FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching

Yaohua Wang[⋆] Lois Orosa[†] Xiangjun Peng[⊙⋆] Yang Guo[⋆] Saugata Ghose[◇‡] Minesh Patel[†]
Jeremie S. Kim[†] Juan Gómez Luna[†] Mohammad Sadrosadati[§] Nika Mansouri Ghiasi[†] Onur Mutlu[†‡]

[⋆]*National University of Defense Technology* [†]*ETH Zürich* [⊙]*Chinese University of Hong Kong*
[◇]*University of Illinois at Urbana–Champaign* [‡]*Carnegie Mellon University* [§]*Institute of Research in Fundamental Sciences*

Main memory, composed of DRAM, is a performance bottleneck for many applications, due to the high DRAM access latency. In-DRAM caches work to mitigate this latency by augmenting regular-latency DRAM with small-but-fast regions of DRAM that serve as a cache for the data held in the regular-latency (i.e., slow) region of DRAM. While an effective in-DRAM cache can allow a large fraction of memory requests to be served from a fast DRAM region, the latency savings are often hindered by inefficient mechanisms for migrating (i.e., relocating) copies of data into and out of the fast regions. Existing in-DRAM caches have two sources of inefficiency: (1) their data relocation granularity is an entire multi-kilobyte row of DRAM, even though much of the row may never be accessed due to poor data locality; and (2) because the relocation latency increases with the physical distance between the slow and fast regions, multiple fast regions are physically interleaved among slow regions to reduce the relocation latency, resulting in increased hardware area and manufacturing complexity.

We propose a new substrate, FIGARO, that uses existing shared global buffers among subarrays within a DRAM bank to provide support for in-DRAM data relocation across subarrays at the granularity of a single cache block. FIGARO has a distance-independent latency within a DRAM bank, and avoids complex modifications to DRAM (such as the interleaving of fast and slow regions). Using FIGARO, we design a fine-grained in-DRAM cache called FIGCache. The key idea of FIGCache is to cache only small, frequently-accessed portions of different DRAM rows in a designated region of DRAM. By caching only the parts of each row that are expected to be accessed in the near future, we can pack more of the frequently-accessed data into FIGCache, and can benefit from additional row hits in DRAM (i.e., accesses to an already-open row, which have a lower latency than accesses to an unopened row). FIGCache provides benefits for systems with both heterogeneous DRAM banks (i.e., banks with fast regions and slow regions) and *conventional homogeneous DRAM banks (i.e., banks with only slow regions).*

Our evaluations across a wide variety of applications show that FIGCache improves the average performance of a system using DDR4 DRAM by 16.3% and reduces average DRAM energy consumption by 7.8% for 8-core workloads, over a conventional system without in-DRAM caching. We show that FIGCache outperforms state-of-the-art in-DRAM caching techniques, and that its performance gains are robust across many system and mechanism parameters.

1. Introduction

DRAM has long been the dominant technology for main memory systems. As many modern applications require greater amounts of DRAM to hold increasing amounts of data, manufacturers are increasing the capacity of DRAM chips via manufacturing process technology scaling. However, unlike capacity, DRAM access latency has not decreased significantly for decades, as latency improvements are traded off to instead decrease the cost-per-bit of DRAM [14, 80, 81, 106, 111, 134]. To understand why, we study the high-level organization of a DRAM chip, as shown in Figure 1. The chip consists of multiple DRAM *banks* (eight in DDR4 DRAM [52]), where each bank is comprised of multiple homogeneous *subarrays* (i.e., two-dimensional tiles) [72] of DRAM cells. Each DRAM cell stores a bit of data in the form of charge. Reads and writes cannot be performed directly on the cell, as the cell holds only a limited amount of charge (in order to keep the cell area small), and this amount is too small to drive the I/O circuitry. Instead, a cell in a subarray is connected via a *bitline* to the subarray's *local row buffer* (consisting of sense amplifiers) [72, 82]. A local row buffer is used to sense, amplify, and hold the contents of one row of DRAM. Each subarray has its own local row buffer, which consumes a relatively large area compared to a row of DRAM cells. To amortize this area and achieve low cost-per-bit, a commodity DRAM connects many DRAM cells to each sense amplifier on a single bitline (e.g., 512–2048 cells per bitline). Doing so results in a long bitline to accommodate the number of attached DRAM cells, and a long bitline has high parasitic capacitance. Bitline capacitance has a direct impact on DRAM access latency: the longer the bitline, the higher the parasitic capacitance, and, thus, the longer the latency required to bring the data from a row of DRAM cells into the local row buffer [81]. The local row buffers in a bank are connected to a shared *global row buffer*, which interfaces with the chip's I/O drivers.

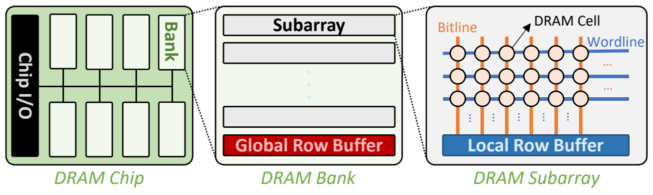
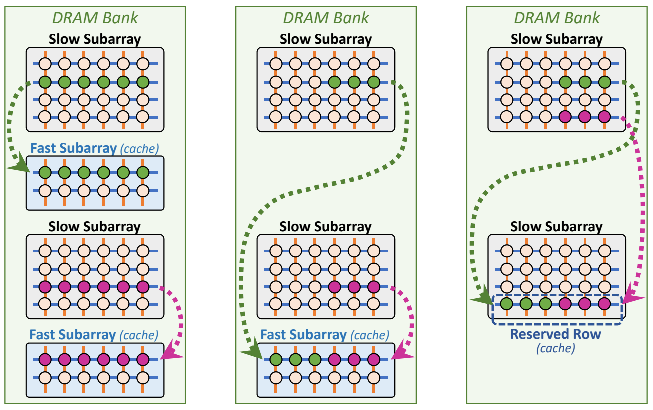


Figure 1. Logical organization of a DRAM chip.

To improve DRAM latency while maintaining low cost-per-bit, prior works modify the DRAM organization to implement an *in-DRAM cache* [15, 81, 94, 134]. Many of these works take the approach shown in Figure 2a, where they introduce *heterogeneous subarrays* into DRAM banks. In such a bank, one type of subarray (labeled a *slow subarray* in the figure) has a regular (i.e., slow) access latency and large capacity, while a second type of subarray (labeled *fast subarray*) has a low access latency but small capacity (i.e., the subarray's bitlines are kept short to reduce parasitic capacitance and, thus, latency). An in-DRAM cache maintains a copy of a subset of rows from the slow subarrays in the fast subarrays, typically caching the *hottest* (i.e., most frequently accessed, or most recently used) rows to increase the probability that



(a) State-of-the-Art In-DRAM Cache (b) FIGCache Using Fast Subarrays (c) FIGCache Using Slow Subarrays

Figure 2. (a) State-of-the-art in-DRAM cache in a heterogeneous bank with many fast subarrays interleaved among slow subarrays; (b) FIGCache in a heterogeneous bank with fewer fast subarrays; (c) FIGCache in a conventional bank with no fast subarrays.

a memory request can be served by the fast subarrays (i.e., with low latency).

Unfortunately, existing in-DRAM cache designs suffer from the inefficient data relocation operations that copy data between the slow and fast subarrays. There are two main reasons for this inefficiency. First, *the data relocation granularity is the size of an entire DRAM row*. In modern DRAM, a row contains a large amount of data (8 kB in DDR4 [52]). However, it is difficult for an application to access the entire contents of a row when the row is opened, as (1) the application may not have high spatial locality [3, 4, 34, 145], and (2) interference among multiple programs running on a multicore processor limits data reuse from an open row [33, 35, 40, 70, 71, 84, 99, 102, 104, 110, 140, 141, 143, 144, 147, 155]. As a result, when a row in an in-DRAM cache is opened, only a small subset of the cached row is typically accessed before the row is evicted from the cache. Second, *there is a trade-off in current designs between relocation latency and design complexity*. The further away a slow subarray is physically from the fast subarray, the higher the latency that is required to perform the data relocation. To reduce the relocation latency, *many* fast subarrays are employed and *interleaved* among slow subarrays (as shown in Figure 2a), which leads to increased area overhead (e.g., each fast subarray requires its own local row buffer and peripheral circuitry [72, 82]) and manufacturing complexity.

To avoid these inefficiencies, we propose a new approach to efficient data relocation support across subarrays within a DRAM bank that uses (mostly) existing structures within a modern DRAM device. As shown in Figure 1, all of the subarrays in a bank share a single *global row buffer*. The global row buffer in a bank serves to connect one column's worth of data from an active local row buffer in the same bank to the I/O drivers in a chip. Across a *rank* of chips (i.e., a group of chips operating in lockstep), the global row buffers of a single bank can hold one cache line (i.e., 64 bytes) of data. We make the **key observation** that the global row buffer in a DRAM bank is interconnected with *all* of the local row buffers of the subarrays in the bank. By safely relaxing some constraints in the operation of the DRAM chip, we can use the global row buffer to facilitate fine-grained relocation across subarrays (i.e., copying only a single column of data in a DRAM chip, which translates to copying a cache block in a DRAM rank). Using this insight, we design a substrate called *FIGARO*. FIGARO operations are performed by enabling two local row buffers to transfer data in an *unaligned* manner between each other (i.e., data from one column in the source local row buffer can be written to a *different* column in the destination local row buffer) via the global row buffer, without any use of the off-chip memory channel. By making novel use of existing structures within DRAM, we implement FIGARO with only modest changes to the peripheral logic within DRAM (<0.3% chip area overhead), without any changes to the cell arrays.

Based on FIGARO, we propose a fine-grained in-DRAM cache (*FIGCache*), as shown in Figure 2b. FIGCache avoids the pitfalls of state-of-the-art in-DRAM cache designs [15, 81, 94, 134]. The *key idea* of FIGCache is to cache only a portion of a DRAM row (i.e., a *row segment*) via FIGARO, instead of caching an entire DRAM row at a time. This *row segment granularity* based caching approach yields three benefits. First, it increases the performance of in-DRAM caches, because a single in-DRAM cache row (in the fast subarray) can now contain small fragments of *multiple* DRAM rows that are likely to be accessed before the fragment is evicted from the cache. By significantly reducing the amount of cache space wasted on unaccessed data, both the in-DRAM cache hit rate and row buffer hit rate increase substantially. Second, it simplifies the in-DRAM cache design. FIGARO has a *distance-independent* relocation latency within a DRAM bank, reducing the number of fast subarrays needed to keep the latency low compared to state-of-the-art in-DRAM caches (e.g., FIGCache provides benefits with only two fast subarrays per bank) and eliminating the need to interleave fast subarrays among slow subarrays. Third, it allows in-DRAM caching to provide potential benefit for conventional DRAM chips that contain only slow subarrays (as shown in Figure 2c). Even without subarrays with lower access latencies, FIGARO allows us to use a small number of rows in a slow subarray to contain the most frequently-accessed fragments of select DRAM rows. This increases the row buffer hit rate significantly, allowing a greater fraction of memory requests to be served with the lower row hit latency (as a row already open in a row buffer has a lower access latency than an unopened row). As we discuss in Section 6, FIGCache can help mitigate security attacks such as RowHammer [21, 29, 68, 69, 107, 108] and row buffer based side channel attacks [118], in addition to its performance benefits.

Our evaluations show that on a system with both fast subarrays and slow subarrays (Figure 2b), FIGCache improves performance by 16.3% and reduces DRAM energy by 7.8%, on average across 20 eight-core workloads, over a conventional system without in-DRAM caching. FIGCache outperforms a state-of-the-art in-DRAM cache design [15], with an average performance improvement of 4.6% for 8-core workloads. We show that even in a system *without* any fast subarrays (Figure 2c), if we reserve 64 of the DRAM rows in a slow subarray as an in-DRAM cache, FIGCache provides considerable performance gain (12.5% on average). We demonstrate that the performance benefits of FIGCache are robust across many system and mechanism parameters (e.g., cache capacity, caching granularity, replacement policy, hot data identification policy). We conclude that FIGCache is a robust and efficient mechanism to reduce DRAM access latency.

We make the following contributions in this work:

- We propose FIGARO, an efficient substrate that enables fine granularity (i.e., column granularity) data relocation across subarrays in a memory bank, at a latency that is independent of the distance of subarrays from each other. FIGARO uses (mostly) existing structures in a modern DRAM chip, with its modifications requiring <0.3% chip area overhead.

