Virtually Pipelined Network Memory

Banit Agrawal Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara
{banit,sherwood}@cs.ucsb.edu

Abstract

We introduce virtually-pipelined memory, an architectural technique that efficiently supports high-bandwidth, uniform latency memory accesses, and high-confidence throughput even under adversarial conditions. We apply this technique to the network processing domain where memory hierarchy design is an increasingly challenging problem as network bandwidth increases. Virtual pipelining provides a simple to analyze programing model of a deep pipeline (deterministic latencies) with a completely different physical implementation (a memory system with banks and probabilistic mapping). This allows designers to effectively decouple the analysis of their algorithms and data structures from the analysis of the memory buses and banks. Unlike specialized hardware customized for a specific data-plane algorithm, our system makes no assumption about the memory access patterns. In the domain of network processors this will be of growing importance as the size of the routing tables, the complexity of the packet classification rules, and the amount of packet buffering required, all continue to grow at a staggering rate. We present a mathematical argument for our system's ability to provably provide bandwidth with high confidence and demonstrate its functionality and area overhead through a synthesizable design. We further show that, even though our scheme is general purpose to support new applications such as packet reassembly, it outperforms the state of the art in specialized packet buffering architectures.

1 Introduction

While consumers reap the benefits of ever increasing network functionality, the technology underlying these advances require armies of engineers and years of design and validation time. The demands placed on a high throughput network device are significantly different than those encountered in the desktop domain. Network components need to reliably service traffic even under the worst conditions [22, 17, 12, 27, 14], yet the underlying memory components on which they are built are often optimized for common case performance. The problem is that network processing, at the highest throughputs, requires massive amounts of memory bandwidth with worst-case throughput guarantees. A new packet may arrive every three nanoseconds for OC-3072, and each packet needs to be buffered, classified into a service class, looked up in the forwarding table, queued for switching, rate controlled, and potentially even scanned for content. Each of these steps may require multiple dependent accesses to large irregular data structures such as trees, sparse bit-vectors, or directed graphs, usually from the same memory hierarchy. To make things worse the size of the data structures grow significantly with the line rate (40 gbps to 160 gbps). Routing tables have grown from 100K to 360K prefixes and classification rules have grown from 2000 to 5000 rules. Network devices will become increasingly reliant on high density commodity DRAM to remain competitive in both pricing and performance.

In this paper, we present virtually pipelined network memory (VPNM), an idea that shields algorithm and processor designers from the complexity inherent to commodity memory DRAM devices which are optimized for common case performance. The pipeline provides a programming model and timing abstraction which greatly eases analysis. A novel memory controller upholds that abstraction and handles all the complexity of memory system, including bank conflicts, bus scheduling, and worst case caching. This frees the programmer from having to worry about any of these issues, and the memory can be treated as a flat deeply pipelined memory with fully deterministic latency no matter what the memory access pattern is. Building a memory controller that can create such an illusion requires that we solve several major problems:

- Multiple Conflicting Requests: Two memory requests that access the same bank in memory will be in conflict, and we will need to stall at least one request. To hide these conflicts, our memory controller uses per-bank queues along with a randomized mapping to ensure that independent memory accesses have a statistically bounded number of bank conflicts. (Section 3.2)
- Reordering of Requests: To resolve bank conflicts requests need be reordered, but a virtual pipeline requires deterministic (in-order) behavior. Since the latencies of all memory accesses are normalized through novel specialized queues, distributed reordering of accesses can be done which create the appearance of fully pipelined memory. (Sections 3.3 and 4.1)

- Redundant Requests: As repeated requests for the same data cannot be randomized to different banks, normalizing the latency for these requests could create the need for gigantic queues. Instead we have built a novel form of merging queues that combines redundant requests and acts as a cache, but provides data playback at the right times to maintain the illusion of a pipeline. (Section 3.4)
- Worst Case Analysis: In addition to the architectural challenges listed above, reasoning about the worst case behavior of our system requires careful mathematical analysis. We show that it is provably hard for even a perfect adversary to create stalls in our virtual pipeline with greater effectiveness than random chance. (Sections 5.1 and 5.2)

To quantify the effectiveness of our system, we have performed a rigorous mathematical analysis, executed detailed simulation, created a synthesizable version, and estimated hardware overheads. In order to show that our approach will actually provide both high performance and ease of programming, we have implemented a packet buffering scheme as a demonstration of performance, and a packet reassembler as a demonstration of usefulness, both using our memory system. We show that despite the generality of our approach (it does not assume the head-read tail-write property) it compares favorably with several special-purpose packet buffering architectures in both performance and area (Section 5.4).

2 Related Work

Dealing with timing variance in the memory system has certainly been addressed in several different ways in the past. Broadly the related work can be broken up into two groups: scheduling and bank conflict reduction techniques that work in the common case, and special purpose techniques that aim to put bounds on worst case performance on particular classes of access patterns.

Common-case DRAM Optimizations – Memory bank conflicts not only effect the total throughput available from a memory system, they can also significantly increase the latency of any given access. In the traditional processing domain, memory latency can often be the most critical factor in terms of determining performance and several researchers have proposed hiding this latency with bank-aware memory layout, prefetching, and other architectural techniques [11, 21]. While latency is critical, traditional machines are far more tolerant of non-uniform latencies and reordering because many other mechanisms are in place to ensure the proper execution order is preserved. For example, in the streaming memory controller (SMC) architecture, memory conflicts are reduced by servicing a set of requests from one stream before switching to a different stream [16]. A second example is the memory scheduling algorithm where memory bandwidth is maximized by reordering various command requests [25]. In the vector processing domain [10], a long stream requires conflict free access for larger number of strides. Rau et al. [24] use randomization to spread the accesses around the memory system, and through the use of Galois fields show that it is possible to have a pseudo-random function that will work as well on any possible stride. While address mapping such as skewing or linear transformations can be used for constant stride, out of order accesses can efficiently handle a larger number of strides [20]. Corbal et al. [5] present a command vector memory system (CVMS) where a full vector request is sent to the memory system instead of individual addresses to provide higher performance.

While these optimizations are incredibly useful, industrial developers working on devices for the core will not adopt them due to the fact that certain deterministic traffic patterns could cause performance to sink drastically. Dropping a single packet can have a enormous impact on network throughput (as this causes a cascade of events up the network stack) and the customer needs to be confident that the device will uphold its line rate. In this domain it would be ideal if there was a *general purpose* way to control banked access such that conflicts *never* occur. Sadly this is not possible in the general case [4]. However, if the memory access patterns can be carefully constrained, algorithms can be developed which solve certain special cases.

