Staged Reads : Mitigating the Impact of DRAM Writes on DRAM Reads *

Niladrish Chatterjee
University of Utah
nil@cs.utah.edu

Naveen Muralimanohar
HP Labs
naveen.muralimanohar@hp.com

Rajeev Balasubramonian
University of Utah
rajeev@cs.utah.edu

Al Davis
University of Utah
ald@cs.utah.edu

Norman P. Jouppi
HP Labs
norm.jouppi@hp.com

Abstract

Main memory latencies have always been a concern for system performance. Given that reads are on the critical path for CPU progress, reads must be prioritized over writes. However, writes must be eventually processed and they often delay pending reads. In fact, a single channel in the main memory system offers almost no parallelism between reads and writes. This is because a single off-chip memory bus is shared by reads and writes and the direction of the bus has to be explicitly turned around when switching from writes to reads. This is an expensive operation and its cost is amortized by carrying out a burst of writes or reads every time the bus direction is switched. As a result, no reads can be processed while a memory channel is busy servicing writes. This paper proposes a novel mechanism to boost read-write parallelism and perform useful components of read operations even when the memory system is busy performing writes. If some of the banks are busy servicing writes, we start issuing reads to the other idle banks. The results of these reads are stored in a few registers near the memory chip's I/O pads. These results are quickly returned immediately following the bus turnaround. The process is referred to as a Staged Read *because it decouples a single read operation into two stages, with the first step being performed in parallel with writes. This innovation can also be viewed as a form of prefetch that is internal to a memory chip. The proposed technique works best when there is bank imbalance in the write stream. We also introduce a write scheduling algorithm that artificially creates bank imbalance and allows useful read operations to be performed during the write drain. Across a suite of memory-intensive workloads, we show that Staged Reads can boost throughput by up to 33% (average 7%) with an average DRAM access latency improvement of 17%, while incurring a very small cost (0.25%) in terms of memory chip area. The throughput improvements are even greater when considering write-intensive workloads (average 11%) or future systems (average 12%).*

*This work was supported in parts by NSF grants CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, HP, and the University of Utah.

1 Introduction

Main memory latencies have always been a major performance bottleneck for high-end systems. This bottleneck is expected to grow in the future as more cores on a chip must be fed with data. Already, many studies [11, 26] have shown the large contribution of queuing delays to overall memory latency. A number of studies have focused on memory scheduling and have tried to optimize throughput and fairness [11, 18, 21, 26, 29]. However, only a few optimizations have targeted writes; for example, the Eager Writeback optimization [16] tries to scatter writes so that write activity does not coincide with read activity, and the Virtual Write Queue optimization [37] combines memory scheduling and cache replacement policies to create a long burst of writes with high row buffer hit rates.

Generally, read operations are given higher priority than writes. When the memory system is servicing reads, the DIMMs drive the off-chip data bus and data is propagated from the DIMMs to the processor. Since writes are not on the critical path for program execution, they are buffered at the processor's memory controller. When the write buffer is nearly full (reaches a high water mark), writes have to be drained. The data bus is turned around so that the processor is now the data bus driver and data is propagated from the processor to the DIMMs. This bus turnaround delay (tWTR) has been of the order of 7.5 ns for multiple DDR generations [14, 20, 37]. Frequent bus turnarounds add turnaround latency and cause bus underutilization which eventually impacts queuing delay. Therefore, to amortize the cost of bus turnaround, a number of writes are drained in a single batch until a low water mark is reached. During this time, reads have no option but to wait at the memory controller; the uni-directional nature of the bus prevents reads from opportunistically reading data out of idle banks. Thus, modern main memory systems offer nearly zero read-write parallelism within a single channel.

This paper attempts an optimization that allows reads to perform opportunistic prefetches while writes are being serviced. This is not a form of speculation; the read operation is simply being decoupled into two stages and the stage that does not require the data bus is being performed in tandem with writes. We refer to this optimization as a *Staged Read*. The two stages are coupled via registers near the memory chip's I/O pads that store the prefetched cache

line. This not only minimizes the latency for the more critical second stage (the second stage does not incur delay for memory chip global wire traversal), but is also less disruptive to memory chip design. Prior work [42] has identified the I/O pad area as being most amenable to change and that area already accommodates some registers that help with scheduling.

With the proposed optimization, while writes are being serviced at a few banks, other banks can perform the first stage of read operations. As many prefetches can be performed as the prefetch registers provided at the I/O pads. After the bus is turned around to service reads, these prefetched results are quickly returned in the subsequent cycles without any idling. The Staged Read optimization is most effective when only a few banks are busy performing writes. We therefore modify the write scheduling algorithm to force bank imbalance and create opportunities for Staged Reads. This ensures that the memory system is doing useful read work even when it is busy handling writes.

Such read-write parallelism becomes even more important in future write-constrained systems when (i) writes are more frequent in chipkill systems [41, 45], (ii) writes take longer (because of new NVM cells [22, 33]), (iii) turnaround delays are more significant [37]. Our results show an average improvement of 7% in throughput for our baseline modern system (along with an average DRAM access latency reduction of 17%) and this improvement can grow to 12% in future systems. Applications that are write-intensive (about half of the simulated benchmarks suite) show a 11% improvement in throughput with our innovation.

2 Background & Motivation

2.1 Main Memory Background

The main memory system is composed of multiple channels (buses), each having one or more DIMMs. For most of this study, we will assume that the DIMMs contain multiple DRAM chips, although, the proposed design will apply for other memory technologies as well. When servicing a cache line request, a number of DRAM chips in a *rank* work in unison. Each rank is itself partitioned into multiple *banks*, each capable of servicing requests in parallel. Ranks and banks enable memory-level parallelism, although, each data transfer is eventually serialized on the memory bus. The most recently accessed row of a bank can be retained in a row buffer, which is simply a row of sense-amps associated with each array. The row is then considered "open". If subsequent accesses deal with cache lines in an open row (a row buffer hit), they can be serviced sooner and more efficiently.

A memory chip is organized into many banks; each bank is organized into many arrays. The I/O pads for a chip are placed centrally on a memory chip [42]. From here, requests and data are propagated via tree-like interconnects to individual arrays involved in an access. To maximize density, the arrays have a very regular layout and are sized to be large. When a read request is issued, the bitlines for the corresponding row must be first PRECHARGED (if they haven't already been previously precharged). An ACTIVATE command is then issued to read the contents of a row into the row buffer. There is significant overfetch in this stage: to service a single 64 byte cache line request, about 8 KB of data is read into a row buffer. It is prohibitively expensive to ship this overfetched data on global wires, so the row buffer is associated with the arrays themselves. Finally, a column-select or CAS command is issued that selects a particular cache line from the row buffer and communicates it via global wires to the I/O pads. In the subsequent cycles, the cache line is transmitted to the processor over the off-chip memory bus. Each of these three major components (PRECHARGE, ACTIVATE, CAS) take up roughly equal amounts of time, approximately 13 ns each in modern DDR3 systems [7], and the data transfer takes about 10 ns.

The memory scheduler has to consider resource availability and several timing constraints when issuing commands. Generally, the memory scheduler prioritizes reads over writes, accesses to open rows, and older requests over younger ones. DRAM writes are generated as a result of write-back operations from the LLC. Since writes are not on the processor's critical path, the memory-controller is not required to complete the write operation immediately and buffers the data in a write queue. One of the many timing constraints is the write turnaround delay (tWTR) that is incurred every time the bus changes direction when switching from a write to a read. Writes and reads are generally issued in bursts to amortize this delay overhead. Writes are buffered until the write queue reaches a high water mark (or there are no pending reads); the bus is then turned around and writes are drained until a low water mark is reached.