- We propose FIGCache, an efficient in-DRAM cache based on FIGARO. FIGCache caches fragments of a DRAM row at the granularity of a row segment, which can be as small as a cache block. Doing so significantly improves in-DRAM caching performance over state-of-the-art in-DRAM caches. Unlike prior works, FIGCache can be implemented in DRAM chips with both heterogeneous (i.e., slow and fast) subarrays and homogeneous (i.e., only slow) subarrays.
- We comprehensively evaluate the performance and energy efficiency of FIGCache. We show that it substantially improves both the performance and energy efficiency of single-core and multi-core systems with DDR4 DRAM, and that it outperforms state-of-the-art in-DRAM caches.

2. Background

We provide background about DRAM organizations and operations to understand how FIGARO works. For more information, we refer the readers to prior works that cover DRAM in detail [14, 15, 16, 33, 39, 40, 41, 61, 62, 63, 65, 67, 72, 73, 74, 80, 81, 82, 88, 89, 97, 126, 127, 130, 131, 147].

DRAM Organization. A modern main memory subsystem consists of one or more memory *channels*, where each channel contains a *memory controller* that manages a dedicated subset of DRAM modules. The modules in a single channel share an off-chip bus that is used to issue commands and transfer data between DRAM modules and memory controller, which typically resides in the processor. Each module is made up of multiple DRAM chips, which are grouped into one or more *ranks*. For modern x8 DRAM chips in the same rank, there are typically 8 chips that hold data (with some modules containing an additional chip for error-correcting codes, or ECC [58, 98, 115, 116]). All chips belonging to the same rank operate in *lockstep* (i.e., the same command is issued and performed by all chips simultaneously), and one row of DRAM cells are distributed across all of the chips within a rank. The chips in a rank, in combination, provide 64 bytes of data (and 8 bytes of ECC code for modules with the extra chip) for each memory request. As Figure 1 shows, a chip is divided into multiple *banks*, which can serve memory requests (i.e., loads or stores) in parallel and independently of each other. Each bank typically consists of 32–64 two-dimensional arrays of DRAM cells called *subarrays* [17, 72, 127, 128, 131].

In this work, we focus on data movement operations *across* subarrays within a bank. Figure 3 provides more detail about the subarray structure. Each subarray typically contains 512–2048 rows of DRAM cells, which are connected to a *local row buffer* (LRB). The LRB consists of a set of *sense amplifiers* that are used to open (i.e., activate) one row at a time in the subarray. Each vertical line of cells is connected to one sense amplifier in the LRB via a *local bitline* wire. Cells within a row share a *wordline*. All of the LRBs in a bank are connected to a shared *global row buffer* (GRB) [15, 48, 64, 72, 101], which is much narrower than the LRB. The GRB is connected to the LRBs using a set of *global bitlines* [48, 82]. The GRB is composed of high-gain sense amplifiers that detect and amplify perturbations caused by a single LRB on the global bitlines [48]. The GRB width is usually correlated with the data output width of the chip (e.g., in an x8 data/ECC chip, which sends 8 bits of data/ECC for each of the eight data bursts that make up one read, the GRB is 64-bit). Since the GRB is much narrower than an LRB, a single *column* (i.e., a small number of bits; 64 in an x8 chip) of the LRB is selected

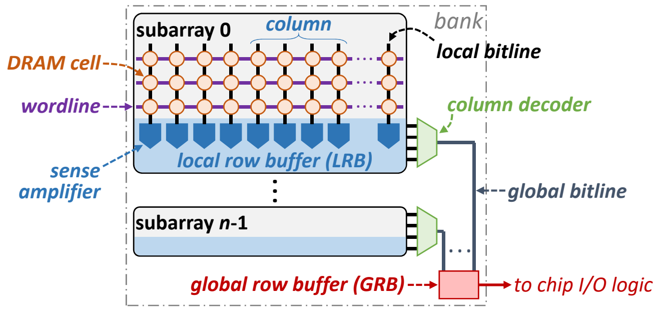


Figure 3. Detailed DRAM bank and subarray organization.

using a *column decoder* to connect to the GRB. The column is chosen based on the memory address requested by the DRAM command that is being performed.

DRAM Operations. The memory controller issues four commands to access and update data within DRAM. First, the memory controller *activates* the DRAM row containing the data. The ACTIVATE command latches the selected DRAM row into the LRB of the subarray that contains the row. Second, once the activation finishes, the memory controller issues a READ or WRITE command, which operates on a column of data. On a READ, one column of the LRB is selected using the column decoder and is sent to the GRB via global bitlines. The GRB then drives the data to the chip I/O logic, which sends the data out of the DRAM chip to the memory controller. While a row is activated, the memory controller can issue subsequent READ/WRITE commands to access other columns of data from the LRB if there are other memory requests to the same row. This is called a *row buffer hit*. Finally, the controller *precharges* the LRB and the subarray by issuing a PRECHARGE command to prepare all of the bitlines for a subsequent ACTIVATE command to a different row.

The latencies of the above commands is governed by timing parameters defined in an industry-wide standard [49, 51, 52, 72, 76, 81], which is set largely depending on the length of local bitlines in the subarray. This is because every local bitline has an associated parasitic capacitance whose value is proportional to the length of the bitline. This parasitic capacitance increases the subarray operation latencies during ACTIVATE and PRECHARGE [81].

3. Existing In-DRAM Cache Designs

DRAM manufacturers often choose a long bitline length to accommodate a greater number of rows (and, thus, increase DRAM capacity) [81]. To alleviate long DRAM latencies that result from longer bitlines, prior works propose in-DRAM caches [15, 81, 94, 134]. The key idea of an in-DRAM cache is to introduce heterogeneity into DRAM, where one region has a fast access latency with short local bitline length, while the other has bitline length and access latency same as regular (i.e., slow) DRAM. To yield the highest performance benefits, the fast region is used as an in-DRAM cache for hot data. We discuss three main approaches that prior works take in building in-DRAM caches.

Heterogeneous Subarray Based Design. Tiered-Latency (TL) DRAM [81] divides a subarray into fast (near) and slow (far) segments that have short and long bitlines, respectively, by adding bitline isolation transistors between the two segments. The fast segment can serve as an in-DRAM cache. A row can be quickly copied between the two segments via the bitlines, via a back-to-back activation operation resembling RowClone [127]. The main disadvantage of TL-DRAM [81] comes from the intrusive nature of the bitline isolation transistors inside the subarray. Isolation transistors are different from the existing cell access transistors in DRAM, which are specially designed with low leakage [94, 123]. When placed in the middle of a subarray, isolation transistors require large cost and can affect DRAM yield. For DRAM chips that use the popular open-bitline architecture [60, 88, 97], TL-DRAM increases the area overhead significantly (by 3.15%) [15]. As a result, such isolation-transistor-based in-DRAM cache designs can potentially face a relatively high barrier to adoption by commercial DRAM vendors.

Heterogeneous Bank Based Design Without Data Relocation Support. CHARM [134] introduces heterogeneity *within* each *bank* by designing a few fast subarrays with (1) short bitlines for faster data sensing, and (2) close placement to the chip I/O for faster data transfer. Fast subarrays maintain the same cell array structure as traditional DRAM, leading to simple design and with little effect on DRAM yield. To fully exploit the potential of fast subarrays, CHARM uses an OS-based scheme to statically allocate frequently used data to fast subarrays based on program profiling information. The main shortcoming of CHARM is that data relocation between fast and slow subarrays must be done through the memory channel using the narrow global data bus of DRAM, which incurs high latency and reduces opportunities to use *dynamic* in-DRAM cache management polices that adapt to dynamic program phase changes (and that requires more frequent data relocation than static profiling-based policies). This substantially limits the potential benefits of CHARM, and makes the overall performance gain of in-DRAM cache depend heavily on the effectiveness of the static, profiling based cache management policy.

Heterogeneous Bank Based Design With Bulk Data Relocation Support. By taking advantage of DRAM structures, DAS-DRAM [94] and LISA-VILLA [15] extend the functionality of CHARM [134] with in-DRAM bulk data relocation mechanisms. These mechanisms dynamically relocate data between fast and slow subarrays without using the narrow global data bus, enabling faster and more efficient relocation. This allows for the efficient implementation of dynamic in-DRAM cache management policies. Specifically, DAS-DRAM enables DRAM row relocation across subarrays in a bank through a row of relocation cells in each subarray. The LISA substrate [15] (upon which the LISA-VILLA in-DRAM caching mechanism is built) further improves the relocation latency with wide inter-subarray links, serving as a direct data relocation path between especially physically-adjacent subarrays. Unfortunately, the overall performance of state-of-the-art in-DRAM caches is greatly limited by two characteristics of the existing in-DRAM data relocation support.

First, the data relocation granularity is large and fixed (i.e., an entire DRAM row is relocated at a time). Due to the limited row buffer locality exhibited by many programs [33], most of the in-DRAM cache hits are actually to only a small subset of a cached DRAM row, leaving the rest of the DRAM row untouched before the row is evicted from the cache (i.e., most of the row is brought into the cache without providing any benefit). The interference among concurrently running programs in a multicore system further hurts the row buffer locality [33, 35, 40, 70, 71, 84, 99, 102, 104, 110, 140, 141, 143, 144, 147, 155]. Thus, caching an entire DRAM row is usually not necessary and leads to poor utilization of the in-DRAM cache (i.e., fast subarray) space. Note that while a cached row can take advantage of low latencies in the fast subarray, its row buffer hit rate does not change, as the contents of the cached row (and therefore its locality behavior) remain the same as the source row in the slow subarray.

Second, data relocation latency increases substantially as the physical relocation distance increases. Each relocation requires the relocated row to be written to each intermediate subarray between the source subarray and the destination subarray. As a result, the further away a slow subarray is physically from the fast subarray, the higher the latency is for the data relocation into and out of the in-DRAM cache. To mitigate this distance-dependent latency, both DAS-DRAM and LISA-VILLA add *multiple* fast subarrays into DRAM banks, physically interleaving the fast subarrays among slow subarrays to reduce the average distance between a slow subarray and its closest fast subarray. Doing so greatly increases the area overhead (e.g., each new subarray requires additional peripheral circuitry, such as decoders and a local row buffer) and manufacturing complexity.

As a result, while DAS-DRAM and LISA-VILLA represent the state-of-the-art for in-DRAM caches, their inefficiencies significantly impact the benefits and practicality of the mechanisms.

4. FIGARO Substrate

To solve the inefficiencies of state-of-the-art in-DRAM cache designs, we propose *Fine-Grained In-DRAM Data Relocation* (FIGARO), a new substrate that enables fine granularity data relocation across the subarrays in a bank at a distance-independent latency. FIGARO can relocate data at the column granularity in a bank (i.e., 64 bits in an x8 DRAM chip, which corresponds to 64-byte cache block granularity in a rank). FIGARO significantly improves in-DRAM caching in two ways: (1) it enables caching at the granularity of what we call a *row segment* (consisting of one or more contiguous cache blocks), and thus a single in-DRAM cache row can now contain row segments from multiple DRAM rows, leading to higher cache utilization and higher row buffer hit rates; and (2) it reduces the need for a large number of fast subarrays per bank (e.g., we use only two for our default configuration in this paper), and they no longer need to be interleaved among normal subarrays, leading to low area overhead and low manufacturing complexity.

4.1. FIGARO Design

FIGARO is built upon the key observation (as we discuss in Section 2) that all of the private per-subarray local row buffers (LRBs) in a bank are connected to a single shared global row buffer (GRB). By taking advantage of this connectivity, FIGARO can perform column-granularity data relocation across subarrays at a distance-independent latency, without using the off-chip memory channel.

Transferring Data Between Two Local Row Buffers. To relocate data, FIGARO introduces a new DRAM command, `RELOC` (*relocate column*). Within a DRAM chip, `RELOC` copies one column of data from the LRB of one subarray to the LRB of another subarray within the same bank, via the GRB. Recall from Section 2 that as eight x8 data chips work together in lockstep in a rank, the GRB across all chips is 64 bytes, and, thus, a `RELOC` command in such a rank-based system copies one cache block (i.e., 64 bytes). The `RELOC` command has two parameters: (1) the source address and (2) the destination address. The source address in `RELOC` consists of only the column address (since the source is an LRB containing an already-activated row, as we describe below), while the destination address consists of the destination subarray index (since the destination is a not-yet-active LRB), and the corresponding column address. Note that our new command can be easily added by using one of the undefined encodings reserved for future use in the DRAM standard [52], similar to new DRAM commands that have been proposed in previous studies [15, 72, 127, 128].

