# DSM: A Case for Hardware-Assisted Merging of DRAM Rows with Same Content

SEYED ARMIN VAKIL GHAHANI*, Pennsylvania State University, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
JAGADISH B. KOTRA, AMD Research, USA

The number of cores and the capacities of main memory in modern systems have been growing significantly. Specifically, memory scaling, although at a slower pace than computation scaling, provided opportunities for very large DRAMs with Terabytes (TBs) capacity. Consequently, addressing the performance and energy consumption bottlenecks of DRAMs is more important than ever.

DRAM memory refresh operation is one of the main contributing factors to the memory overheads, especially for large capacity DRAMs used in modern servers and emerging large-scale data centers. This paper addresses the memory refresh problem by leveraging the fact that most cloud servers host virtualized systems that use similar kernels, libraries, etc. We propose and experimentally evaluate a novel approach that exploits this observation to address the DRAM refresh overhead in such systems.

More specifically, in this work, we present DSM, a light-weight hardware extension in memory controller to detect the pages with same content in memory and refresh only one of them and redirect the requests to the others to this page. Our detailed experimental analysis shows that the proposed DSM design can reduce 99$^{th}$ percentile memory access latency by up to 2.01x, and it also reduces the overall memory energy consumption by up to 8.5%.

CCS Concepts: • **Hardware → Dynamic memory**; **Power estimation and optimization**; **Memory and dense storage**; • **Software and its engineering →** *Memory management*.

Additional Key Words and Phrases: DRAM; Memory Refresh; Virtualized systems; KSM

**ACM Reference Format:**
Seyed Armin Vakil Ghahani, Mahmut Taylan Kandemir, and Jagadish B. Kotra. 2020. DSM: A Case for Hardware-Assisted Merging of DRAM Rows with Same Content. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2, Article 33 (June 2020), 26 pages. https://doi.org/10.1145/3392151

## 1 INTRODUCTION

Thanks to Moore's law, compute cores have evolved from high Instruction Level Parallelism (ILP)-based uni-core systems to Thread Level Parallelism (TLP)-based multi-core and many-core systems. Additionally, modern cloud environments started employing several heterogeneous computing resources, including CPUs, GPUs, and FPGAs [2, 16, 33, 35, 59, 64]. To cope with the memory demands imposed by these computing elements, data center environments started deploying high capacity memory resources or dis-aggregated memories [50, 51] that run into several Terabytes

*Seyed Armin Vakil Ghahani was mentored by Jagadish Kotra on this project.

Authors' addresses: Seyed Armin Vakil Ghahani, arminvakil@psu.edu, Pennsylvania State University, State College, PA, 16802, USA; Mahmut Taylan Kandemir, kandemir@psu.edu, Pennsylvania State University, State College, PA, 16802, USA; Jagadish B. Kotra, jagadish.kotra@amd.com, AMD Research, Austin, Texas, 78735, USA.


2476-1249/2020/6-ART33 $15.00
https://doi.org/10.1145/3392151

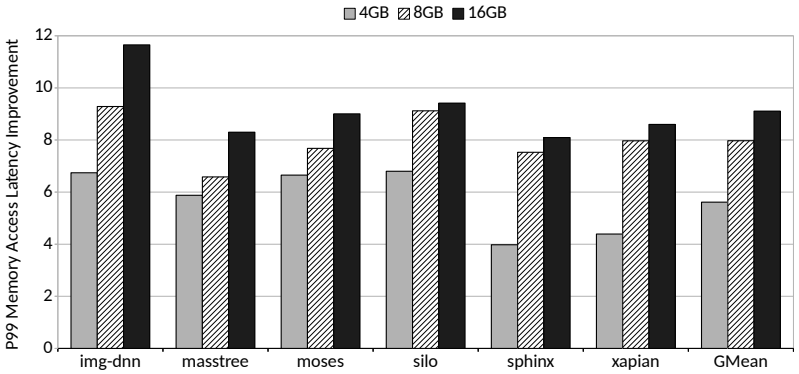


Fig. 1. The 99$^{th}$ percentile memory access latency improvement when memory refresh is disabled in the system for 4GB, 8GB, and 16GB DRAMs.