Removing Bank Conflicts in Special Cases – One of the most important special cases that has been studied is packet buffering. Packet buffering is one of the most memory intensive operations that networks need to deal with [17, 12], and high speed DRAM is the only way to provide both the density and performance required by modern routers. However, in recent years researchers have shown special methods for mapping these queues onto banks of memory in such a way that conflicts are either unlikely [14, 2, 22, 29] or impossible [12, 17]. These techniques rely on the ability to carefully monitor the number of places in memory where a read or write may occur without a bank conflict, and to *schedule* memory requests around these conflicts in various ways. For example, in [12], a specialized structure similar to a reorder buffer is used to schedule accesses to the heads and tails of the different packet buffer queues. The technique combines clever algorithms with careful microarchitectural design to ensure worst case bounds on performance are always met in the case of packet buffering. Randomization has also been considered in the packet buffering space as well. For example, Kumar et. al. present a technique for buffering large packets by randomly distributing parts of the packet over many different memory channels [19]. However, this technique can handle neither small packets nor bank conflicts. Another important special case is dataplane algorithms which may also suffer from memory bank conflict concerns. Whether these banks of memory are on or off chip, supporting multiple non-conflicting banked memory accesses requires a significant amount of analysis and planning. For example, a conflict reduced tree-lookup engine was proposed by Baboescu et. al. [1]. A tree is broken into many sub-trees, each of which are then mapped to parts of a rotating pipeline. They prove that optimally allocating the sub-trees is NP-complete and present a heuristic mapping instead. Similarly in [15], a conflict-free hashing technique is proposed for longest prefix match (LPM) where conflicts are taken care of at an algorithmic level. While the above methods are very powerful, they all require careful layout (by the programmer or hardware designer) of each data structure into the particular bank structure of the system and allow neither changes to the data structures nor sharing of the memory hierarchy.

While our approach may have one stall on average once every 10^{13} cycles (on the order of hours), the benefit is that no time has to be spent considering the effect of banking on already complex data structures. As we will describe in Section 3, a virtually pipelined network memory system uses cryptographically strong randomization, several new types of queues, and careful probabilistic analysis to ensure that deterministic latency is efficiently provided with provably strong confidence.

3 Virtually Pipelined Memory

Vendors need to have confidence that their devices will operate at the advertised line rates regardless of operating conditions, including when under denial of service attack by adversaries or in the face of unfortunate traffic patterns. For this reason, most high-end network ASICs do not use any DRAM for data-plane processing because the banks make worst-case analysis difficult or impossible. The major exception to this rule is packet buffering; even today it requires an amount of memory that can only be satisfied through DRAM and a great deal of effort has been expended to map packet buffering algorithms into banks with worst-case bounds. Later in Section 5.4.1 we compare our implementation against several special purpose architectures for packet buffering.

3.1 DRAM Banks

Modern DRAM designs try to expose the internal bank structure so accesses can be interleaved and the effective bandwidth can be increased [6, 13]. The various types of DRAM differ primarily in their interfaces at the chip and bus level [8, 7, 23], but the idea of banking is always there. Experimental evidence [23] indicates that on average PC133 SDRAM works at 60% efficiency and DDR266 SDRAM works at 37% efficiency, where 80 to 85% of the lost efficiency is due to the bank conflicts. To help address this problem RDRAMs expose many more banks [23]. For example, in Samsung Rambus MR18R162GDF0-CM8 each RDRAM device can contain up to 32 banks and each RIMM module can contain up to 16 such devices, so the module can have up to $32 * 16 = 512$ independent banks [26].

A bank conflict occurs when two accesses require different rows in the same bank. Only one can be serviced at a time, and hence one will be delayed by L time steps. L is the ratio of bank access time to data transfer time – in other words it is the number of accesses that will have to be skipped before a bank conflict can be resolved. Throughout this paper we conservatively assume that there is one transfer per cycle and we select the value of L=20 [26, 30]. If L is smaller then our approach will be even more efficient.

3.2 Building a Provably Strong Approach

To prove that our approach will deliver throughput with high confidence, we consider the best possible adversary and show that adversary will never be able to construct a sequence of accesses that perform poorly. First, we map the data to banks in permutations that are provably as good as random. Universal hashes [3], an idea that has been extended by the cryptography community, provides a way to ensure that an adversary cannot figure out the hash function without direct observation of conflicts. The virtual pipeline then prevents an adversary from even seeing those conflicts through specialized queues. This latency normalization not only allows us to formally reason about our system, it also shields the processor from the problem of reordering, and greatly simplifies data structure analysis. While the latency of any given memory access will be increased significantly over the best possible case, the memory bandwidth delivered by the entire scheme is almost equal to the case where there are no bank conflicts. While this may make little sense in the latency-intolerant world of desktop computing, in the network space this can be a huge benefit.

3.3 Distributed Scheduling around Conflicts

While Universal Hashing provides the means to prevent our theoretical adversary from constructing sets of conflicting accesses with greater than random probability, even in a random assignment of data to banks a relatively large number of bank conflicts can occur due to the Birthday Paradox [18]. In fact if there was no queuing used, then it would take only $O(\sqrt{B})$ accesses before the first stall would occur if there are B banks. Clearly we will need to schedule around these conflicts in order to keep the virtual pipeline timing abstraction. In our implementation, a controller for each bank is used, and each bank handles requests in-order, but each bank is handled independently so the requests to different banks may be handled out-of-order. Each bank controller is then in charge of ensuring that for every access at time t, it returns the result at time $t + D$ for some *fixed* value of D. As long as this holds, there is no need for the programmer to worry about the fact that there is even such a thing as banks.

One major benefit of our design is that the memory scheduling and reordering can be done in a fully parallel and independent manner. If each memory bank has its own controller, there is exactly one request per cycle, and each controller ensures that the result of a request is returned exactly D cycles later, then there is no need to coordinate between the controllers. When it comes to return the result at time $t + D$, a bank controller will know that it is always safe to