Figure 4 illustrates how FIGARO relocates one column of data from a source subarray (subarray A) to a destination subarray (subarray B) in a DRAM chip using the RELOC command. First, the memory controller issues an ACTIVATE command to one row in subarray A (❶ in Figure 4), which copies data from the selected row to subarray A's local row buffer (LRB). Second, the memory controller issues a RELOC command. The RELOC command relocates one column of data (A3 in Figure 4) from subarray A's LRB to subarray B's LRB. To do this, RELOC selects the desired column of data from subarray A's LRB (column 3) using A's column decoder (❷), which loads the column into the global row buffer (GRB; ❸), and at the same time connects the GRB to subarray B using B's column decoder, which places the column of data from subarray A in the correct column (column 1) of subarray B's LRB (❹). Multiple RELOC commands can be issued at this point, copying multiple columns of data from the activated source row to subarray B's LRB. Third, the memory controller issues an ACTIVATE command to subarray B (❺), overwriting only the corresponding column in the activated row with the new data (i.e., A3). Fourth, the memory controller issues a PRECHARGE command to prepare the entire bank for future accesses (not shown in the figure).

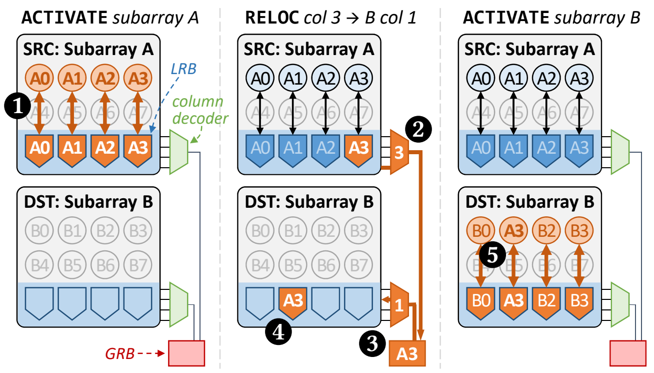


Figure 4. An example of data relocation using FIGARO.

During the RELOC command, FIGARO relies on the fact that the GRB has a higher drive strength than the LRB [48]. Therefore, when the destination LRB is connected to the GRB (❸ in Figure 4), the GRB has enough drive strength to induce charge perturbation to the idle (i.e., precharged) bitlines of the destination subarray, allowing the destination LRB to sense and latch this perturbation even though we are not activating the destination subarray. The GRB will also help to quickly drive the corresponding local sense amplifiers and the bitlines in the destination subarray to a stable state (either V_{dd} or 0). Therefore, when the destination row is activated, the DRAM cells connected to the bitlines in a stable state will be overwritten, while all other cells in the row will maintain their original values [15, 81, 127] (as seen in ❺ in Figure 4). This requires *no* modification to existing DRAM. Note that for DRAM modules that contain an additional chip for ECC information, since the data chips and ECC chip operate in lockstep (Section 2), the corresponding ECC code is transferred together with the data during the relocation process.

Distance-Independent Latency of RELOC. The latency of existing data relocation substrates in a DRAM bank [15, 94] is distance-dependent because these substrates perform time-consuming sensing of intermediate local row buffers during data relocation, as the data moves from the LRB of one subarray to the next. As a result, the relocation latency (and energy) is directly dependent on the number of intermediate local row buffer operations. Unlike existing relocation substrates that use local bitlines and isolation transistors for data movement [15, 94], the latency of RELOC comes mainly from the sensing of the GRB and the driving of the destination LRB, both via the global bitlines. While the latency difference between relocation operations to different subarrays is dependent on the length of global bitlines, a longer global bitline length has a relatively small impact on the relocation latency, as global bitlines are made of metal with lower capacitance and resistance than local bitlines [134].

Similar to standard READ/WRITE operations in DRAM, whose latencies are set to accommodate worst-case accesses to the furthest subarray, we set the RELOC latency based on the worst case (i.e., the latency of relocating data between two subarrays that are the furthest away from each other when connected via the global row buffer). We use this worst-case latency (plus a safety margin; see Section 4.2) as the timing parameter for RELOC.

Issuing Multiple Activations Without a Precharge. To activate *src* and *dst* rows one after another *without* precharging either row, we must relax the existing constraint that only one row in a bank can be active at a time. Existing memory controllers do *not* allow another ACTIVATE command to be issued to an already-activated bank because the row decoder hierarchy (i.e., the global and local row decoders) *cannot* simultaneously drive two wordlines [72]. While a row is active, the wordline corresponding to the row needs to remain asserted, so that the cells of the row remain connected to the LRB. In existing DRAM chips, the decoder hierarchy latches and drives *only* one-row address, which the memory controller provides along with an ACTIVATE command. To enable FIGARO, we employ a similar technique to prior work on subarray-level parallelism [72]: we add a latch to the decoding logic of each subarray to store an additional row address for holding the source row of RELOC, and extend the local row decoder of each subarray, such that it can choose between the row address in this latch and the conventional row address bus (to identify the destination row of RELOC).

Enabling Unaligned Data Relocation. In a DRAM chip, the column decoder latches and drives *only* one column address per bank, which determines which portion of the LRB is connected to the GRB. Because conventional DRAM activates only one subarray at a time, the column decoder sends a single column address to *all* LRBs in the bank.

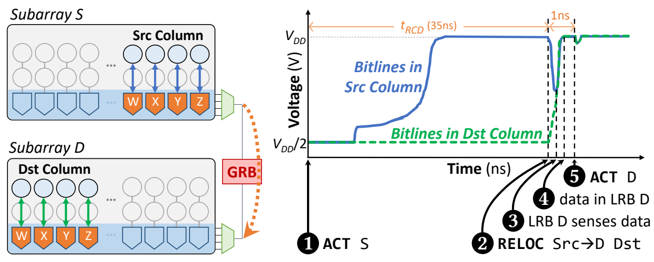
To enable *unaligned* data relocation (i.e., relocating data from column A in the *src* subarray LRB to column B in the *dst* subarray LRB, where $A \neq B$), we need to modify the column decoding logic. When the memory controller sends *two* column addresses simultaneously (one for the source subarray in RELOC, and the other for the destination subarray), we add a multiplexer to the column decoder of each subarray, to allow the decoder to choose which of the two column addresses it reads (based on whether the subarray contains the *src* or the *dst* LRB). As the existing address bus in DRAM is wide enough to transfer two column addresses at once [72], we do not need to change the physical DRAM interface.[1]

[1]RELOC uses 21 bits to express the column addresses: 7 bits to identify the source column in the open row of the bank, 7 bits to identify the destination subarray index, and 7 bits to identify the destination column.

4.2. Latency and Energy Analysis

We perform detailed circuit-level SPICE simulations to find the latency of the RELOC operation. We analyze a SPICE-level model of the entire cell array of a modern DRAM chip (i.e., row decoder, cell capacitor, access transistor, sense amplifier, bitline capacitor and resistor) with 22 nm DRAM technology parameters, based on an open-source DRAM SPICE model [39]. We use 22 nm PTM low-power transistor models [8, 148] to implement the access transistors and the sense amplifiers. In our SPICE simulations, we run 108 iterations of Monte-Carlo simulations with a $\pm5\%$ margin on every parameter of each circuit component, to account for manufacturing process variation and for the worst-case cells in DRAM. Across all iterations, we observe that RELOC operates correctly. We report the latency of RELOC based on the Monte-Carlo simulation iteration with the highest access latency.

To aid the explanation of our SPICE simulation of RELOC, we use an example that performs RELOC from the *Src* column of LRB S in subarray S to the *Dst* column of LRB D in subarray D through the GRB, as shown in Figure 5a. Figure 5b shows the voltage of the bitlines for both the *Src* column (which holds data value 1 in each cell) and the *Dst* column during the RELOC process over time according to our SPICE simulation. We explain this RELOC process step by step.



(a) Example **RELOC** Operation (b) Illustration of **RELOC** Bitline Voltages

Figure 5. Detailed RELOC operation and timing.

First, before the RELOC command is issued, an ACTIVATE (ACT) command is sent to subarray S at time 0 (❶). After 35 ns (based on the standard-specified t_{RAS} parameter [52]; ❷), the bitlines are fully restored to V_{DD}. Second, the memory controller sends the RELOC command to relocate (copy) data from LRB S to LRB D through the GRB. RELOC turns on the connection between the *Src* column in LRB S and the *Dst* column in LRB D. Third, after a small amount of time (❸), the voltage of the source bitlines in the *Src* column first drops, as these fully-driven bitlines share charge with the precharged bitlines in the *Dst* column though the GRB. This causes the corresponding sense amplifiers in LRB D to sense the charge difference and start amplifying the perturbation, during which the GRB helps amplification with higher drive strength. In a very short time (less than 1 ns), bitlines in the *Dst* column are fully driven with the value that is originally stored in LRB S (❹). Finally, an ACTIVATE command is sent to subarray D (❺), overwriting the DRAM cells connected to the bitlines in the *Dst* column, while maintaining the existing values of the other cells in the row [15, 81, 127].

Using SPICE simulations, we find that the latency of RELOC is 0.57 ns (accounting for the worst case of relocating data via the global row buffer between the two subarrays that are the furthest away from each other). We add a guardband to the RELOC latency, similar to what DRAM manufacturers do to account for process and temperature variation (e.g., the ACTIVATE timing, t_{RCD}, has been observed to have extra margins of 13.3% [12] and 17.3% [80]). We add a conservative 43% guardband for RELOC on top of our SPICE simulation results, resulting in a 1 ns latency. This results in a total latency of 63.5 ns to relocate one column (i.e., two ACTIVATEs, one RELOC, and one PRECHARGE). We estimate the energy consumption of a one-cache-block (rank-level) FIGARO data relocation operation to be 0.03 μJ, using the Micron power calculator [100].

5. Fine-Grained In-DRAM Cache Design

FIGARO can improve the efficiency of in-DRAM caches [15, 94] by enabling (1) the ability to relocate data into and out of the cache at the fine granularity of a *row segment* instead of an entire row, resulting in higher performance; and (2) designs that avoid the need for a large number of fast (yet low-capacity) subarrays interleaved among slow subarrays and thus easier to manufacture, resulting in lower area overhead and lower complexity. We use FIGARO as the foundation of a new in-DRAM cache called *FIGCache* (fine-grained in-DRAM cache). FIGCache co-locates hot row segments from slow subarrays into a small number of rows that serve as a cache. To manage the cache, FIGCache uses a tag store (FTS) in the memory controller to hold metadata about currently-cached segments, and employs a simple policy for identifying which segments should be brought into the cache (Section 3). When a row segment needs to be brought into the cache, FIGCache uses multiple RELOC commands (one for each cache block in the segment) to copy data from the slow subarray into the cache. Likewise, a dirty evicted row segment is written back from the cache to its location in the slow subarray using RELOC commands. Rows serving as the cache can either be implemented using small fast subarrays, reserved rows within slow subarrays, or fast rows within a subarray (Section 5.2).

5.1. FIGCache Tag Store

A row segment is brought into FIGCache to lower the latency for subsequent accesses to the segment. For each memory request, in order to know whether it should be serviced by the cache or by the slow subarrays, the memory controller needs to store information about which row segments are currently cached. To this end, we introduce a FIGCache tag store (FTS) in the memory controller. As shown in Figure 6, we maintain a fixed portion of the tag store for each bank. Within each portion, there is a separate entry for each in-DRAM cache slot in the bank (where each fixed-size slot is the size of one row segment). For each entry, FTS holds four fields: (1) a tag holding the original address of the row segment; (2) a valid bit (*V* in the figure); (3) a dirty bit (*D*); and (4) a *benefit* counter (*Benefit*), which is used for cache replacement. In a bank, FIGCache acts as a fully-associative cache, and, thus, the entries within each portion of the FTS are maintained as a fully-associative structure.

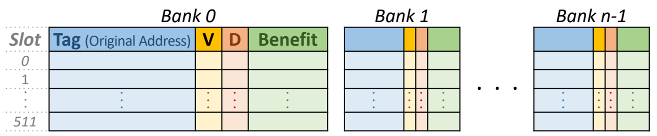


Figure 6. FIGCache tag store (FTS).

