



# Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses

Vivek Seshadri, Thomas Mullins, Amirali Boroumand,  
Onur Mutlu, Phillip B. Gibbons\*, Michael A. Kozuch<sup>†</sup>, Todd C. Mowry

Carnegie Mellon University <sup>†</sup>Intel Labs

## Abstract

Many data structures (e.g., matrices) are typically accessed with multiple access patterns. Depending on the layout of the data structure in physical address space, some access patterns result in non-unit strides. In existing systems, which are optimized to store and access cache lines, non-unit strided accesses exhibit low spatial locality. Therefore, they incur high latency, and waste memory bandwidth and cache space.

We propose the Gather-Scatter DRAM (GS-DRAM) to address this problem. We observe that a commodity DRAM module contains many chips. Each chip stores a part of every cache line mapped to the module. Our idea is to enable the memory controller to access multiple values that belong to a strided pattern from different chips using a single read/write command. To realize this idea, GS-DRAM first maps the data of each cache line to different chips such that multiple values of a strided access pattern are mapped to different chips. Second, instead of sending a separate address to each chip, GS-DRAM maps each strided pattern to a small *pattern ID* that is communicated to the module. Based on the pattern ID, each chip independently computes the address of the value to be accessed. The cache line returned by the module contains different values of the strided pattern gathered from different chips. We show that this approach enables GS-DRAM to achieve near-ideal memory bandwidth and cache utilization for many common access patterns.

We design an end-to-end system to exploit GS-DRAM. Our evaluations show that 1) for in-memory databases, GS-DRAM obtains the best of the row store and the column store layouts, in terms of both performance and energy, and 2) for matrix-matrix multiplication, GS-DRAM seamlessly enables SIMD optimizations and outperforms the best tiled layout. Our framework is general, and can benefit many modern data-intensive applications.

\*Phillip B. Gibbons was with Intel Labs for much of this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).

MICRO-48, December 05-09, 2015, Waikiki, HI, USA

Copyright 2015 ACM. ISBN 978-1-4503-4034-2/15/12...\$15.00.

DOI: <http://dx.doi.org/10.1145/2830772.2830820>

## Categories and Subject Descriptors

B.3.1 [Memory Structures]: Semiconductor Memories

## Keywords

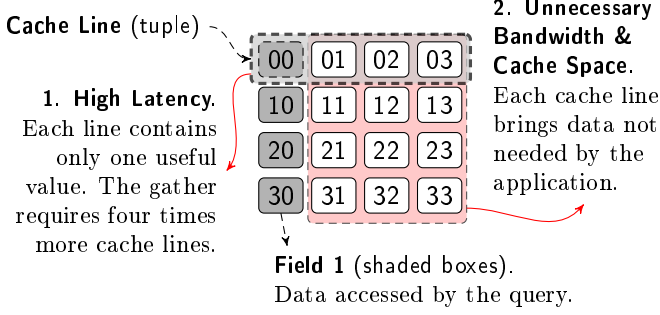
Strided accesses, DRAM, Memory bandwidth, In-memory databases, SIMD, Performance, Energy, Caches

## 1. Introduction

Many data structures are accessed with multiple access patterns. Depending on the layout of the data structure in the physical address space, some access patterns result in large non-unit strides. For instance, *in-memory databases* [18, 22, 43] exhibit such strided accesses frequently. While a database can be laid out as a row store or a column store [45], accessing a column from a row store (or vice versa) results in non-unit strided accesses. Most databases typically run two kinds of queries: *transactions*, which access many fields from a few rows, or *analytics*, which access one or few fields from many rows. Regardless of the layout, one of these two query types would result in non-unit strided accesses. We use the term *strided accesses* to refer to non-unit strides.

Existing systems are inefficient in performing strided accesses because the memory subsystem, i.e., caches and main memory, is optimized to store and communicate data at a cache line granularity. Unfortunately, the cache line size (typically 64 bytes) is usually much larger than the size of the individual data item involved in a strided access (typically 4 or 8 bytes). Consequently, strided accesses have two adverse effects on performance: 1) high latency, and 2) unnecessary memory bandwidth and cache space consumption.

Figure 1 illustrates these two problems using a query that accesses a *single field* in a database table. The figure shows the first four tuples of the table. We assume each tuple contains four fields and fits in a cache line (as shown in the figure). The query accesses only the first field of the table. First, each cache line contains only one useful piece of data (shaded boxes in the figure). As a result, the processor must fetch four times more cache lines than necessary to access the required data. This significantly increases the latency of the operation, thereby directly degrading the performance of the program performing the operation. Second, although the program



**Figure 1: Problems in accessing the first field (shaded boxes) from a table in a cache-line-optimized memory system. The box “ij” corresponds to the  $j^{th}$  field of the  $i^{th}$  tuple.**

needs only the first field of the table, each cache line access also brings along the remaining fields of the table into the cache. As a result, the strided access results in inefficient use of off-chip memory bandwidth and on-chip cache space, both of which are precious shared resources in modern systems. This results in high interference to other concurrently-running programs in the system, thereby degrading overall system performance. While the program can lay out the data structure to suit the dominating access pattern, if multiple access patterns occur frequently, existing systems cannot avoid strided accesses. For example, an emerging database workload, referred to as Hybrid Transaction/Analytical Processing (HTAP) [3], runs both transactions and real-time analytics on the same version of the database table.

Given the importance of strided access patterns, several prior works (e.g., Impulse [9, 53], Adaptive/Dynamic Granularity Memory Systems [51, 52]) have proposed solutions to improve the performance of strided accesses. Unfortunately, prior works [9, 51, 52] require the off-chip memory interface to support fine-grained memory accesses [4, 5, 8, 50, 55] and, in some cases, a sectored cache [30, 42]. These approaches significantly increase the cost of the memory interface and the cache tag store, and potentially lower the utilization of off-chip memory bandwidth and on-chip cache space.

Our goal is to design a mechanism that 1) improves the performance (cache hit rate and memory bandwidth consumption) of strided accesses, and 2) works with commodity DRAM modules and traditional non-sectored caches with very few changes.

To this end, we propose the Gather-Scatter DRAM (GS-DRAM), a substrate that allows the memory controller to gather or scatter data with strided access patterns (specifically power-of-2 strides) efficiently. Our observation is the following. Modern DRAM modules consist of multiple chips. Each chip within a module contains a part of every cache line stored in the module. Our idea is to allow the memory controller to access multiple

values of a strided access pattern from different chips with a single read or write command. However, there are two challenges in implementing this idea.

First, if multiple pieces of data required by an access pattern are present in the same chip (referred to as *chip conflicts*), then the memory controller must necessarily issue one read or write for *each* such piece of data. To address this problem, GS-DRAM uses a simple data shuffling mechanism that modifies the way data of a cache line is mapped to different DRAM chips within a rank. Our mapping scheme incurs minimal chip conflicts for any power-of-2 strided access pattern. Second, in existing DRAM interfaces, all the chips within a rank access the *same* address in response to a read or write command. To break this restriction, in GS-DRAM, with each read or write command, the memory controller provides a modifier called the *pattern ID*. Based on the pattern ID, each DRAM chip uses a simple *column translation logic* to compute a new column address, and accesses the corresponding piece of data. We show that a handful of pattern IDs and a simple column translation logic (only two bitwise operations per chip) enables the memory controller to efficiently gather or scatter any power-of-2 strided access pattern. For instance, in the example of Figure 1, the memory controller can access a cache line that contains only the first field of four tuples from the table, i.e., **00 10 20 30**, with nearly the same latency as accessing any cache line from DRAM.

For applications to exploit our GS-DRAM substrate, our mechanism needs support from various layers of the system stack. The application instructs the processor to use a specific access pattern by using the appropriate pattern ID with the `load` and `store` instructions. Similarly, the processor uses the pattern ID (along with the cache line address) to locate *gathered* cache lines. Finally, we observe that many applications have only two primary access patterns, and use this observation (as a restriction) to simplify the support required to maintain coherence of gathered cache lines. Section 4 discusses our end-to-end system design and its associated cost.