2.2 Simulation Methodology

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	16-core, 3.2 GHz
Re-Order-Buffer	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache	4MB/64B/8-way, shared, 10-cycle
Coherence Protocol	Snooping MESI
DRAM Parameters	
DRAM Device Paramters	MT41J128M8 DDR3-800 [7], 8 banks/device 16384 rows/bank, x8 part
DRAM Configuration	2 64-bit Channels 1 DIMM/Channel (unbuffered, non-ECC) 2 Ranks/DIMM, 8 devices/Rank
Row-Buffer Size	8KB per bank
Active row-buffers per DIMM	8
Total DRAM Capacity	4 GB
DRAM Bus Frequency	1600MHz
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size	48 entries per channel
High/Low Watermarks	32/16

Table 1. Simulator parameters.

We use the Wind River Simics [4, 24] simulation platform for our study. Table 1 details the salient features of the simulated processor and memory hierarchy. We model an out-of-order processor using Simics' *ooo-micro-arch* mod-

ule and use a heavily modified *trans-staller* module for the DRAM/PCM simulator. The DRAM simulator closely follows the model described by Gries in [7] and shares features with the DRAMSim framework [43].

In this work, we model a modest multi-core (16 core) system with two channels to limit simulation time. The memory controller models a First-Ready-First-Come-First-Served (FR-FCFS) scheduling policy and models the timing parameters described in Table 2. The interplay of these timing parameters is crucial for evaluating DRAM bank management as maintaining the restrictions imposed by the parameters will significantly impact bank usage [20]. The parameters tRAS, tRRD, and tFAW are essential because they impose restrictions on how frequently accesses can be made to the same bank (or same rank) if the accesses are not row hits. In our simulator, our bank usage model adheres to these constraints.

Parameter	DRAM / PCM	Parameter	DRAM / PCM
tRCD	13.5ns / 55ns	tCAS	13.5ns
tRP	13.5ns	tWR	13.5ns / 125ns
tRAS	36ns / 55ns	tRRD	7.5ns
tRTRS	2 Bus Cycles	tFAW	40ns
tWTR	7.5ns	tCWD	6.5ns

Table 2. Timing Parameters [7, 22]

The DRAM device model and timing parameters were derived from [7, 20]. We model multiple ranks per memory channel, each rank has several banks (each with its own row-buffer). The data and address bus models are accurately designed to simulate contention and bus turnaround delays. The DRAM pipeline model is equipped to handle both reads and writes. In the baseline model, writes are enqueued in the write queue on arrival and the write queue gets drained by a specific amount upon reaching a high water mark. The simulator's command scheduling mechanism can overlap commands to different banks (and ranks) to take maximum advantage of the bank level parallelism in the access stream.

DRAM address mapping parameters for our platform (i.e., number of rows/columns/banks) were adopted from the Micron data sheet [7] and the open row address mapping policy from [20] is used in the baseline. We use this address mapping scheme because this results in the best performing baseline on average when compared to other commonly used address interleaving schemes [20, 43].

Our techniques are evaluated with full system simulation of a wide array of memory-intensive benchmarks. We use multithreaded workloads (each core running 1 thread) from the PARSEC [13] (*canneal, fluidanimate*), OpenMP NAS Parallel Benchmark [12] (*cg, is, ep, lu, mg, sp*) and SPECJVM [9] (*lu.large, sor.large, sparse.large, derby*) suites along with the STREAM [3] benchmark. We also run multiprogrammed workloads from the SPEC CPU 2006 suite (*bzip2, dealII, gromacs, gobmk, hmmer, leslie3d, libquantum, omnetpp, perlbench, soplex and xalancbmk*). We selected applications from these benchmark suites that exhibited last level cache MPKI greater than 2 and could work with a total 4 GB of main memory. Each of these single threaded workloads are run on a single core - so essentially each workload is comprised of 16 copies of the benchmark running on 16 cores. We also run

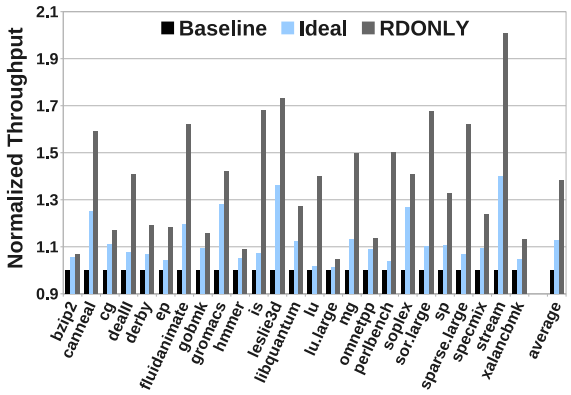


Figure 1. Room For Performance Improvement

a workload designated as specmix which consists of the following single threaded SPEC CPU 2006 applications : *bzip2, bwaves, milc, leslie3d, soplex, sjeng, libquantum, gobmk*. We chose to model cache space per core (4 MB for 16 cores) and memory channels per core (2 channels for 16 cores) that are slightly lower than those in modern systems. This allows us to create the memory pressure per channel that may be representative of a future many-core processor without incurring the high simulation times of such a many-core processor.

For multi-threaded applications, we start simulations at the beginning of the parallel-region/region-of-interest of the application, whereas for the multiprogrammed SPEC benchmarks, we fast forward the simulation by 2 billion instructions on each core before taking measurements. We run the simulations for a total of 1 million DRAM read accesses after warming up each core for 5 million cycles. One million DRAM read accesses correspond to roughly 270 million program instructions on average. For comparing the effectiveness of the proposed schemes, we use the total system throughput defined as $\sum_i (IPC^i_{shared}/IPC^i_{alone})$ where IPC^i_{shared} is the IPC of program i in a multi-core setting. IPC^i_{alone} is the IPC of program i on a stand-alone single-core system with the same memory system.

2.3 Motivational Results

We start by characterizing the impact of writes on overall performance. Figure 1 shows normalized IPC results for a few different memory system models. The left-most bar represents the baseline model with write queue high/low water marks of 32/16. The right-most bar *RDONLY* represents a model where writes take up zero latency and impose zero constraints on other operations. This represents an upper bound on performance that is clearly unattainable, but shows that write handling impacts system performance by 36% on average for our memory-intensive programs. The bar in the middle represents an oracular scheme that is more realistic and closer to the spirit of the Staged Read optimization. It assumes that while writes are being serviced, all pending reads can be somehow prefetched (regardless of bank conflicts), and these prefetched values can be returned in successive cycles following the bus turnaround. This bar is referred to as *Ideal* in the rest of the paper and shows room for a 13% average improvement.

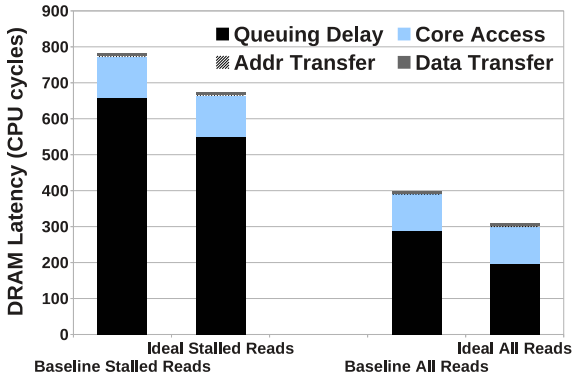


Figure 2. DRAM Latency Breakdown

Figure 2 shows the break-up of the DRAM access latencies of the baseline and the Ideal cases. On the left of the graph, the two bars show the average latencies encountered by reads that have to wait for the write-queue to drain. By finishing the bank access of the reads in parallel with the writes, the Ideal configuration can substantially lower the queuing delay, showing that ramping up the read-pipeline after the write-to-read turnaround is inefficient in the baseline. The impact of this speed-up is noticed in the reduced overall queuing delay for all reads in the system as shown in the two right-most bars in Figure 2.