An FTS entry is set as valid when a row segment is relocated to the corresponding in-DRAM cache slot. For every memory request, the memory controller looks up the FTS portion associated with the bank of the corresponding request to determine whether or not the request is a hit in the in-DRAM cache. If the request is a FIGCache hit (i.e., an FTS entry matches the row segment ID of the request), its corresponding entry's benefit counter is incremented if the value is not saturated (we empirically set the counter size to 5 bits in this work), and the memory controller redirects the request to the in-DRAM cache. If the request is a write, the entry's dirty bit is set. We next discuss how FIGCache misses are handled.

Choosing Row Segments to Insert Into the Cache. We rely on a very simple policy, called *insert-any-miss*, to identify when a row segment should be inserted into the cache: every FIGCache miss for a memory request triggers a row segment relocation into the cache. This policy is designed to achieve the highest utilization of the in-DRAM cache. While more sophisticated policies (e.g., adding only those segments whose access frequencies exceed a certain threshold) can be used to limit the number of segments inserted into FIGCache, such policies typically spend additional area and energy on calculating statistics. We evaluate the sensitivity of FIGCache to various insertion policies in Section 9, and find that the performance gain of FIGCache is robust to different policies, and that our simple insert-any-miss policy performs well.

Cache Replacement Policy. FIGCache manages cache replacement at the row granularity. When a new row segment needs to be inserted, and no free segments are available in the cache, FIGCache calculates the cumulative benefit of each in-DRAM cache row, by summing together the benefit values of each cached row segment in the row.[2] The row with the lowest total benefit score is selected for eviction. FIGCache maintains a register that holds the ID of the row to be evicted (6 bits in our configuration), and maintains a single bitvector (8 bits in our configuration) that tracks which row segments in the row have not yet been evicted. When a new in-DRAM cache row is selected for eviction, the bitvector is set to all ones, marking all of the row segments in the selected row for eviction. From the marked row segments, the one with the lowest individual benefit score is evicted, making room for the segment being inserted, and its corresponding bit in the bitvector is cleared. The other row segments remain marked for eviction in the bitvector, and the next time that a row segment needs to be inserted, the marked row segment with the lowest score is evicted. This process continues for every insertion until no more marked row segments remain, at which point a new row is selected for eviction.

We choose to perform eviction at a row granularity in order to take advantage of temporal locality across row segments. The benefits of FIGCache increase when multiple row segments in an open FIGCache row are accessed, as memory accesses to open rows are faster than memory accesses to closed rows. By evicting all of the segments in a row, we can pack the row with row segments that are accessed close in time to each other, increasing the chance (due to locality) that the segments will again be accessed together, thus increasing the row buffer hit rate in the in-DRAM cache. We compare our row-granularity replacement policy with commonly-used replacement policies that can be applied at the row segment granularity in Section 9, and show that our row-granularity replacement policy achieves higher performance due to the higher row buffer hit rate it enables.

5.2. In-DRAM Cache Design

Building In-DRAM Cache with Fast Subarrays. One way to implement the in-DRAM cache is to add fast subarrays to a DRAM bank, in addition to the regular (i.e., slow) subarrays, similar to prior works [15, 94, 134]. A fast subarray achieves low access latency by reducing the bitline length [81]. Unlike prior works [15, 94, 134], whose direct connections between subarrays can incur highly-distance-dependent latencies for data relocation (causing designers to interleave many fast subarrays among slow subarrays to bound the relocation latency), FIGARO provides a *distance-independent* relocation latency (Section 4.1), as all relocation operations go through the global row buffer and global bitlines that are shared across all subarrays in a bank. This allows FIGCache to employ only a small number of fast subarrays (we use only two per bank in this work), which reduces both the area overhead (fewer subarrays per bank lead to fewer peripheral circuitry blocks that are needed for that bank) and manufacturing complexity (fewer fast subarrays lead to less design and placement complexity).

Building In-DRAM Cache with Slow Subarrays. As FIGARO facilitates the co-location of multiple hot row segments into the same DRAM row, the row buffer hit rate is expected to increase, thus lowering the average DRAM latency. Our row-granularity replacement policy further increases the likelihood of increased row buffer hit rates. As a result, with a low-overhead relocation mechanism, FIGCache can improve performance even *without* the aid of reduced-latency subarrays. This enables us to build the in-DRAM cache in conventional homogeneous DRAM chips without introducing heterogeneity into DRAM banks.

We propose to reserve a small number of DRAM rows per bank in a slow subarray, to serve as the in-DRAM cache. Note that DRAM row reservation is a widely-used optimization method in both academia [31, 39, 126, 128, 141] and industry [95]. One potential issue with using rows in an existing slow subarray is that FIGARO cannot efficiently relocate data *within the same subarray*. As a result, to avoid the overheads of relocation, we simply do not cache any row segments from the same subarray that FIGCache's rows reside in. Given that existing DRAM chips employ a large number of subarrays (i.e., 32 to 64) in each bank [17, 72, 128], the loss of caching opportunity is negligible. An alternative can be to reserve DRAM rows in two subarrays, and relocate the row segments of one of those subarrays to the reserved rows in the other subarray. However, to simplify the cache management logic, we do not evaluate such a setup in our work.

Building In-DRAM Cache with Fast Rows in a Subarray. Two recent works, CROW [39] and CLR-DRAM [97], use the idea of *cell coupling*, where the same bit is written to into two or more cells along the same bitline [39] or wordline [97] within a subarray. Cell coupling reduces the access latency when the coupled cells are activated together, as all of the coupled cells now drive their charge simultaneously, increasing the speed at which the data value can be sensed by a sense amplifier. As a result, a row of coupled cells acts as a fast DRAM row, enabling a similar effect as fast subarrays (i.e., low-latency access) without the need for a separate subarray.

Based on the structures proposed in CROW [39] and CLR-DRAM [97] to write the same bit into multiple cells concurrently, FIGARO can be extended to relocate data from a conventional slow DRAM row to a fast row. When relocating data from global row buffer to the destination local row buffer, `RELOC` can utilize the mechanisms proposed in existing works [39, 97] so that each bit in the global row buffer can be written into multiple cells (i.e., cells in the fast DRAM row) in the destination subarray. We leave evaluation of such a mechanism to future work.

Tiered-Latency (TL) DRAM [81] enables fast rows within a subarray by adding isolation transistors along the bitlines of

[2]We can use the Dirty-Block Index [125] to simplify the summing operation, as it can help to efficiently maintain per-row benefit scores.

the subarray. When the isolation transistors are open, only a small number of rows remain connected to the local row buffer, providing similar performance to a fast subarray. To build FIGCache on top of TL-DRAM, we can use `RELOC` to cache data from the slow rows of one subarray into the fast rows of a different subarray, as `RELOC` cannot relocate data when the source and destination are in the same subarray without incurring additional overheads (i.e., the use of a second subarray to serve as an intermediate buffer). We leave a detailed implementation and evaluation of a TL-DRAM-based FIGCache to future work.

6. Other Use Cases for FIGARO and FIGCache

We believe that FIGARO and FIGCache can enable multiple new use cases (other than FIGCache in DDRx DRAM). We briefly discuss two such cases, and leave it to future work to design and evaluate mechanisms that enable these use cases.

FIGARO with Emerging DRAM Technologies. Although we evaluate FIGARO and FIGCache for DDR4 DRAM, both solutions can be applied to other DRAM-based memory technologies with similar bank organizations to DDR4, such as 3D-stacked High-Bandwidth Memory (HBM) [1,77,78] and GDDR5 memory for GPUs [50].

Mitigating DRAM Security Vulnerabilities with FIGCache. FIGCache can be used to reduce the vulnerability of DRAM to row-buffer-conflict-based attacks. We briefly examine two potential vulnerabilities: (1) RowHammer and (2) side channel attacks in DRAM.

RowHammer [21,29,68,69,107,108] is a vulnerability that takes place when two or more rows in the same bank are accessed frequently. These frequent accesses cause the two (or more) rows to be repeatedly open and closed due to row buffer conflicts, hammering (i.e., inducing bit flips in) the data stored in neighboring DRAM rows. FIGCache helps to reduce the impact of RowHammer because FIGCache dynamically relocates frequently-accessed data into a single row. Frequently-accessed row segments can be cached by FIGCache in the same in-DRAM cache row, which eliminates the need to repeatedly open and close the DRAM rows that hold each segment. FIGCache reduces the probability that RowHammer can take place on the in-DRAM cache rows, as FIGCache's cache insertion policy keeps row segments accessed around the same time as one another in a single row, significantly reducing the frequency at which multiple in-DRAM cache rows need to be opened/closed (see Section 8.1).

A DRAM-row-based side channel attack can be used by a malicious program to locate and monitor the memory accesses of a victim program without the victim's knowledge or permission [118]. In a scenario where the attacker's data is located in the same bank as data belonging to the victim, a side channel can be established by monitoring the access time variation caused by row buffer conflicts. DRAMA [118] demonstrates that this access time variation, coupled with knowledge of where the attacker's data resides in physical memory, can be used to determine when the victim is accessing its data, revealing information such as when a user is performing each keystroke while entering a URL into the address bar of a browser. The attack works because the attacker can observe information about row hits and misses to specific DRAM rows where its data is co-located with that of the victim. FIGCache breaks this ability by caching select portions of DRAM rows, which alters the row hit and miss patterns for the cached data. Because the attack depends on precise row hit/miss information, FIGCache's caching behavior can mitigate the attack.

We leave the evaluation of both attacks and potential mitigation techniques using FIGCache to future work.

7. Experimental Methodology

We evaluate FIGCache using a modified version of Ramulator [73, 122], a cycle-accurate DRAM simulator, coupled with our in-house processor simulator. We collect user-level application traces using Pin [96]. Table 1 shows a summary of our system configuration. We set the default row segment size as 1/8th of a DRAM row, and study the effect of various row segment sizes on the performance of FIGCache (Section 9). For the fast subarray design, we use the open-source SPICE model developed for LISA-VILLA [15], where slow and fast subarrays have 512 and 32 DRAM rows, respectively, and where timing parameters for activation (t_{RCD}), precharge (t_{RP}), and restoration (t_{RAS}) in fast subarrays can be reduced by 45.5%, 38.2%, and 62.9% respectively.

Processor	8 cores, 3.2 GHz, 3-wide issue, 256-entry inst. window 8 MSHRs/core, L1 4-way 64 kB, L2 8-way 256 kB
Last-Level Cache	2 MB/core, 64 B cache block, 16-way
Memory Controller	64-entry RD/WR request queues, FR-FCFS scheduling [121, 158]
DRAM	DDR4, 800 MHz bus frequency, 1 channel for single-core/4 channels for eight-core, 1 rank, 4 bank groups with 4 banks each, 64 subarrays per bank, 8 kB row size, 4 GB capacity per channel, address interleaving: {row, rank, bankgroup, bank, channel, column}
FIGARO	rank-level `RELOC` granularity: 64 B, `RELOC` latency: 1 ns
FIGCache	row segment: 1/8th of DRAM row (16 cache blocks), fast subarray reduces t_{RCD} / t_{RP} / t_{RAS} by 45.5% / 38.2% / 62.9% [15], in-DRAM cache size: 64 rows per bank
LISA-VILLA	in-DRAM cache size: 512 rows per bank

Table 1. Simulated system configuration.

To evaluate energy consumption, we model all major components of our evaluated system based on prior works [11, 147], including CPU cores, L1/L2/last-level caches, off-chip interconnects, and DRAM. We use several tools for this, including McPAT 1.0 [87] for the CPU cores, CACTI 6.5 [105] for the caches, Orion 3.0 [57] for the interconnect, and a modified version of DRAMPower [13] for DRAM.

As shown in Table 2, we evaluate twenty single-thread applications from the TPC [142], MediaBench [30], Memory Scheduling Championship [19], Biobench [7], and SPEC CPU 2006 [137] benchmark suites. We classify the applications into two categories: memory intensive (greater than 10 last-level cache misses per kilo-instruction, or MPKI) and memory non-intensive (less than 10 MPKI). To evaluate the effect of FIGCache on a multicore system, we form 20 eight-core multiprogrammed workloads. We vary the load on the memory system by generating workloads where 25%, 50%, 75%, and 100% of the applications are memory intensive. To demonstrate the performance improvement of FIGCache on multithreaded workloads, we evaluate *canneal* and *fluidanimate* from PARSEC [10], and *radix* from SPLASH-2 [150]. For both the single-core applications and eight-core workloads, each core executes at least one billion instructions. We report the instruction-per-cycle (IPC) speedup for single-core applications, and weighted speedup [133] as the system performance metric [28] for the eight-core workloads. For multithreaded workloads, we execute the entire application, and report the improvement in execution time.

