High-Bandwidth Network Memory System Through Virtual Pipelines

Banit Agrawal and Timothy Sherwood, *Member, IEEE*

***Abstract*—As network bandwidth increases, designing an effective memory system for network processors becomes a significant challenge. 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. Simply relying on large, fast SRAMs alone is not likely to be scalable or cost-effective. Instead, trends point to the use of low-cost commodity DRAM devices as a means to deliver the worst-case memory performance that network data-plane algorithms demand. While DRAMs can deliver a great deal of throughput, the problem is that memory banking significantly complicates the worst-case analysis, and specialized algorithms are needed to ensure that specific types of access patterns are conflict-free.**

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. Virtual pipelining provides a simple-to-analyze programming 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. 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.

***Index Terms*—Bank conflicts, DRAM, mean time to stall, memory, memory controller, MTS, network, packet buffering, packet reassembly, universal hashing, virtual pipeline, VPNM.**

I. Introduction

WHILE consumers reap the benefits of ever-increasing network functionality, the technology underlying these advances requires 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 [2]–[6], 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 3 ns 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 these data structures are continuing to grow significantly, memory for buffering has been increasing almost proportionally with the line rate (40 to 160 Gbps), routing tables have grown from 100 to 360 K prefixes, and classification rules have increased from 2000 to 5000 rules in recent years. 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 that are optimized for common-case performance. The pipeline provides a programming model and timing abstraction that greatly eases analysis. A novel memory controller upholds that abstraction and handles all the complexity of the memory system, including bank conflicts, bus scheduling, and bursts of accesses to the exact same address. 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 (Fig. 1). 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. (Sections III-B and IV-A2)
- Reordering of requests: To resolve bank conflicts, requests need to be reordered, but our virtual pipeline presents a deterministic (in-order) interface. The latencies of all memory accesses are normalized through specialized queues, and accesses are reordered in a distributed (per-bank) manner to create the appearance of fully pipelined memory. (Sections III-C and IV-A4)

Manuscript received August 28, 2007; revised June 17, 2008; approved by IEEE/ACM Transactions on Networking Editor T. Wolf. First published May 29, 2009; current version published August 19, 2009. This work was supported in part by NSF Career Grant CCF-0448654 and NSF-CT-T 0524771. This work is an extended version of the authors' work in MICRO 39, 2006.

The authors are with the Computer Science Department, University of California, Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: banit@cs.ucsb.edu; sherwood@cs.ucsb.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2008.2008646

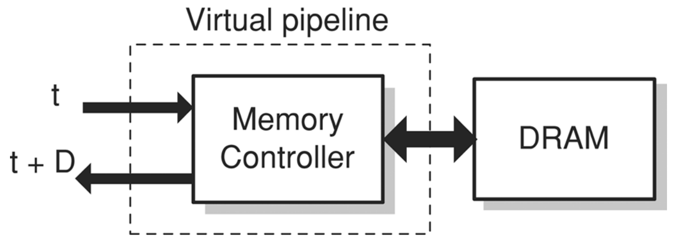


Fig. 1. High-level block diagram of virtual pipeline where each access to DRAM is normalized to a deterministic latency (D).

- 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. (Sections III-D and IV-A1)
- 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 V-B and V-C)

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 VI).

II. 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 for 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 [7], and other architectural techniques [8]–[11]. While latency is critical, traditional machines are far more tolerant of nonuniform 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 [12]. A second example is the memory scheduling algorithm, where memory bandwidth is maximized by reordering various command requests [13]. In the vector processing domain [14], a long stream requires conflict-free access for larger number of strides. Rau *et al.* [15] 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 pseudorandom 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 [16]. Corbal *et al.* [17] 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 an 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 were a *general-purpose* way to control banked access such that conflicts *never* occur. Sadly, this is not possible in the general case [18]. However, if the memory access patterns can be carefully constrained, algorithms can be developed that 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 [3], [4], 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 packet buffer queues onto banks of memory such that conflicts are either unlikely [6], [19], [2], [20] or impossible [4], [3]. 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 [4], 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. For example, Kumar *et al.* present a technique for buffering large packets by randomly distributing parts of the packet over many different memory channels [21]. However, this technique can handle neither small packets nor bank conflicts. Another important special case is data-plane algorithms that may also suffer from memory-bank conflict concerns. Whether these banks of memory are on- or off-chip, supporting multiple nonconflicting 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.* [22]. A tree is broken into many subtrees, each of which is then mapped to parts of a rotating pipeline. They prove that optimally allocating the subtrees is NP-complete and present a heuristic mapping instead. Similarly in [23], a conflict-free hashing technique is proposed for longest prefix match (LPM), where conflicts are handled 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 III, a VPNM 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.

III. 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 VI-A, we compare our implementation against several special-purpose architectures for packet buffering.

A. Dram Banks

Modern DRAM designs try to expose the internal bank structure so accesses can be interleaved and the effective bandwidth can be increased [24], [25]. The various types of DRAM differ primarily in their interfaces at the chip and bus level [26]–[28], but the idea of banking is always there. Experimental evidence [28] indicates that on average PC133 SDRAM works at 60% efficiency and DDR266 SDRAM works at 37% efficiency, where 80%–85% of the lost efficiency is due to the bank conflicts. To help address this problem, RDRAMs expose many more banks [28]. 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 [29].

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 $L = 20$ [29], [30]. If L is smaller, then our approach will be even more efficient. Besides bank conflicts, DRAM is also unavailable for incoming requests during the refresh operation. However, there are techniques such as concurrent refresh [31], where refresh operation can be done simultaneously with read/write operation and, with further optimizations, unavailability due to refresh operation could be eliminated.

