Split Write DRAM: Reducing DRAM Access Latency by Mitigating Read-Write Interference

K Chitra
Indian Institute of Technology
Guwahati, India
k.chitra@iitg.ac.in

Aryabartta Sahu
Indian Institute of Technology
Guwahati, India
asahu@iitg.ac.in

Arjun Dey
Nutanix Inc.
Bengaluru, India
arjundey01@gmail.com

Minesh Patel
Rutgers University
New Jersey, USA
minesh.patelh@gmail.com

Abstract

Modern memory controllers use a write queue to optimistically defer DRAM write operations until a rank is idle to avoid disrupting latency-critical reads. Unfortunately, when the write queue reaches capacity without adequate idle time available, all pending read operations suffer a severe performance penalty while waiting for the memory controller to perform a mandatory bus turnaround and drain the outstanding writes.

To mitigate the performance impact of write draining, we introduce *Split Writes*, which divide write operations into two phases: Data Transfer and Row Access. Split Writes enable the memory controller to first transfer the write data to a small, fast buffer—called the *Split Write Cache (SWC)*—located within the DRAM chip during write draining. Data Transfer avoids row activation or precharge, significantly reducing the latency of write draining. Later, the memory controller can exploit bank idle time, which is more common than rank idle time, to opportunistically write data back from the split write caches to DRAM rows.

Unlike prior techniques that delay writes or modify cache behaviour, Split Writes enable timely write draining while minimizing read disruption. Furthermore, Split Writes preserve protocol correctness, requiring modest architectural changes while improving system responsiveness under high write pressure.

CCS Concepts

• **Hardware → Memory and dense storage**; • **Computer systems organization** → *Processors and memory architectures.*

Keywords

Memory, DRAM, Memory controller, DRAM Controller, Bank conflict, Row Buffer conflict, Row Buffer Policies

1 Introduction

Memory access latency is a key system design parameter that determines performance for a wide range of modern workloads. Memory controllers in DRAM-based systems heavily optimize for latency-critical read operations using techniques such as caching, prefetching, and memory access scheduling. However, these optimizations are fundamentally limited by contention between read and write operations that share the same DRAM bus due to packaging and electrical constraints. The bus can only send data in one direction at a time, so any transition between read and write operations incurs additional latency (typically 10–12 ns [11]) known as the *bus turnaround time*. This causes priority inversion where write operations necessarily block latency-critical reads regardless of bank availability, ultimately degrading application performance and reducing memory bandwidth utilization.

To amortize bus turnaround overhead, the memory controller typically serves reads and writes in bursts by buffering them in read and write queues, respectively. Read operations are prioritized while writes are deferred until either the DRAM rank is idle or the write queue fills beyond a threshold value (e.g., 80%) known as the high watermark. During rank idle time, the memory controller may preform a *minor write drain* to opportunistically write back part of the write queue. However, any read request that arrives during a minor write drain suffers a latency penalty because the memory controller must first interrupt the write drain or allow it to complete and then perform a bus turnaround before the read may be served. In performance-sensitive settings, a memory controller may prefer to avoid minor write drains altogether to avoid inadvertently delaying latency-critical reads [12]. In contrast, if the high watermark is reached, the memory controller performs a *major write drain* to flush the majority of the write queue down to a lower threshold value (e.g., 20%) known as the low watermark. A major write drain severely impacts performance because its long latency starves all pending read operations, regardless of their origin or criticality.

Unfortunately, a combination of higher core counts, larger write queues, and increasing memory demands in modern systems exacerbates the impact that major write drains have on overall system performance. Therefore our goal in this work is to alleviate the performance impact of major write drains with low hardware cost. Existing techniques such as increasing the write queue size [4], leveraging the LLC as a temporary write buffer [8], or modifying cache eviction policies [5, 6], often introduce additional hardware complexity or degrade cache efficiency and still fail to fully eliminate read-write contention during periods of write buffer saturation.

Our key idea in this work is to *fundamentally reduce the latency of write operations* so that write drains complete quickly and efficiently, reducing the latency impact to performance-critical reads. Our work is based on the insight that DRAM write access latency is dominated by row access operations rather than data bus transfers. By decoupling the two, the memory controller can take row operations off the critical path of major write drains.