Category	Benchmark Name
Memory Intensive	zeusmp, leslie3d, mcf, GemsFDTD, libquantum bwaves, lbm, com, tigr, mum
Memory Non-Intensive	h264ref, bzip2, gromacs, gcc, bfssandy grep, wc-8443, sjeng, tpcc64, tpch2

Table 2. Benchmarks used for single-core and multiprogrammed workloads.

8. Evaluation

We evaluate four realistic configurations to understand the benefits of FIGCache:

- *Base*: a baseline system with conventional DDR4 DRAM;
- *LISA-VILLA* [15]: a state-of-the-art in-DRAM cache;
- *FIGCache-Slow*: our in-DRAM cache with cache rows stored in 64 reserved rows of one existing *slow* subarray (i.e., a system with conventional homogeneous DRAM subarrays);
- *FIGCache-Fast*: our in-DRAM cache with cache rows stored in *two* small *fast* subarrays (with a total of 64 rows).

We also evaluate two idealized configurations to examine the impact of certain system parameters:

- *FIGCache-Ideal*: an unrealistic version of FIGCache-Fast where the row segment relocation latency is *zero*; and
- *LL-DRAM*: a system where *all* subarrays in the DRAM chips are fast (i.e., low latency).

8.1. Performance

Figures 7 and 8 show the performance improvement over Base for our single-thread applications (using a one-core system) and eight-application multiprogrammed workloads (using an eight-core system), respectively. In both figures, we group the applications and workloads based on memory intensity (see Section 7). We make four observations from the figures.

First, both FIGCache-Slow and FIGCache-Fast *always* improve performance over Base. For our single-thread applications, FIGCache-Fast provides an average speedup over Base of 1.5% (up to 2.9%) for memory non-intensive applications, and 16.1% (up to 22.5%) for memory intensive applications. For our multiprogrammed workloads, FIGCache-Fast improves the weighted speedup over Base by an average of 3.9%, 12.9%, 21.8%, and 27.1% for workloads in the 25%, 50%, 75%, and 100% memory intensive categories, respectively. Across all 20 eight-core workloads, the average performance improvement of FIGCache-Fast is 16.3%. FIGCache-Fast achieves speedups for our three multithreaded applications as well (not shown in the figure), with an average improvement of 16.8% over Base. Despite not having cache rows with faster access times, FIGCache-Slow retains a large fraction of the benefits of FIGCache-Fast, with an average performance gain of 5.9% and 12.4% for single-thread and multiprogrammed workloads, respectively.

Second, we observe that compared with LISA-VILLA, which employs 16 fast subarrays and interleaves them among the normal subarrays, FIGCache-Fast provides 4.7% higher performance averaged across our 20 eight-core workloads, despite employing *only two* fast subarrays. This is because even though FIGCache-Fast has much fewer fast subarrays per bank, FIGCache-Fast caches only 1/8th of a row at a time and co-locates multiple row segments with high expected temporal locality in a single cache row. The increased row buffer hit rate in the in-DRAM cache (see analysis below) provides most of FIGCache-Fast's benefits over LISA-VILLA. These benefits also allow FIGCache-Slow to outperform LISA-VILLA by 1.9% on average across all of our multiprogrammed workloads, even though FIGCache-Slow has *no* fast subarrays at all. We conclude that reducing the granularity of caching and co-locating multiple row segments into a single cache row is greatly effective for improving the performance of an in-DRAM cache.

Third, the benefits of FIGCache-Fast and FIGCache-Slow increase as workload memory intensity increases. On average, compared to Base, FIGCache-Fast and FIGCache-Slow provide 27.1% and 20.6% speedup for 100% memory intensive eight-core workloads, respectively, whereas they achieve more modest speedups of 3.9% and 3.2%, respectively, for 25% memory intensive workloads. There are multiple reasons for the increased benefits for memory intensive workloads: these workloads (1) are more likely to generate requests that compete for the same memory bank (i.e., they induce bank conflicts by accessing different rows), which FIGCache can potentially alleviate by gathering the accessed row segments of each conflicting row into a single cache row; and (2) may in some cases be more sensitive to DRAM latency. The potential

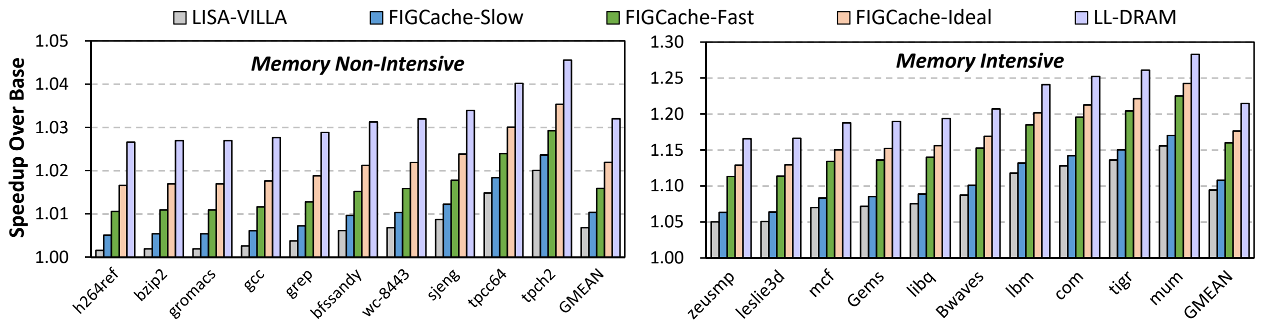


Figure 7. Performance of in-DRAM caching mechanisms for single-thread applications, normalized to Base.

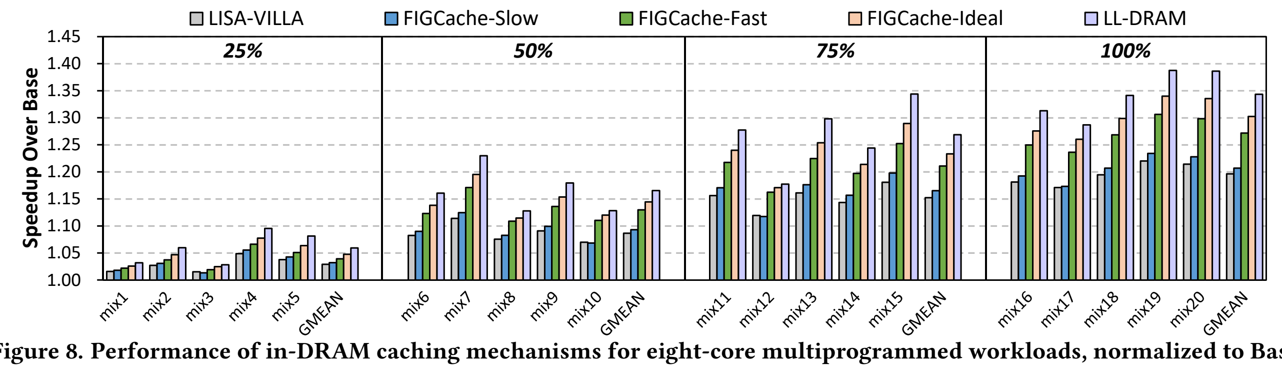


Figure 8. Performance of in-DRAM caching mechanisms for eight-core multiprogrammed workloads, normalized to Base.

correlation between bank conflicts and FIGCache effectiveness is corroborated by the fact that our eight-core multiprogrammed workloads achieve much larger performance improvements than our single-core applications. Individual applications in multiprogrammed workloads are likely to interfere with each other, thus exacerbating bank conflicts [33, 35, 40, 70, 71, 84, 99, 102, 104, 110, 140, 141, 143, 144, 147, 155], which FIGCache can help to alleviate.

Fourth, FIGCache-Fast approaches the ideal performance improvement of *both* FIGCache-Ideal and LL-DRAM, coming within 1.9% and 4.6% respectively, on average, for our eight-core system. These improvements indicate that the latency of cache insertion in FIGCache is low. When a FIGCache miss occurs, the memory controller opens the row containing the data that is being requested. While the row is open, the memory controller uses `RELOC` operations to relocate the row segment data into the cache. Since the row is already open, the first ACTIVATE command discussed in Section 4.2 is not needed, which greatly reduces the time required for relocation. The resulting relocation latency is low enough that FIGCache can, in many cases, behave similarly to low-latency DRAM, without the associated challenges of low-latency DRAM (e.g., small capacity, high cost).

Overall, we conclude that FIGCache significantly reduces DRAM latency and outperforms a state-of-the-art in-DRAM caching mechanism, while approaching the performance of a low-latency DRAM design with only fast subarrays.

Cache Hit Rate. Figure 9 illustrates the in-DRAM cache hit rate of LISA-VILLA, FIGCache-Slow, and FIGCache-Fast, averaged across each workload category. We observe that despite having fewer or no fast subarrays, and having significantly fewer rows reserved for caching, FIGCache-Slow and FIGCache-Fast have comparable cache hit rates to LISA-VILLA across all workloads. This is because due to the limited row buffer locality in many applications, caching an entire DRAM row (as opposed to a row segment), leads to inefficient cache utilization since most of each cached row is not used. The finer granularity employed by FIGCache eliminates much of this inefficient utilization without sacrificing the cache hit rate with a smaller cache. FIGCache-Slow results in a slightly lower cache hit ratio than FIGCache-Fast because, as we discuss in Section 5.2, FIGCache-Slow does not cache row segments from the subarray where the reserved rows are allocated.

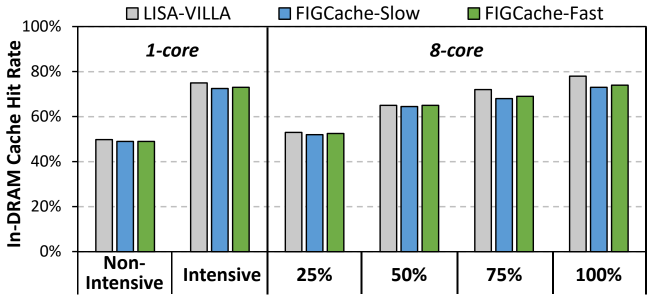


Figure 9. In-DRAM cache hit rate of LISA-VILLA, FIGCache-Slow, and FIGCache-Fast.

Row Buffer Hit Rate. Unlike with the cache hit rate, FIGCache-Slow and FIGCache-Fast both have significantly-higher (18% higher on average) row buffer hit rates for the entire DRAM system than LISA-VILLA, as we observe in Figure 10. This is due to two reasons: (1) the smaller row segment granularity used by FIGCache; and (2) our benefit-based

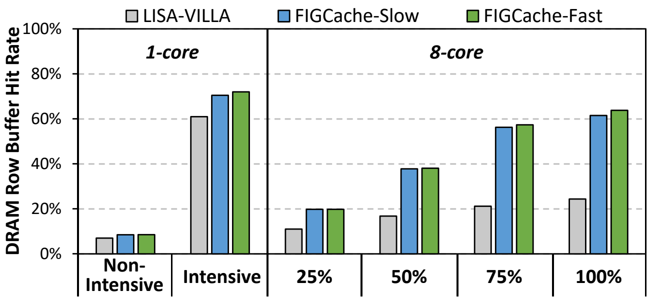


Figure 10. DRAM row buffer hit rate of LISA-VILLA, FIGCache-Slow, and FIGCache-Fast.

cache replacement policy (Section 5.1), which increases the row buffer hit rate by taking into account the temporal locality of multiple row segments during co-location. In contrast, LISA-VILLA caches an entire DRAM row at a time, and thus the row buffer hit rate cannot be improved fundamentally beyond the existing row buffer hit rate of the original row. As a result, LISA-VILLA can benefit *only* from the reduced latencies of a fast subarray. We conclude that both FIGCache-Slow and FIGCache-Fast are effective at improving row buffer hit rate due to their ability to efficiently co-locate multiple row segments from different source rows into a single in-DRAM cache row.