(TBs) of capacity [4, 5, 18, 20, 23, 34, 63]. To address this demand, memory capacity has increased from several hundred Megabytes (MBs) to Gigabytes (GBs) to several Terabytes (TBs) of DRAM memory. Although computation and memory capacity have scaled over the years, the number of pins connecting DRAM memory to computation cores have not scaled at the same pace, causing DRAM memory bandwidth per core to reduce over time [6–9]. This leads to a reduced number of memory channels per core, allowing memory scaling to confine to a channel. The ramifications of such scaling in DRAM capacity per channel has resulted in an increased number of rows per DRAM bank.

This type of memory scaling has presented unique problems for DRAM-based memories where the data is volatile and requires a periodic refresh. Unfortunately, DRAM refresh poses significant challenges with the scaled memories, especially since the retention time has not changed and remained constant. Consequently, a greater number of DRAM rows are to be refreshed within the same retention time, causing DRAM refresh to present energy and performance bottlenecks.

First, DRAM refresh contributes to a noticeable portion of power consumption in data centers. Due to the massive scale at which these data centers operate, power consumption plays a significant role in the Total Cost of Ownership (TCO) of these environments. DRAM power is slated to consume approximately 18% of the total power in a data center environment [11, 22, 27, 49, 81]. Further, of the total DRAM power, DRAM refresh accounted for 10-50% in 2Gb-64Gb DRAM devices [38, 55, 68]. Consequently, DRAM power optimizations play a crucial role in reducing the overall cost of ownership in data center environments.

Second, DRAM refresh imposes performance hiccups in memory access latency, which is expected to be even more problematic in larger DRAMs. One of the important performance metrics in cloud environments is tail latency of the system that plays a major role in the Quality-of-Service (QoS) of these systems [19]. Prior work [19, 21, 47, 60, 67, 83] show that different components in the system contribute to the tail latency, from hardware to application-level sources [47]. Memory access latency could adversely impact the tail latency for memory-intensive applications, as reported by Li *et al.* [47]. One of the significant memory events that contributes to memory access tail latency is the memory refresh operation, which is expected to be even more problematic in larger DRAMs. Figure 1 shows the improvement of the 99$^{th}$ percentile memory access latency of the system when memory refresh is disabled for different DRAM capacities. Therefore, DRAM refresh contributes significantly in the performance as well as consumed energy owing to the increased memory capacity in the data center environments.

In this paper, we present, *DRAM Same-Row Merging (DSM)*, a hardware extension that alleviates the overhead of DRAM refreshes. Our design leverages content similarity in DRAM rows to *minimize DRAM refresh overheads*. More specifically, DSM architecture skips refreshing rows that contain same data values, thereby saving DRAM refresh energy as well as reducing the adverse impact of DRAM refresh latency on the critical path of the on-demand requests stalled by refresh operations. This opportunity has been leveraged in software space by Kernel Same page Merging (KSM) [53, 79] for virtualized systems. KSM uses this information to de-duplicate pages containing the same content to save memory capacity. However, KSM leads to high performance penalties and significant tail latency overheads [74] as it breaks huge pages, and needs TLB shoot-downs [46, 65], which DSM completely eliminates.

Overall, in this work, we make the following contributions:

- We present a detailed analysis of the replication of memory contents for virtualized systems and discuss how to exploit this information for reducing the memory refresh overhead in the system.
- Based on the characterization results, we propose a low-overhead hardware extension (DSM) to skip refreshing duplicated rows in memory.
- We evaluate DSM on tail-latency sensitive and SPEC2006 applications and observe up to 2.01x improvement in $99^{th}$ percentile memory access latency by skipping up to 47.1% of the memory refresh commands. Also, DSM improves IPC by up to 4.1% and reduces overall memory energy consumption by up to 8.5%. We also estimate up to 8.2% improvement of $99^{th}$ percentile application latency.