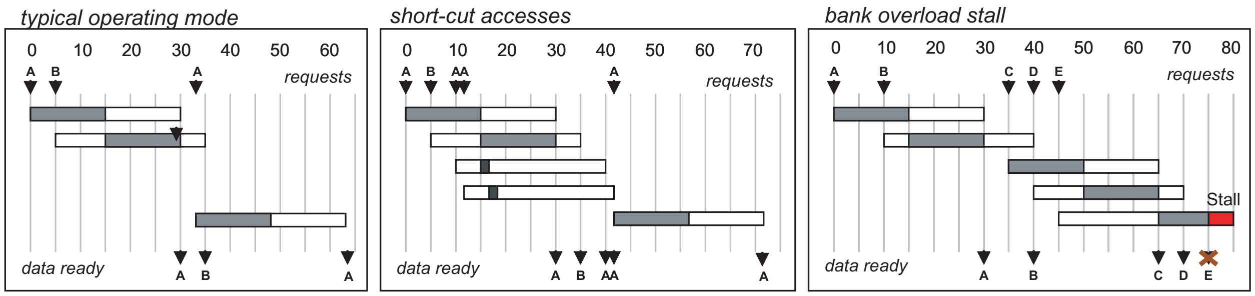


Figure 1: An example of how each bank controller will normalize the latency of memory accesses to a fixed delay ($D = 30$). In all three graphs the x-axis is cycles and each memory access is shown as a row. The light white boxes are the times during which the request is "in the pipeline", while the dark grey box is the actual time that it takes to access the bank (L=15). In this way a certain number of bank conflicts can be hidden as long as there are not too many requests in a short amount of time. The graph on the left shows normal operation, while the middle graph shows what happens when there are redundant requests for a single bank which therefore don't require bank access. The graph on the right shows what happens when there are too many requests to one bank (A-E) in a short period of time thus causing a stall. Later in the analysis section we will also refer to Q which is the maximum number of overlapping requests that can be handled, in this case Q is 30/15 = 2.

send the data to the interface because by definition it was the only one to get a request at time t.

3.4 What Can Go Wrong

If there are B banks in the system then any one bank will only have a 1 in B chance of getting a new request on any given cycle. [1] The biggest thing that can go wrong is that we get so many requests to one bank that one of the queues fill up and we need to stall. Reducing the probability of this happening even for the worst cases access pattern requires careful architectural design and mathematical analysis. In fact there are two ways in which a bank can end up getting more than $1/B$ of the requests.

The first way is that it could be unlucky, and just due to randomness more than $1/B$ of the requests go to a single bank (because we map them randomly). By keeping access queues, we can ensure that the latency is normalized to D to handle simultaneously occurring bank conflicts. How large that number is, and how long it will take to happen in practice are discussed extensively in Section 5. In practice we find that normalizing D to 1000 nanoseconds is more than enough, and is several orders of magnitude less than a typical router latency of 2 milliseconds. While this a typical example, the actual value of D is dependent on L and the size of bank access queue as described in Section 4. While there is a constant added delay to D due to universal hashing, the hash function can be fully pipelined and then it will not be any big impact to the normalized delay D.

The second way we could get many accesses to one bank is that there could be repeated requests for the same memory line. The invariant that a request at time t is satisfied at time $t + D$ must hold for this case as well, and in Section 5.2 we describe how to design a special merging queue to address this second problem. The idea behind our merging queue is that redundant memory accesses are combined into a single access and a *single queue entry* internally. If an access A comes at t_1 and a redundant access A comes at t_2, a reply still needs to be made at $t_1 + D$ and at $t_2 + D$ even though internally only *one queue entry* for A is maintained. In addition to handling the repeating pattern "A,A,A,A,..." we need to handle "A,B,A,B,..." with only two queue entries. In fact if we need to handle Q bank conflicts without a stall, then we will need to handle up to Q different sets of redundant accesses. In Figure 1 we show how the virtually pipelined network memory works altogether for different type of accesses.

[1]This is not to say that each bank will be responsible for exactly $1/B$ of the requests as in round robin. Round robin will not work here because requests must be satisfied by one bank that contains their memory. Although we get to pick the mapping between memory lines and banks, the memory access pattern will determine which actual memory lines are requested.

4 Implementing the Interface

At a high level, the memory controller implementing our virtual pipeline interface is essentially a collection of decoupled memory bank controllers. Each bank controller handles one memory bank, or one group of banks that act together as a single logical bank. Figure 2 shows one possible implementation where a memory controller contains all of the bank controllers on-chip, and they all share one bus. This would require no modification to the bus or DRAM architecture.

The performance of our controller is limited by the single bus to the memory banks. If we have to service one memory request per cycle, then we need to have one outgoing access on each cycle to the memory bus and the bus will become a bottleneck. Hence, to keep up with the incoming address per cycle and to prevent any accumulation of requests in the bank controller due to mismatched throughputs, we need to support slightly more memory bus transaction/second than

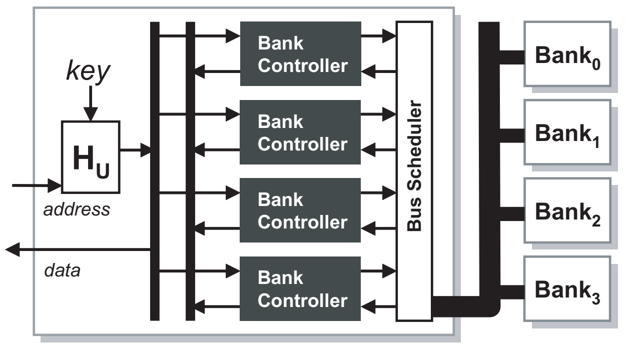


Figure 2: Memory controller block diagram. After an access is randomized from universal hash engine (H_U), it is directed to the corresponding bank controller for further processing.

allowed on the interface bus. We call the ratio of the request rate on the interface bus and request rate of memory bus as bus scaling ratio (R). The value of 'R' is chosen slightly higher than 1 to provide slightly higher access rate on the memory side compared to the interface side. This mismatch ensures that idle slots in the schedule do not accumulate slowly over time. A round-robin scheduler arbitrates the bus by granting access to each bus controller every B cycles, where B is the number of banks. It might happen that some of the round-robin slots are not used when there is no access for the particular bank or the memory bank is busy, although with further analysis or a split-bus architectures this inefficiency can be eliminated.

Once the determination of which bank a particular memory request needs to access, the request is handed off to the appropriate bank controller which takes care of everything for that bank. Almost all of the latencies in the system are fully deterministic so there is no need to employ a complicated scheduling mechanism. The only time the latencies are not fully deterministic is when there are a sufficient number of memory accesses to a single bank in a sufficiently small amount of time that cause the latency normalizing technique to stall. However, as we will show in Section 5, the parameters of the architecture can be chosen such that this happens extremely infrequently (on the order of once every trillion requests in the worst case).

Since stalls happen so infrequently and because the stall time is also very low (in the worst case a full memory access latency of about 100 nanoseconds), stalls can be handled in one of two ways. The first way is to simply stall the controller, where the slowdown would not even be a fraction of a percent, while the other alternative is to simply drop the packet (which would be noise compared to packet-loss due to many other factors). In either case, an attacker cannot leverage information about a stall unless they can a) observe the exact instant of the stall, b) remember the exact sequence of accesses that caused the stall and c) are able to replay the stall causing events with minor changes (to look for more multiple collisions). With randomization due to universal mapping, and a very high value of Mean-time-to-stall (around 10^{13} as described in Sections 5.1 and 5.2), the ability to do this will be practically impossible. If such attacks are believed to be a threat, a further (and sightly more costly) option is to change the universal mapping function and reordering the data on the occurrence of multiple stalls (an expensive operation, but certainly possible with frequency on the order of once a day).