B. 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 such an adversary can never tractably construct a sequence of accesses that performs poorly. First, we map the data to banks in permutations that are provably as good as random. Universal hashes [32], an idea that has been extended by the cryptography community, provide a way to ensure that an adversary cannot figure out the hash function without direct observation of conflicts. These hashing techniques are very similar to block ciphers, although the complexity is significantly reduced. This reduction comes from the fact that a universal hash only needs to output $\log_2 B$ bits for B banks, as opposed to the 64- or 128-bit outputs required from a block cipher. While a strong cryptographic hash ensures that collisions are impossible for an adversary to discover (which necessarily requires a very large range), a universal hash (which works over small ranges, such as the number of banks in our system) only ensures that collisions do not occur with greater than uniformly random probability.

For bank mapping, we need perfect universal hashing in the sense that it needs to appear perfectly random (not perfectly balanced), and universal hashing provides this guarantee. There could be a chance that we get unlucky and all accesses might result in bank conflicts, but this is true in the same way that a hacker might get lucky and guess a cryptographic key on the first try. The probability of this happening even when an attacker is trying very hard is what is important, and that can be significantly lowered if there is no direct observation of conflicts. The virtual pipeline provides exactly this functionality and prevents an adversary from 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, this can be a huge benefit in the network space.

C. Distributed Scheduling Around Conflicts

While universal hashing provides the means to prevent our theoretical adversary from constructing access sequences that result in more conflicts than a randomly generated sequence, even in a random assignment of data to banks, a relatively large number of bank conflicts can occur due to the Birthday Paradox [33]. 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

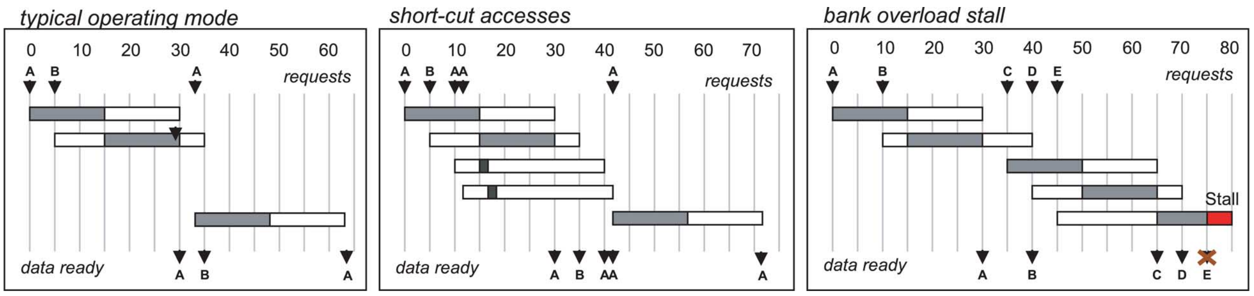


Fig. 2. An example of how each bank controller will normalize the latency of memory accesses to a fixed delay $(D = 30)$. In all three graphs, the x-axis is cycles, and each memory access is shown as a row. The light white boxes are the times during which the request is "in the pipeline," while the dark gray 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 do not 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 $30/15 = 2$.

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

D. 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 fills up and we need to stall. Reducing the probability of this happening even for the worst-case access pattern requires careful architectural design and mathematical analysis. In fact, there are two ways 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 V. In practice, we find that normalizing D to about 1000 ns is more than enough and is also several orders of magnitude less than a typical router processing latency of 0.5 to 2 ms [34]. 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 IV. 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 V-C, 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 Fig. 2, we show how the VPNM 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 each request must be satisfied by the one bank that contains its 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.

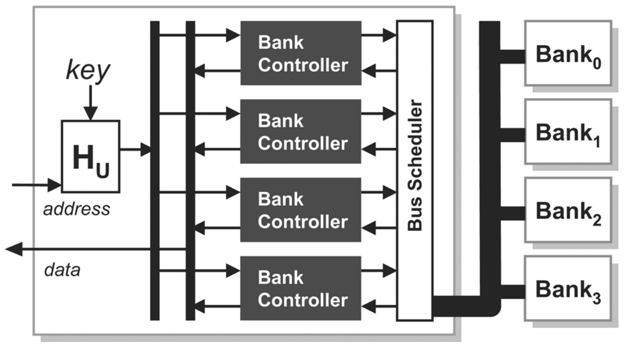


Fig. 3. Memory controller block diagram. After an access is mapped to the proper bank by the universal hash engine (H_U), it is sent to the corresponding bank controller for further processing.

IV. 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. Fig. 3 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 a slightly higher number of memory bus transactions per second than 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 bank 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 architecture, this inefficiency can be eliminated.

After the determination of which bank a particular memory request needs to access, the request is handed off to the appropriate bank controller that handles all the timing and scheduling for that one 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 V, 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 the stall time is also very low (in the worst case, a full memory access latency), 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: 1) observe the exact instant of the stall; 2) remember the exact sequence of accesses that caused the stall; and 3) 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 V-B and V-C), 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 reorder the data on the occurrence of multiple stalls (an expensive operation but certainly possible with frequency on the order of once a day).

A. Bank Controller Architecture

Solving the challenges described in Section I 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 Fig. 4. 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, and sending data to the interface at a prespecified time slot to ensure the deterministic latency, and each of these components is designed to address one or more of the challenges mentioned earlier. We now describe each of these components in detail.

1) 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 nonredundant 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, we associate a counter with each address and data. An incrementer/decrementer is associated with each counter to keep track of the number of unserviced requests for the corresponding address. The buffer contains K rows, where each row contains an address of A bits, a 1-bit address valid flag, a counter of C bits, and data words of W bits. The number of rows K plays a pivotal role in deciding the stall rate of the system, which we discuss in detail in Section V-B. A and W can be assigned any design-specific values, whereas the value of C depends on the overall deterministic latency (D). It should be at least $\log_2 D$ (for $D = 160$, $C = 8$ bits). A row is called *free* when the counter of that row is zero. A *first-zero* circuit finds out the first *free* row. The *free* row gets assigned to a new read request, and the address is written to the address content addressable memory (CAM). The *free* row is updated using the first-zero circuit at the same time. The data words are buffered in the specified row whenever the read access to memory bank completes. The buffering of data words in delay storage buffer ensures that we can handle repeated requests to the same address and can provide the data words to the interface side after a deterministic latency (D).