We quantitatively evaluate GS-DRAM with two applications. First, for an in-memory database, we compare the performance of GS-DRAM to a row store and a column store on a variety of workloads with transactions and analytics. Our evaluations show that GS-DRAM provides the best of both the row store and column store layouts for all workloads. More specifically, GS-DRAM performs 2X better than the row store for analytical queries and 3X better than the column store for transactions. Second, for a General Matrix-Matrix (GEMM) multiplication workload, GS-DRAM avoids the need for a software gather to exploit SIMD optimizations, and outperforms the best tiled layout by 10%. Section 5 discusses these results and other applications of GS-DRAM.

This paper makes the following contributions.

- We propose GS-DRAM, a substrate that exploits the commodity DRAM architecture to enable the memory controller to efficiently gather or scatter data with strided access patterns. GS-DRAM can be implemented with negligible changes to commodity DRAM modules and no changes to DRAM chips.
- We discuss and address several challenges (e.g., cache coherence) in designing an end-to-end system to exploit the GS-DRAM substrate. Our mechanism requires minimal changes to existing hardware (less than 0.6% cache area cost).
- We demonstrate the power of GS-DRAM by evaluating its benefits on an emerging in-memory database application, and a well-optimized matrix-matrix multiplication workload. GS-DRAM outperforms state-of-the-art mechanisms for both applications.

## 2. Background on DRAM

DRAM-based main memory is a multi-level hierarchy of structures. At the highest level, each processor consists of one or more DRAM channels. Each channel has a dedicated command, address, and data bus. One or more memory modules can be connected to each DRAM channel. Each memory module contains a number of DRAM chips. As the data output width of each DRAM chip is low (typically 8 bits for commodity DRAM), multiple chips are grouped together to form a rank. All chips within a rank share the command and address buses, but each chip has its own dedicated data bus. As a result, any command sent to a rank is processed by all the chips within the rank, thereby increasing the data width of the rank. Figure 2 shows the organization of a rank with 4 commodity DRAM chips (we use a 4-chip DRAM rank just to simplify the explanation of our mechanism. Our evaluations, however, use standard 8-chip 64-bit DRAM ranks). With an 8-bit data width for each chip, the data width of the rank is 32 bits.

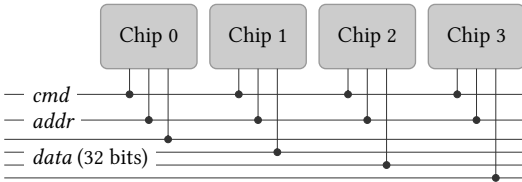


Figure 2: Organization of a DRAM rank with 4 chips.

Internally, each chip consists of multiple banks. Although each bank is further divided into smaller structures [12, 26, 28, 41, 47], for the purposes of understanding our mechanism, the following abstraction of a bank is sufficient. Each bank consists of many rows of DRAM

cells, and a row buffer that caches the last accessed row from the bank. Each row contains many cache lines, each identified by the corresponding column address.

When the memory controller receives an access to a cache line (32 bytes), it first determines the bank  $B$ , row address  $R$ , and the column address  $C$  of the cache line within the DRAM hierarchy. As the data of each cache line is equally split across all four chips within the rank, the memory controller maintains a mapping scheme to determine which parts of the cache line are mapped to which chips. One simple mapping scheme maps the  $i^{\text{th}}$  8 bytes of each cache line to the  $i^{\text{th}}$  chip.

To read the cache line from DRAM, the memory controller performs the following three steps. First, the controller issues a PRECHARGE command to the bank  $B$ . This step prepares the bank for a new access and can be skipped if the bank is already in the *precharged* state. Second, the controller issues an ACTIVATE command to the bank with the address  $R$ . This command instructs all the chips in the rank to copy the data from the corresponding row of DRAM cells to the row buffer within the bank. This step can also be skipped if the row  $R$  within the bank is already activated. Finally, to access the cache line, the controller issues a READ command to the bank with the address  $C$ . Upon receiving the command, each chip accesses the corresponding column of data (8 bytes) from the row buffer and transfers it on the data bus, 8 bits at a time. In DDR DRAM technology, data can be transferred on the bus in each half cycle. As a result, the transfer takes 4 bus cycles. Once the data is transferred, the memory controller assembles the required cache line based on the cache-line-to-chip mapping scheme, and sends the cache line back to the processor. A WRITE operation is similar, except the data is transferred from the memory controller to DRAM.<sup>1</sup>

## 3. The Gather-Scatter DRAM

We will use the database example in Section 1 (Figure 1) to describe the challenges and the design of our mechanism. To avoid confusion with DRAM terminology, we will refer to the database rows and columns as *tuples* and *fields*, respectively. To recap, our example consists of a database table with many tuples. Each tuple contains four fields and fits exactly in a cache line. Therefore, accessing any tuple from the table involves only a single cache line read. On the other hand, accessing (only) a single field from all the tuples requires the processor to read all the fields from all the tuples.

Our goal is to design a DRAM substrate that will enable the processor to access a field of the table (stored

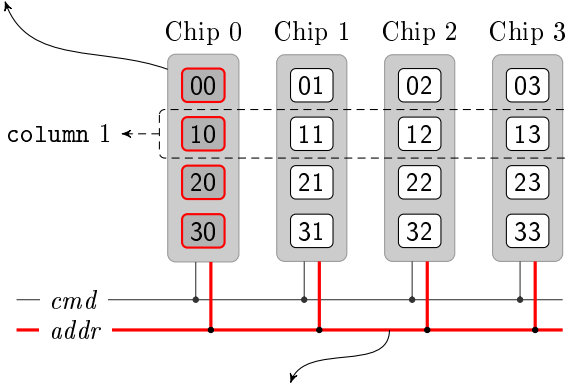
<sup>1</sup>For detailed discussion DRAM organization and operation, we refer the reader to other works (e.g., [23, 26, 27, 28, 31, 41, 49, 54]).

in tuple-major order) across all tuples, without incurring the penalties of existing interfaces. More specifically, if the memory controller wants to read the first field of the first four tuples of the table, it must be able to issue a *single command* that fetches the following gathered cache line: 00 10 20 30. At the same time, the controller must be able to read a tuple from memory (e.g., 00 01 02 03) with a single command. Our **idea** is to enable the controller to access multiple values from a strided access from *different* chips within the rank with a single command. However, there are two challenges in implementing this idea.

### 3.1. Challenges in Designing GS-DRAM

Figure 3 shows the two challenges. We assume that the first four tuples of the table are stored from the beginning of a DRAM row. Since each tuple maps to a single cache line, the data of each tuple is split across all four chips. Based on the mapping scheme described in Section 2, the  $i^{th}$  field of each tuple is mapped to the  $i^{th}$  chip.

**Challenge 1:** The first field from all the tuples of the table are mapped to the *same* chip. The memory controller must issue one **READ** for *each* value. The first field of the tuples should be distributed across all chips for the controller to gather them with minimum **READs**.



**Challenge 2:** All the chips use the *same* address for every **READ/WRITE** command. The memory controller needs more flexibility to gather different access patterns.

Figure 3: The two challenges in designing GS-DRAM.

**Challenge 1: Reducing chip conflicts.** The simple mapping mechanism maps the first field of *all* the tuples to Chip 0. Since each chip can send out only one field (8 bytes) per **READ** operation, gathering the first field of the four tuples will necessarily require four **READs**. In a general scenario, different pieces of data that are required by a gather operation will be mapped to different chips. When two such pieces of data are mapped to the same chip, it results in what we call a *chip conflict*. Chip con-

licts increase the number of **READs** required to complete the gather operation. Therefore, we have to map the data structure to the chips in a manner that minimizes the number of chip conflicts for target access patterns.

**Challenge 2: Communicating the access pattern to the module.** As shown in Figure 3, in today’s systems, when a column command is sent to a rank, all the chips select the *same* column from the activated row and send out the data. If the memory controller needs to access the first tuple of the table and the first field of the four tuples each with a *single READ* operation, we need to break this constraint and allow the memory controller to potentially read *different* columns from different chips using a single **READ** command. One naive way of achieving this flexibility is to use multiple address buses, one for each chip. Unfortunately, this approach is very costly as it significantly increases the pin count of the memory channel. Therefore, we need a simple and low cost mechanism to allow the memory controller to efficiently communicate different access patterns to the DRAM module.