3 Staged Reads

3.1 Proposed Memory Access Pipeline

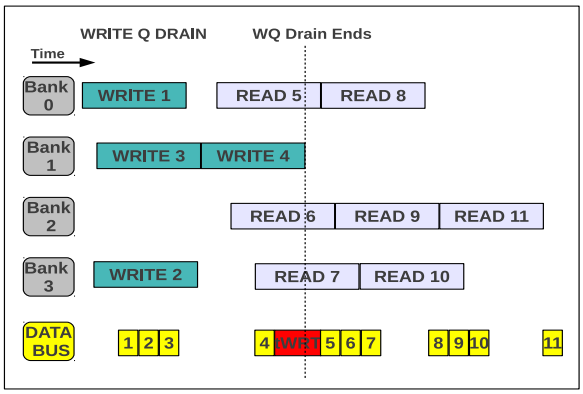
Baseline Scheduling. We assume a baseline scheduling process that is already heavily optimized. When the write queue is draining, we first schedule row buffer hits when possible and prioritize older writes otherwise, while maximizing bank-level parallelism. For most of the write drain process, reads are not issued. As we near the end of the write drain process, as banks are released after their last write, we start issuing PRECHARGE, ACTIVATE, and CAS for the upcoming reads. These are scheduled such that the data is ready for transfer on the bus immediately after the bus is turned around. The pipeline is shown in Figure 3a. Note how the operations for READ-5 begin before the bus is turned around (shown in red by tWTR) and the data transfer for READ-5 happens immediately after the tWTR phase.

Optimization Opportunity. The key to the Staged Read optimization is that there are bus idle slots soon after the bus starts servicing reads (right after data transfer 7 in Figure 3a), and bank idleness when servicing writes (Bank 2 in Figure 3a). Bus idleness when servicing reads happens when the reads conflict for the same banks (Reads-6, 9, 11 all conflict for Bank 2). These idle bus slots could have been filled if some of the reads for Bank 2 could have been prefetched during Bank 2's idle time during the write phase. The baseline scheduling policy can already start issuing up to one read per bank before the bus turnaround happens (for example, Read-5, 6, 7 in Figure 3a). Thus, each bank already does some limited prefetch, whereby the bank access latency of some reads are hidden, with the prefetched lines remaining in the bank's row buffer. The Staged Read optimization is intended to provide prefetch beyond this single read per bank.

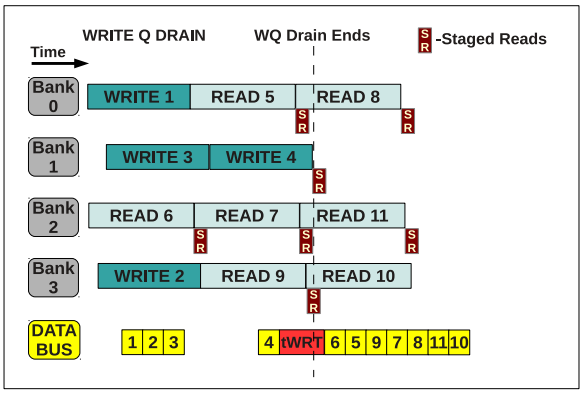
Timing with Staged Reads. Figure 3b shows how Reads-6, 9, 11 for Bank 2 (and all other reads except for Bank 1 reads) are moved further to the left. As soon as a bank is done servicing writes, it starts to service reads. If the read finishes before the bus is turned around, the resulting cache line is saved in a Staged Read register near the IO pads (shown by the SR box in Figure 3b). After the bus is turned around, data blocks are either returned via the normal process (i.e., from the sense amplifier row-buffer through a traditional column-read command) or from Staged Read registers. As seen in Figure 3b, the bus is kept busy for a number of cycles following the turnaround. Many studies, including ours, show that bank conflicts are the source of bus idle cycles and consequently long queuing delays. Hence, such prefetch operations have a favorable impact on the latency of many subsequent reads.

3.2 Staged Read Implementation

Organization and Overheads of Staged Read Registers. As our results show later, most programs do well with 16/32 Staged Read registers. Assuming 64 byte cache lines spread across a rank of eight chips, this corresponds to a storage overhead of only 128/256 bytes per DRAM chip. This is much more efficient than prior proposals that have advocated the use of row buffer caches within DRAM chips [15, 34]. The proposed optimization is much less invasive than row buffer caches for several reasons. First, row buffer cache entries retain entire rows, each about 8 KB in size (this is the rank or DIMM level row-buffer, each constituent chip in a rank has a 1KB row-buffer). Second, if row buffer caches are placed centrally near the IO pads, an enormous amount of overfetch energy and latency is incurred in moving the entire row to the central row buffer cache. If row buffer caches are distributed among arrays, the area and layout of highly optimized arrays is impacted. DRAM chips employ a limited number of metal layers and it is a challenge to introduce a latch or SRAM structure for the row buffer cache directly adjacent to the arrays themselves. Third, row buffer caches are speculative – entries are retained in the hope that future accesses will reuse data in these entries. The Staged Read optimization does not suffer from any of these problems. The registers only store the specific cache line that will be requested in the near future. The registers can be located centrally near the IO pads and be shared by all banks as shown in the physical floorplan depicted in Figure 4a. This is feasible because their overall capacity is small and no additional data (compared to the baseline) is being shipped across global wires to the IO pads. Thus, the prefetch does not impact overfetch energy on a DRAM chip and the only energy penalty is the cost of reading data in and out of Staged Read registers. The proposal also does not impact the layout of the dense array structures, which represent the bulk of the memory chip area footprint. The IO pad area of a DRAM chip occupies a central strip on the DRAM chip. It contains an IO gating structure that is shared by all banks and by reads and writes. In order to promote read-write parallelism with Staged Reads, the Staged Read registers must be placed directly before the on-chip global wires for data reach the IO gating structure. It is well-known that changes to a DRAM chip must be extremely cost sensitive.



(a) Baseline behavior for reads and writes.



(b) Staged Read behavior for reads and writes.

Figure 3. Timing for reads and writes with and without the Staged Read optimization.

Vogelsang [42] points out that changes to a DRAM chip are most costly when instituted in the bitline sense-amplifier stripe, followed by in the local wordline driver stripe, then in the column logic, and finally in the row logic and center stripe. We are therefore limiting our modifications to the least invasive portion of the DRAM chip.

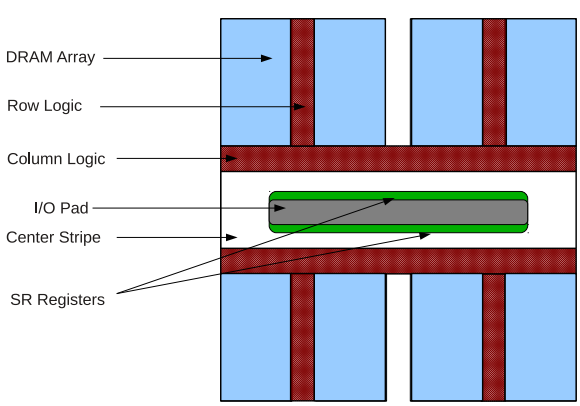
Area Overhead. At a 32 nm process, a 256 byte register file (corresponding to 32 staged read registers) requires 1000 sq. microns [25]. Most of the overhead can be attributed to a new channel that must be implemented between each bank and the staged read register pool as shown in Figure 4b. DDR3 has a burst length of eight, meaning that each bank within an x8 device will be sending 64 bits of data to the I/O pads. As DRAM core frequency is less than off-chip DDR bus frequency, to sustain high bandwidth, all the 64 bits are sent from a bank to the I/O pads in parallel. This data is then serialized and sent 8 bits at a time through the DDR bus. For a wire pitch of 2.5F (where F is the feature size), a channel with 64 wires will have a pitch of 5.1 microns. Even after considering the overhead of eight such channels, for eight banks connected to the staged register through a mux, the net area overhead is approximately less than 0.25% in a 50 sq. mm DRAM device. Note that DRAM layout is heavily optimized for area, and the actual overhead can deviate slightly from the above based on how transistors are laid and wire pitch employed for buses and staged registers.