8.2. System Energy Consumption

Figure 11 shows the overall system energy consumption for Base, FIGCache-Slow, and FIGCache-Fast, averaged across each workload category. We break down the system energy into the energy consumed by the CPU, caches (L1, L2, and LLC), off-chip interconnect (labeled *off-chip* in the figure), and DRAM.

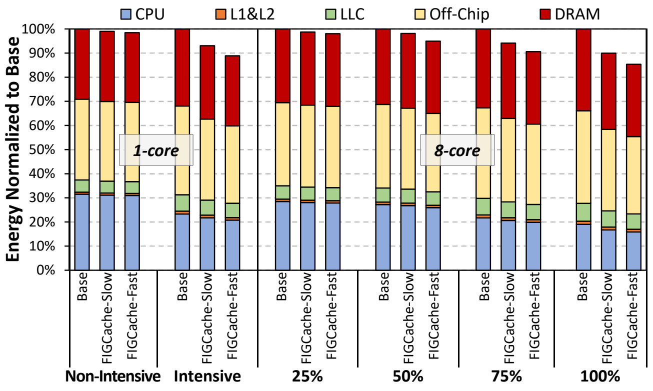


Figure 11. Energy and energy breakdown of LISA-VILLA, FIGCache-Slow, and FIGCache-Fast, normalized to Base.

We draw two observations from the figure. First, for each workload category, both FIGCache-Slow and FIGCache-Fast consume less energy than Base. For the memory intensive single-core applications, FIGCache-Slow and FIGCache-Fast reduce the system energy consumption by an average of 6.9% and 11.1%, respectively, compared to Base. Second, we observe that the energy reduction comes from two sources: (1) improved row buffer hit rate, which helps to amortize the energy of ACTIVATE and PRECHARGE commands on many memory accesses; and (2) reduced execution time, which saves static energy across each component. For FIGCache-Fast, there is a third source of energy reduction, as the faster ACTIVATE and PRECHARGE commands enabled by the fast subarrays further reduce both dynamic and static energy.

Overall, we conclude that FIGCache is effective at reducing system energy consumption.

8.3. Hardware Overhead

DRAM Area and Power Overhead. FIGARO adds a column address MUX, row address MUX, and a row address latch to each DRAM subarray. Our RTL-level evaluation using a 22 nm technology shows that each column MUX occupies an area of 4.7 μm^2 and consumes 2.1 µW, while each row MUX occupies an area of 18.8 μm^2 and consumes 8.4 µW. Each row address latch stores the 40-bit partially predecoded row address, and occupies an area of 35.2 μm^2 with a power consumption of 19.1 µW. For the system configurations described in Table 1, the overall area overhead is less than 0.3% of an entire DRAM chip. The overall power consumption is negligible as an activation consumes 51.2 mW [72].

FIGCache-Fast introduces two fast subarrays per bank as an inclusive in-DRAM cache, which is transparent to the operating system. Each fast subarray contains 32 rows (vs. 512 rows in each slow subarray). Using area estimates from prior works [15, 94], we calculate that a fast subarray, including cells and sense amplifiers, requires 22.6% of the area of a slow subarray. As a result, in our DRAM configuration (see Table 1) where each bank has 64 slow subarrays, the two fast subarrays introduced by FIGCache-Fast add 0.7% to the area of the DRAM chip. In comparison, LISA-VILLA [15] adds 16 fast subarrays to each bank, which have an area overhead of 5.6% of the DRAM chip. FIGCache-Slow has a lower area overhead than FIGCache-Fast, as it uses rows in existing subarrays instead of adding new subarrays, eliminating the area required for additional sense amplifiers. As a result, FIGCache-Slow has an area overhead of only 0.2% in the DRAM chip.

Memory Controller. On the memory controller side, we add the FTS (Section 5), which incurs modest storage overhead. We assume one FTS portion per bank, where each portion has 512 entries. Each entry of FTS consists of a row segment address tag, a 5-bit benefit counter, and the dirty and valid bits. The width of the tag is dependent on the number of cached row segments in one bank. For the configuration in Section 7, there are 256K row segments per bank (32K DRAM rows per bank, 8 row segments per DRAM row), which requires a tag size of 19 bits. In total, each entry requires 26 bits. Therefore, for each channel in our DRAM configuration (see Table 1), which contains 16 banks with 512 FTS entries per bank, the total storage required for the FTS is 26.0 kB. Note that compared to LISA-VILLA [15], the additional cost of FTS is only the 3-bit row segment index per entry. Using McPAT [87], we compute the total area of all FTS tables to be 0.496 mm^2 at the 22 nm technology node, which is only 1.44% of the area consumed by the 16 MB last-level cache.

We evaluate the access time and power consumption of FTS using CACTI [105]. We find that the access time is only 0.11 ns, which is small enough that we do not expect it to have a significant impact on the overall cycle time of the memory controller. To determine power consumption, we analyze the FTS activity for our applications, accounting for all of the major table operations. Using CACTI [105] and assuming a 22 nm technology node, we find that the table consumes 0.187 mW on average. This is only 0.07% of the average power consumed by the last-level cache. We include this additional power consumption in our system energy evaluations.

9. Sensitivity Studies

In this section, we evaluate our design with various configurations, including different cache capacities, row segment sizes, cache replacement policies and hot row segment identification policies. As FIGCache-Slow has similar trends with FIGCache-Fast for these configurations, we show results for only FIGCache-Fast.

9.1. In-DRAM Cache Capacity

We examine how the number of fast subarrays in each DRAM bank affects performance. Figure 12 shows the speedup of FIGCache-Fast over Base as we vary the number of fast subarrays (*FS* in the figure) from 1 to 16. We make two observations from the figure. First, FIGCache-Fast's performance improvement increases with increasing in-DRAM cache capacity. A larger number of fast subarrays reduces the number of evictions, and has the potential to provide more opportunities for FIGCache-Fast to reduce access latency for rows that would otherwise be evicted from a smaller in-DRAM cache. Second, more fast subarrays provide diminishing returns on FIGCache's performance improvement, even though they come with additional storage and complexity overheads. For example, increasing the number of fast subarrays from 2 to 4 and from 4 to 8 improves performance by less than 2.7% and 0.8%, respectively, for 100% memory intensive eight-core workloads. We implement two fast subarrays per bank to achieve a balance between performance improvement and in-DRAM storage overhead.

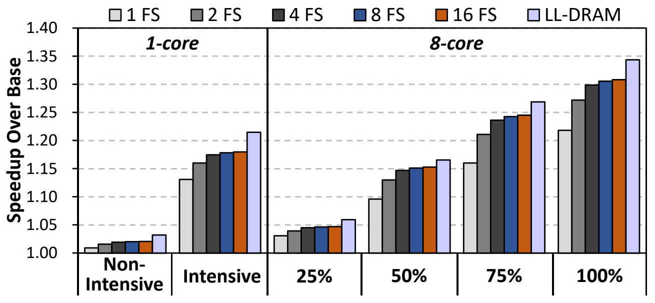


Figure 12. Performance with different cache capacities.

9.2. Row Segment Size

We vary the size of a row segment to understand its impact on performance. While a larger row segment size can potentially expose a greater number of opportunities for exploiting spatial locality within a DRAM row, there are three downsides: (1) many applications do not make use of the contents of an entire row when the row is open, causing a row segment size that is too large to lead to cache underutilization; (2) the caching latency increases, as a larger row segment requires more `RELOC` operations to be issued; and (3) for a given in-DRAM cache size, a larger row segment size means fewer row segments can be cached. Figure 13 shows the performance of FIGCache-Fast with row segment sizes ranging from 8 cache blocks (i.e., 512 B, 1/16th of a DRAM row) to 128 cache blocks (i.e., 8 kB, the entire row). We make two observations from the figure. First, we find that FIGCache-Fast performs slightly *worse* than LISA-VILLA [15] when the row segment size is an entire DRAM row (128 cache blocks). This is due to the higher data relocation latency required by FIGCache, as 128 `RELOC` operations are needed, and highlights the benefits of smaller row segment sizes. Second, we find a peak in performance at a row segment size of 16 cache blocks (i.e., 1 kB, 1/8th of a DRAM row), as it outperforms other row segment sizes across all of our workload categories, and, thus, we choose this as the row segment size in our implementation. Note that while we do not evaluate it, FIGCache can be modified

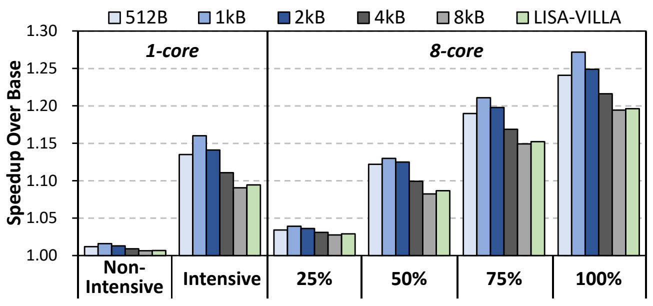


Figure 13. Performance with different row segment sizes.

to support heterogeneous and/or dynamic row segment sizes (as opposed to the static row segment size that we currently use). We leave such a design to future work.

9.3. In-DRAM Cache Replacement Policy

As we discuss in Section 5.1, we implement a new row-granularity benefit-based cache replacement policy for FIGCache, where the eviction granularity (an entire row) differs from the insertion granularity (a single row segment). The different eviction and insertion granularities allow us to improve opportunities for exploiting temporal locality across row segments in an in-DRAM cache row by packing recently-accessed row segments together into a single cache row. To understand the benefits of our policy, we evaluate how FIGCache performs with three other commonly-used replacement policies. Figure 14 shows the performance (normalized to Base) of FIGCache-Fast using our replacement policy (*RowBenefit* in the figure), along with FIGCache-Fast's performance using: (1) *SegmentBenefit*, a traditional benefit-based policy [81] where the granularity of eviction is the same as that of insertion (a row segment for FIGCache), and only the one row segment with the lowest benefit score anywhere in the in-DRAM cache is evicted; (2) *LRU*, a traditional policy that evicts the least-recently-used row segment; and (3) *Random*, a policy that evicts a row segment at random from any row in the cache.

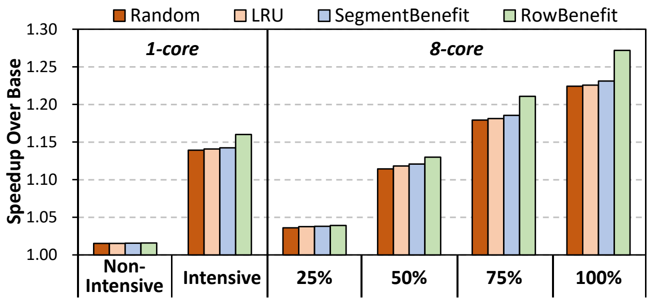


Figure 14. Performance with different in-DRAM cache replacement policies for FIGCache.

We make two observations from the figure. First, FIGCache-Fast outperforms Base by more than 12.5% on average across both single-thread and multithreaded workloads with all four cache replacement policies, indicating the benefits of fine-granularity in-DRAM caching regardless of the exact replacement policy employed. Second, our RowBenefit policy either performs the same as, or outperforms, all three commonly-used policies, with its benefits increasing as workloads become more memory intensive. The RowBenefit policy improves the performance of FIGCache-Fast by 4.1% over the next-best policy (SegmentBenefit) for 100% memory intensive eight-core workloads, due to its increased row buffer hit rate from successfully improving temporal locality in in-DRAM cache rows. We conclude that our fine-grained in-DRAM cache with its row-granularity replacement policy is effective at capturing temporal locality across cached row segments.

9.4. Row Segment Insertion Policy

We use a simple *insert-any-miss* policy to identify which row segments to cache (as we discuss in Section 5.1), where we insert *every* row segment that misses in the in-DRAM cache into the cache. However, it is possible to be more judicious in deciding which row segments should be inserted into the cache. One example is increasing the threshold of the number of consecutive cache misses to the row segment before the segment is inserted. While a higher threshold can potentially reduce cases where a row segment is accessed only once across a large time period (in which case it cannot benefit from caching), it can also (1) reduce the benefits of caching (by waiting too long to cache a row segment with high temporal locality), and (2) require additional metadata (as accesses to uncached row segments now need to be tracked). To understand the potential benefits of a more judicious insertion policy, we evaluate different threshold values (where a value of 1 is our policy of caching a row segment after a miss to it), ideally assuming that the additional storage required for higher thresholds does not introduce additional latency.