In the following sections, we propose a simple mechanism to address the above challenges with specific focus on power-of-2 strided access patterns. While non-power-of-2 strides (e.g., odd strides) pose some additional challenges (e.g., alignment), a similar approach can be used to support non-power-of-2 strides as well.

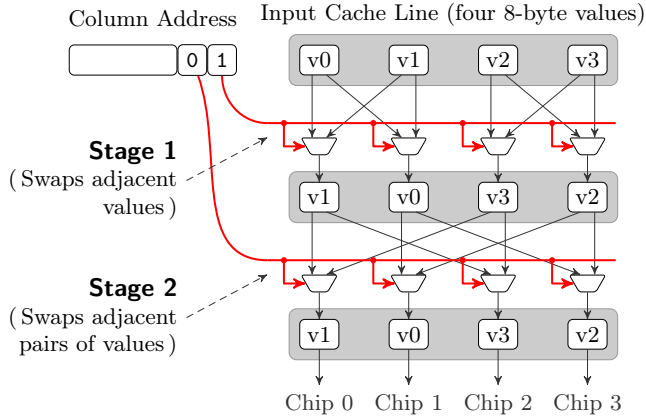
### 3.2. Column ID-based Data Shuffling

To address challenge 1, i.e., to minimize chip conflicts, the memory controller must employ a mapping scheme that distributes data of each cache line to different DRAM chips with the following three goals. First, the mapping scheme should be able to minimize chip conflicts for a number of access patterns. Second, the memory controller must be able to succinctly communicate an access pattern along with a column command to the DRAM module. Third, once the different parts of the cache line are read from different chips, the memory controller must be able to quickly assemble the cache line. Unfortunately, these goals are conflicting.

While a simple mapping scheme (e.g., the one described in Section 2) enables the controller to assemble a cache line by concatenating the data received from different chips, this scheme incurs a large number of chip conflicts for many frequently occurring access patterns (e.g., any power-of-2 stride  $> 1$ ). On the other hand, pseudo-random mapping schemes [38] potentially incur a small number of conflicts for almost any access pattern. Unfortunately, such pseudo-random mapping schemes have two shortcomings. First, for any cache line access, the memory controller must compute which column of data to access from each chip and communicate this information to the chips along with the column command.

With a pseudo random interleaving, this communication may require a separate address bus for each chip, which would significantly increase the cost of the memory channel. Second, after reading the data, the memory controller must spend more time assembling the cache line, increasing the overall latency of the READ operation.

We propose a simple *column ID-based data shuffling* mechanism that achieves a sweet spot by restricting our focus to power-of-2 strided access patterns. Our shuffling mechanism is similar to a butterfly network [13], and is implemented in the memory controller. To map the data of the cache line with column address  $C$  to different chips, the memory controller inspects the  $n$  least significant bits (LSB) of  $C$ . Based on these  $n$  bits, the controller uses  $n$  stages of shuffling. Figure 4 shows an example of a 2-stage shuffling mechanism. In Stage 1 (Figure 4), if the LSB is set, our mechanism groups adjacent 8-byte values in the cache line into pairs and swaps the values within each pair. In Stage 2 (Figure 4), if the second LSB is set, our mechanism groups the 8-byte values in the cache line into quadruplets, and swaps the adjacent *pairs* of values. The mechanism proceeds similarly into the higher levels, doubling the size of the group of values swapped in each higher stage. The shuffling mechanism can be enabled only for those data structures that require our mechanism. Section 4.3 discusses this in more detail.

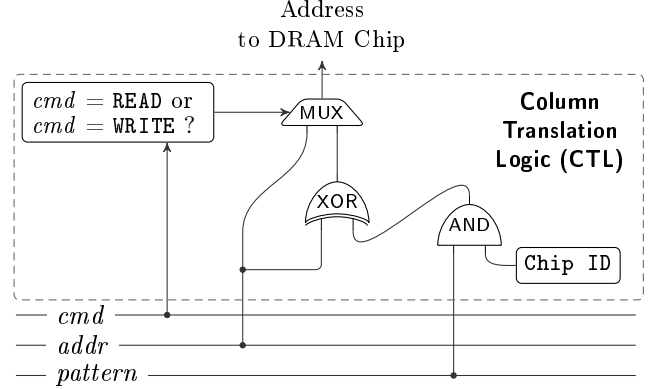


**Figure 4: 2-stage shuffling mechanism that maps different 8-byte values within a cache line to a DRAM chip. For each mux, 0 selects the vertical input, and 1 selects the cross input.**

With this simple multi-stage shuffling mechanism, the memory controller can map data to DRAM chips such that *any* power-of-2 strided access pattern incurs zero chip conflicts for values within a single DRAM row.

### 3.3. Pattern ID: Low-cost Column Translation

The second challenge is to enable the memory controller to flexibly access different column addresses from differ-



**Figure 5: Column Translation Logic (CTL). Each chip has its own CTL. The CTL can be implemented in the DRAM module (as shown in Figure 6). Each logic gate performs a bitwise operation of the input values.**

ent DRAM chips using a single READ command. To this end, we propose a simple mechanism wherein the memory controller associates a *pattern ID* with each access pattern. It provides this pattern ID with each column command. Each DRAM chip then independently computes a new column address based on 1) the issued column address, 2) the chip ID, and 3) the pattern ID. We refer to this mechanism as *column translation*.

Figure 5 shows the column translation logic for a single chip. As shown in the figure, our mechanism requires only two bitwise operations per chip to compute the new column address. More specifically, the output column address for each chip is given by  $(\text{Chip ID} \& \text{Pattern ID}) \oplus \text{Column ID}$ , where Column ID is the column address provided by the memory controller. In addition to the logic to perform these simple bitwise operations, our mechanism requires 1) a register per chip to store the chip ID, and 2) a multiplexer to enable the address translation only for column commands. While our column translation logic can be combined with the column selection logic already present within each chip, our mechanism can also be implemented within the DRAM module with *no* changes to the DRAM chips.

Combining this pattern-ID-based column translation mechanism with the column-ID-based data shuffling mechanism, the memory controller can gather or scatter any power-of-2 strided access pattern with no waste in memory bandwidth.

### 3.4. GS-DRAM: Putting It All Together

Figure 6 shows the full overview of our GS-DRAM substrate. The figure shows how the first four tuples of our example table are mapped to the DRAM chips using our data shuffling mechanism. The first tuple (column ID = 0) undergoes no shuffling as the two LSBs of the column

ID are both 0 (see Figure 4). For the second tuple (column ID = 1), the adjacent values within each pairs of values are swapped (Figure 4, Stage 1). Similarly, for the third tuple (column ID = 2), adjacent pair of values are swapped (Figure 4, Stage 2). For the fourth tuple (column ID = 3), since the two LSBs of the column ID are both 1, both stages of the shuffling scheme are enabled (Figure 4, Stages 1 and 2). As shown in shaded boxes in Figure 6, the first field of the four tuples (i.e., **00 10 20 30**) are mapped to *different* chips, allowing the memory controller to read them with a single **READ** command. The same is true for the other fields of the table as well (e.g., **01 11 21 31**).

The figure also shows the per-chip column translation logic. To read a specific tuple from the table, the memory controller simply issues a **READ** command with pattern ID = 0 and an appropriate column address. For example, when the memory controller issues the **READ** for column ID 2 and pattern 0, the four chips return the data corresponding to the columns (2 2 2 2), which is the data in the third tuple of the table (i.e., **22 23 20 21**). In other words, pattern ID 0 allows the memory controller to perform the default read operation. Hence, we refer to pattern ID 0 as the *default pattern*.

On the other hand, if the memory controller issues a **READ** for column ID 0 and pattern 3, the four chips return the data corresponding to columns (0 1 2 3), which precisely maps to the first field of the table. Similarly, the other fields of the first four tuples can be read from the database by varying the column ID with pattern 3.