Effect on Regular Reads. With our proposed implementation, after a row is activated and brought into the row-buffer and a cache-line is read from it, it has to choose one of two paths, i.e., either the regular bus to the I/O pins or the bus that feeds into the Staged Read registers. As shown in Figure 4b, this is accomplished by a simple de-multiplexer to choose between one of the two paths which introduces a 1FO4 gate delay to every read (regular and staged). However this is less than 1% of the DRAM read latency and hence has negligible impact on the performance of non-staged reads.

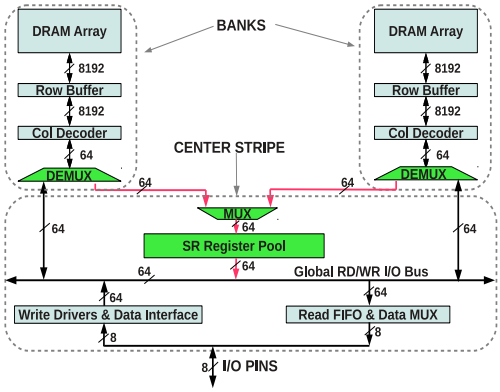
New Memory Commands. The results in a Staged Read register must be accessed with a new low latency instruction. In addition to the conventional CAS instruction, we now have CAS-SR and SR-Read instructions. For Staged Reads, the CAS-SR brings a specified cache line to a specified Staged Read register. The SR-Read moves the contents of the specified Staged Read register to the processor. Assuming a common pool of Staged Read registers that are shared by all banks, both new instructions must specify a few bits to identify the Staged-Read register being handled. The memory controller must track in-progress reads and their corresponding Staged Read registers. The address/command bus is never oversubscribed because each cmd/address transfer is a single cycle operation compared to a 4 cycle data burst and we observe that the address bus has an average utilization of 15% in the baseline. In the baseline, a CAS command is accompanied by a single address transfer (column address). For staged reads, this is replaced by a CAS-SR command and two address transfers (column address and destination SR register-id) and a SR-Read command accompanied by a source SR register-id at the end of the WQ drain. The increased activity on the address/command bus while performing Staged Reads pushes the utilization up to 24%.

Implementability. A sign that such a proposal is implementable is the fact that some high-performance DRAM chips have introduced buffering at the IO pads [5, 20]. Since the IO gating structure is shared by reads and writes and since reads can begin only after the last write has moved past the IO gating structure, Rambus Direct RDRAM devices introduced a write buffer at the IO pads so that the incoming data could be quickly buffered and the IO gating structure can be relinquished sooner for use by reads [20]. Our optimization is similar in structure, but the logical behavior is very different. In Rambus devices, the buffering is happening for writes on their way in so they can get out of the way of important reads. In our Staged Read optimization, buffering is happening for reads on their way out so they can get out of the way of an ongoing write burst.

Targeting Niche Markets. Given that commodity DRAM chips are highly cost-constrained, there is a possibility that such innovations, in spite of their minimal cost impact, may be rejected for the high-volume commodity market. However, there are several DRAM memory products that are produced for niche markets where either performance or energy is given a higher priority than cost. Such memory products may either be used in supercomputer or dat-



(a) Floorplan Of DRAM Chip.



(b) Logical Organization (SR overhead circuitry is shown in red and greeen).

Figure 4. Organization of Staged Read Registers

acenter settings, or in the mobile market. Samsung's LP-DRAM [8, 10] is an example of a low-power chip and Micron's RLDRAM [6] is an example of a high-performance memory chip. It is expected that the marketplace for such niche DRAM products might grow as the memory hierarchy starts to incorporate multiple memory technologies (DRAM, eDRAM, PCM, STT-RAM, Memristors, etc.). In such hybrid memory hierarchies, the focus on cost may shift to the PCM sub-system, while the DRAM sub-system may be expected to provide low latency with innovations such as Staged Reads. Recent papers [2, 40, 44] also advocate the use of a 3D stack of memory chips and a logic die. The interface die can be used for many auxiliary activities such as scheduling, refresh, wear leveling, interface with photonics, row buffer caching, etc. If such a design approach becomes popular, Staged Read registers could be placed on the logic die, thus further minimizing their impact on commodity DRAM chip layouts.

3.3 Exploiting Staged Reads - Memory Scheduler

From the description in Section 3.2, we see that for staged reads to be beneficial, there have to be enough opportunities for the controller to schedule staged reads to idle banks. The opportunity is high if there are some banks that are not targeted by the current write stream and those same banks are targeted by the current pending reads. While we see in Section 4 that such opportunities already exist in varying amounts for different benchmarks, we devise a memory scheduler policy that actively creates such bank imbalance. This best ensures that useful read work is performed during every write drain phase.

The write scheduler first orders all banks based on the simple metric: pending writes - pending reads. Banks are picked in order from this list to construct a set of writes that, once drained, will help the write queue reach its low water mark. Thus, we are draining writes to banks that have many pending writes; banks that have many pending reads are being kept idle. With the above scheduling policy, referred to as the *Write Imbalance (WIMB)* scheduler, during every write drain phase, a bank will roughly alternate between primarily handling writes or primarily handling Staged Read operations. For example, in a 4-bank system, in one write drain phase, a number of writes may be sent to banks 0 and 1, while banks 2 and 3 are busy handling staged reads. In the next write drain phase, banks 2 and 3 are favored for write drains (because their pending write queue has grown), while banks 0 and 1 may handle staged reads.

4 Evaluation

4.1 Results

In this section, we analyze the performance impact of our innovation and also present a sensitivity analysis of staged reads. We present results for the following different configurations.

- **Baseline :** These experiments model the baseline DRAM pipeline and memory controller described in Section 2.2 (Table 1) . The memory controller has a 48 element write queue (for each channel) which is drained once it reaches a high water mark of 32, until the occupancy drops to 16. In the baseline model, there is no provision for staged reads, which means that following a column-read command, the data is read out from the sense amplifiers and sent out over the I/O pins.
- **SR_X :** These configurations refer to systems that consist of DRAM chips and controllers that are, at a maximum, equipped to handle **X** staged read requests per rank. The timing specifications for staged reads are as described in Section 3. We consider the following values of X : 16, 32, and infinite.
- **SR_32+WIMB :** This refers to the configuration where the memory controller's write-scheduling policy is modified to direct writes at a small subset of banks in a rank. This exposes more free banks that can be used by the staged read mechanism.
- **Ideal :** As described earlier in Section 2.3, we also show a bar for the Ideal case as a reference. The ideal case assumes that all pending reads can be prefetched into an infinite set of Staged Read registers while they wait for a write drain cycle to finish.