4.1 Bank Controller Architecture

Solving the challenges described in the introduction requires a carefully designed bank controller. In particular, it must be able to queue the bank requests, store the data to a constant delay, and handle multiple redundant requests.

The architecture block diagram of our bank controller is shown in Figure 3. From the figure we can see that the bank controller consists of five main components described with the text next to each block. The primary tasks of these components include queuing input requests, initiating a memory request, sending data to the interface at a pre-specified time slot to ensure the deterministic latency and each of these components is designed to address one or more of challenges mentioned earlier.

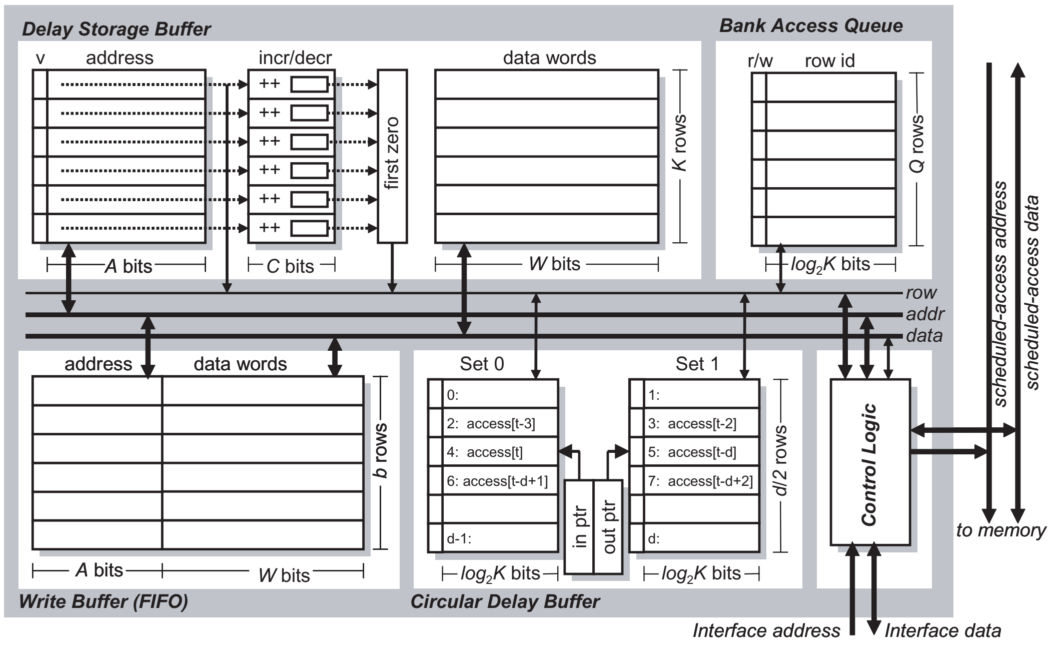
4.2 Controller Operations

At a high level each memory request goes through 4 states: pending, accessing, waiting, and completed. New requests start out as *pending*, and when the proper request is actually sent out to the DRAM, the request is *accessing*. When the result returns from DRAM the request is *waiting* (until D total cycles have elapsed), and finally the request is *completed* and results are returned to the rest of the system.

When a new read request comes in, all the valid addresses of the address CAM in the delay storage buffer are searched. On a match (a redundant access), the matched row counter is incremented and the id of matched row is written to the circular delay buffer (along with its valid bit). On a mismatch, a free row is determined using the *first zero* circuit and it is updated with the new address and the counter is initialized to one. The id of the corresponding free row is written to the circular delay buffer. During this mismatch case, we also add the row id combined with '0' bit (read) to the bank access queue (where it waits to become *accessing*). On an incoming write request, the write address and data is added to write buffer FIFO. A '1' bit (write) is written to the bank access queue. The row id is unused in this case as we access the write buffer in FIFO order. It is also searched in the address CAM and on a match, the address valid flag is unset. But this row cannot be used for a new read request and will service previous read requests until the counter reaches zero since the data until the current cycle is still valid. When the counter reaches zero, then there are no pending requests for that row and the row can serve as free row for the new requests.

Delay Storage Buffer -- The delay storage buffer stores the address of each pending and accessing request, and stores the address and data of waiting requests. Each non-redundant request will have an entry allocated for it in the delay buffer for a total of *D* cycles. To account for repeated requests to the same address, a counter is associated with each address and data. The buffer contains *K* rows, where each row contains an address of *A* bits, a one-bit address valid flag, a counter of *C* bits, and data words of *W* bits. The data words are buffered in these rows whenever the read access to memory bank completes, and one row is needed for each *unique* access

Bank Access Queue -- The bank access queue keeps track of all pending read and write requests that require access to the memory bank. It can store up to *Q* interleaved read or write requests in FIFO order. To avoid keeping *Q* copies of the address and data, each entry is just the index of a target row in the delay storage buffer.



Write Buffer -- The write buffer is organized as FIFO structure, which stores the address and data of all incoming write requests. Unlike read request, we need not need to wait for the write requests to complete. We only need to buffer the write request until it gets scheduled to access the memory bank.

Circular Delay Buffer -- The circular delay buffer stores the request identifier of every incoming read request and triggers the final result to be written the output interface after a deterministic latency (*D*). This circular delay buffer is the only component which is accessed every cycle irrespective of the input requests. Note that if we just stored the full data here, instead of a pointer to the delay storage buffer, then we would need to have a huge number of bytes to buffer all the data (2 to 3 orders of magnitude more).

Control Logic -- The control logic handles the necessary communication between components (while the interconnect inside the bank controller is drawn as a bus for simplicity, in fact it is a collection of direct point-to-point connections).

Figure 3: Architecture block diagram of the VPNM bank request controller.

During each cycle, the controller scans the bank access queue and reads from the circular delay buffer. If the bank controller is granted to schedule a memory bank request, then the first request in the bank access queue is dequeued for access. In the case of a read access, the address is read from the delay storage buffer and put on the memory bank address bus. While in the case of write access, the address and the data words are dequeued from the write buffer and the write command is issued to the memory bank. In the case of no incoming read requests in the current cycle, the control logic invalidates the current entry of circular delay buffer. On every cycle, it also reads the D-cycle delayed request-id from the circular delay buffer. If it is valid, then the data is read from the data words present in delay storage buffer and the data is put on the interface bus. Since we do one read and one write operation to circular delay buffer every cycle, it is designed as a 2-set (single-ported) architecture with *in* and *out* pointers to save the power consumption. While there are some additional aspects of the bank controller design, we cannot fully describe all of the low level implementation aspects in this paper due to space limitations.