### 3.5. GS-DRAM Parameters

GS-DRAM has three main parameters: 1) the number of chips in each module, 2) the number of shuffling stages in the data shuffling mechanism, and 3) the number of

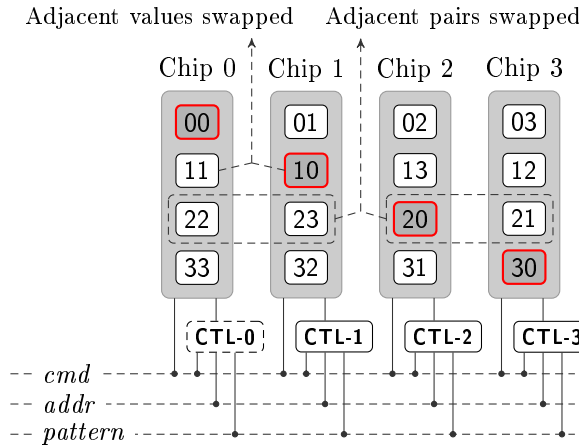


Figure 6: GS-DRAM Overview. CTL-*i* is the column translation logic with Chip ID = *i* (Figure 5).

bits of pattern ID. While the number of chips determines the size of each cache line, the other two parameters determine the set of access patterns that can be efficiently gathered by GS-DRAM. We use the term  $\text{GS-DRAM}_{c,s,p}$  to denote a GS-DRAM with *c* chips, *s* stages of shuffling, and *p* bits of pattern ID.

Figure 7 shows all possible cache lines that can be gathered by  $\text{GS-DRAM}_{4,2,2}$ , with the four possible patterns for column IDs 0 through 3. For each pattern ID and column ID combination, the figure shows the index of the four values within the row buffer that are retrieved from the DRAM module. As shown in the figure, pattern 0 retrieves contiguous values. Pattern 1 retrieves every other value (stride = 2). Pattern 2 has a dual stride of (1,7). Pattern 3 retrieves every 4th value (stride = 4). In general, pattern  $2^k - 1$  gathers data with a stride  $2^k$ .

Pattern = 0 Stride = 1	DRAM Col ID	Cache line retrieved	Pattern = 1 Stride = 2	DRAM Col ID	Cache line retrieved
	0	0 1 2 3		0	0 2 4 6
	1	4 5 6 7		1	1 3 5 7
	2	8 9 10 11		2	8 10 12 14
	3	12 13 14 15		3	9 11 13 15
Pattern = 2 Stride = 1,7	DRAM Col ID	Cache line retrieved	Pattern = 3 Stride = 4	DRAM Col ID	Cache line retrieved
	0	0 1 8 9		0	0 4 8 12
	1	2 3 10 11		1	1 5 9 13
	2	4 5 12 13		2	2 6 10 14
	3	6 7 14 15		3	3 7 11 15

Figure 7: List of cache lines gathered by  $\text{GS-DRAM}_{4,2,2}$  for all possible patterns for column IDs 0–3. Each circle contains the index of the 8-byte value inside the logical row buffer.

While we showed a use case for pattern 3 (in our example), we envision use-cases for other patterns as well. Pattern 1, for instance, can be useful for data structures like key-value stores. Assuming an 8-byte key and an 8-byte value, the cache line (Patt 0, Col 0) corresponds to the first two key-value pairs. However the cache line (Patt 1, Col 0) corresponds to the first four keys, and (Patt 1, Col 1) corresponds to the first four values. Similarly, pattern 2 can be used to fetch odd-even pairs of fields from an object with 8 fields.

Our mechanism is general. For instance, with  $\text{GS-DRAM}_{8,3,3}$  (i.e., 8 chips, 3 shuffling stages, and 3 bits of pattern ID), the memory controller can access data with seven different patterns (e.g., pattern 7 results in a stride of 8). Section 6 discusses other simple extensions to our approach to enable more fine-grained gather access patterns, and larger strides.

### 3.6. Ease of Implementing GS-DRAM

In Section 5, we will show that GS-DRAM has compelling performance and energy benefits compared to existing DRAM interfaces. These benefits are augmented by the fact that GS-DRAM is simple to implement. First, our data shuffling mechanism is simple and has low latency. Each stage involves only data swapping and takes at most one processor cycle. Our evaluations use GS-DRAM<sub>8,3,3</sub>, thereby incurring 3 cycles of additional latency to shuffle/unshuffle data for each DRAM write/read. Second, for GS-DRAM<sub>\*,\*,p</sub>, the column translation logic requires only two p-bit bitwise operations, a p-bit register to store the chip ID, and a p-bit multiplexer. In fact, this mechanism can be implemented as part of the DRAM module *without* any changes to the DRAM chips themselves. Finally, third, GS-DRAM requires the memory controller to communicate only k bits of pattern ID to the DRAM module, adding only a few pins to each channel. In fact, the column command in existing DDR DRAM interfaces already has a few spare address pins that can potentially be used by the memory controller to communicate the pattern ID (e.g., DDR4 has two spare address pins for column commands [21]).

## 4. End-to-end System Design

In this section, we discuss the support required from the rest of the system stack to exploit the GS-DRAM substrate. In this paper, we propose a mechanism that leverages support from different layers of the system stack to exploit GS-DRAM: 1) on-chip caches, 2) the instruction set architecture, and 3) software. It is also possible for the processor to dynamically identify different access patterns present in an application and exploit GS-DRAM to accelerate such patterns transparently to the application. As our goal in this work is to demonstrate the benefits of GS-DRAM, we leave the design of such an automatic mechanism for future work. The following sections assume a GS-DRAM<sub>\*,\*,p</sub>, i.e., a p-bit pattern ID.

### 4.1. On-Chip Cache Support

Our mechanism introduces two problems with respect to on-chip cache management. First, when the memory controller gathers a cache line from a non-zero pattern ID, the values in the cache line are *not* contiguously stored in physical memory. For instance, in our example (Figure 1), although the controller can fetch the first field of the first four tuples of the table with a single READ, the first field of the table is not stored contiguously in physical memory. Second, two cache lines belonging to different patterns may have a partial overlap. In our example (Figure 1), if the memory controller reads the first

tuple (pattern ID = 0, column ID = 0) and the first field of the first four tuples (pattern ID = 3, column ID = 0), the two resulting cache lines have a common value (the first field of the first tuple, i.e., **00**).

One simple way to avoid these problems is to store the individual values of the gathered data in *different* physical cache lines by employing a sectored cache [30] (for example). However, with the off-chip interface to DRAM operating at a wider-than-sector (i.e., a full cache line) granularity, such a design will increase the complexity of the cache-DRAM interface. For example, writebacks may require read-modify-writes as the processor may not have the entire cache line. More importantly, a mechanism that does not store the gathered values in the same cache line cannot extract the full benefits of SIMD optimizations because values that are required by a single SIMD operation would now be stored in *multiple* physical cache lines. Therefore, we propose a simple mechanism that stores each gathered cache line from DRAM in a single physical cache line in the on-chip cache. Our mechanism has two aspects.

**1. Identifying non-contiguous cache lines.** When a non-contiguous cache line is stored in the cache, the cache controller needs a mechanism to identify the cache line. We observe that, in our proposed system, each cache line can be uniquely identified using the cache line address and the pattern ID with which it was fetched from DRAM. Therefore, we extend each cache line tag in the cache tag store with  $p$  additional bits to store the pattern ID of the corresponding cache line.

**2. Maintaining cache coherence.** The presense of overlapping cache lines has two implications on cache coherence. First, before fetching a cache line from DRAM, the controller must check if there are any dirty cache lines in the cache which have a partial overlap with the cache line being fetched. Second, when a value is modified by the processor, in addition to invalidating the modified cache line from the other caches, the processor must also invalidate all other cache lines that contain the value that is being modified. With a number of different available patterns, this can be a complex and costly operation.

Fortunately, we observe that many applications that use strided accesses require only two pattern IDs per data structure, the default pattern and one other pattern ID. Thus, as a trade-off to simplify cache coherence, we restrict each data structure to use only the zero pattern and one other pattern ID. To implement this constraint, we associate each virtual page with an additional  $p$ -bit pattern ID. Any access to a cache line within the page can use either the zero pattern or the page's pattern ID. If multiple virtual pages are mapped to the same physical page, the OS must ensure that the same alternate pattern ID is used for all mappings.