The key difference between SR_Inf and Ideal is that SR_Inf continues to faithfully model bank conflicts and other timing constraints. So some pending reads in SR_Inf

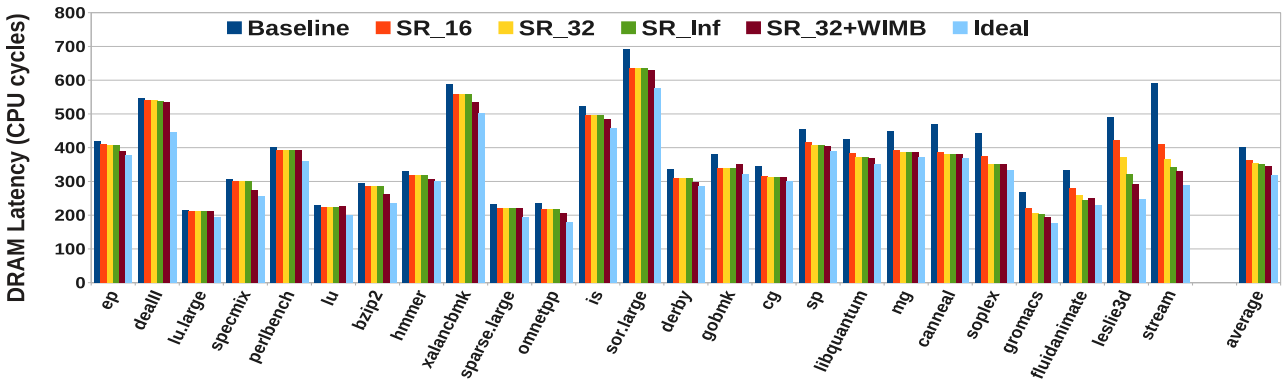


Figure 5. DRAM Latency Impact of Staged Reads

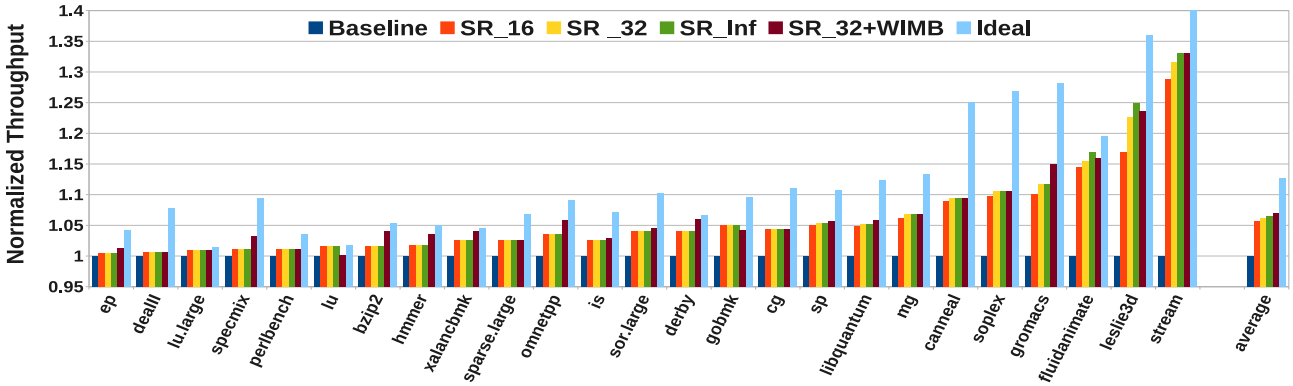


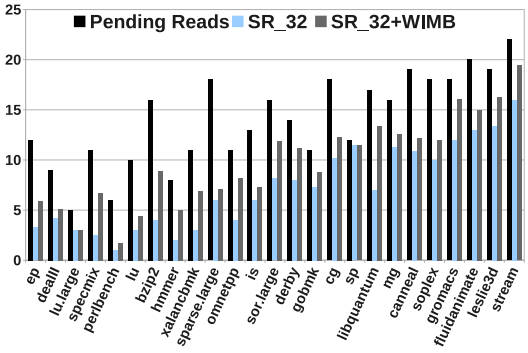
Figure 6. Performance Impact of Staged Reads

may not have the opportunity to issue their prefetch before the write drain is complete.

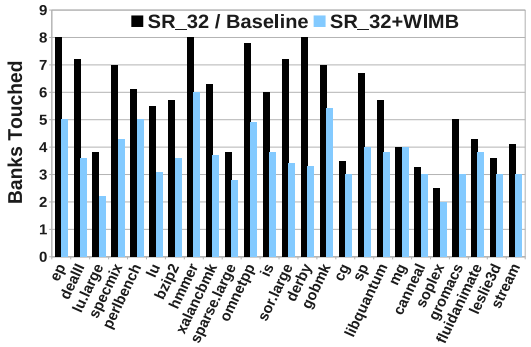
Figure 5 shows the impact of staged reads on average DRAM latency. Figure 6 shows the impact on normalized weighted throughput. The benchmarks are ordered from left to right based on the throughput improvement caused by the SR_32 configuration. Some applications, such as ep, dealII, and lu.large do not exhibit much improvement with staged reads, even with an infinite register pool - whereas applications such as stream, leslie3d, fluidanimate, show marked improvements. The sensitivity of applications to our innovation is dependent on whether during a write drain cycle, there are enough reads that can be parallelized using staged reads and whether these reads would have introduced data bus bubbles in the baseline because of bank conflicts. To help understand the performance characteristics of the different configurations, we plot the average number of reads stalled during write queue drain cycles and the number of staged reads completed with 32 staged-read registers in Figure 7a. In Figure 7b, we plot the average number of banks that are engaged by writes during write queue drains.

The best indicator for Ideal performance is the number of pending reads during each write induced stall period. The performance of the Ideal configuration is high in all cases where there are a large number of pending reads (the first series in Figure 7a). In a practical setting, however, it might not be possible for the staged read mechanism to drain all these pending reads. There might not be enough bank imbalance between reads and writes for the staged reads to schedule read prefetches. Thus, an application like sor.large shows marked improvement with Ideal configuration (Figure 6), because it has a high number of pending reads (Figure 7a). In reality, these reads can not be drained by staged reads as the writes in *sor.large* also touch a large number of banks (Figure 7b), thereby reducing the opportunity to carry out staged reads.

Applications that have a high read bandwidth demand would naturally see many reads queuing up during write drain cycles - but the MPKI alone can not explain the response of an application to staged reads. The improvements obtained are influenced by bank imbalance between writes and reads. For the best performing applications with the SR_32 scheme, such as *stream, leslie3d,* and *fluidanimate*, there are relatively larger number of queued reads during each write-queue drain cycle (Figure 7a). A large fraction of these can be drained by staged reads, because few banks are touched by the writes during write queue drains in these applications (Figure 7b), leaving other banks ready to service staged reads. Applications that have a favorable combination of large number of pending reads and small number of banks touched by writes benefit the most from regular staged reads with the baseline scheduler. On the other hand, applications such as *lu.large* and *perlbench* have very few outstanding reads during write drains, while applications like *omnetpp* and *is* have the writes spread evenly over a large number of banks - leading to lower benefits.



(a) Average Number of Reads Stalled By Write Drains and Number of Staged Reads Completed Per Rank



(b) Average Number of Banks Touched By Writes Per Drain Cycle Per Rank

Figure 7. Statistics to help explain improvements with Staged Read optimization

Increasing the number of staged-read registers improves the latency for some benchmarks (Figure 6a). For example, by going from SR_16 to SR_32, the average latency of accesses drops by approximately 8% for applications like *stream* and *leslie3d*. For most other applications, 16 registers are enough to handle all pending reads, an observation verified by Figure 7a. For the benchmarks we evaluated, SR_32 performs as well as SR_Inf on all occassions except for *stream*, *leslie3d*, and *fluidanimate*, where during some drain cycles, more than 32 SR registers can be useful. But this is not a common occurrence.