## 2 BACKGROUND

In this section, we present a brief primer on the basic organization of DRAM, cover some details on how DRAM cells are refreshed, go over the in-memory computing techniques, and finally touch upon de-duplication in software space.

### 2.1 Basic DRAM Organization

DRAMs are organized hierarchically and are controlled by on-chip integrated memory controllers, as illustrated in Figure 2. Each memory controller manages Dual In-line Memory Modules (DIMMs) connected to it over a DRAM channel. Each DIMM is further divided into ranks, which are in turn composed of banks. For example, if there are two ranks per DIMM and eight banks per rank, we have a total of 16 banks per DIMM. Each DRAM bank consists of DRAM rows which are a series of DRAM cells connected horizontally by word-lines and vertically by bit-lines. DRAM cells are density-optimized and are comprised of 1T-1C cells. The access transistor (1T) and charge capacitor (1C) cells are arranged as a 2D mesh, connected by the horizontal word-lines and vertical bit-lines. Further, the bit-lines connect the DRAM cells to sense amplifiers.

Upon encountering memory access, memory controller issues a DRAM row activate $t_{RAS}$ command that connects all the DRAM cells connected to a DRAM row word-lines to the bit-lines. This causes the charge in DRAM rows to flow into the bit-lines, causing a deviation in the charge on the bit-lines. This deviation in charge in the bit-lines is sensed and amplified by the sense amplifiers, enabling the amplified charge to reside in the DRAM row-buffer. Once the $t_{RAS}$ duration elapses, memory controller issues a $t_{CAS}$ command, which reads the data corresponding to the cache line to on-chip over the memory channel. If the data needs to be read from a different row in the same bank and the row-buffer contains an open row, memory controller issues a precharge ($t_{PRE}$) command, which closes the currently open row in the row-buffer, before issuing an activate command corresponding to the next read or write.

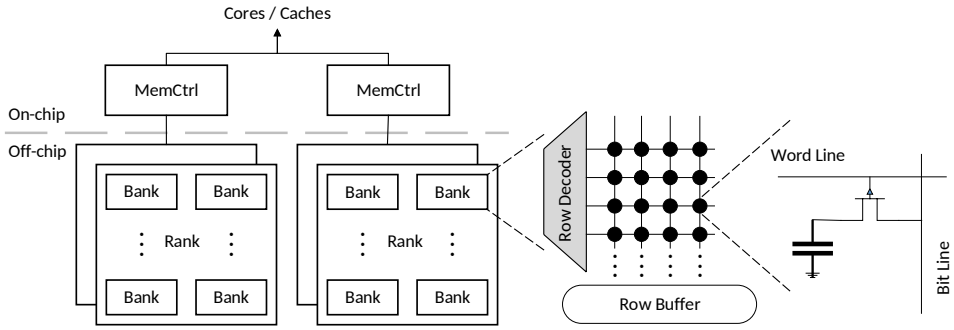


Fig. 2. DRAM organization.

## 2.2 DRAM Refresh

DRAM cells are volatile and lose charge over time. To maintain data integrity, DRAM cells need to be refreshed periodically. The data retention time of a DRAM is referred to as DRAM refresh window ($t_{REFW}$). This DRAM retention time ($t_{REFW}$) is typically in the order of several milliseconds, depending on the process variation of the DRAM cells and the operating temperature. For a DRAM operating in temperatures less than 85 deg. C, the $t_{REFW}$ is typically 64 msec, while it becomes 32 msec for a system operating at temperatures greater than 85 deg. C. Thus, depending on the operating temperature and the cell process variation, rows in a DRAM need to be refreshed with a certain frequency. A DRAM refresh command to a row essentially activates the row into the row-buffer and precharges it.