Before fetching a cache line from DRAM with a pattern, the memory controller must only look for dirty cache lines from the other pattern. Since all these cache lines belong to the same DRAM row, this operation is fast and can be accelerated using simple structures like the Dirty-Block Index [40]. Similarly, when the processor needs to modify a shared cache line, our mechanism piggybacks the other pattern ID of the page along with the *read-exclusive* coherence request. Each cache controller then locally invalidates the cache lines from the other pattern ID that overlap with the cache line being modified. For GS-DRAM<sub>c,\*,\*</sub>, our mechanism requires  $c$  additional invalidations for each read-exclusive request.

#### 4.2. Instruction Set Architecture Support

To enable software to communicate strided access patterns to the processor, we introduce a new variant of the load/store instruction, called `pattload/pattstore` that enable the application to specify the pattern ID. These new instructions takes the following form:

```
pattload reg, addr, patt
pattstore reg, addr, patt
```

where `reg` is the destination register, `addr` is the address of the data, and `patt` is the pattern ID.

To execute a `pattload` or `pattstore`, the processor first splits the `addr` field into two parts: the cache line address (`caddr`), and the offset within the cache line (`offset`). Then the processor sends out a request for the cache line with address-pattern combination (`caddr`, `patt`). If the cache line is present in the on-chip cache, it is sent to the processor. Otherwise, the request reaches the memory controller. The memory controller identifies the row address and the column address from `caddr` and issues a `READ` command for a cache line with pattern ID `patt`. If the memory controller interleaves cache lines across multiple channels (or ranks), then it must access the corresponding cache line within each channel (or rank) and interleave the data from different channels appropriately before obtaining the required cache line. The cache line is then stored in the on-chip cache and is also sent to the processor. After receiving the cache line, the processor reads or updates the data at the `offset` to or from the destination or source register (`reg`).

Note that architectures like x86 allow instructions to directly operate on memory by using different addressing modes to specify memory operands [1]. For such architectures, common addressing modes may be augmented with a pattern ID field, or instruction prefixes may be employed to specify the pattern.

#### 4.3. System and Application Software Support

Our mechanism requires two pieces of information from the software for each data structure: 1) whether the

data structure requires the memory controller to use the shuffling mechanism (Section 3.2) (we refer to this as the *shuffle flag*), and 2) the alternate pattern ID (Section 3.3) with which the application will access the data structure. To enable the application to specify this information, we propose a new variant of the `malloc` system call, called `pattmalloc`, which includes two additional parameters: the shuffle flag, and the pattern ID. When the OS allocates virtual pages for a `pattmalloc`, it also updates the page tables with the shuffle flag and the alternate pattern ID for those pages.

Once the data structure is allocated with `pattmalloc`, the application can use the `pattload` or `pattstore` instruction to access the data structure efficiently with both the zero pattern and the alternate access pattern. While we can envision automating this process using a compiler optimization, we do not explore that path in this paper. Figure 8 shows an example piece of code before and after our optimization. The original code (line 5) allocates an array of 512 objects (each object with eight 8-byte fields) and computes the sum of the first field of all the objects (lines 8 and 9). The figure highlights the key benefit of our approach.

In the program without our optimization (Figure 8, left), each iteration of the loop (line 9) fetches a different cache line. As a result, the entire loop accesses 512 different cache lines. On the other hand, with our optimization (Figure 8, right), the program first allocates memory for the array using `pattmalloc` (line 5), with the shuffle flag enabled and an alternate pattern ID = 7 (i.e., stride of 8). The program then breaks the loop into two parts. Each iteration of the outer loop (line 8) fetches a single strided cache line that contains only values from the first field. The loop skips the other fields ( $i += 8$ ). The inner loop (lines 9-11) iterates over values within each strided cache line. In the first iteration of the inner loop, the `pattload` instruction with pattern ID 7 fetches a cache line with a stride of 8. As a result, the remaining seven iterations of the inner loop result in cache hits. Consequently, with our optimization, the entire loop accesses only 64 cache lines. As we will show in our evaluations, this reduction in the number of accessed cache lines directly translates to reduction in latency, bandwidth consumption, and cache capacity consumption, thereby improving overall performance.

#### 4.4. Hardware Cost

In this section, we quantify the changes required by our mechanism, specifically GS-DRAM<sub>8,3,3</sub> (Section 3.5), to various hardware components. On the DRAM side, first, our mechanism requires the addition of the column translation logic (CTL) for each DRAM chip. Each CTL requires a 3-bit register for the Chip ID, a 3-bit bitwise



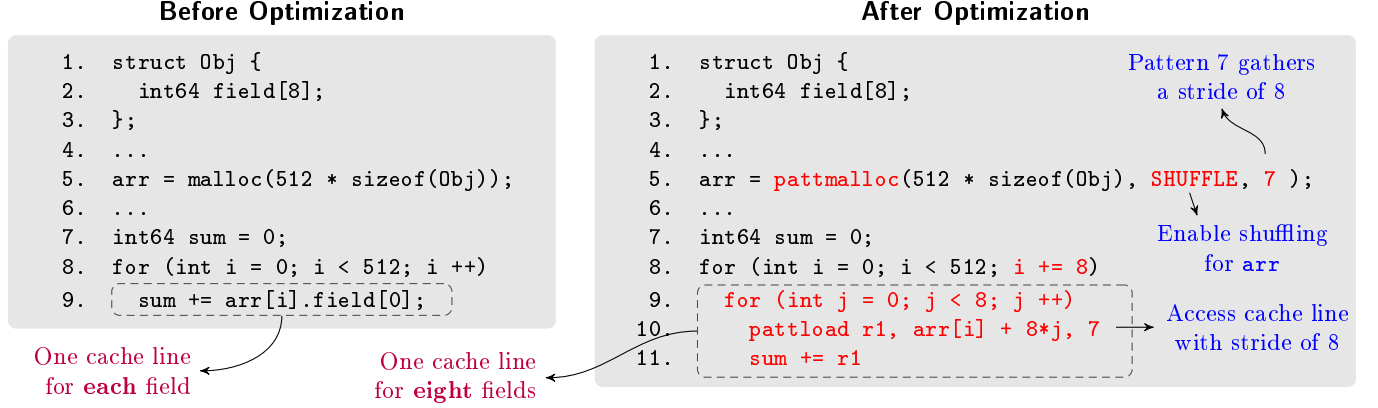


Figure 8: Example code without and with our optimization.

AND gate, a 3-bit bitwise XOR gate and a 3-bit bitwise multiplexer. Even for a commodity DRAM module with 8 chips, the overall cost is roughly 72 logic gates and 24 bits of register storage, which is negligible compared to the logic already present in a DRAM module. Second, our mechanism requires a few additional pins on the DRAM interface to communicate the pattern ID. However, existing DRAM interfaces already have some spare address bits, which can be used to communicate part of the pattern ID. Using this approach, a 3-bit pattern ID requires only one additional pin for DDR4 [21].

On the processor side, first, our mechanism requires the controller to implement the shuffling logic. Second, our mechanism augments each cache tag entry with the pattern ID. Each page table entry and TLB entry stores the shuffle flag and the alternate pattern ID for the corresponding page (Section 4.1). For a 3-bit pattern ID, the cost of this addition is less than 0.6% of the cache size. Finally, the processor must implement the `pattload` and `pattstore` instructions, and the state machine for invalidating additional cache lines on read-exclusive coherence requests. The operation of `pattload/pattstore` is not very different from that of a regular `load/store` instruction. Therefore, we expect the implementation of these new instructions to be simple. Similarly, on a write, our mechanism has to check only eight cache lines (for GS-DRAM with 8 chips) for possible overlap with the modified cache line. Therefore, we expect the invalidation state machine to be relatively simple. Note that a similar state machine has been used to keep data coherent in a virtually-indexed physically-tagged cache in the presence of synonyms [2].

## 5. Applications and Evaluations

To quantitatively evaluate the benefits of GS-DRAM, we implement our framework in the Gem5 simulator [7], on top of the x86 architecture. We implement the `pattload`

instruction by modifying the behavior of the `prefetch` instruction to gather with a specific pattern into either the `rax` register (8 bytes) or the `xmm0` register (16 bytes). None of our evaluated applications required the `pattstore` instruction. Table 1 lists the main parameters of the simulated system. All caches uniformly use 64-byte cache lines. While we envision several applications to benefit from our framework, in this section, we primarily discuss and evaluate two applications: 1) an in-memory database workload, and 2) general matrix-matrix multiplication workload.