2 Background

Each DRAM bank contains two-dimensional arrays of DRAM *cells* organized as *rows* and *columns [2]. Wordlines* are used to activate a chosen row, and *bitlines* are used to access the data from a column within that row. Each bank is internally divided into *subarrays* (groups of rows), and each subarray has a row-width collection of sense amplifiers (logically known as a subarray's *local row buffer*) that detect and cache the data values stored within an active row's cells. The local row buffer serves all accesses to the active row, but only one row buffer may operate at a time with a given bank. Therefore, consecutive accesses to different rows with in the same bank must be serialized.

To serve a memory request that accesses data at a specific row and column address, the *memory controller* issues the following commands to a bank:

i) ACTIVATE: The controller issues an ACTIVATE command along with the row address to open the row. This connects the memory cells in the specified row with their respective bitlines, causing a voltage perturbation on each bitline that the local sense amplifiers sense and amplify to a CMOS-readable value.

ii) READ/WRITE: The controller then issues a READ or WRITE command along with the column address to access the designated column. The specified column from the local sense amplifiers is selected via *selection lines* by asserting the *CSEL signals* set by the *column decoder*, connecting the corresponding bitlines to the *I/O lines*. For a read operation, a column-width number of *global sense amplifiers* read the selected columns onto the I/O lines, forwarding the data to the memory chip's output pins. For a write operation, the data is written to the local row buffer via I/O lines driven by a circuit called the *write driver*, which overwrites the selected columns of DRAM cells via the bitlines. The DRAM data bus is bidirectional, which means that the DRAM chip drives the bus during read operations and the memory controller drives it during write operations. The read-to-write and write-to-read turnaround times (tRTW and tWTR, respectively) are key timing parameters that specify the minimum delay needed when switching the data bus direction between read and write operations.

iii) PRECHARGE: Before activating a new row in a bank, the controller must first reset the bitline voltage to a specific voltage by issuing a PRECHARGE command [3].[1]

3 Related Work

Significant work addresses the performance degradation in DRAM systems due to the contention between reads and writes over the shared memory bus. Prior work mitigates this bottleneck through various hardware modifications and memory access scheduling strategies. However, these approaches either introduce complexity, degrade read performance, or fail to eliminate read-write contention under heavy write pressure.

Virtual Write Queue[4] extends the last-level cache (LLC) to act as a virtual write buffer, allowing delayed writebacks to DRAM. Although it helps reduce write-induced blocking, this approach risks LLC pollution, increasing read miss rates and harming read performance—especially under mixed or read-heavy workloads. Rank-aware write buffering [6] redirects writes to less-active ranks to create idle windows in other ranks, primarily for power savings. However, it does not target read latency or queuing delay directly, and relies on complex coordination between cache replacement and rank state tracking. It also becomes less effective when memory traffic is balanced across ranks.

Adaptive Burst Writes [9] and Last-Write Prediction [8] attempt to tune writeback behavior dynamically to reduce read interference and write volume, respectively. These techniques are either prediction based or require runtime adaptation and may struggle to respond to sudden workload shifts. Other solutions, like write coalescing and filtering, improve memory bandwidth efficiency but do not directly address the blocking nature of write-backs on reads.

[1] If the requested row is already present in the row buffer, we can directly read or write without an additional PRECHARGE or ACTIVATE. This is called a row buffer hit. Otherwise, it is called a *row buffer miss*.

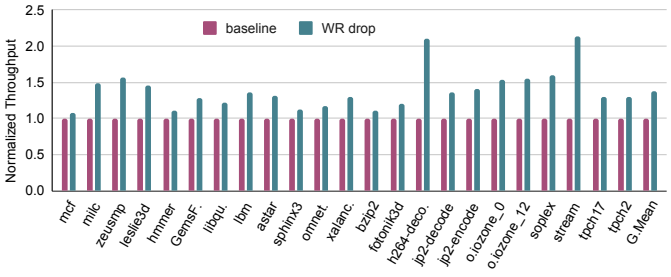

Figure 1: Best-case IPC improvement possible by mitigating read-write interference during write draining

4 Motivation

Our work differs significantly from prior solutions by targeting the DRAM access latency during write drains rather than attempting to delay or overlap writeback timing. By temporarily transferring data into a quickly-accessible buffer, Split Write DRAM shortens the write drain phase and restores read service earlier, improving system responsiveness without altering cache behavior or requiring complex controller prediction logic. This method reduces read-write interference, minimizes turnaround penalties, and maintains higher read throughput without significantly impacting memory system complexity.