DRAM memory controllers split the $t_{REFW}$ retention time into small refresh intervals ($t_{REFI}$) and issue a refresh command once every $t_{REFI}$. Typically, $t_{REFI}$ is in the order of several microseconds. Depending on the operating temperature, the $t_{REFI}$ is either 7.8 usec or 3.9 usec. Further, depending on the type of the DRAM (e.g., regular DDR-x [used in desktops/servers] versus LP-DRAM [used in mobile environments]), there are two types of refresh granularities: (i) all-bank refresh and (ii) per-bank refresh. In all-bank refresh, all the banks in a rank are refreshed with a single refresh command. Thus, none of the banks in a rank is available for servicing the on-demand requests during a refresh operation. In comparison, in the per-bank refresh, a refresh command refreshes only the DRAM rows in a single bank, thereby enabling the other banks in the same rank to be available to serve on-demand accesses.

We want to emphasize that DRAM refresh operations involve activating and pre-charging all the DRAM rows in a refresh window of few milliseconds to maintain data integrity and, as a result, they adversely affect the performance of the system. The refresh operation (in an all bank refresh) locks the whole rank and consequently degrades the memory system throughput [55]. Moreover, the memory access latency also increases as the accesses to the ranks that are being refreshed encounter delays [15, 31, 55, 61, 62]. Further, DRAM refresh operations also degrade the energy efficiency in the system [15, 55]. Finally, the negative impact of DRAM refresh operations is expected to be even more problematic with increasing density [15, 41, 43, 54, 55, 62].

## 2.3 RowClone

A large body of work has been explored in the in-memory and near-memory designs to address the *Memory Wall* problem [1, 25, 26, 28, 37, 42, 45, 48, 57, 66, 72, 76]. One of the main advantages of in-memory computation is the reduction of required bandwidth and energy to move data between processor and memory. Bulk data copy is one of the common bandwidth-intensive operations that

has been studied by Seshadri *et. al* [72, 73]. RowClone is a technique developed for copying a bulk of data from one location in physical memory to another, *without the need of bringing the data on-chip* [72]. Our proposal leverages RowClone hardware modification to copy data in case of write accesses to rows that are merged as explained later in Section 4.

### 2.4 Kernel Same-Page Merging

Virtualization is widely employed today in both desktop and data center applications. Virtual machines (VMs) are especially appealing for server consolidation to fully utilize the underlying physical machine resources in a safe manner [30]. In such an environment, kernel same-page merging (KSM) [3] reduces the memory footprint by *de-duplicating* pages that have the same content.

When KSM detects two pages with the same content, if neither of the two pages was merged before, it modifies one of them to "*kpage*" (KSM page), makes it read-only and frees the other page [53]. If one of the pages is already a *kpage*, it just frees the other page. In the next step, KSM changes the page table entry (PTE) of the other page to point to the *kpage*. Consequently, both page table entries point to the same page in the system from now on. Upon a write request to a *kpage*, an exception occurs, and OS copies the content of the page to a free page in memory and updates the page table accordingly – a procedure similar to copy-on-write in Linux.

KSM could lead to high performance degradation and high tail latency in the system [74]. First, KSM consumes CPU cycles to execute de-duplication and pollutes caches to calculate the checksum of pages and perform comparison among them. Skarlatos *et al.* [74] proposed a hardware extension called PageForge, which addresses this problem by calculating checksum and comparing pages in memory controller. However, both KSM and PageForge cause TLB shoot-downs upon every de-duplication and un-merging of a de-duplicated page. In addition, KSM and PageForge aggressive de-duplication approach typically results in breaking huge pages [46], which in turn leads to increased cost for address translation, a critical overhead for virtualized systems. Therefore, in common case, KSM is configured to only perform de-duplication when the amount of free memory reaches below some threshold, 20% by default.