Processor	1-2 cores, x86, in-order, 4 GHz
L1-D Cache	Private, 32 KB, 8-way associative, LRU policy
L1-I Cache	Private, 32 KB, 8-way associative, LRU policy
L2 Cache	Shared, 2 MB, 8-way associative, LRU policy
Memory	DDR3-1600, 1 channel, 1 rank, 8 banks
	Open row, FR-FCFS [39, 56], GS-DRAM <sub>8,3,3</sub>

Table 1: Main parameters of the simulated system.

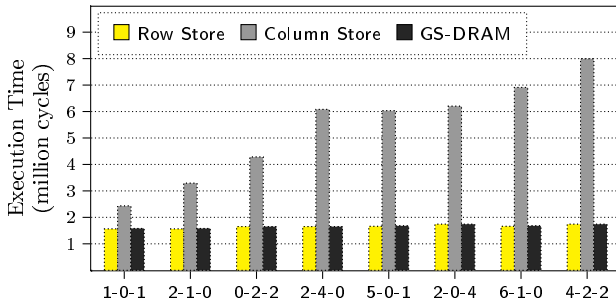
### 5.1. In-Memory Databases

In-memory databases (IMDB) (e.g., [18, 22, 43]) provide significantly higher performance than traditional disk-oriented databases. Similar to any other database, an IMDB may support two kinds of queries: *transactions*, which access many fields from a few tuples, and *analytics*, which access one or few fields from many tuples. As a result, the storage model used for the database tables heavily impacts the performance of transactions and analytical queries. As mentioned in Section 1, while a row-oriented organization (*row store*) is better for transactions, a column-oriented organization [45] (*column store*) is better for analytics. Increasing need for both fast transactions and fast real-time analytics has given rise to a new workload referred to as *Hybrid Trans-*

*action/Analytical Processing* (HTAP) [3]. In an HTAP workload, both transactions and analytical queries are run on the *same version* of the database. Unfortunately, neither the row store nor the column store provides the best performance for both transactions and analytics.

With our GS-DRAM framework, each database table can be stored as a row store in memory, but can be accessed at high performance *both* in the row-oriented access pattern *and* the field-oriented access pattern.<sup>2</sup> Therefore, we expect GS-DRAM to provide the *best of both row and column layouts* for both kinds of queries. We demonstrate this potential benefit by comparing the performance of GS-DRAM with both a row store layout (Row Store) and a column store layout (Column Store) on three workloads: 1) a transaction-only workload, 2) an analytics-only workload, and 3) an HTAP workload. For our experiments, we assume an IMDB with a single table with one million tuples and no use of compression. Each tuple contains eight 8-byte fields, and fits exactly in a 64B cache line. (Our mechanism naturally extends to any table with power-of-2 tuple size.)

**Transaction workload.** For this workload, each transaction operates on a randomly-chosen tuple, accessing  $i$ ,  $j$ , and  $k$  fields in the read-only, write-only, and read-write mode, respectively. Figure 9 compares the performance (execution time) of GS-DRAM, Row Store, and Column Store on the transaction workload for various values of  $i$ ,  $j$ , and  $k$  (x-axis). The workloads are sorted based on the total number of fields accessed by each transaction. For each mechanism, the figure plots the execution time for running 10000 transactions.



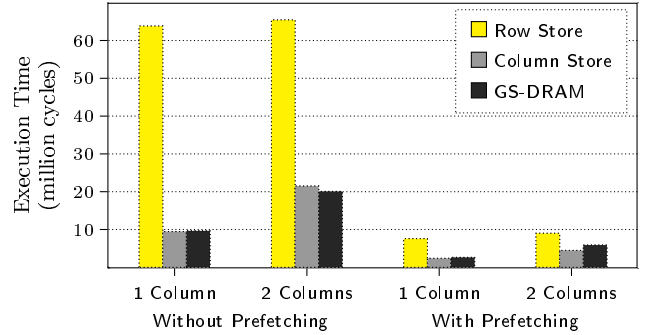
**Figure 9: Transaction Workload Performance: Execution time for 10000 transactions. The x-axis indicates the number of read-only, write-only, and read-write fields for each workload.**

We draw three conclusions. First, as each transaction accesses only one tuple, it accesses only one cache line. Therefore, the performance of Row Store is almost the same regardless of the number of fields

<sup>2</sup>GS-DRAM requires the database to be structured (i.e., not have any variable length fields). This is fine for most high-performance IMDBs as they handle variable length fields using fixed size pointers for fast data retrieval [19, 32]. GS-DRAM will perform at least as well as the baseline for unstructured databases.

read/written by each transaction. Second, the performance of Column Store is worse than that of Row Store, and decreases with increasing number of fields. This is because Column Store accesses a different cache line for each field of a tuple accessed by a transaction, thereby causing a large number of memory accesses. Finally, as expected, GS-DRAM performs as well as Row Store and 3X (on average) better than Column Store for the transactions workload.

**Analytics workload.** For this workload, we measure the time taken to run a query that computes the sum of  $k$  columns from the table. Figure 10 compares the performance of the three mechanisms on the analytics workload for  $k = 1$  and  $k = 2$ . The figure shows the performance of each mechanism without and with prefetching. We use a PC-based stride prefetcher [6] (with prefetching degree of 4 [44]) that prefetches data into the L2 cache. We draw several conclusions from the results.



**Figure 10: Analytics Workload Performance: Execution time for running an analytics query on 1 or 2 columns (without and with prefetching).**

First, prefetching significantly improves the performance of all three mechanisms for both queries. This is expected as the analytics query has a uniform stride for all mechanisms. Second, the performance of Row Store is roughly the same for both queries. This is because each tuple of the table fits in a single cache line and hence, the number of memory accesses for Row Store is the same for both queries (with and without prefetching). Third, the execution time of Column Store increases with more fields. This is expected as Column Store needs to fetch more cache lines when accessing more fields from the table. Regardless, Column Store significantly outperforms Row Store for both queries, as it causes far fewer cache line fetches compared to Row Store. Finally, GS-DRAM, by gathering the columns from the table as efficiently as Column Store, performs similarly to Column Store and significantly better than Row Store both without and with prefetching (2X on average).

**HTAP workload.** For this workload, we run one analytics thread and one transactions thread concurrently on the same system operating on the *same* ta-

ble. The analytics thread computes the sum of a single column, whereas the transactions thread runs transactions (on randomly chosen tuples with one read-only and one write-only field) until the analytics thread completes. We measure 1) the time taken to complete the analytics query, and 2) the transaction throughput. Figures 11a and 11b plot these results, without and with prefetching.

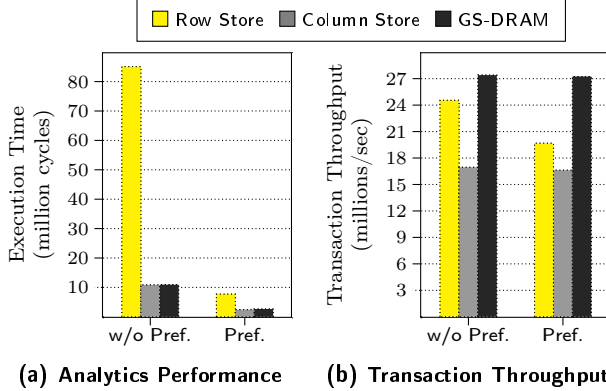


Figure 11: HTAP (without and with prefetching) (transactions: 1 read-only, 1 write-only field; analytics: 1 column)

First, for analytics, prefetching significantly improves performance for all three mechanisms. GS-DRAM performs as well as Column Store. Second, for transactions, we find that GS-DRAM not only outperforms Column Store, in terms of transaction throughput, but it also performs better than Row Store. We traced this effect back to inter-thread contention for main memory bandwidth, a well-studied problem (e.g., [17, 24, 25, 34, 35, 46]). The FR-FCFS [39, 56] memory scheduler prioritizes requests that hit in the row buffer. With Row Store, the analytics thread accesses all the cache lines in a DRAM row, thereby starving requests of the transaction thread to the same bank (similar to a memory performance hog program described in [33]). In contrast, by fetching just the required field, GS-DRAM accesses *8 times fewer* cache lines per row. As a result, it stalls the transaction thread for much smaller amount of time, leading to higher transaction throughput than Row Store. The problem becomes worse for Row Store with prefetching, since the prefetcher makes the analytics thread run even faster, thereby consuming a larger fraction of the memory bandwidth.