Figure 15 shows FIGCache-Fast's average performance, normalized to Base, for four threshold values (1, 2, 4, 8). We make two observations from the figure. First, increasing the threshold from 1 to 2 minimally increases the performance of memory non-intensive workloads, though further threshold increases can result in worse performance than a threshold of 1. Second, for memory intensive workloads, a higher threshold leads to *worse* performance, by decreasing the number of cache hits (latter not shown). Therefore, we conclude that a threshold of 1 (i.e., our simple insert-any-miss policy) is effective for performance.

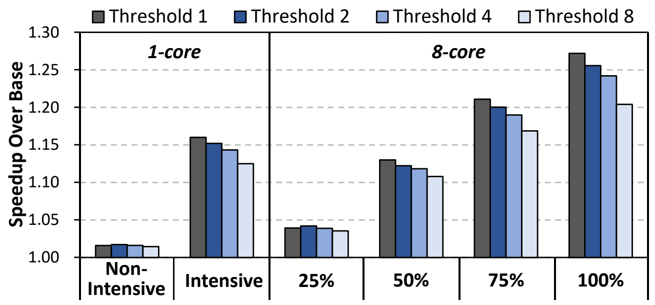


Figure 15. Performance with different row segment insertion thresholds.

10. Related Work

To our knowledge, this work is the first to propose an efficient fine-grained in-DRAM data relocation substrate, which enables a new *fine-grained* in-DRAM cache design. We already quantitatively demonstrate that *FIGCache* outperforms the most closely-related state-of-the-art in-DRAM cache design, LISA-VILLA [15]. In this section, we briefly discuss other related works that propose (1) other designs for in-DRAM caches, (2) in-DRAM data relocation support, (3) designs that improve the row buffer hit rate; and (4) DRAM latency and power reduction mechanisms.

In-DRAM Caching Mechanisms. As we discuss in Section 3, there are three main approaches that prior works take in building in-DRAM caches: (1) a heterogeneous subarray based design (Tiered-Latency DRAM [81]), (2) a heterogeneous bank based design without data relocation support (CHARM [134]), and (3) a heterogeneous bank based design with bulk data relocation support (DAS-DRAM [94] and LISA-VILLA [15]). Like FIGCache, these works build their in-DRAM caches out of DRAM cells. Several earlier works [38,42,44,45,59,120] on cached DRAM integrate SRAM caches into the DRAM modules, usually at very high area overhead [72, 81].

Similar to traditional caching mechanisms that relocate data into a dedicated cache, CROW [39], CLR-DRAM [97], and Multiple Clone Row DRAM (MCR-DRAM) [20] decrease the access latency for frequently-accessed DRAM rows by coupling multiple cells together for a single bit of data, thus increasing the amount of charge that is driven to a sense amplifier when a row is activated. As we discuss in Section 5.2, FIGCache can be built on top of the hardware mechanisms that CROW and CLR-DRAM use to manage the fast rows. While FIGCache can also be integrated with MCR-DRAM, such a design can become more complex, as MCR-DRAM depends on the OS to manage which pages are assigned to its fast rows [20].

In-DRAM Data Relocation Support. The DAS-DRAM [94] and LISA [15] substrates provide support for bulk data migration across subarrays, as we discuss in Section 3. Another mechanism for bulk data relocation in DRAM is RowClone-FPM [127]. However, as RowClone-FPM relocates data only within a subarray, it can not be used to build an in-DRAM cache that caches data from multiple subarrays in a bank. RowClone-PSM [127] is a mechanism that relocates data at column granularity across different DRAM banks, using the shared global data bus inside DRAM (which connects to the memory channel). Unfortunately, by using the global data bus, RowClone-PSM blocks memory requests to *all* banks during data relocation, reducing the overall bank-level parallelism [75, 110]. If RowClone-PSM is used to relocate 4 kB of data between two subarrays in separate banks, it decreases system performance by 24% compared to using a conventional `memcpy` operation [15]. RowClone-PSM's performance is even *lower* for data relocation between subarrays in the *same* bank, as this requires two RowClone-PSM operations (one moving data from the source subarray to a second bank that serves as an intermediate buffer, and another moving data from the second bank to the destination subarray in the original bank) [127]. Network-on-Memory (NoM) [119] overcomes this inter-bank limitation of RowClone-PSM with fast and efficient data relocation across banks within 3D-stacked DRAM, via the use of higher connectivity between banks provided by a network in the logic layer. FIGARO is orthogonal to NoM.

Mechanisms to Improve Row Buffer Hit Rate. Several works mitigate the negative effects of low row buffer hit rates by reducing the amount of activated data, either by enabling partial row buffer activation, designing smaller row buffers, or by in-DRAM data layout or transfer transformations. Examples of these works include fine-grained activation [22], Half-DRAM [155], selective bitline activation [143], partial row activation [84], efficient 3D-stacked DRAM designs [18, 113], gather-scatter DRAM [129], data reorganization in 3D-stacked DRAM [5,6], and row buffer locality aware caching in hybrid memories [152]. FIGCache is orthogonal to these designs, and can be combined with them to reduce the amount of unused activated data both in cached rows and in non-cached rows. At the software level, prior work proposes to reduce the size of a memory page in the operating system to what it calls micro-pages [141], in order to improve spatial locality within a page. The reduced page size allows for multiple micro-pages to fit into a single DRAM row, and increases the row buffer hit rate by co-locating heavily-used micro-pages into the same row. While this approach is similar to how FIGCache collects multiple cached row segments into a single DRAM row, micro-pages do not have hardware support for relocation, and must instead use high-latency `memcpy` operations through the memory controller to relocate data. Other techniques to improve the row buffer hit rate include changing the memory scheduling policy (e.g., [9, 32, 46, 47, 55, 56, 70, 71, 103, 109, 110, 112, 121, 139, 140, 144, 153, 158]) to result in more row buffer hits or introducing new memory allocation policies [24, 54, 56, 90, 104, 114, 145, 146, 151, 154] to reduce inter-thread interference at the row buffer. These techniques are orthogonal to FIGCache.

DRAM Latency and Power Reduction. To reduce DRAM access latency, prior works enable reduced DRAM timing parameters by exploiting the charge level of DRAM cells [23, 40, 67, 89, 117, 132, 147, 156] or by driving bitlines with charge from multiple cells that contain the same data [20, 39, 97]. Several other works [12, 14, 23, 65, 66, 79, 80] employ optimized timing parameters that take advantage of variation in and across DRAM chips to speed up DRAM accesses. Aside from latency reduction, recent studies propose to reduce DRAM row activation and I/O power consumption through efficient row buffer designs (e.g., multiple sub-row buffers [35], row buffer caches [43, 92, 93, 149], eager writeback [53, 76, 83, 138]), sub-rank memory [2, 157], silent writeback elimination [85, 86], special data encoding schemes [34, 124, 135, 136], an OS-based scheduler to select different power modes [26], a page-hit-aware low power design [91], and DRAM voltage and/or frequency scaling [16, 25, 27, 36, 37, 106]. FIGCache provides a new solution for DRAM latency and power reduction, which can potentially be combined with these existing approaches.

11. Conclusion

In this work, we observe that existing in-DRAM cache designs are inefficient due to (1) the coarse granularity (i.e., a DRAM row) at which they cache data and (2) hardware designs that result in high area overhead and manufacturing complexity. We eliminate these inefficiencies by introducing FIGARO, a new, low-cost DRAM substrate that enables data relocation (i.e., copying) at the granularity of a DRAM column within a chip (cache block within a rank) with only minor modifications to existing peripheral circuitry in commodity DRAM chips. Using FIGARO, we build FIGCache, a fine-grained in-DRAM cache, which greatly improves overall performance and energy consumption, and has a significantly simpler design than existing in-DRAM caches. We believe and hope that future works and architectures can exploit the FIGARO substrate to enable more use cases and application-/system-level performance and energy benefits.

Acknowledgments

We thank the anonymous reviewers, SAFARI group members for the feedback and the stimulating research environment. This work was supported by a Hunan Province Science and Technology Planning project (No. 2019RS2027), a National University of Defense Technology research project (No. 18/19-QNCXJ-WYH), and the industrial partners of SAFARI, especially Google, Huawei, Intel, Microsoft, and VMware.

References

[1] Advanced Micro Devices, Inc., "High Bandwidth Memory." https://www.amd.com/en/technologies/hbm

[2] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Improving System Energy Efficiency with Memory Rank Subsetting," *TACO*, 2012.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.

[6] B. Akin, J. C. Hoe, and F. Franchetti, "HAMLeT: Hardware Accelerated Memory Layout Transform Within 3D-Stacked DRAM," in *HPEC*, 2014.

[7] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *ISPASS*, 2005.

[8] Arizona State Univ., NIMO Group, "Predictive Technology Model," 2012. http://ptm.asu.edu/

[9] R. Ausavarungnirun, K. K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[11] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[12] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.

[13] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "Towards Variation-Aware System-Level Power Estimation of DRAMs: An Empirical Approach," in *DAC*, 2013.

[14] K. K. Chang, A. Kashyap, H. Hassan, , S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[15] K. K. Chang, P. J. Nair, , D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[16] K. K. Chang, A. G. Yaglikci, A. Agrawal, N. Chatterjee, S. Ghose, A. Kashyap, H. Hassan, D. Lee, M. O'Connor, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[17] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[18] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017.

[19] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Memory Scheduling Championship (MSC)," 2012. https://www.cs.utah.edu/~rajeev/jwac12/

[20] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.

[21] L. Cojocar, J. S. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *IEEE S&P*, 2020.

[22] E. Cooper-Balis and B. Jacob, "Fine-Grained Activation for Power Reduction in DRAM," *IEEE Micro*, 2010.

[23] A. Das, H. Hassan, and O. Mutlu, "VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency," in *DAC*, 2018.

[24] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.

[25] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *ICAC*, 2011.

[26] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Scheduler-Based DRAM Energy Management," in *DAC*, 2002.

[27] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "MemScale: Active Low-Power Modes for Main Memory," in *ASPLOS*, 2011.

[28] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.

[29] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE S&P*, 2020.

[30] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II Video: Expediting the Next Generation of Video Systems Research," *MICPRO*, 2009.

[31] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.

[32] S. Ghose, H. Lee, and J. F. Martínez, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.

[33] S. Ghose, T. Li, N. Hajinazar, D. Senol Cali, and O. Mutlu, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.

[34] S. Ghose, A. G. Yağlıkçı, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," *SIGMETRICS*, 2018.

[35] N. D. Gulur, R. Manikantan, M. Mehendale, and R. Govindarajan, "Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities," in *ICS*, 2012.

[36] J. Haj-Yahya, M. Alser, J. Kim, A. G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu, "SysScale: Exploiting Multi-Domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors," in *ISCA*, 2020.

[37] J. Haj-Yahya, Y. Sazeides, M. Alser, E. Rotem, and O. Mutlu, "Techniques for Reducing the Connected-Standby Energy Consumption of Mobile Devices," in *HPCA*, 2020.

[38] C. A. Hart, "CDRAM in a Unified Memory Architecture," in *COMPCON*, 1994.

[39] H. Hassan, M. Patel, J. S. Kim, A. G. Yaglikci, N. Vijaykumar, N. Mansourighiasi, S. Ghose, and O. Mutlu, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.

[40] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.

[41] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.

[42] A. Hegde, N. K. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin, "VL-CDRAM: Variable Line Sized Cached DRAMs," in *CODES+ISSS*, 2003.

[43] E. Herrero, J. González, R. Canal, and D. Tullsen, "Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments," *IEEE TC*, 2013.

[44] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory," *IEEE Micro*, 1990.

[45] W.-C. Hsu and J. E. Smith, "Performance of Cached DRAM Organizations in Vector Supercomputers," in *ISCA*, 1993.

[46] I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers," in *MICRO*, 2004.

[47] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.

[48] K. Itoh, *VLSI Memory Chip Design.* Springer Science & Business Media, 2013.

[49] JEDEC Solid State Technology Assn., *JESD79-3F: DDR3 SDRAM Standard*, July 2012.

[50] JEDEC Solid State Technology Assn., *JESD212C: Graphics Double Data Rate (GDDR5) SGRAM Standard*, February 2016.

[51] JEDEC Solid State Technology Assn., *JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard*, March 2017.