We model the benefits that Split Writes can achieve using the key DRAM timing parameters that determine the overall latency of a DRAM write operation, which comprises activating a row, writing the data, and precharging the bank:

- t_{RCD}: Row to Column Delay—the time required to move data from DRAM cells to the sense amplifiers during the row activation process.
- t_{CWD}: Column Write Delay—the delay between issuing a write command and the point at which the memory controller places data onto the data bus.
- t_{CCD}: Column-to-Column Delay—the minimum time required between two column commands to different banks or within the same bank.

- t_{WR}: Write Recovery Time—the minimum time between the end of a write burst and the start of a precharge command; this includes the time required for the write operation to release I/O gating resources.
- t_{RP}: Row Precharge Time—the time required to precharge a DRAM bank, preparing it for another row access.

Assuming a DRAM row hit ratio of α, the latency incurred by draining N writes can be modeled as:

$$N((1-\alpha)(t_{RCD} + t_{CWD} + t_{CCD} + t_{WR} + t_{RP}) + \alpha \cdot t_{CCD}) \quad (1)$$

Split Writes avoid the need for t_{RCD}, t_{WR}, and t_{RP} during write queue draining, enabling the memory controller to save up to $N(1-\alpha)(t_{RCD} + t_{WR} + t_{RP})$ total time. The memory controller can then exploit bank-level parallelism to hide the latency of the Data Write phase of the Split Write operations, thereby vastly reducing the overall latency of a major write drain.

We conduct an oracle study using Ramulator2 [1], an open source cycle accurate DRAM simulator. We model two configurations: i) a baseline, where both major and minor write drains are allowed, and ii) a write drop variant (WR-drop), where writes incur zero latency during major write drains and therefore do not block reads, while minor write draining is still allowed. We use an out-of-order core running at a *4 GHz* clock frequency, with a last level cache (LLC) of 8 MB, 64 B cache line size, and 8-way associativity. The memory subsystem is configured as a *DDR4-8Gb ×8 device with 1 channel, 2 ranks, 4 bank groups, 4 banks, 64K rows, and 1K columns*. The memory controller employs *FRFCFS* scheduling with *64-entry read/write queues*. Benchmarks are drawn from SPEC CPU2006 [13], STREAM, TPC, MemBench [14], which include multimedia and OS workloads. Each benchmark is simulated for one billion instructions and performance is reported as instruction throughput, measured in instructions per cycle (IPC).

We characterize the impact of writes on overall performance. Figure 1 presents normalized IPC results, where the left bar corresponds to the baseline model with both major and minor write drains enabled, and the right bar corresponds to the wite drop (WR drop) model, where writes incur zero latency and impose no constraints on reads during major write drains. This configuration represents an upper bound on performance, showing an average improvement around 37%, indicating that read operations are significantly blocked by writes in the baseline design.

4.1 Split Write DRAM Overview

The central goal of our work is to minimize the latency overhead of write operations so that bulk write drains no longer stall latency-sensitive read requests. Conventional DRAM systems suffer from the fact that completing a write operation requires a full row cycle (activation, data transfer, and precharge), where the row operations—not the data bus transfer—dominate the overall latency. Our key observation is that this tight coupling between data transfer and row access unnecessarily prolongs write drains and severely limits the ability of the memory controller to prioritize reads.

Our implementation of this idea is called Split Writes, which introduces small write caches called *Split Write Caches (SWCs)* within each memory chip to absorb writes quickly when draining writes, deferring the actual DRAM row access to a more opportune time.