**Energy.** We use McPAT [29] and DRAMPower [10, 11] (integrated with Gem5 [7]) to estimate the processor and DRAM energy consumption of the three mechanisms. Our evaluations show that, for transactions, GS-DRAM consumes similar energy to Row Store and 2.1X lower than Column Store. For analytics (with prefetching enabled), GS-DRAM consumes similar energy to Column Store and 2.4X lower energy (4X without prefetching) than Row Store. (As different mechanisms

perform different amounts of work for the HTAP workload, we do not compare energy for this workload.) The energy benefits of GS-DRAM come from 1) lower overall processor energy consumption due to reduced execution time, and 2) lower DRAM energy consumption due to significantly fewer memory accesses.

Figure 12 summarizes the performance and energy benefits of GS-DRAM over Row Store and Column Store for the transaction workload and the analytics workload. We conclude that GS-DRAM provides the best of both the row store and the column store layouts.

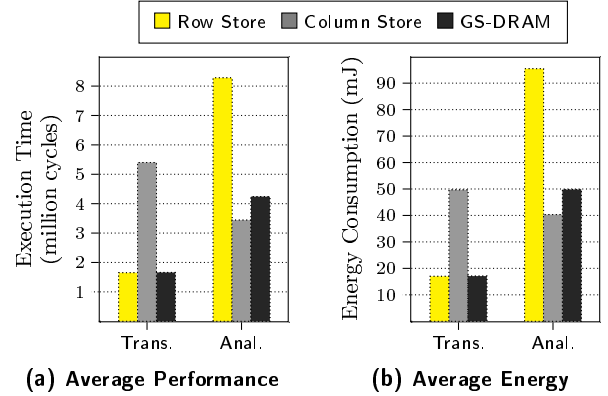


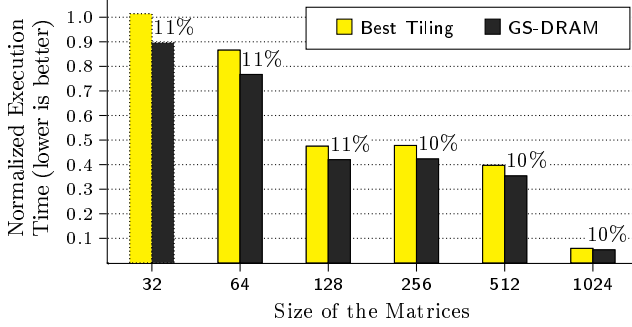
Figure 12: Summary of performance and energy consumption for the transactions and analytics workloads

## 5.2. Scientific Computation: GEMM

General Matrix-Matrix (GEMM) multiplication is an important kernel in many scientific computations. When two  $n \times n$  matrices  $A$  and  $B$  are multiplied, the matrix  $A$  is accessed in the row-major order, whereas the matrix  $B$  is accessed in the column-major order. If both matrices are stored in row-major order, a naive algorithm will result in poor spatial locality for accesses to  $B$ . To mitigate this problem, matrix libraries use two techniques. First, they split each matrix into smaller tiles, converting the reuses of matrix values into L1 cache hits. Second, they use SIMD instructions to speed up each vector dot product involved in the operation.

Unfortunately, even after tiling, values of a column of matrix  $B$  are stored in different cache lines. As a result, to exploit SIMD, the software must gather the values of a column into a SIMD register. In contrast, GS-DRAM can read each tile of the matrix in the column-major order into the L1 cache such that each cache line contains values gathered from one column. As a result, GS-DRAM naturally enables SIMD operations, without requiring the software to gather data into SIMD registers.

Figure 13 plots the performance of GEMM with GS-DRAM and with the best-performing tiled version normalized to a non-tiled version for different sizes ( $n$ )



**Figure 13: GEMM Multiplication: Performance of GS-DRAM and the best tiled-version (normalized to a non-tiled baseline). Values on top indicate percentage reduction in execution time of GS-DRAM compared to tiling.**

of the input matrices. We draw two conclusions. First, as the size of the matrices increases, tiling provides significant performance improvement by eliminating many memory references. Second, by seamlessly enabling SIMD operations, GS-DRAM improves the performance of GEMM multiplication by 10% on average compared to the best tiled baseline. Note that GS-DRAM achieves 10% improvement over a heavily-optimized tiled baseline that spends most of its time in the L1 cache.

### 5.3. Other Applications

We envision GS-DRAM to benefit many other applications like key-value stores, graph processing, and graphics. Key-value stores have two main operations: *insert* and *lookup*. The *insert* operation benefits from both the key and value being in the same cache line. On the other hand, the *lookup* operation benefits from accessing a cache line that contains only keys. Similarly, in graph processing, operations that update individual nodes in the graph have different access patterns than those that traverse the graph. In graphics, multiple pieces of information (e.g., RGB values of pixels) may be packed into small objects. Different operations may access multiple values within an object or a single value across a large number of objects. The different access patterns exhibited by these applications have a regular stride and can benefit significantly from GS-DRAM.

## 6. Extensions to GS-DRAM

In this section, we describe three simple extensions to GS-DRAM: 1) programmable shuffling, 2) wider pattern IDs, and 3) intra-chip column translation. These extensions (together or individually) allow GS-DRAM to 1) express more patterns (e.g., larger strides), 2) gather or scatter data at a granularity smaller than 8 bytes, and 3) enable ECC support.

### 6.1. Programmable Shuffling

Although our shuffling mechanism uses the least significant bits of the column ID to control the shuffling stages, there are two simple ways of explicitly controlling which shuffling stages are active. First, we can use a *shuffle mask* to disable some stages. For example, the shuffle mask 10 disables swapping of adjacent values (Figure 4, Stage 1). Second, instead of using the least significant bits to control the shuffling stages, we can choose different combinations of bits (e.g., XOR of multiple sets of bits [14, 48]). To enable programmable shuffling, we add another parameter to GS-DRAM called the *shuffling function*,  $f$ . For GS-DRAM <sub>$c,s,p,f$</sub> , the function  $f$  takes a column ID as input and generates an  $n$ -bit value that is used as the control input to the  $n$  shuffling stages. The function  $f$  can be application-specific, thereby optimizing GS-DRAM for each application.

### 6.2. Wider Pattern IDs

Although a wide pattern ID comes at additional cost, using a wider pattern ID allows the memory controller to express more access patterns. However, the column translation logic (CTL) performs a bitwise AND of the chip ID and the pattern ID to create a modifier for the column address. As a result, even if we use a wide pattern ID, a small chip ID disables the higher order bits of the pattern ID. Specifically, for GS-DRAM <sub>$c,s,p$</sub> , if  $p > \log c$ , the CTL uses only the least significant  $\log c$  bits of the pattern ID. To enable wider pattern IDs, we propose to simply widen the chip ID used by the CTL by repeating the physical chip ID multiple times. For instance, with 8 chips and a 6-bit pattern ID, the chip ID used by CTL for chip 3 will be 011-011 (i.e., 011 repeated twice). With this simple extension, GS-DRAM can enable more access patterns (e.g., larger strides).

### 6.3. Intra-Chip Column Translation

Although we have assumed that each DRAM bank has a single wide row-buffer, in reality, each DRAM bank is a 2-D collection of multiple small *tiles* or MATs [26, 47, 54]. Similar to how each chip within a rank contributes 64 bits to each cache line, each tile contributes equally to the 64 bits of data supplied by each chip. We can use the column translation logic within each DRAM chip to select different columns from different tiles for a single READ or WRITE. This mechanism has two benefits. First, with the support for intra-chip column translation, we can gather access patterns at a granularity smaller than 8 bytes. Second, with DIMMs that support ECC, GS-DRAM may incur additional bandwidth to read all the required ECC values for non-zero patterns. However, if we use a chip that supports intra-chip column selection for ECC, accesses with non-zero patterns can gather the data from the eight data chips and gather the

ECC from the eight tiles within the ECC chip, thereby seamlessly supporting ECC for all access patterns.