On average, the best performing scheme is the SR_32+WIMB configuration that yields a 7% improvement in throughput. By actively creating idle periods for some banks that have a lot of pending reads, this configuration offers a good opportunity for these reads to be completed using staged reads. As seen in Figure 7b, the average number of banks touched by writes decreases due to the biased write-scheduling policy for almost all cases. For applications like *gromacs, derby, omnetpp* the bank imbalance is increased favorably and this is reflected in the additional benefit obtained by SR_32+WIMB over regular SR_32 - more staged reads are completed for these applications (Figure 7a) with the novel write-scheduling policy. Applications like *cg* and *mg* that already touched a small number of banks do not derive any additional benefits over SR_32 with the modified write-scheduling policy. There are also applications such as *gobmk* and *lu* where the improvement with SR_32 and a regular write-scheduling policy is greater than the SR_32+WIMB policy. This can be explained by the fact that in these benchmarks, the explicitly reduced bank footprint of writes leads to a lengthening of the average write-queue drain cycle - which diminishes the benefits of staged reads. However, in none of the cases do we see any performance degradation compared to the baseline.

The gap between the Ideal configuration and SR_Inf in Figure 6 cannot be bridged by parallelizing reads and writes. This results from reads waiting on the same banks as targeted by the writes - a problem that can not be alleviated with regular staged reads, but which is abstracted away in the Ideal configuration. By using the SR_32+WIMB configuration, this is ameliorated to a certain extent for most applications.

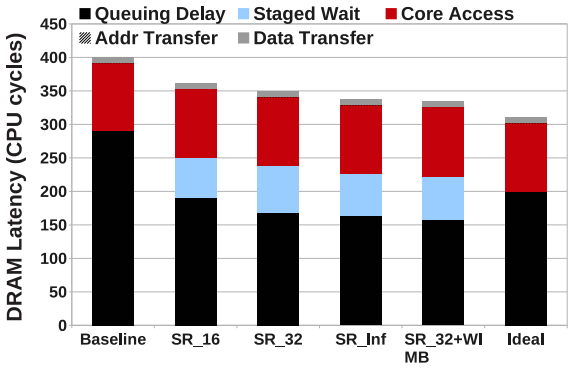
Overall, we witness a 17% reduction in average DRAM latency across the benchmark suite resulting in a 6.2% improvement in overall system throughput with 32 staged read registers - this grows to 7% with the modified write-scheduling policy. Half of the simulated benchmarks yield an improvement higher than 3% and for these write-intensive benchmarks, the average throughput improvement is 11% with SR_32.

Figure 8 sheds further light into the DRAM latency impact of staged-reads. Figure 8a shows the DRAM latency breakdown for DRAM read requests. We see that read requests waiting for a write queue drain to finish have very long queuing delays in the baseline which are brought down substantially by using staged reads (Figure 8a) leading to lowering of overall DRAM latency. However, even after a read goes through the staged read phase, it has to wait in the staged-read register for some amount of time before it can be sent out over the bus, which we refer to as the staged-wait latency. Recall that when a read request goes into the staged-read register, it has already finished its bank activity and the next request to the bank can start immediately. By reducing the bank-wait for pending read requests, the bus utilization after the write-queue drain is increased.

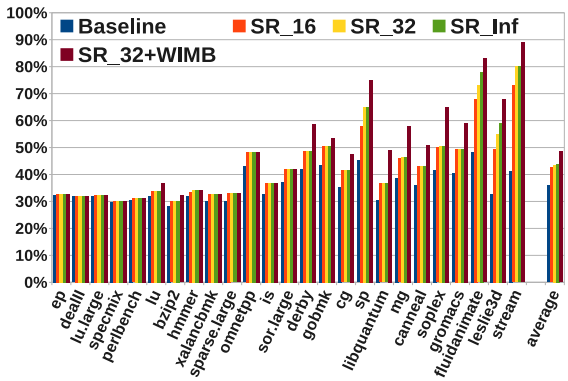
Figure 8b shows the bus utilization in the period following the write queue drain till all the reads that had arrived before the end of the write-queue drain are completed. With staged reads, the utilization in this period increases by as much as 35% for STREAM and about 22% on average. Applications that demonstrate the maximum bus utilization in this period with staged-reads also derive the maximum benefit. We don't show the bus utilization in this window achieved by the Ideal configuration because it is 100% for all applications by design.

4.2 Sensitivity Analysis

To estimate the extent of the influence of our choice of DRAM parameters on the performance of staged reads, we tested the following factors that can potentially impact the efficacy of staged reads - write queue high and low water marks and a higher number of banks.



(a) DRAM Latency Breakdown For All Read Requests



(b)Bus Utilization Immediately After WQ Drain

Figure 8. Analysis of Staged Reads

4.2.1 Write Queue Parameters

The choice of the high and low water marks for the write queue will determine the duration and frequency of write drain cycles. This in turn determines for how long (and how many) reads are stalled due to the write queue drain and also how frequently this disruption occurs, respectively. It also determines if enough bank imbalance is observed, both during and after the write drain.

We ran simulations with various values of the write queue drain parameters. With large values of the high water mark, draining of writes can be delayed - potentially reducing the adverse impacts of writes. In such a case, it is also important to drain the write queue by a large amount each time, because otherwise the gap between write-queue drains will not be reduced. We present results for two different configurations with high water marks of 16 and 128 and low water marks of 8 and 64, respectively.

However, it is important to note that a baseline system (not capable of staged reads) which employs high/low water marks of 32/16 performs better on average compared to the 16/8 and 128/64 configurations. We find that by frequently initiating write-queue drains (i.e., a smaller high water mark) , bandwidth hungry applications get penalized so that the IPC drops by about 2% on average. Again, by using a large value for the high water mark, we risk stalling some critical reads at the head of the out-of-order core's reorder buffer for a long duration. Thus with a high value for the high water mark, some applications perform better than the baseline (i.e., high/low water mark 32/16) and some applications perform worse, leading to a 1.4% performance degradation on average. Therefore, for the workloads we simulated, the best performing write queue configuration is the one used for the baseline.

We see that in both the 16/8 and 128/64 cases, staged reads can offer performance improvements as shown in Figure 9. The results show the same trends as before, but the improvements are slightly lower (just under 5% on average) with SR_32 (which has no scheduler induced write-imbalance). With low water marks, there are just not enough pending reads that can be expedited with staged-reads. In fact, even with the write-scheduler creating write-imbalance, there is no extra improvement due to the dearth of writes. On the other hand, with high water-marks, there is less write imbalance with a regular scheduler, which can be alleviated by SR_32+WIMB as it has a much larger pool of writes to choose from. This leads to the SR_32+WIMB creating more oppurtunities for staged reads - finally yielding a 7.2% improvement over a non-staged read baseline.

4.2.2 More Banks

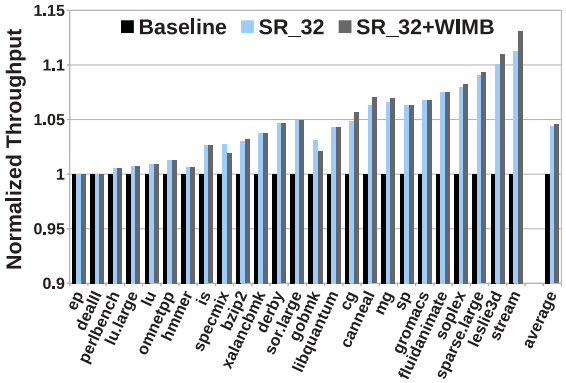
The efficacy of staged-reads increases if the reads pending on a write-drain cycle are directed at banks that do not have many writes going to them. With larger number of banks, the possibility of the memory controller being able to find more opportunities for scheduling staged reads increases. On the other hand, if more banks exist, the baseline suffers from fewer bank conflicts for reads and fewer data bus bubbles following the write drain. So there are competing trends at play with more banks. We carry out simulations where the same 4GB capacity as the baseline system is split into twice the number of banks (i.e., 16 banks/rank). We observe that a regular DRAM system (without staged reads) with more banks performs better than the baseline by about 3.1%. Applying our SR_32 configuration on this improved system yields an average improvement of about 4.3%. Due to increased bank parallelism, the performance benefits with SR_32+WIMB are comparable to regular staged reads. Thus, as a performance optimization for next generation memories, it is more effective to add Staged Read registers than to double the number of banks.