## 3 MOTIVATION

Server architectures consist of a high number of cores and large capacity DRAMs [70] or disaggregated memories [50, 51]. However, the energy and performance overheads of existing DRAM devices increase with larger capacity and DRAM refresh contributes significantly to this overhead.

Memory refreshes contribute to a considerable portion of the energy consumption in the system, which becomes an even more significant concern as systems employ higher capacity DRAMs [5, 38, 41, 54, 55, 68]. Figure 3 summarizes this trend for a range of DRAM capacities. As shown in this figure, the percentage of energy spent refreshing memory increases with larger DRAM capacities, emphasizing the fact that reducing the memory refresh overhead leads to higher energy savings as systems move toward larger DRAM capacities.

Moreover, DRAM refresh impacts performance negatively as the DRAM density increases [5, 38, 43, 55, 68]. In this context, Qureshi et. al [68] show that the potential speedup that could be achieved by removing memory refreshes is about 4% in an 8Gb device and increases to 54% in a 64Gb device. Figure 1 plots the $99^{th}$ percentile memory access latency improvement for the Tailbench suite [40] when disabling DRAM refresh (*Ideal DRAM*) for three different DRAM chip densities. It can be observed from these results that memory access tail latency improvement brought by removing memory refresh overhead increases significantly as the chip densities become larger.

In this work, we address both energy and performance overheads brought by DRAM refreshes in server architectures by reducing the unnecessary memory refresh operations, leveraging the fact that cloud server architectures mostly host virtualized machines. VMs typically use the same

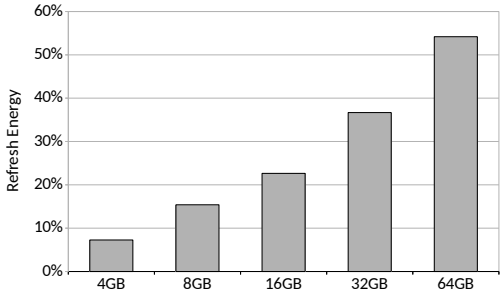


Fig. 3. DRAM refresh contribution to total DRAM energy consumption for 4GB, 8GB, 16GB, 32GB, and 64GB capacities.

set of kernels, libraries, and even applications and, as a result, there can be many rows in memory that have the same content.

Our design detects the rows with the same content and it refreshes only one row, instead of refreshing all of them. This opportunity is exploited by KSM [3] in virtualized systems domain to reduce memory footprint. However, KSM causes high performance penalties as it leads to splintering large 2MB pages into base 4KB pages (to allow de-duplication) and consequently TLB shootdowns [46, 65]. The use of KSM in such virtualized systems executing latency-sensitive applications could increase the 95th percentile latency by up to 5X [74], which is the key performance metric for such applications [19, 40, 78]. Our approach, on the other hand, uses same content values in virtualized systems for an entirely different purpose.

Figure 4 plots the pages that have the same content in the Tailbench suite [40] and different mixes of Tailbench and SPEC2006 [32] applications by KSM from the time the system boots-up until the program ends. For this figure, we evaluated four VMs (each 2GB memory) on Intel Core i7-8700K with 8GB main memory and let KSM to de-duplicate as many pages as it can in 5 minutes. After this bootup time, we execute different mixes of benchmarks on 4 VMs, as described with more details in Section 6. One can see from these results that the number of de-duplicated pages can be from kernels and libraries before the benchmark execution and also, after the benchmark execution in some benchmarks. For example, in Img-DNN, 260K and 150K pages are de-duplicated by KSM *before* and *during* the benchmark execution, respectively.

Based on these observations, we leverage the pages with same content in memory to reduce the total number of DRAM refreshes for both reducing energy consumption and improving performance.