[52] JEDEC Solid State Technology Assn., *JESD79-4B: DDR4 SDRAM Standard*, June 2017.

[53] M. Jeon, C. Li, A. L. Cox, and S. Rixner, "Reducing DRAM Row Activations with Eager Read/Write Clustering," *TACO*, 2013.

[54] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, “Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems,” in *HPCA*, 2012.

[55] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *ISCA*, 2013.

[56] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *ASPLOS*, 2013.

[57] A. B. Kahng, B. Lin, and S. Nath, “Explicit Modeling of Control and Data for Improved NoC Router Estimation,” in *DAC*, 2012.

[58] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi, “Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling,” in *The Memory Forum*, 2014.

[59] G. Kedem and R. P. Koganti, “WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines,” Duke Univ. Dept. of Computer Science, Tech. Rep. CS-1997-03, 1997.

[60] B. Keeth, *DRAM Circuit Design: Fundamental and High-Speed Topics.* John Wiley & Sons, 2007.

[61] S. Khan, D. Lee, Y. Kim, A. Alameldeen, C. Wilkerson, and O. Mutlu, “The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study,” in *SIGMETRICS*, 2014.

[62] S. Khan, D. Lee, and O. Mutlu, “PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM,” in *DSN*, 2016.

[63] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, “Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content,” in *MICRO*, 2017.

[64] R. Kho, D. Boursin, M. Brox, P. Gregorius, H. Hoenigschmid, B. Kho, S. Kieser, D. Kehrer, M. Kuzmenka, U. Moeller, P. Petkov, M. Plan, M. Richter, I. Russell, K. Schiller, R. Schneider, K. Swaminathan, B. Weber, J. Weber, I. Bormann, F. Funfrock, M. Gjukic, W. Spirkl, H. Steffens, J. Weller, and T. Hein, “75nm 7Gb/s/Pin 1Gb GDDR5 Graphics Memory Device with Bandwidth-Improvement Techniques,” in *ISSCC*, 2009.

[65] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, “Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines,” in *ICCD*, 2018.

[66] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, “The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices,” in *HPCA*, 2018.

[67] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, “D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput,” in *HPCA*, 2019.

[68] J. S. Kim, M. Patel, A. G. Yaglikçi, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, “Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques,” in *ISCA*, 2020.

[69] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors,” in *ISCA*, 2014.

[70] 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.

[71] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” in *MICRO*, 2010.

[72] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, “A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM,” in *ISCA*, 2012.

[73] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A Fast and Extensible DRAM Simulator,” in *IEEE CAL*, 2015.

[74] S. Koppula, L. Orosa, A. G. Yaglikçi, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, “EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM,” in *MICRO*, 2019.

[75] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, “Improving Memory Bank-Level Parallelism in the Presence of Prefetching,” in *MICRO*, 2009.

[76] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, “DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems,” in *Univ. of Texas at Austin, HPS Research Group, Tech. Rep. TR-HPS-2010-2*, 2010.

[77] D. U. Lee, K. W. Kim, K. W. Kim, K. S. Lee, S. J. Byeon, J. H. Kim, J. H. Cho, J. Lee, and J. H. Chun, “A 1.2 V 8 Gb 8-Channel 128 GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective I/O Test Circuits,” *JSSC*, 2015.

[78] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, “Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost,” *TACO*, 2016.

[79] D. Lee, S. M. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, “Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms,” *SIGMETRICS*, 2017.

[80] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, “Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case,” in *HPCA*, 2015.

[81] 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.

[82] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, “Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM,” in *PACT*, 2015.

[83] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, “Eager Writeback – A Technique for Improving Bandwidth Utilization,” in *MICRO*, 2000.

[84] Y. Lee, H. Kim, S. Hong, and S. Kim, “Partial Row Activation for Low-Power DRAM System,” in *HPCA*, 2017.

[85] Y. Lee, S. Kim, S. Hong, and J. Lee, “Skinflint DRAM System: Minimizing DRAM Chip Writes for Low Power,” in *HPCA*, 2013.

[86] K. M. Lepak and M. H. Lipasti, “On the Value Locality of Store Instructions,” in *ISCA*, 2000.

[87] 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.

[88] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, “An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms,” in *ISCA*, 2013.

[89] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, “RAIDR: Retention-Aware Intelligent DRAM Refresh,” in *ISCA*, 2012.

[90] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, “A Software Memory Partition Approach for Eliminating Bank-Level Interference in Multicore Systems,” in *PACT*, 2012.

[91] S. Liu, S. O. Memik, Y. Zhang, and G. Memik, “A Power and Temperature Aware DRAM Architecture,” in *DAC*, 2008.

[92] G. H. Loh, “3D-Stacked Memory Architectures for Multi-Core Processors,” in *ISCA*, 2008.

[93] G. H. Loh, “A Register-File Approach for Row Buffer Caches in Die-Stacked DRAMs,” in *MICRO*, 2011.

[94] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, “Improving DRAM Latency with Dynamic Asymmetric Subarray,” in *MICRO*, 2015.

[95] W.-M. Lu, B.-F. Hung, and M.-S. Huang, “Method for Controlling a DRAM,” U.S. Patent Appl. 12/116,208, 2009.

[96] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *PLDI*, 2005.

[97] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yaglikçi, L. Orosa, J. Park, and O. Mutlu, “CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off,” in *ISCA*, 2020.

[98] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, “Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory,” in *DSN*, 2014.

[99] J. Meza, J. Li, and O. Mutlu, “A Case for Small Row Buffers in Non-Volatile Main Memories,” in *ICCD*, 2012.

[100] Micron Technology, Inc., “Calculating Memory System Power for DDR3,” Technical Note TN-41-01, 2007.

[101] Y. Moon, Y.-H. Cho, H.-B. Lee, B.-H. Jeong, S.-H. Hyun, B.-C. Kim, I.-C. Jeong, S.-Y. Seo, J.-H. Shin, S.-W. Choi, H.-S. Song, J.-H. Choi, K.-H. Kyung, Y.-H. Jun, and K. Kim, “1.2V 1.6Gb/s 56nm 6F2 4Gb DDR3 SDRAM with Hybrid-I/O Sense Amplifier and Segmented Sub-Array Architecture,” in *ISSCC*, 2009.

[102] T. Moscibroda and O. Mutlu, “Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems,” in *USENIX Security*, 2007.

[103] J. Mukundan and J. F. Martínez, “MORSE: Multi-objective Reconfigurable Self-Optimizing Memory Scheduler,” in *HPCA*, 2012.

[104] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, “Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning,” in *MICRO*, 2011.

[105] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A Tool to Model Large Caches,” *HP Laboratories*, 2009.

[106] O. Mutlu, “Memory Scaling: A Systems Architecture Perspective,” in *IMW*, 2013.
[107] O. Mutlu, “The RowHammer Problem and Other Issues We May Face As Memory Becomes Denser,” in *DATE*, 2017.
[108] O. Mutlu and J. S. Kim, “RowHammer: A Retrospective,” *TCAD*, 2020.
[109] O. Mutlu and T. Moscibroda, “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,” in *MICRO*, 2007.
[110] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,” in *ISCA*, 2008.
[111] O. Mutlu and L. Subramanian, “Research Problems and Opportunities in Memory Systems,” in *SUPERFRI*, 2014.
[112] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, “Fair Queuing Memory Systems,” in *MICRO*, 2006.
[113] M. O’Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, “Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems,” in *MICRO*, 2017.
[114] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh, “Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems,” in *ASPLOS*, 2013.
[115] M. Patel, J. Kim, T. Shahroodi, H. Hassan, and O. Mutlu, “Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics,” in *MICRO*, 2020.
[116] M. Patel, J. S. Kim, H. Hassan, and O. Mutlu, “Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices,” in *DSN*, 2019.
[117] M. Patel, J. S. Kim, and O. Mutlu, “The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions,” in *ISCA*, 2017.
[118] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks,” in *USENIX Security*, 2016.
[119] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, “NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories,” *IEEE CAL*, 2020.
[120] S. Rixner, “Memory Controller Optimizations for Web Servers,” in *MICRO*, 2004.
[121] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory Access Scheduling,” in *ISCA*, 2000.
[122] SAFARI Research Group, “Ramulator: A DRAM Simulator — GitHub Repository.” https://github.com/CMU-SAFARI/ramulator
[123] T. Schloesser, F. Jakubowski, J. v. Kluge, A. Graham, S. Slesazeck, M. Popp, P. Baars, K. Muemmler, P. Moll, K. Wilson, A. Buerke, D. Koehler, J. Radecker, E. Erben, U. Zimmermann, T. Vorrath, B. Fischer, G. Aichmayr, R. Agaiby, W. Pamler, T. Schuster, W. Bergner, and W. Mueller, “6F2 Buried Wordline DRAM Cell for 40nm and Beyond,” in *IEDM*, 2008.
[124] H. Seol, W. Shin, J. Jang, J. Choi, J. Suh, and L. S. Kim, “Energy Efficient Data Encoding in DRAM Channels Exploiting Data Value Similarity,” in *ISCA*, 2016.
[125] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “The Dirty-Block Index,” in *ISCA*, 2014.
[126] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Fast Bulk Bitwise AND and OR in DRAM,” *IEEE CAL*, 2015.
[127] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, 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.
[128] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology,” in *MICRO*, 2017.
[129] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses,” in *MICRO*, 2015.
[130] V. Seshadri and O. Mutlu, “Simple Operations in Memory to Reduce Data Movement,” in *Advances in Computers, Volume 106*, 2017.
[131] V. Seshadri and O. Mutlu, “In-DRAM Bulk Bitwise Execution Engine,” in *Advances in Computers*, 2020, available as arXiv:1905.09822 [cs.AR].
[132] W. Shin, J. Yang, J. Choi, and L.-S. Kim, “NUAT: A Non-Uniform Access Time Memory Controller,” in *HPCA*, 2014.
[133] A. Snavely, D. M. Tullsen, and G. Voelker, “Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor,” in *ASPLOS*, 2000.
[134] Y. H. Son, S. O, Y. Ro, J. W. Lee, and J. H. Ahn, “Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations,” in *ISCA*, 2013.
[135] Y. Song and E. Ipek, “More Is Less: Improving the Energy Efficiency of Data Movement via Opportunistic Use of Sparse Codes,” in *MICRO*, 2015.
[136] M. R. Stan and W. P. Burleson, “Bus-Invert Coding for Low-Power I/O,” *TVLSI*, 1995.
[137] Standard Performance Evaluation Corporation, “SPEC CPU® 2006.” https://www.spec.org/cpu2006/
[138] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, “The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies,” in *ISCA*, 2010.
[139] 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.
[140] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, “BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling,” *TPDS*, 2016.
[141] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, “Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement,” in *ASPLOS*, 2010.
[142] Transaction Processing Performance Council, “TPC Benchmarks Overview.” http://www.tpc.org/information/benchmarks5.asp
[143] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, “Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores,” in *ISCA*, 2010.
[144] H. Usui, L. Subramanian, K. K. Chang, and O. Mutlu, “DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators,” *TACO*, 2016.
[145] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, “The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs,” in *ISCA*, 2018.
[146] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, “A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory,” in *ISCA*, 2018.
[147] Y. Wang, A. Tavakko, L. Orosa, S. Ghose, N. M. Ghiasi, M. Patel, J. S. Kim, H. Hassan, M. Sadrosadati, and O. Mutlu, “Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration,” in *MICRO*, 2018.
[148] Wei Zhao and Yu Cao, “New Generation of Predictive Technology Model for Sub-45nm Design Exploration,” in *ISQED*, 2006.
[149] D. H. Woo, N. H. Seong, and H.-H. S. Lee, “Pragmatic Integration of an SRAM Row Cache in Heterogeneous 3-D DRAM Architecture Using TSV,” *TVLSI*, 2013.
[150] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *ISCA*, 1995.
[151] M. Xie, D. Tong, K. Huang, and X. Cheng, “Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning,” in *HPCA*, 2014.
[152] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” in *ICCD*, 2012.
[153] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, “Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures,” in *MICRO*, 2009.
[154] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, “PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms,” in *RTAS*, 2014.
[155] 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.
[156] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang, “Restore Truncation for Performance Improvement in Future DRAM Systems,” in *HPCA*, 2016.
[157] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, “Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency,” in *MICRO*, 2008.
[158] 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,” U.S. Patent 5,630,096, 1997.