4.3 Projecting for Future Main Memory Trends

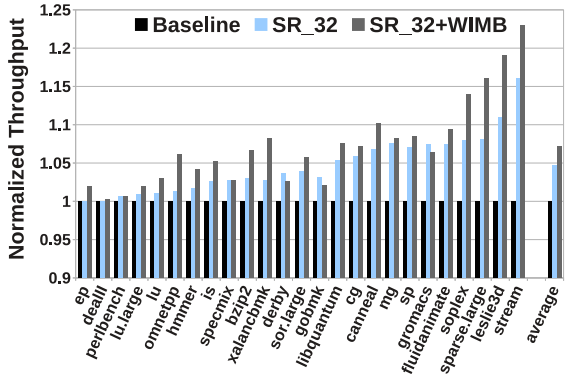
In this section, we evaluate if the Staged Read optimization will be more compelling in future memory systems. We examine a number of processor configurations that might represent these future trends: memory systems with reliability support, non-volatile memories, and fewer channels per core.

4.3.1 Higher Write Traffic

With errors in DRAM becoming a major source of concern [35, 45], DIMMs equipped with error protection measures are being employed in data-centers. In one possible chipkill implementation (to overcome the failure of one DRAM chip on a rank), each DIMM can have a separate



(a) High Water Mark = 16, Low Water Mark = 8



(b) High Water Mark = 128, Low Water Mark = 64

Figure 9. Staged Reads With Different Write Queue Sizes

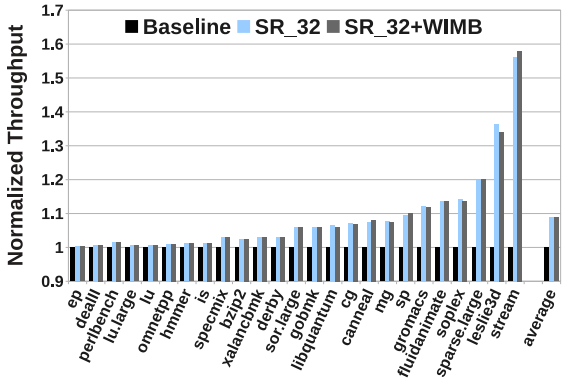


Figure 10. Staged Reads with RAID-5 like Chipkill Protection

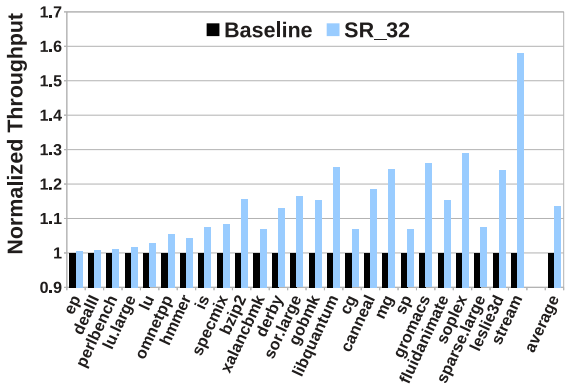


Figure 11. Staged Reads with PCM Main Memory

chip that stores either parity information per byte or an ECC code per 64-bit word. On each read, the information in the extra chip can help with error detection and possibly correction. If the information is not enough to correct the error (as may happen with multi-bit errors), a second-tier protocol is invoked. In RAID-like fashion, parity is also maintained across DIMMs. If a DIMM flags an uncorrectable error, information from all DIMMs is used to reconstruct the lost data. Such RAID-like schemes have been implemented in real systems [1] and will likely be used more often in the future as error rates increase near the limits of scaling. As is seen in any RAID-5 system, every write to a cache line now requires us to read two cache lines and write two cache lines. This causes a significant increase in write traffic.

We simulate a RAID-5 like system where a fifth DIMM stores the parity information for the other 4 DIMMs. The baseline system in such a scenario encounters double the write traffic as seen in a non-ECC scenario. This, coupled with the increased number of reads makes staged reads more compelling. We see that the throughput increases by an average of 9% (Figure 10) by using 32 staged-read registers. Compared to a non-reliable baseline, we see a shorter gap between write-queue drain cycles which improves the benefits of Staged Reads. Using SR_32+WIMB, the average throughput does not increase beyond what is provided by SR_32 - since a data write and it's corresponding ECC code write have to be completed together, the write-scheduler is not able to re-order the writes to expose any additional staged read opportunities.

4.3.2 PCM

Recent work [22, 31, 32] advocates the use of PCM as a viable main memory replacement. Similarly, other NVM technologies with read and write latencies longer than those of DRAM are also being considered [39]. We assume that the PCM main memory is preceded by a 16 MB L3 eDRAM cache (L3 average latency of 200 cycles). PCM chip timing parameters are summarized in Table 2; the PCM read and write latencies are approximately 2X and 4X higher than the corresponding DRAM latencies due to the high values of the row-activation (tRCD) and write-recovery (tWR) timing parameters.

The higher read and write latencies make the Staged Read optimization more compelling. We observe an average 26% reduction in memory latency because of a sharp drop in queuing delay. This translates to an average 12% throughput improvement (Figure 11), with the Stream benchmark showing a 58% improvement. On the other hand, SR_32+WIMB does not offer as much advantage as regular SR_32 - the performance improvement is 9.5% on average. The performance drop (compared to SR_32) is due to the high penalty of lengthening the write-queue drain cycle-time in PCM. By restricting bank-parallelism

within writes, SR_32+WIMB ends up with bank conflicts on the targeted banks.

4.3.3 Number of Channels

With a greater number of channels, the pressure of pending reads on each channel would reduce and the benefits of Staged Reads would also diminish. With a quad-channel configuration, for our workload suite, the benefits of SR_32 is 3.3% and that of SR_32+WIMB is about 3.0%. However, the ITRS [19] projects an increase in the number of cores, but no increase in the number of off-chip pins. Hence, we expect that the number of channels per core will actually decrease in the future. If we instead assume that 16 cores share a single channel, the improvement with SR_32 jumps up to 8.4% because of the greater role played by queuing delays, while introducing write imbalance improves throughput by 9.2%.

5 Related Work

A large body of work has looked at techniques to improve the latency of DRAM reads and the maximum achieveable bandwidth of DRAM systems. Researchers have proposed techniques to mitigate long DRAM latencies through intelligent data placement [38, 46] and optimized scheduling [21, 28, 29].

An example of additional buffering on DRAM chips is the use of row buffer caches. Row buffer caches primarily help by increasing row buffer hit rates. Hikada et al. [15] propose a DRAM architecture with an on-chip SRAM cache, called Cache-DRAM, that maintains recently opened row-buffers to decrease access latency (by eliminating precharging time) and increasing bandwidth utilization. Hsu et al. [17] evaluate the trade-offs between caching one entire row-buffer in SRAM and caching multiple regular data cache lines in a set-associative cache. They also propose the use of address interleaving schemes to spread accesses over multiple row buffers to improve bank parallelism. Zhang et al. [47] propose another variation of cached DRAM where the on-memory cache contains multiple large cache lines buffering multiple rows of the memory array. However, all these techniques suffer from the problems described in Section 3.2 and are not specifically targeted at optimizing write behavior.