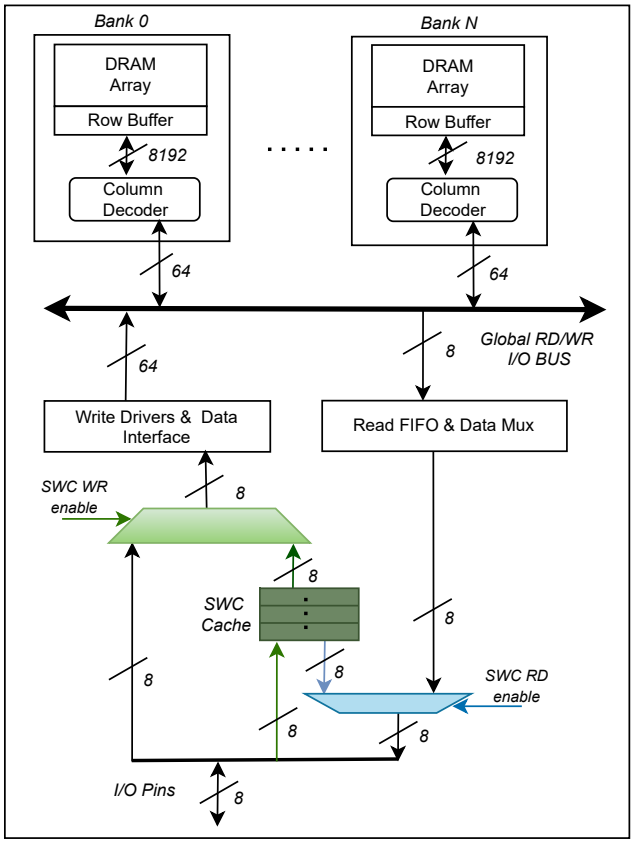

Figure 2: Organization of Split Write Cache for an x8 Split-Write DRAM chip

This enables the data transfer to occur quickly and avoids blocking the rank for the entire duration of a full DRAM write operation. The buffered writes can then be flushed into the DRAM array in the background or during a future idle period of that bank (no need to wait until the rank is idle). As a result, a latency-critical read operation may be served earlier, and if its data is present in an SWC, may be served without requiring a DRAM row access at all. This greatly alleviates the penalty that write draining imposes on pending read operations.

4.2 Split Write DRAM Architecture

Figure 2 illustrates the DRAM architecture augmented with a split write cache. Each DRAM chip incorporates a dedicated cache of 64 entries, with each entry size 64 bits. The cache is positioned in the center stripe region of the DRAM chip, requiring modifications only to I/O circuitry and no changes to the core storage arrays. To visualize the architectural modifications, Figure 2 highlights the updated datapath, including the read path in blue and the write path in green. The proposed architecture is fully compatible with conventional DRAM operations, also supporting the new split write operation. This is achieved by adding a split write cache (SWC) and two multiplexers. The multiplexers are responsible for selecting

between read and write operations that are served by the SWC instead of directly accessing the DRAM bank.

(i) *Write Operations:* The multiplexers determine whether the data written into the DRAM array originates from the SWC or directly from the memory controller. When the SWC write enable signal is asserted, the data is supplied from the SWC; otherwise, it is provided directly by the memory controller.

(ii) *Read Operations:* The second multiplexer controls data retrieval by selecting either the DRAM array or the SWC as the source of the requested data. If the requested data is present in the SWC, an internal SWC read enable signal get asserted, and the data is directly fetched from the cache. Otherwise, the read request is served from the DRAM array.

4.3 Split Write DRAM Commands

The proposed split write architecture extends the baseline DRAM command specification. Conventional DRAM commands such as ACTIVATE, READ, and WRITE remain fully supported to ensure compatibility with existing memory protocols. The newly introduced SWC commands operate either in conjunction with or as alternatives to these baseline operations in order to perform split write operations.

(1) WRITE and *SWC_WRITE*: A conventional DRAM WRITE operation writes the data directly from the memory controller write buffer into the DRAM array via write drivers. In contrast, during a major write drain, an *SWC_WRITE* places the incoming data into the SWC, deferring the update of the DRAM array.

(2) *SWC_FLUSH*: This command combines with a conventional WRITE operation to commit buffered data from the SWC into the DRAM array. To perform the flush, the SW memory controller first activates the required row (row address comes from the SWC table) and then issues the *SWC_FLUSH* command along with the corresponding column address and SWC index.

(3) READ and *SWC_READ*: A conventional READ command retrieves data from the DRAM array following an ACTIVATE command. *SWC_READ* fetches data directly from the SWC when the requested entry is present, thereby bypassing both row activation and READ. If the requested data is not available in the SWC, the SW memory controller must instead perform a conventional READ operation to retrieve the data from the DRAM array.

4.4 Mechanism

The split write DRAM controller maintains a *split write cache table (SWC Table)* that contains the cache index, the ID of the row and column it holds, and a valid/invalid bit. The SWC Table uses a parallel associative lookup (CAM-style) that can be accessed within a single controller clock cycle, can be overlapped with request queue handling. The lookup is small and parallelized, it does not lie on the critical path of timing sensitive DRAM operations.

Write Operation in Split Write DRAM: The memory controller performs a Split Write in two steps:

(1) **Data Transfer:** The controller transfers the write data into a *Split Write Cache (SWC)* within the target DRAM chip without performing any row operations. At this stage, the write is considered "drained" from the memory controller's perspective. This operation is carried out by issuing the *SWC_WRITE* command along with the corresponding SWC index, which is sent through the address pins.