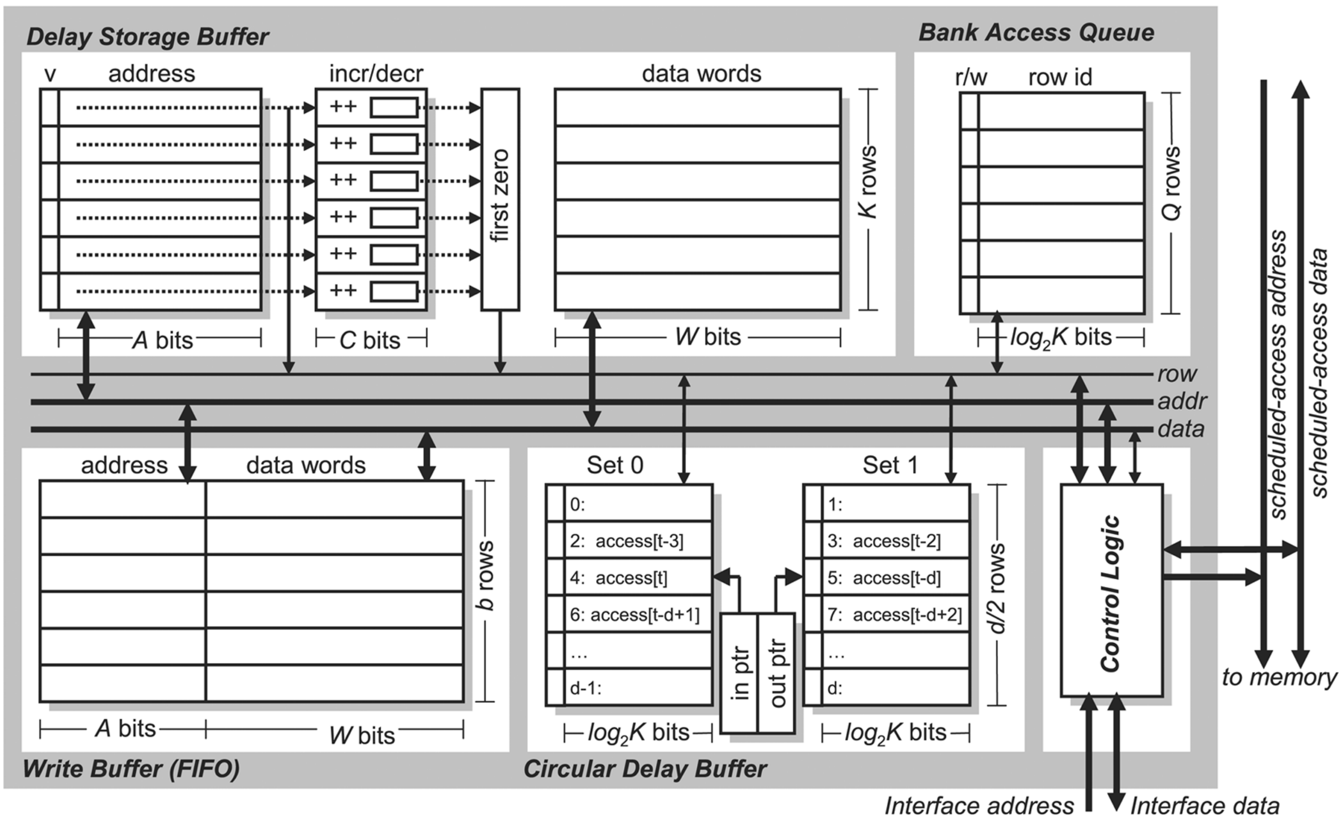
2) Bank Access Queue: The bank access queue keeps track of all pending read and write requests that require access to the memory bank. It can store up to Q interleaved read or write requests in FIFO order. The value of Q is also crucial to dictate the stall rate of the system, which we discuss in detail in Section V-C. To minimize the area overhead, we just keep a few bits to store the requests. There is 1 bit per entry to specify the type of access, i.e., whether the request is a read or write request. The remaining $\log_2 K$ bits per entry are used only in the case of read accesses to specify the corresponding row in delay storage buffer (that way we do not require duplicate copy of address).

3) Write Buffer: The write buffer is organized as a FIFO structure, which stores the address and data of all incoming write requests. Unlike read requests, we do not need to wait for the write requests to complete. We only need to buffer each write request until it gets scheduled to access the memory bank. The write buffer can contain a maximum of b write requests, where each request row consists of A bits of address and W bits of data. In our case, the value of b is chosen as $Q/2$, assuming equal read and write operations (to make a close comparison to packet buffering).

4) Circular Delay Buffer: The circular delay buffer stores the request identifier (id) of every incoming read request in SRAM memory. Then, the read request is served to the output interface after the deterministic latency (D). Hence, we need a FIFO of length D, where. on one end. the request identifier (which specifies a row in delay storage buffer) for new read requests is added and, on the other end, the request identifier is read to serve the data to the output interface. This operation requires one read port and one write port in SRAM memory. Hence, two-ported memory design increases the area overhead

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).

Fig. 4. Architecture block diagram of the VPNM bank controller.

and power consumption. This circular delay buffer is the only component that is accessed every cycle irrespective of the input requests. This problem motivates the design of our 2-set SRAM architecture, where *in* and *out* pointers are used for writing and reading, respectively. The size of each SRAM set is $D/2$, and the width of each set entry is $1 + \log_2 K$ bits. The valid bit is 1 bit to signify any valid read requests appeared in that cycle, whereas $\log_2 K$ bits represent the encoded row id of a row in delay storage buffer. 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).