4.3 Stall Conditions

The aim of the VPNM bus controller architecture is to provide a provably small stall rate in the system through randomization, but the actual stall rate is a function of the parameters of the system. There are three different cases which require a stall to resolve, each of which is influenced by a different subset of the parameters.

- Delay storage buffer stall - The number of rows (K) in delay storage buffer are limited and a row has to be reserved for D cycles for one data output. Hence, if there are no free rows and it cannot reserve a row for a new read request, then it results in a delay storage buffer stall. This stall is mainly dependent on the following parameters 1) Number of rows (K) in delay storage buffer 2) Deterministic delay (D) 3) Number of banks (B). The deterministic delay is determined using the access latency (L) and the bank request queue size (Q), and this stall analysis is presented in Section 5.1.

- Bank request queue stall - When a new non-repeating read/write request comes to a bank and the size of the bank access queue is already Q, then the new request cannot be accommodated in the queue. This condition results in bank request queue stall. There are three main parameters which control this stall - 1) average input rate, which is equal to $1/B$, where B is the number of banks. 2) Queue size (Q) 3) the output rate, which is decided by the ratio (R) of frequency on the memory side and frequency on the interface side. In Section 5.2 we discuss exactly how to perform the confidence analysis for this stall.
- Write buffer stall - Write buffer (WB) stall happens when a write request cannot be added in the write buffer. As we keep the write buffer equal to half of bank request queue size, the chances of stall rate in write buffer is much less than the stall rate in bank request queue. The analysis of the write buffer stall is similar to the analysis of bank request queue and does not dominate the overall stall, so we will only discuss about the bank request queue and delay storage buffer stall in our mathematical analysis in Section 5.

5 Analysis of Design

The Virtually Pipelined Network Memory can stall in the three ways described in Section 4.3. In any of these cases, the buffer will have to stall, and it will not be able to take a new request that cycle. Because we randomize the mapping we can formally analyze the probability of this happening and because we use the cryptographic idea of universal hashing we know that there is no deterministic way for an adversary to generate conflicts with greater than random probability unless they can directly see them. We ensure that the conflicts are not visible through latency normalization (queuing both before and after a request) unless many many different combinations are tried. We quantify this number, and the confidence we place in our throughput, as the Mean Time to Stall (MTS). It is important to maximize the MTS, a job we can perform through optimization of the parameters described in Section 4 and summarized in Table 1.

To evaluate the effect of these parameters on MTS, we performed three types of analysis: Simulation (for functionality), Mathematical (for MTS), and Design (to quantify the hardware overhead). To get an understanding of the execution behavior of our design, and to verify our mathematical models, we have built functional models in both C and Verilog and we have synthesized our design using synopsys design compiler. However, in this paper we concentrate on the mathematical analysis of delay storage buffer stall and bank access queue stall, the calculation of the mean time to stall (MTS) for both these cases, and a high level analysis of the hardware required.

Table 1: Parameters for the analysis of our controller

Q	number of entries in the bank access queue
K	number of rows in the delay storage buffer
B	number of banks in the system
L	latency of accessing one bank
D	delay to which all memory accesses are normalized
R	frequency scaling ratio

5.1 Delay Storage Buffer Stall

A delay buffer entry is needed to store the data associated with an access for the duration of D. A buffer will overflow if there are more requests assigned to it over a period of D cycles than there are places to store those requests. To calculate the Mean Time to Stall (MTS) we need to determine the expected amount of time we will have to wait until one of the B banks gets K or more requests over D cycles. The mapping of requests to banks is random so we can treat the bank assignments as a random sequence of integers $(a_1, a_2, \ldots, a_T)$, where each a_i is drawn from the uniform distribution on $\{1, 2, \ldots, B\}$.

If we want to know the probability of stall after T cycles, then for any $i \leq T - D + 1$, we can detect a stall happening when at least $K - 1$ of the symbols $a_{i+1}, \ldots, a_{i+D-1}$ are equal to a_i; the probability of this is $\binom{D-1}{K-1} \cdot (\frac{1}{B})^{K-1}$, so the probability of not having a delay buffer overfill over the given interval is $1 - \binom{D-1}{K-1} \cdot (\frac{1}{B})^{K-1}$. Since we are only concerned with the probability that *at least one* stall occurs and not how many, we can conservatively estimate the probability of no stall occurring over the entire sequence as $(1 - \binom{D-1}{K-1} \cdot (\frac{1}{B})^{K-1})^{T-D+1}$. This method assumes that stalls are independent, when in fact they are positively correlated, and it actually counts some stalls multiple times. Solving for a probability of 50% that a stall can happen, the Mean Time to Stall is:

$$MTS = \frac{\log(\frac{1}{2})}{\log(1 - (\binom{D-1}{K-1} \cdot \frac{1}{B})^{K-1}))} + D$$

Figure 4 shows the impact of number of entries in storage delay buffer (K) on this stall. We take the value of $R = 1.3$ in this case. Since B and Q are interrelated for this analysis, we select the optimal combination of B and Q. We set the higher limit of the MTS value to 10^{16} in all of our analysis [2]. Figure 4 shows that for $B = 32$, the curve rises sharply with K and we can get a MTS of 10^{12} for $K = 32$. The curve for $B = 64$ follows very closely to the curve for $B = 32$. Hence, having $B = 32$ is optimal in our case. For lower number of banks ($B < 32$), we need much higher values of K to even reach a MTS value of 10^8.

5.2 Bank Access Queue Stall

Performing an analysis similar to that presented in Section 5.1 will not work for the bank access queue because

[2] An MTS of 10^{12} is around one stall every 15 minutes with a very aggressive bus transaction speed of 1 GHz

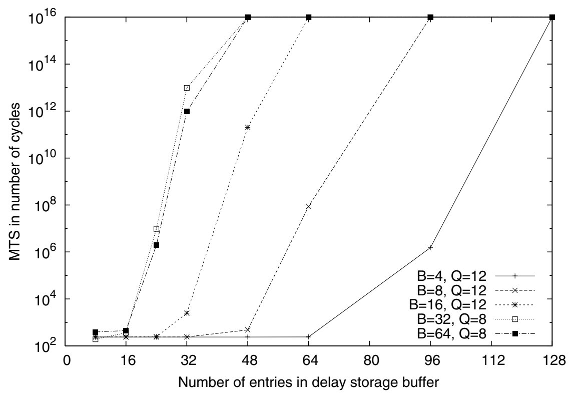
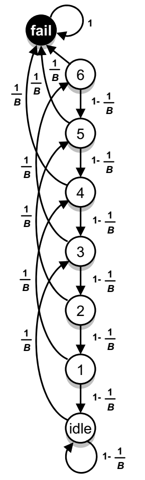