## 7. Related Work

To our knowledge, this is the first work to exploit DRAM architecture to improve the spatial locality of strided accesses and provide efficient gather/scatter support by introducing simple data mapping and logic changes. In this section, we compare GS-DRAM to several prior works that propose mechanisms to improve the performance of such strided and gather/scatter accesses.

Carter et al. [9] propose Impulse, a mechanism to export gather operations to the memory controller. In their system, applications specify a *gather mapping* to the memory controller (with the help of the OS). To perform a gather access, the controller assembles a cache line with only the values required by the access pattern and sends the cache line to the processor, thereby reducing the bandwidth between the memory controller and the processor. Impulse has two shortcomings. First, with commodity DRAM modules, which are optimized for accessing cache lines, Impulse cannot mitigate the wasted memory bandwidth consumption between the memory controller and DRAM. Impulse requires a memory interface that supports fine-grained accesses (e.g., [4, 5, 8, 50, 55]), which significantly increases the system cost. Second, Impulse punts the problem of maintaining cache coherence to software. In contrast, our mechanism 1) works with commodity DRAM modules with very few changes, and 2) provides coherence of gathered cache lines transparent to software.

Yoon et al. [51, 52] propose the Dynamic Granularity Memory System (DGMS), a memory interface that allows the memory controller to dynamically change the granularity of memory accesses in order to avoid unnecessary data transfers for accesses with low spatial locality. Similar to Impulse, DGMS requires a memory interface that supports fine-grained memory accesses (e.g., [4, 5, 8, 50, 55]) and a sectored cache [30, 42]. In contrast, GS-DRAM works with commodity DRAM modules and non-sectored caches with very few changes.

Prior works (e.g., [6, 15, 16, 36, 37, 44]) propose prefetching for strided accesses. While prefetching reduces the latency of such accesses, it does not avoid the waste in memory bandwidth and cache space. He et al. [20] propose a model to analyze the performance of gather-scatter accesses on a GPU. To improve cache locality, their model splits gather-scatter loops into multiple passes such that each pass performs only accesses from a small group of values that fit in the cache. This mechanism works only when multiple values are *actually* reused by the application. In contrast, GS-DRAM fetches

*only* useful values from DRAM, thereby achieving better memory bandwidth and cache utilization.

## 8. Conclusion

We introduced *Gather-Scatter DRAM*, a low-cost substrate that enables the memory controller to efficiently gather or scatter data with different non-unit strided access patterns. Our mechanism exploits the fact that multiple DRAM chips contribute to each cache line access. GS-DRAM maps values accessed by different strided patterns to different chips, and uses a per-chip column translation logic to access data with different patterns using significantly fewer memory accesses than existing DRAM interfaces. Our framework requires no changes to commodity DRAM chips, and very few changes to the DRAM module, the memory interface, and the processor architecture. Our evaluations show that GS-DRAM provides the best of both the row store and the column store layouts for a number of in-memory database workloads, and outperforms the best tiled layout on a well-optimized matrix-matrix multiplication workload. Our framework can benefit many other modern data-intensive applications like key-value stores and graph processing. We conclude that the GS-DRAM framework is a simple and effective way to improve the performance of non-unit strided and gather/scatter memory accesses.

## Acknowledgments

We thank the anonymous reviewers and the members of the SAFARI group for their feedback. We acknowledge the generous support of Intel, Google, Nvidia, and Samsung. This work is supported in part by NSF grants 0953246, 1212962, 1320531, and 1409723, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

## References

- [1] Intel 64 and IA-32 Architectures Software Developer's Manual. <http://download.intel.com/design/processor/manuals/253665.pdf>, Vol. 1, Chap. 3.7.
- [2] Alpha 21264 Microprocessor Hardware Reference Manual. <http://h18000.www1.hp.com/cpq-alphaserver/technology/literature/21264hrm.pdf>, 1999.
- [3] Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. <https://www.gartner.com/doc/2657815/hybrid-transactionanalytical-processing-foster-opportunities>, 2014.
- [4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future Scaling of Processor-memory Interfaces. In *SC*, 2009.
- [5] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore DIMM: An Energy Efficient Memory Module with Independently Controlled DRAMs. *IEEE CAL*, January 2009.
- [6] J.-L. Baer and T.-F. Chen. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE TC*, 1995.

- [7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
- [8] T. M. Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro*, 30(2):70–79, March 2010.
- [9] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA*, 1999.
- [10] K. Chandrasekar, B. Akesson, and K. Goossens. Improved Power Modeling of DDR SDRAMs. In *DSD*, 2011.
- [11] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, , and K. Goossens. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. <http://www.drampower.info>.
- [12] K. K.-W. Chang, D. Lee, Z. Chisti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.
- [13] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [14] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A Flexible Data Organization in Parallel Memories. In *ICPP*, 1985.
- [15] J. W. C. Fu and J. H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *ISCA*, 1991.
- [16] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO*, 1992.
- [17] S. Ghose, H. Lee, and J. F. Martinez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *ISCA*, 2013.
- [18] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010.
- [19] H-Store. Anti-Caching. <http://hstore.cs.brown.edu/documentation/deployment/anti-caching/>, 2015.
- [20] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient Gather and Scatter Operations on Graphics Processors. In *SC*, 2007.
- [21] JEDEC. DDR4 SDRAM Standard. <http://www.jedec.org/standards-documents/docs/jesd79-4a>, 2013.
- [22] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *VLDB*, 2008.
- [23] B. Keeth, R. J. Baker, B. Johnson, and F. Lin. *DRAM Circuit Design: Fundamental and High-speed Topics*. Wiley-IEEE Press, 2007.
- [24] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.
- [25] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.
- [26] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM. In *ISCA*, 2012.
- [27] D. Lee, Y. Kim, G. Pechhimenko, S. Khan, V. Seshadri, K. K.-W. Chang, and O. Mutlu. Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*, 2015.
- [28] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.
- [29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*, 2009.
- [30] J. S. Liptay. Structural Aspects of the System/360 Model 85: II the Cache. *IBM Syst. J.*, 7(1):15–21, March 1968.
- [31] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.
- [32] MemSQL. Datatypes. <http://docs.memsql.com/4.0/ref/datatypes/>, 2015.
- [33] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In *SS*, 2007.
- [34] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.
- [35] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.
- [36] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *HPCA*, 2004.
- [37] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers As a Secondary Cache Replacement. In *ISCA*, 1994.
- [38] B. R. Rau. Pseudo-randomly Interleaved Memory. In *ISCA*, 1991.
- [39] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.
- [40] V. Seshadri, A. Bhowmick, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry. The Dirty-block Index. In *ISCA*, 2014.
- [41] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pechhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.
- [42] A. Sez nec. Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost. In *ISCA*, 1994.
- [43] N. Shamgunov. The MemSQL In-Memory Database System. In *VLDB*, 2014.
- [44] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.
- [45] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. In *VLDB*, 2005.
- [46] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost. In *ICCD*, 2014.
- [47] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramanian, A. Davis, and N. P. Jouppi. Rethinking DRAM Design and Organization for Energy-constrained Multi-cores. In *ISCA*, 2010.
- [48] M. Valero, T. Lang, and E. Ayguadé. Conflict-free access of vectors with power-of-two strides. In *ICS*, 1992.
- [49] T. Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*, 2010.
- [50] F.A. Ware and C. Hampel. Improving Power and Data Efficiency with Threaded Memory Modules. In *ICCD*, 2006.
- [51] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive Granularity Memory Systems: A Tradeoff Between Storage Efficiency and Throughput. In *ISCA*, 2011.
- [52] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The Dynamic Granularity Memory System. In *ISCA*, 2012.
- [53] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse Memory Controller. *IEEE TC*, November 2001.
- [54] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie. Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation. In *ISCA*, 2014.
- [55] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *MICRO*, 2008.
- [56] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. Patent 5630096, 1997.