As we have to provide a fixed latency of D cycles between an *in* operation and an *out* operation, the write pointer *in* chases the read pointer *out* by exactly one cycle delay.} This *in* pointer chasing accounts for $(D-1)$ delay. Then, we latch the output data for one more cycle to achieve deterministic latency (D). The two-set architecture requires only one read-write port, and both sets can be accessed in parallel (one for *in* operation and the other for *out* operation). This reduces the overall delay and provides considerable energy and area savings. A sample of pointer switching with different time of accesses is shown in the two sets of circular delay buffer in Fig. 4. The end result is that this buffer lets us normalize the latency to D very efficiently.

5) 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.) The control logic also controls the interface-side request handling and memory-side request scheduling. It keeps an encoded read row for which the memory is currently being accessed so that, on the completion of the request, the data can be written to this row in delay storage buffer. It also invalidates the *in* entry in the circular delay buffer in case there is no read request in the current cycle.

B. Controller Operations

At a high level, each memory request goes through four 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 the 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 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. However, this row cannot be used for a new read request until all previous read requests are serviced and the counter reaches zero because 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.

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. 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.

C. 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 that require a stall to resolve, each of which is influenced by a different subset of the parameters.

TABLE I
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

1) 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: a) number of rows (K) in delay storage buffer; b) deterministic delay (D); and c) 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 V-B.
2) Bank access queue stall: When a new nonrepeating 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 access queue stall. There are three main parameters that control this stall: a) average input rate, which is equal to $1/B$, where B is the number of banks; b) queue size (Q); and c) the output rate, which is decided by the ratio (R) of frequency on the memory side and frequency on the interface side. In Section V-C, we discuss exactly how to perform the confidence analysis for this stall.
3) 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 the bank access queue size, the chances of stall rate in write buffer are much less than the stall rate in bank access queue. The analysis of the write buffer stall is similar to the analysis of bank access queue and does not dominate the overall stall, so we will only discuss the bank access queue and delay storage buffer stall in our mathematical analysis in Section V.

V. ANALYSIS OF DESIGN

The VPNM can stall in the three ways described in Section IV-C. 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. Additionally, 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 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 IV and summarized in Table I.

To evaluate the effect of these parameters on MTS, we performed three types of analysis: simulation (for functionality),

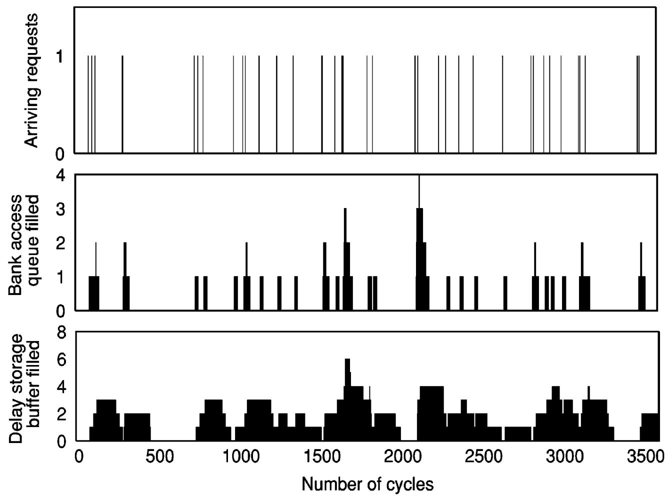


Fig. 5. Bank request controller simulation using $Q = 4$, $K = 8$, and $B = 128$.

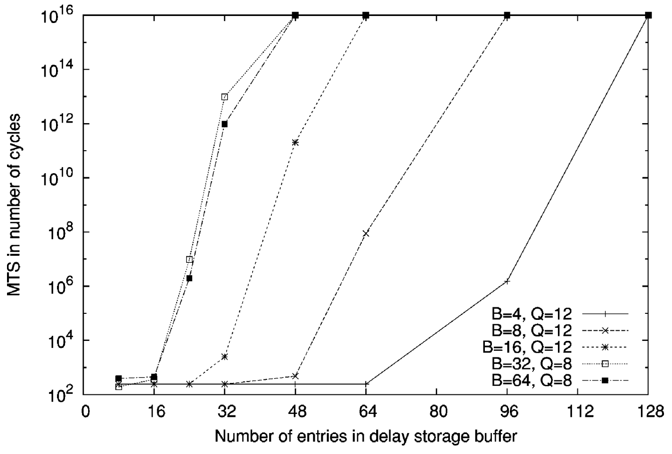


Fig. 6. MTS variation with number of entries in delay storage buffer (K) for memory controller with $R = 1.3$.

mathematical (for MTS), and design (to quantify the hardware overhead). To get an understanding of the execution behavior of our design and verify our mathematical models, we have built functional models in both C and Verilog and 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 MTS for both these cases, the effect of normalized delay, and a high-level analysis of the hardware required.

A. *Simulation of the Design*

As described earlier, the number of rows in a delay storage buffer and the size of bank access queue are important design parameters to decide the stall rate of the system. In this section, we present a simple simulation with random accesses just to provide a look into the intuition behind our approach. Fig. 5 shows the resource utilization over time for one bank controller. In this illustration, the number of banks is 128, the number of rows in delay storage buffer is 8, and bank access queue size is 4. The top bar shows the arrival of requests to the controller, which is a random process. The utilization of the two critical resources described above are in the lower two graphs. As seen, in the window of 2200–2300 cycles, a burst of requests comes to the controller and almost fills the bank access queue, and four rows are reserved in the delay storage buffer. Subsequently, the number of busy rows and pending entries decrease because input burst requests are serviced, and there is no new incoming request for some time. We could have seen a stall in a case when a burst of requests arrive in short time such that it fills up either the delay storage buffer or the bank access queue. However, queue/buffer parameters are large enough in this case to not witness such a stall. In the next sections, we see how these architectural parameters effect the stall rate through a detailed mathematical analysis.