Figure 4: MTS variation with number of entries in delay storage buffer (K) for memory controller with $R = 1.3$



$$I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M = \begin{bmatrix} 1-\frac{1}{B} & 0 & 0 & \frac{1}{B} & 0 & 0 & 0 & 0 \\ 1-\frac{1}{B} & 0 & 0 & 0 & \frac{1}{B} & 0 & 0 & 0 \\ 0 & 1-\frac{1}{B} & 0 & 0 & 0 & \frac{1}{B} & 0 & 0 \\ 0 & 0 & 1-\frac{1}{B} & 0 & 0 & 0 & \frac{1}{B} & 0 \\ 0 & 0 & 0 & 1-\frac{1}{B} & 0 & 0 & 0 & \frac{1}{B} \\ 0 & 0 & 0 & 0 & 1-\frac{1}{B} & 0 & 0 & \frac{1}{B} \\ 0 & 0 & 0 & 0 & 0 & 1-\frac{1}{B} & 0 & \frac{1}{B} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Markov Model that captures the fail probability of a Bank Access Queue with $L = 3$ and $Q = 2$. With probability $1/B$ a new request will arrive at any given bank causing their to be L more cycles worth of work.

there is no fixed window of time over which we can analyze the system combinatorially. There is state involved because the queue may cause a stall or not depending on the amount of work left to be done by the memory bank. To analyze the stall rate of the bank access queue we determined that the queue essentially acts as a probabilistic state machine.

To do the analysis, we need to combine this abstract state machine with the probabilities that any transition will occur. Each cycle a new request will come to a given bank controller with probability $\frac{1}{B}$ and the probability that there will be no new request is $1 - \frac{1}{B}$. The probabilistic state machine that we are left with, is a Markov Model. In Figure 5 we can see the probabilistic model stored both as a directed graph, and in adjacency matrix form labeled M.

The adjacency matrix form has a very nice property: given an initial starting state at cycle zero, stored as the vector I, to calculate the probability of landing in any state at cycle one, we simply multiply I by M. In the example given, there is probability P of being in state 2, 1-P of still being in the idle state. This process can then be repeated, and to get the distribution of states after t time steps we simply multiply I by M t times, which is of course IM^t. Note that the stall state is an absorbing state, so the probability of being in that state should tell us the probability of there ever being a bank overflow on any of the t cycles. To calculate that probability, we simply need to calculate M^t.

We use this analysis to figure out the impact of bank request queue size (Q) on MTS. The effect of normalized delay D can also be directly seen as D is directly proportional to Q. If we decrease/increase the value of D, then we have to decrease/increase the value of Q accordingly. For our memory controller with a value of $R = 1.3$, the MTS graph is shown in Figure 6. We find that for $B = 32$ and $B = 64$, the curve for MTS is almost the same. We can clearly see from the figure that a lower number of banks ($B < 32$) can only provide a maximum MTS value of 10^2 for even larger values of Q. Hence, an SDRAM with its small number of banks cannot achieve a reasonable MTS. However, for $B = 32$ and $B = 64$, we see an exponential increase in MTS with the increasing value of Q. We can get an MTS of 10^{14} for $Q = 64$ using 32 or 64 banks. If any application that does not demand a high value of MTS, but requires a lower value of normalized delay, then we can use the system with a lower value of Q and with 32/64 banks. We did not calculate the MTS values for $B >= 128$ because the large matrix size makes our analysis very difficult (the matrix requires more than 2 GB of main memory).

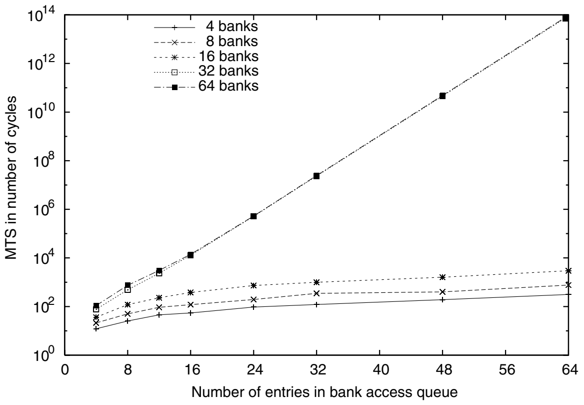


Figure 6: MTS variation with number of entries in bank access queue (Q) for our controller with $R = 1.3$

5.3 Hardware Estimation

The structures presented in Section 4 ensure that only probabilistically well formed modes of failure are possible and that exponential improvements in MTS can be achieved for linear increases in resources. While the analysis above allows us to formally analyze the stall rate of our system, without a careful estimate of the area and power overhead it is hard to understand the tradeoffs fully. To explore this de-

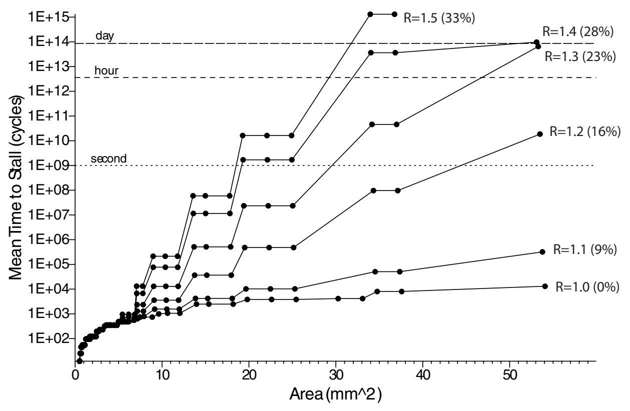


Figure 7: MTS with area overhead for our memory controller for different frequency ratios (R)

sign space, we developed a hardware overhead analysis tool for our bank controller architecture that takes these design parameters (B,L,K,Q,R,$tech$) as inputs and provides area and energy consumption for the set of all bank controllers. We use a validated version of the Cacti 3.0 tool [28] and our synthesizable Verilog model to design our overhead tool and use $0.13\mu m$ CMOS technology to evaluate the hardware overhead.