The impact of better write queue management on DRAM system performance was articulated in a recent paper from Stuecheli et al. [37]. The authors propose a coordinated cache and memory management policy that exposes the contents of cache-lines in the LLC to the memory controller and hence provides it with more opportunities to find writes that can cause row buffer hits. The large (virtual) write queue also provides opportunities for long write bursts to improve bus utilization. Staged Reads and VWQ are orthogonal for the most part: VWQ tries to reduce the time spent in writes and Staged Reads try to hide write imbalance and read imbalance. So the two techniques could be combined to further alleviate the write bottleneck. Lee et al. [23] proposed a mechanism to reduce write interefernce that is similar to VWQ in which dirty lines from the LLC are selected for pre-emptive eviction if they are to the same row in the DRAM bank.

For PCM systems, Qureshi et al. describe write-cancellation and write-pausing to prevent reads from being stalled by iterative writes [30]. But these writes have to eventually happen and such re-tries may increase the overall bank occupancy of writes.

Lee et al. describe eager write-back [16], in which dirty lines are pre-emptively sent to the DRAM from the LLC during bus idle cycles. This reduces bandwidth contention between reads and writes that arise typically during cache replacement. Natarajan et al. [27] describe several write scheduling policies that try to schedule writes opportunistically when the read activity is low. Borkenhagen and Vanderpool [14] suggest techniques to predict the arrival of a read, to stifle the write queue drain. Shao et al. [36] describe BASR where the goal is to hide the write latency by either preempting writes with reads or by piggybacking writes to some row that has been opened by an ongoing read. BASR does nothing to improve the parallelism between reads and writes, which is the central focus of our paper. Further, piggybacking opportunities will be limited in future many-core processors because of reduced locality. In fact all the above techniques will be less applicable in future multi-cores. Activity on a memory bus is expected to be less bursty when the bus is shared by multiple programs. Hence, future multi-cores will likely see fewer bus idle cycles and fewer opportunities to drain writes without posing any interference.

6 Conclusions

We show that write handling in modern DRAM main memory systems can account for a large portion of overall execution time. This bottleneck will grow in future memory systems, especially with more cores, chipkill support or NVM main memories. This requires that mechanisms be developed to boost read-write parallelism. We show that the Staged Read optimization is effective at breaking up a traditional read into two stages, one of which can be safely overlapped with writes. We show average improvements of 7% in throughput for modern memory systems (accompanying a 17% reduction in DRAM access latency) and up to 12% for future systems. The proposed implementation has been designed to cater to the cost sensitivity of DRAM chips. We introduce less than a kilo-byte of buffering near the IO pads, similar to structures that have been employed for other functionalities in some prior high performance DRAM chips. We therefore believe that the Staged Read optimization is worth considering for DRAM chips designed for the high performance segment.

7 Acknowledgements

We thank the reviewers (especially our shepherd, Hillery Hunter) and members of the Utah Arch group for their suggestions to improve this work.

References

[1] HP ProLiant Server Memory. http://h18000.www1.hp.com/products/servers/technology/memoryprotection.html.

[2] Hybrid Memory Cube, Micron Technologies. http://www.micron.com/innovations/hmc.html.

[3] STREAM - Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[4] Wind River Simics Full System Simulator. http://www.windriver.com/products/simics/.

[5] 64M-bit Virtual Channel SDRAM Data Sheet. Technical report, NEC, 2003.

[6] Exploring the RLDRAM II Feature Set. Technical Report TN-49-02, Micron Technologies Inc., 2004.

[7] Micron DDR3 SDRAM Part MT41J128M8 . http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf, 2006.

[8] Low-Power Versus Standard DDR SDRAM. Technical Report TN-46-15, Micron Technologies Inc., 2007.

[9] Java Virtual Machine Benchmark, 2008. Available at http://www.spec.org/jvm2008/.

[10] Mobile DRAM Power-Saving Features and Power Calculations. Technical Report TN-46-12, Micron Technologies Inc., 2009.

[11] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *Proceedings of PACT*, 2010.

[12] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[13] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of IISWC*, 2008.

[14] J. Borkenhagen and B. Vanderpool. Read Prediction Algorithm to Provide Low Latency Reads with SDRAM Cache, 2004. United States Patent Application, Number US 2004/6801982 A1 .

[15] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima. The Cache DRAM architecture: A DRAM with an On-chip Cache Memory. *Micro, IEEE*, 10(2), 1990.

[16] H. hsin Lee and G. Tyson. Eager writeback - a technique for improving bandwidth utilization. In *In Proceedings of MICRO*, 2000.

[17] W.-C. Hsu and J. E. Smith. Performance of cached dram organizations in vector supercomputers. In *Proceedings of ISCA*, 1993.

[18] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of ISCA*, 2008.

[19] ITRS. International Technology Roadmap for Semiconductors, 2009 Edition.

[20] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.

[21] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of MICRO*, 2010.

[22] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of ISCA*, 2009.

[23] C. Lee, V. Narasimhan, E. Ebrahimi, O. Mutlu, and Y. Patt. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. Technical report, High Performance Systems Group, University of Texas at Austin, 2010.

[24] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[25] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of MICRO*, 2007.

[26] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of MICRO*, 2007.

[27] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-Processor Environment. In *Proceedings of WMPI*, 2004.

[28] K. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings HPCA*, 2004.

[29] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling - Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of ISCA*, 2008.

[30] M. Qureshi, M. Franceschini, and L. Lastras. Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing. In *Proceedings of HPCA*, 2010.

[31] M. Qureshi, M. Franceschini, L. Lastras-Montano, and J. Karidis. Morphable Memory System: A Robust Architecture for Exploiting Multi-Level Phase Change Memory. In *Proceedings of ISCA*, 2010.

[32] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *Proceedings of MICRO*, 2009.

[33] M. Qureshi, V. Srinivasan, and J. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of ISCA*, 2009.

[34] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of MICRO*, 2004.

[35] B. Schroeder et al. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of SIGMETRICS*, 2009.

[36] J. Shao and B. Davis. A Burst Scheduling Access Reordering Mechanism. In *Proceedings of HPCA*, 2007.

[37] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John. The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In *Proceedings of ISCA*, 2010.

[38] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *Proceedings of ASPLOS-XV*, 2010.

[39] F. Tabrizi. Non-volatile STT-RAM: A True Universal Memory. http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2009/20090813_ThursPlenary_Tabrizi.pdf.

[40] A. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. Combining Memory and a Controller with Photonics through 3D-Stacking to Enable Scalable and Energy-Efficient Systems. In *Proceedings of ISCA*, 2011.

[41] A. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *Proceedings of ISCA*, 2010.

[42] T. Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *Proceedings of MICRO*, 2010.

[43] D. Wang et al. DRAMsim: A Memory-System Simulator. In *SIGARCH Computer Architecture News*, September 2005.

[44] D. H. Woo et al. An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth. In *Proceedings of HPCA*, 2010.

[45] D. Yoon and M. Erez. Virtualized and Flexible ECC for Main Memory. In *Proceedings of ASPLOS*, 2010.

[46] Z. Zhang, Z. Zhu, and X. Zhand. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *Proceedings of MICRO*, 2000.

[47] Z. Zhang, Z. Zhu, and X. Zhang. Cached DRAM for ILP Processor Memory Access Latency Reduction. *Micro, IEEE*, 21(4), 2001.