B. *Delay Storage Buffer (DSB) Stall*

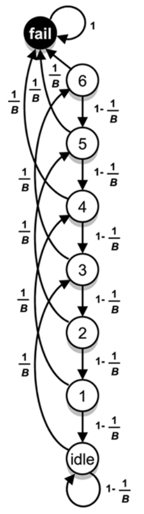
A delay buffer entry is needed to store the data associated with an access for the duration of D cycles. A buffer will overflow if there are more requests assigned to it over a period of D cycles than there are places to store those requests. To calculate the MTS, we need to determine the expected amount 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 $\binom{D-1}{K-1} \cdot \left(\frac{1}{B}\right)^{K-1}$, so the probability of not having a delay buffer overfill over the given interval is $1 - \binom{D-1}{K-1} \cdot \left(\frac{1}{B}\right)^{K-1}$. 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 $(1 - \binom{D-1}{K-1} 1 \cdot \left(\frac{1}{B}\right)^{K-1})^{T-D+1}$. 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 MTS is

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

Fig. 6 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 analyses.[2] Fig. 6 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.

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



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

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

Fig. 7. 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 there to be L more cycles worth of work.

C. Bank Access Queue (BAQ) Stall

Performing an analysis similar to that presented in Section V-B will not work for the bank access queue because 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. Each state in this machines corresponds to an amount of work left for the queue to perform (the number of cycles left until the queue would be empty). The transitions between these states are probabilistic, based on whether a new request arrives at the bank controller on any given cycle. If there are no requests in the queue, then we are at a ground state with no work to be done. If no new request comes on a cycle, then there is one cycle less work for the queue to do. We know this because all the accesses are serialized for a given bank, and as long as there is something in the queue, work is being done toward its completion. If a new request comes in, then there is $L-1$ more cycles worth of work to do. Finally, the queue will overflow if there is more than QL worth of work do, as that is the maximum amount that can be stored.

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 Fig. 7, we can see the probabilistic model stored both as a directed graph and in an 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 Fig. 8. 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 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).

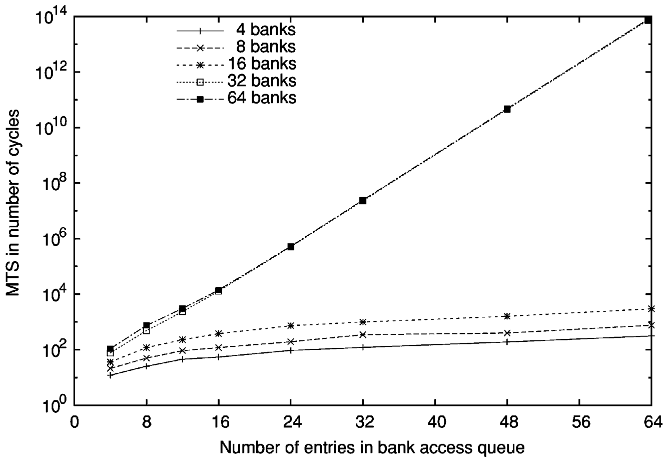


Fig. 8. MTS variation with number of entries in bank access queue (Q) for our controller with $R = 1.3$.

D. Overall Stall Rate

Now that we have analyzed delay storage buffer (DSB) stall and bank access queue (BAQ) stall, we present how total stall rate is effected based on different parameters. The overall system MTS is decided by whether DSB stalls first or BAQ stalls first, and hence, it is the minimum of the two. To illustrate the tradeoff between these two stall rates, Fig. 9 shows the MTS as a function of both the normalized delay and the total number of delay buffers. The normalized delay has very crucial role since it puts pressure on the delay buffers but eases bank access queue.

The number of banks is fixed to 32, and the bus scaling ratio to 1.3. We vary the normalized delay up to 1.5 μs (shown on x-axis). The normalized delay (D) also represents the size of bank access queue (Q) because it is directly proportional to Q. On the y-axis, we vary the number of rows in the DSB from 8 to 128. The MTS is shown on the z-axis, which is in log scale.

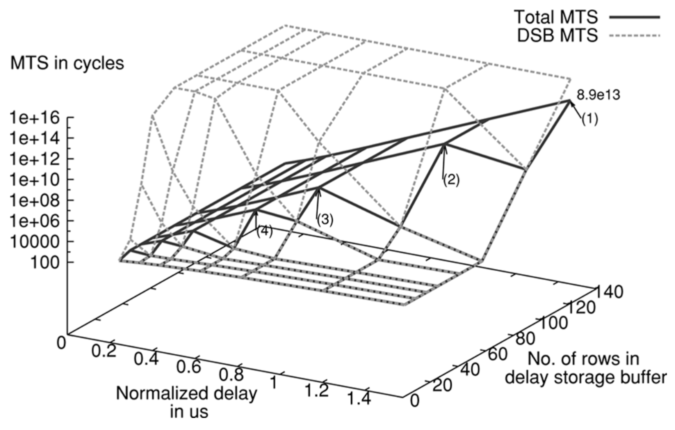


Fig. 9. Effect of normalized delay and DSB's rows on total stall rate. While total MTS is decided by DSB MTS in the lower triangular surface (high delay, less rows), best design points are shown near the diagonal where DSB and BAQ compete for deciding the total stall rate.