5.3.1 Optimal Parameters

Since area overhead is one of the most critical concerns as it directly affects the cost of the system, we take the total area overhead of *all* the bank controllers as our key design parameter to decide the value of MTS. As a point of reference, one bank controller (which then needs to be replicated per bank) with $L = 20$, $K = 24$, and $Q = 12$, occupies 0.15 mm^2. We run the hardware overhead tool for several thousand configurations with varying architectural parameters and consider the Pareto optimal design points in terms of area, MTS, and bandwidth utilization (R). We also set some baseline required values of MTS, which are 1 second (10^9), 1 hour (3.6×10^{12}), and 1 day (8.64×10^{13}) for an aggressive 1 GHz clock frequency. While this is not small, our example parameter set describes a design that targets a very aggressive bandwidth system and compares favorably with past special purpose designs (see Section 5.4)

The Pareto-optimal curve for our memory controller is shown in Figure 7. This figure shows an interesting tradeoff between the MTS and the utilization of effective bandwidth on the memory bus side. If we increase the value of R, then we get better values of MTS with effective lower utilization of memory bus. For $R = 1.3$, we need 23% extra memory bus bandwidth, but with a much better stall rate compared to $R = 1.2$ (16% extra bandwidth). We find that we can choose either $R = 1.3$ (one second MTS=10^9 for about 30 mm^2) or $R = 1.4$ (one hour MTS=3.6×10^{12} for about 30 mm^2) to get the best values of MTS without compromising much of the memory bus speed utilization.

Table 2: Optimal design parameters for best MTS and area overhead combination

Frequency Scaling ratio (R)	Area overhead in mm^2	MTS in cycles	Optimal design parameters	Energy in nJ
1.3	13.6	5.12e+05	B=32, Q=24, K=48	11.09
1.3	19.4	2.34e+07	B=32, Q=32, K=64	13.26
1.3	34.1	4.57e+10	B=32, Q=48, K=96	17.05
1.3	53.2	6.50e+13	B=32, Q=64, K=8	21.51
1.4	13.6	1.14e+07	B=32, Q=24, K=48	10.79
1.4	19.3	1.69e+09	B=32, Q=32, K=64	12.83
1.4	34.0	3.62e+13	B=32, Q=48, K=96	16.38
1.4	53.0	9.75e+13	B=32, Q=64, K=128	20.54

We calculate the optimal parameters from Figure 7 and we find the energy consumption for these optimal parameters. The optimal parameters along with all design constraints are shown in Table 2. The table shows that for R=1.3 and R=1.4, we need around 32 banks, 32 to 48 bank access queue entries, and 64 to 96 storage delay buffer entries with 10 to 20 nJ energy consumption.

5.4 Applications Mapping

To demonstrate the usefulness and generality of our approach, in this section we show how our Virtually Pipelined Network Memory can be easily used to implement two different high-speed memory intensive data-plane algorithms. By implementing Packet Buffering on top of VPNM we can directly compare against special purpose hardware designs in terms of performance. While our approach hides the complexity of banking from the programmer, it can match and even exceed the performance of past work that requires specialized bank-aware algorithms. To further show the usefulness of our system, we have also mapped a Packet Reassembler (used in content inspection) to our design, a memory bound problem for which there is no current bank-safe algorithm known.

5.4.1 Packet Buffering

Packets need to be temporarily buffered from the transmission line until the scheduler issues a request to forward the packet to the output port. According to current industry practice, the amount of buffering required is $2RT$ [17], where R is the line rate and T is the round trip time over the Internet. For 160 gbps line rate and a typical round trip time of 0.2 second [12], the buffer size will be 4 GB. The main challenge in packet buffering is to deal with constantly increasing line rate (10 gbps to 40 gbps and from 40 gbps to 160 gbps) and the number of interfaces (order of hundreds to order of thousands).

Using DRAM as intermediate memory for buffering does not provide full efficiency due to DRAM bank conflicts [22, 12]. In [22], an out-of-order technique has been proposed to reduce the bank conflict to provide packet buffering requirement for 10 gbps. Iyer et al. [17] have used a combination of SRAM and DRAM, where SRAMs are used for storing some head and tail packets for each queue. This combination allow them to buffer packets at 40 gbps using some clever memory management algorithms (for example: earliest critical queue first (ECQF)). But they do not consider the effect of bank conflicts. Garcia et al. [12] take their approach further by providing a DRAM subsystem (CFDS) that can handle bank conflicts (through a long reorder buffer like structure) and schedule a request to DRAM every b cycles, where b can be less than the random access time of DRAM. A comparison of their approach and RADS [17] reveals that CFDS requires less head and tail SRAM and it can provide packet buffering at 160 gbps. The data granularity for DRAM used in [12] is b cells, where the size of one cell is 64 bytes.

Since our architecture can handle any arbitrary access patterns (they don't have to be structured requests directed by a queue management algorithm), the packet buffering will just be a special case of our system to provide one write access and one read access. Instead of keeping large head and tail SRAMs to store packets, we just need to store the head and tail pointers of each queue in SRAM. On a read from a particular queue, the head pointer will be incremented by the packet size, whereas a write to a particular queue will increment the tail pointer by the packet size. Our universal hash hardware unit randomizes the address from these pointers uniformly across different banks. In our approach, a request can be issued per cycle, whereas in [12] a request can be issued every b cycle. Their architecture is very difficult to design for $b = 1$ as they have also said in their paper *"The implementation of RR scheduling logic for OC-3072 and* $b = 1$ *is certainly of difficult viability."*

As we just need to store the head and tail pointers for each queue (rather than actual entries in the queue), we can provide support for a large number of queues (up to 4096 with an SRAM size of 32KB – which can be further increased to support even more queues). We use the same data granularity used in [12] and compare our results with [22], RADS [17], and CFDS [12] by taking into account the throughput, area overhead, normalized delay and maximum number of supported interfaces. The comparison results are provided in Table 3 for 0.13 μm technology. Table 3 shows that our scheme and CFDS scheme [12] can provide data throughput of 160 gbps because memory requests can be scheduled every cycle in our case and every b cycles in CFDS scheme. But our scheme requires about 35% less area, introduces ten times less latency, and can support about five times the number of interfaces compared to the CFDS scheme.

5.4.2 Packet Reassembly

In an intrusion detection/prevention processing node, the content inspection techniques scan each incoming packet for any malicious content. Since most of these technique examine each packet irrespective of the ordering/sequence of packets, they are less effective for intrusion detection because a clever attacker can craft out-of-sequence TCP packets such that the worm/virus signature is intentionally divided on the boundary of two reordered packets. By doing TCP packet reassembly as a preprocessing step, we can ensure that packets are always scanned in-order. In essence packet reassembly provides a strong front end to effective content inspection.