(2) **Data Write:** At an opportune time, the memory controller activates the target DRAM row, instructs the SWC to write the data into the row, and performs a precharge operation after write completion. This process is triggered by issuing the *SWC_FLUSH* command. The memory controller activates the target DRAM row whose identifier is stored in the SWC table. The SWC entry associated with the pending flush is identified using the SWC index. This index, together with the column address, is transmitted through the address pins. An internal SWC write-enable signal configures the write datapath to write the corresponding data from the SWC into the active DRAM row at the specified column address. This operation effectively moves the buffered data from the SWC into its permanent storage location in the array.

On a read request, the SW DRAM controller checks for the entry in the SWC Table. If the data is not present in the SWC, DRAM issues normal READ operations. If data is available in the SWC, the SW DRAM controller take the action as described below.

Read Operation in Split Write DRAM: To serve a memory request when the requested data resides in the split write cache (SWC), the SW DRAM controller issues an *SWC_READ* command along with the corresponding SWC index The SWC index is transmitted through the address pins, as a conventional row address or column address is not required for the *SWC_READ* operation. An internal SWC read-enable signal configures the read data path to read the corresponding data from SWC.

Split writes enable the memory controller to reduce the latency impact that writes impose on reads while preserving access ordering and correctness given that data is eventually committed to the DRAM array. The SWC may be implemented either within the bank circuitry or globally shared across banks based on the designer's goals: shared SWCs improve utilization in the case of nonuniform bank access patterns but require a single larger, slower memory block compared with distributing smaller SWCs across banks.

4.5 Area Overhead

The Split Write DRAM architecture introduces additional hardware overhead in the center stripe region of the DRAM chip, while leaving the core storage array unmodified. The primary source of overhead arises from the split write cache (SWC), where a 64-entry cache with a 64-bit entry size requires 512 B of on-chip storage per DRAM chip. The SWC table in the memory controller incurs an additional overhead of $(Rowaddressbits + Columnaddressbits + Validbit) \times 64 \times$ per rank. For a DDR4–8 Gb × 8 configuration, the SWC table table overhead in the memory controller is $(16 + 10 + 1) \times 64 = 216B$ per rank.

5 Conclusion

We propose split write technique to mitigate the read latency during the write draining. This design ensures that write drains complete with low latency, freeing the controller to resume issuing read requests. The read operations that target recently written data can be satisfied directly from the SWCs, avoiding the cost of activating the DRAM row. Therefore, the split writes give the memory controller greater flexibility to overlap or defer row operations, thereby reducing interference between reads and writes.

References

[1] Luo et al., "Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator," *IEEE CAL*, 2023.
[2] B.Keeth et al., "DRAM Circuit Design," *Wiley-IEEE Press*, 2001.
[3] JEDEC, "JESD79-4C: DDR4 SDRAM Standard", 2020.
[4] Stuecheli et al., "Coordinating DRAM and Last-Level-Cache Policies with the Virtual Write Queue," *IEEE Micro*, 2011.
[5] Mainak et al., "Bandwidth-Aware Last-Level Caching: Efficiently Coordinating Off-Chip Read and Write Bandwidth," *ICCD*, 2019.
[6] Amin et al., "Rank-Aware Cache Replacement and Write Buffering to Improve DRAM Energy Efficiency," *ISLPED*, 2010.
[7] Niladrish et al., "Staged Reads: Mitigating the impact of DRAM writes on DRAM reads," *HPCA*, 2012.
[8] Wang et al., "Improving Writeback Efficiency with Decoupled Last-Write Prediction," *ISCA*, 2012.
[9] Cheng et al., "Adaptive Burst-Writes (ABW): Memory Requests Scheduling to Reduce Write-Induced Interference," *ACM TODES*, 2015.
[10] Wang et al., "Rank Idle Time Prediction Driven Last-Level Cache Writeback," *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2012.
[11] Balasubramonian, Rajeev, "Innovations in the Memory System," *Springer Nature*, 2022.
[12] Lee et al., "DRAM-Aware LLC Writeback:Reducing Write-Caused Interference in MemorySystems," *Carnegie Mellon University. Journal contribution https://doi.org/10.1184/R1/6468701.v1*, 2018.
[13] "SPEC Workloads," *Available at http://www.spec.org/*
[14] "MemBench Workloads," *Available at https://github.com/CMU-SAFARI/MemBen*
[15] Rixner et al., "Memory access scheduling," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.
[16] Ghose et al., "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," *SIGMETRICS*, 2019.