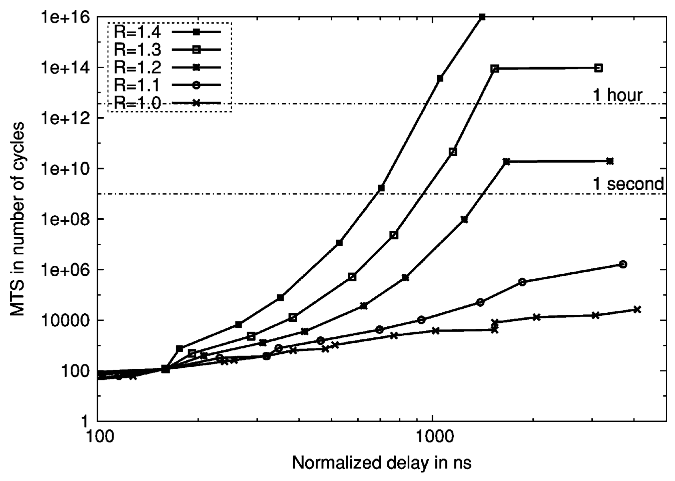


Fig. 10. Best achievable MTS for a particular normalized delay for different frequency scaling ratios (R).

The dotted gray lines show the MTS for the delay storage buffer, while the solid black lines show the total MTS.

When we first take a look at the DSB MTS, we find that as the delay grows, an increasing number of rows are required to achieve the optimal MTS. For example, when delay is less than 0.2 μs, *an* upper limit is reached even with less than 24 rows. However, if delay is increased to more than 1.4 μs, 128 rows are required to reach the upper limit of 10^{16}. The longer the delay, the more delay buffers are required to prevent stalls.

The total MTS is a combination of the DSB stalls and the BAQ stalls. The BAQ MTS is dependent on the size of bank access queue and the normalized delay. With a very large number of DSBs, the total MTS grows exponentially with the normalized delay. This is the same result we have shown earlier in Fig. 8, but here we show that the BAQ stall dictates the total stall rate when the delay is lower and the number of rows is higher (upper triangular surface in figure for total MTS). Hence, both DSB MTS and BAQ MTS are closest along the diagonal line. These are the most balanced design points, and in the figure, they are labeled with numbers (1)–(4).

E. Normalized Delay Analysis

In this section, we present the Pareto-optimal results for normalized delay where we find the best achievable MTS for a particular normalized latency. To find the Pareto-optimal points, we assume a large delay storage buffer and sweep across a large set of design parameters to get the best possible MTS for a given normalized delay. Fig. 10 shows the best MTS possible for different normalized latencies and for different scaling ratios. The normalized delay is on the x-axis in log scale, whereas the y-axis shows the MTS, again in log scale. We find that for $R < 1.2$, it is difficult to achieve a high MTS even if we increase the delay significantly. On the other hand, for $R >= 1.2$, the value of MTS increases exponentially with increasing delay. For $R = 1.2$ case, we can achieve a MTS of greater than 1 s with delay higher than 1 μs, but an MTS of greater than 1 h is not possible. However, in the case of $R = 1.3$ and $R = 1.4$, where we have to sacrifice the bandwidth a little further, we can achieve a much higher MTS (more than an hour) with a lower delay compared to the other cases. While this latency is significant, it is still far less than several existing special-purpose packet buffering approaches (as we will discuss later). For nonbuffering applications, this latency might be hidden through a sufficient amount of thread-level parallelism (swapping between threads with deterministic timing) and may further require more packet buffering (as packets may take longer to egress). We have quantified the extra memory requirements for two high-throughput applications in Section VI, but a larger study of the system-level effects of this latency is outside the scope of this work.

F. Hardware Estimation

The structures presented in Section IV ensure that only probabilistically well-formed modes of stall 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, it is hard to understand the tradeoffs fully without a careful estimate of the area and power overhead. To explore this design space, we developed a hardware overhead analysis tool for our bank controller architecture that takes these design parameters $(B, L, K, Q, R, \mathrm{tech})$ 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 [35] and our synthesizable Verilog model to design our overhead tool and use 0.13 μm CMOS technology to evaluate the hardware overhead.

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 s (10^9) 1 h (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

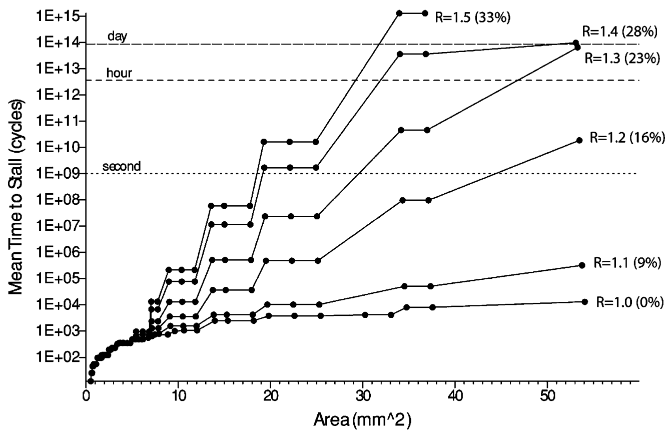


Fig. 11. MTS with area overhead for our memory controller for different frequency ratios (R).

TABLE II
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

and compares favorably with past special purpose designs (see Section VI).

The Pareto-optimal curve for our memory controller is shown in Fig. 11. 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 $R = 1.3$ (1 s MTS $= 10^9$ for about 30 mm^2) or $R = 1.4$ (1 h MTS $= 3.6 \times 10^{12}$ for about 30 mm^2) to get the best values of MTS without compromising much of the memory bus speed utilization.

We calculate the optimal parameters from Fig. 11 and find the energy consumption for these optimal parameters. The optimal parameters along with all design constraints are shown in Table II. The table shows that for $R = 1.3$ and $R = 1.4$, we need around 32 banks, 32–48 bank access queue entries, and 64–96 storage delay buffer entries with 10–20 nJ energy consumption.