Table 3: Comparison of packet buffering schemes with our generalized architecture

Packet buffering scheme	Max. line rate (gbps)	SRAM size (bytes)	Area in mm^2	Total delay in ns	No. of supported interfaces
Aristides et al. [22]	10	520 KB	27.4	-	64000
RADS [17]	40	64 KB	10	53	130
CFDS [12]	160	-	60	10000	850
Our approach	160	320 KB	41.9	960	4096

While Dharmapurikar et al. [9] have proposed a packet reassembly mechanism which is robust even in the presence of adversaries, unlike the state of the art in packet buffering techniques, their algorithm does not consider the presence of memory banks (and thus the bounds on performance are not tight). Of course algorithms designers would rather deal with network problems than mapping their data structures to banks by hand. VPNM provides exactly that ability and we have mapped their technique [9] to a virtually pipelined memory system. Using the same data granularity for DRAM as in [12] and processing 64 bytes or less each cycle, we find the need to perform one DRAM read access for accessing connection record, one DRAM access for accessing the corresponding hole-buffer data structure, one DRAM access to update this data structure, one DRAM access to write the packet, and one DRAM access to finally read the packet in future. Hence, for each 64-byte packet chunk, five DRAM accesses are required. Since our memory system can process requests every cycle, with a 400MHz RDRAM [23] we can get an effective throughput of (400 MHz/5)*64 bytes/sec = 40 Gbps, which is more than enough to feed current generation of content inspection engines. We do require some amount of extra storage space compared to [9] as we need to store each packet in FIFO for the duration of three DRAM accesses ($3 * D$), which requires 72 Kbytes of SRAM.

6 Conclusion

Network systems are increasingly asked to perform a variety of memory intensive tasks at very high throughput. In order to reliably service traffic with guarantees on throughput, even under worst case conditions, specialized techniques are required to handle the variations in latency caused by memory banking. Certain algorithms can be carefully mapped to memory banks in a way that ensures worst case performance goals are met, but this is not always possible and requires careful planning at the algorithm, system, and hardware levels. Instead we present a general purpose technique for separating these two concerns, virtually pipelined network memory, and show that with provably high confidence it can simultaneously solve the issues of bank conflicts and bus scheduling for throughput oriented applications. To achieve this deep virtual pipeline, we had to solve the challenges of multiple conflicting requests, reordering of requests, repeated request, and timing analysis of the system. We have performed rigorous mathematical analysis to show that there is on order of one stall in every 10^{13} memory accesses. Furthermore, we have provided a detailed simulation, created a synthesizable version to validate implementability and estimated hardware overheads to better understand the tradeoffs. To demonstrate the performance and generality of our virtually pipelined network memory we have considered the problem of packet buffering and packet reassembly. For packet buffering application, we find that our scheme requires about 35% less area, about ten times less latency and can support about five times more number of interfaces compared to the best existing scheme for OC-3072 line rate. While we have presented the packet buffering and reassembly implementation using our architecture, in the future we will explore the potential of mapping other data plane algorithms into DRAM including packet classification, packet inspection, application-oriented networking and potentially even a broader class of irregular streaming applications.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments. We also thank John Brevik from UCSB for providing help and useful insights to the mathematical analysis. This work was funded in part by NSF Career Grant CCF-0448654.

References

[1] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 123–133, 2005.

[2] G. A. Bouchard, M. Calle, and R. Ramaswami. Dynamic random access memory system with bank conflict avoidance feature. United States Patents, US 6,944,731, September 2005.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[4] F. Chung, R. Graham, and G. Varghese. Parallelism versus memory allocation in pipelined router forwarding engines. In P. B. Gibbons and M. Adler, editors, *SPAA*, pages 103–111. ACM, 2004.

[5] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 68, 1998.

[6] R. Crisp. Direct Rambus technology: The new main memory standard. *IEEE Micro*, 17(6):18–28, November/December 1997.

[7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 222–233, 1999.

[8] B. Davis, B. L. Jacob, and T. N. Mudge. The new DRAM interfaces: SDRAM, RDRAM and variants. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 26–31, 2000.

[9] S. Dharmapurikar and V. Paxson. Robust TCP reassembly in the presence of adversaries. In *Proceedings of 14th USENIX Security Symposium*, pages 65–80, Baltimore, MD, August 2005.

[10] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 160–170, 1997.

[11] W. fen Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 301, 2001.

[12] J. Garcia, J. Corbal, L. Cerda, and M. Valero. Design and implementation of high-performance memory systems for future packet buffers. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 373, 2003.

[13] M. Gries. A survey of synchronous RAM architectures. Technical Report 71, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Gloriastrasse 35, CH-8092 Zurich, Apr. 1999.

[14] J. Hasan, S. Chandra, and T. N. Vijaykumar. Efficient use of memory bandwidth to improve network processor throughput. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 300–313, 2003.

[15] J. Hasan, V. Jakkula, S. Cadambi, and S. Chakradhar. Chisel: A storage-efficient, collision-free hash-based packet processing architecture. In *Proceedings of The 33rd Annual International Symposium on Computer Architecture (ISCA 33)*, Boston, MA, June 2006.

[16] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct Rambus memory. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 80, 1999.

[17] S. Iyer, R. R. Kompella, and N. McKeown. Designing packet buffers for router linecards. Technical Report TR02-HPNG-031001, Stanford University, Nov. 2002.

[18] E. Jaulmes, A. Joux, and F. Valette. On the security of randomized cbc-mac beyond the birthday paradox limit: A new construction. In *FSE '02: Revised Papers from the 9th International Workshop on Fast Software Encryption*, pages 237–251, 2002.

[19] S. Kumar, P. Crowley, and J. Turner. Design of randomized multichannel packet storage for high performance routers. In *13th Annual Symposium on High Performance Interconnects (Hot Interconnects)*, Palo Alto, CA, August 2005.

[20] T. Lang, M. Valero, M. Peiron, and E. Ayguade. Conflict-free access for streams in multimodule memories. *IEEE Transactions on Computers*, 44(5):634–646, 1995.

[21] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis. Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 39–48, 2000.

[22] A. Nikologiannis and M. Katevenis. Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques. In *In the Proceedings of the IEEE International Conference on Communications (ICC'2001)*, pages 2048–2052, Helsinki, Finland, June 2001.

[23] RamBus. RDRAM Memory: Leading Performance and Value over SDRAM and DDR, 2001.

[24] B. R. Rau. Pseudo-randomly interleaved memory. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 74–83, 1991.

[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, 2000.

[26] Samsung. Samsung Rambus MR18R162GDF0-CM8 512MB 16bit 800MHz datasheet, 2005.

[27] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 288–299, 2003.

[28] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report Western Research Lab (WRL) Research Report, 2001/2.

[29] G. Shrimali and N. McKeown. Building packet buffers with interleaved memories. In *Proceedings of Workshop on High Performance Switching and Routing*, Hong Kong, May 2005.

[30] J. Truong. Evolution of network memory. Samsung Semiconductor, Inc., March 2005.