEE-FOCS, 1999.

[19] A. Broder, M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," IEEE INFOCOM, 2001.

[20] F. Bonomi, M Mitzenmacher, R. Panigrahy, S. Singh, G. Varghese, "Beyond Bloom filters: From approximate membership checks to approximate state machines," ACM SIGCOMM, 2006.