VI. APPLICATIONS MAPPING

To demonstrate the usefulness and generality of our approach, in this section, we show how our memory system can be easily used by high-throughput network applications in edge or core routers. While such commercial routers are almost always custom-designed to meet the worst-case performance, recently there has been a great deal of discussion regarding a new generation of software-centric platforms built around more general-purpose cores both academically (GENI [36]) and commercially [37]. In the case of programmable routers, a large number of thread contexts could be used to fill the virtual pipeline every cycle, fully utilizing the guaranteed high-memory bandwidth. In addition to the ease of scheduling that deterministic latency provides, the biggest advantage here is that nearly complete determinism in our approach provides complete fairness in access to memory bandwidth and ensures total performance isolation in terms of memory system. However, an analysis of this is beyond the scope of this work. While many new applications will be written in software/microcode, many highly memory-intensive applications will be designed at the hardware and firmware levels to achieve the desired performance with worst-case guarantees. To demonstrate the applicability of our approach in this scenario, we have implemented 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.

A. 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, to avoid losing throughput (especially in edge routers), the amount of buffering required is RT [3], where R is the line rate and T is the two-way propagation delay through the router. For 160-Gbps line rate and a typical round-trip time of 0.2 s [4], the buffer size will be 4 GB. The main challenge in packet buffering is to deal with constantly increasing line rate (10–40 Gbps and from 40–160 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 [2], [4]. In [2], an out-of-order technique has been proposed to reduce the bank conflict to provide packet buffering requirement for 10 Gbps. Iyer *et al.* [3] 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)]. However, they do not consider the effect of bank conflicts. Garcia *et al.* [4] 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 [3] reveals that CFDS requires less head and tail SRAM and can provide packet buffering at 160 Gbps. The data granularity for DRAM used in [4] is b cells, where the size of one cell is 64 bytes.

Since our architecture can handle any arbitrary access patterns (they do not have to be structured requests directed by a

TABLE III
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. [2]	10	520 KB	27.4	-	64000
RADS [3]	40	64 KB	10	53	130
CFDS [4]	160	-	60	10000	850
Our approach	160	320 KB	41.9	960	4096

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 [4], 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 32 KB—which can be further increased to support even more queues). We use the same data granularity used in [4] and compare our results to [2], RADS [3], and CFDS [4] by taking into account the throughput, area overhead, normalized delay, and maximum number of supported interfaces. The comparison results are provided in Table III for 0.13 μm technology. Table III shows that our scheme and CFDS scheme [4] 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. However, our scheme requires about 35% less area, introduces 10 times less latency, and can support about five times the number of interfaces compared to the CFDS scheme.

B. 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 techniques 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.

While Dharmapurikar *et al.* [38] have proposed a packet reassembly mechanism that 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, algorithm 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 [38] to a virtually pipelined memory system. Using the same data granularity for DRAM as in [4] and processing 64 bytes or less each cycle, we find the need to perform one DRAM read access for accessing the 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 the future. Hence, for each 64-byte packet chunk, five DRAM accesses are required. Since our memory system can process requests every cycle, with a 400-MHz RDRAM [28], we can get an effective throughput of $(400$ MHz$/5)$*64 bytes/s $= 40$ Gbps, which is more than enough to feed the current generation of content inspection engines. We do require some amount of extra storage space compared to [38] since we need to store each packet in FIFO for the duration of three DRAM accesses $(3 * D)$, which requires 72 kB of SRAM.

VII. 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 requests, 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 and about 10 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, there is a potential for mapping other data-plane algorithms into DRAM, including packet classification, packet inspection, application-oriented networking, software-centric programmable routers, and interesting future directions to look at even a broader class of irregular streaming applications.

REFERENCES

[1] B. Agrawal and T. Sherwood, "Virtually pipelined network memory," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO 39)*, 2006, pp. 197–207.

[2] A. Nikologiannis and M. Katevenis, "Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques," in *Proc. IEEE Int. Conf. Commun. (ICC'2001)*, Helsinki, Finland, Jun. 2001, pp. 2048–2052.
[3] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," Stanford Univ., Tech. Rep. TR02-HPNG-031001, Nov. 2002.
[4] J. Garcia, J. Corbal, L. Cerda, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO 36)*, 2003, pp. 373–386.
[5] T. Sherwood, G. Varghese, and B. Calder, "A pipelined memory architecture for high throughput network processors," in *Proc. 30th Annu. Int. Symp. Comput. Archit. (ISCA'03)*, 2003, pp. 288–299.
[6] J. Hasan, S. Chandra, and T. N. Vijaykumar, "Efficient use of memory bandwidth to improve network processor throughput," in *Proc. 30th Annu. Int. Symp. Comput. Archit. (ISCA'03)*, 2003, pp. 300–313.
[7] W. F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proc. 7th Int. Symp. High-Perform. Comput. Archit. (HPCA'01)*, 2001, pp. 301–312.
[8] S. Rixner, "Memory controller optimizations for web servers," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO 37)*, 2004, pp. 355–366.
[9] Z. Zhu, Z. Zhang, and X. Zhang, "Fine-grain priority scheduling on multi-channel memory systems," in *Proc. 8th Int. Symp. High-Perform. Comput. Archit. (HPCA'02)*, 2002, pp. 107–116.
[10] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism," in *Proc. 13th Int. Symp. High-Perform. Comput. Archit. (HPCA'07)*, 2007, pp. 285–294.
[11] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "Design of a parallel vector access unit for SDRAM memory systems," in *Proc. 6th Int. Symp. High-Perform. Comput. Archit. (HPCA'00)*, 2000, pp. 39–48.
[12] 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 *Proc. 5th Int. Symp. High Perform. Comput. Archit. (HPCA'99)*, 1999, pp. 80–89.
[13] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Annu. Int. Symp. Comput. Archit. (ISCA'00)*, 2000, pp. 128–138.
[14] R. Espasa, M. Valero, and J. E. Smith, "Out-of-order vector architectures," in *Proc. 30th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO 30)*, 1997, pp. 160–170.
[15] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proc. 18th Annu. Int. Symp. Comput. Archit. (ISCA'91)*, 1991, pp. 74–83.
[16] T. Lang, M. Valero, M. Peiron, and E. Ayguade, "Conflict-free access for streams in multimodule memories," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 634–646, May 1995.
[17] J. Corbal, R. Espasa, and M. Valero, "Command vector memory systems: High performance at low cost," in *Proc. 1998 Int. Conf. Parallel Archit. Compilation Tech. (PACT'98)*, 1998, pp. 68–79.
[18] F. Chung, R. Graham, and G. Varghese, "Parallelism versus memory allocation in pipelined router forwarding engines," in *Proc. SPAA*, P. B. Gibbons and M. Adler, Eds., 2004, pp. 103–111.
[19] G. A. Bouchard, M. Calle, and R. Ramaswami, "Dynamic random access memory system with bank conflict avoidance feature," U.S. Patent 6 944 731, Sep. 2005.
[20] G. Shrimali and N. McKeown, "Building packet buffers with interleaved memories," in *Proc. Workshop High Perform. Switching and Routing*, Hong Kong, May 2005, pp. 1–5.
[21] S. Kumar, P. Crowley, and J. Turner, "Design of randomized multichannel packet storage for high performance routers," in *Proc. 13th Annu. Symp. High Perform. Interconnects (Hot Interconnects)*, Palo Alto, CA, Aug. 2005, pp. 100–106.
[22] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proc. 32nd Annu. Int. Symp. Comput. Archit. (ISCA'05)*, 2005, pp. 123–133.
[23] J. Hasan, V. Jakkula, S. Cadambi, and S. Chakradhar, "Chisel: A storage-efficient, collision-free hash-based packet processing architecture," in *Proc. 33rd Annu. Int. Symp. Comput. Archit. (ISCA 33)*, Boston, MA, Jun. 2006, pp. 203–215.
[24] R. Crisp, "Direct Rambus technology: The new main memory standard," *IEEE Micro*, vol. 17, no. 6, pp. 18–28, Nov./Dec. 1997.
[25] M. Gries, "A survey of synchronous RAM architectures," Computer Eng. and Networks Lab. (TIK), ETH Zurich, Switzerland, Tech. Rep. 71, Apr. 1999.
[26] B. Davis, B. L. Jacob, and T. N. Mudge, "The new DRAM interfaces: SDRAM, RDRAM and variants," in *Proc. 3rd Int. Symp. High Perform. Comput. (ISHPC'00)*, 2000, pp. 26–31.
[27] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proc. 26th Annu. Int. Symp. Comput. Archit. (ISCA'99)*, 1999, pp. 222–233.
[28] RamBus, "RDRAM memory: Leading performance and value over SDRAM and DDR," 2001.
[29] Samsung, "Samsung RamBus MR18R162GDF0-CM8 512 MB 16 bit 800 MHz datasheet," 2005.
[30] J. Truong, "Evolution of network memory," Samsung Semiconductor, Inc., Mar. 2005.
[31] T. Kirihata *et al.*, "An 800 MHz embedded dram with a concurrent refresh mode," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2004, pp. 15–19.
[32] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J.Comput. Syst. Sci.*, vol. 18, pp. 143–154, 1979.
[33] E. Jaulmes, A. Joux, and F. Valette, "On the security of randomized CBC-MAC beyond the birthday paradox limit: A new construction," in *Proc. Revised Papers 9th Int. Workshop. Fast Softw. Encryption (FSE'02)*, 2002, pp. 237–251.
[34] "Internet core router test: Looking at latency," 2001 [Online]. Available: http://www.lightreading.com/document.asp?doc_id=4009&page_number=7
[35] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power and area model," Western Research Lab (WRL) Res. Rep., Tech. Rep, 2001/2.
[36] J. Turner, "A proposed architecture for the Geni backbone platform," Mar. 2006.
[37] W. Eatherton, "The push of network processing to the top of the pyramid," 2005, ANCS'05 Keynote.
[38] S. Dharmapurikar and V. Paxson, "Robust TCP reassembly in the presence of adversaries," in *Proc. 14th USENIX Security Symp.*, Baltimore, MD, Aug. 2005, pp. 65–80.

Banit Agrawal received the B.Tech. degree in instrumentation engineering from the Indian Institute of Technology, Kharagpur, India, in 2001, the M.S. degree in computer science from the University of California, Riverside, in 2004, and the Ph.D. degree in computer science from the University of California, Santa Barbara, in 2008.

His research interests include memory design and modeling for high-performance networking, 3-dimensional ICs, dataflow analysis, security/analysis processors, nanoscale architectures, and power-aware architectures.

Dr. Agrawal was the recipient of a Dean's Fellowship during 2007–2008, a best paper award in the International Symposium on Code Generation and Optimization (CGO) 2006, a best paper nomination in the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006, and an IEEE Micro Top Pick Award in the Computer Architecture Conference in 2006.

Timothy Sherwood (S'98–M'03) received the B.S. degree from the University of California, Davis, in 1998, and the M.S. and Ph.D. degrees from the University of California, San Diego, in 2003, where he worked with Prof. B. Calder.

He joined the University of California, Santa Barbara (UCSB) in 2003, where he is currently an Assistant Professor. His prior work on program phase analysis methods (a technique for reasoning about and predicting the behavior of programs over time) has been cited over 350 times and is now used by Intel, Hewlett-Packard (HP), and other industry partners to guide the design of their largest microprocessors. His research interests focus on the area of computer architecture, specifically in the development of novel high-throughput methods by which systems can be monitored and analyzed.

Prof. Sherwood was the recipient of a National Science Foundation (NSF) CAREER Award in 2005, and, for the three consecutive years since joining UCSB, he has received the IEEE Micro Top Pick Award for novel research contributions of significance to the computing industry.