## 4 DSM DESIGN

We propose DSM, a hardware extension that leverages same content rows in memory to address the high performance and energy overhead of DRAM refresh in server environments. Our goal is to detect the rows with same content in DRAM and only refresh one row (referred to as representative row) from each group (of same content rows). Since servers typically host virtualized systems, there is a significant number of rows that have same values in such systems. Thus, we exploit this opportunity and group the rows with the same content as one, removing the need for refreshing all these memory rows.

To collect the required information for such a design goal, we use an important data structure, *Mapping Table*, to guarantee that data are retrievable even if the corresponding row has not been refreshed. We use the term *merged rows* to refer to the rows whose refreshes are skipped while

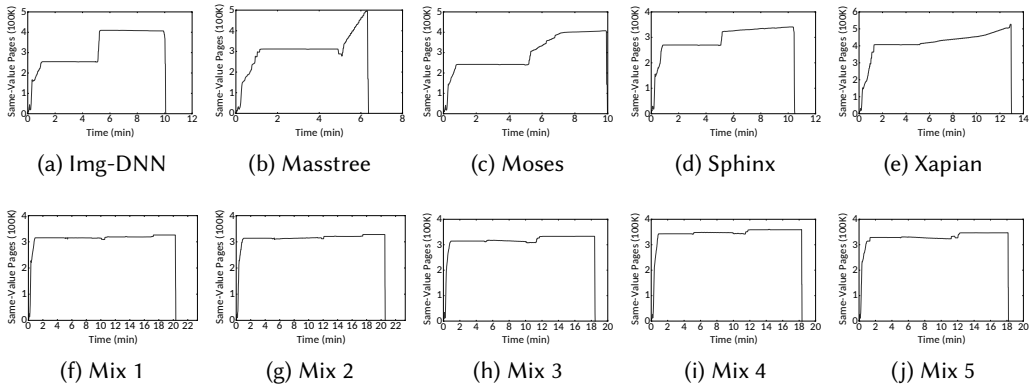


Fig. 4. The number of pages with the same content from the system bootup until the end of applications.

their accesses are redirected to their corresponding representative row by memory controller. In our design, the information stored in the Mapping Table assists memory controller in identifying whether the row is merged or not upon memory access. Memory controller then issues the appropriate commands to keep memory in a consistent state.

### 4.1 Representative Rows

The naïve solution to *group rows with same value as one* is to maintain *row* → *row* mapping for all rows in memory in Mapping Table. Therefore, each entry in Mapping Table shows whether the corresponding row in DRAM is mapped to another row and if so, which row is its representative. Consequently, entries in Mapping Table should be large enough so it could point to any other row in DRAM since any row in DRAM can be a representative row. Memory controller then can use Mapping Table to redirect merged rows' memory accesses to their representatives. Additionally, memory controller uses this information to only refresh representative rows, instead of all the rows that are merged.

However, implementing this naïve approach is *not* practical since it leads to high hardware overhead for Mapping Table. Moreover, this scheme increases the performance penalty in case of a write to a representative row. Upon this event, memory controller should copy the content of the representative row to all rows that are mapped to it. Also, the mapping of all those merged rows has to be updated by memory controller. As a result, the naïve solution must keep track of the rows that are mapped to each representative row.

Instead, we reserve a set of rows, called *R-Row*s, which are not accessible by OS and are meant to keep the content of the representative rows. Therefore, each row can only be mapped to one of the *R-Row*s and we maintain *row* → *R-Row* in Mapping Table. Consequently, each entry in Mapping Table is the index of the *R-Row* that the corresponding row has been mapped to. The rows will be mapped to a reserved index, if they are not merged. Additionally, our proposal will not incur the high overhead of write accesses to representative rows in comparison to the naïve solution. DSM only allows *R-Row*s to be representative, a design choice that guarantees there will not be any write access to representative rows since they are not accessible by OS. Moreover, merged rows are mapped to the *R-Row*s in the same channel which averts cross-channel mappings in DSM architecture.

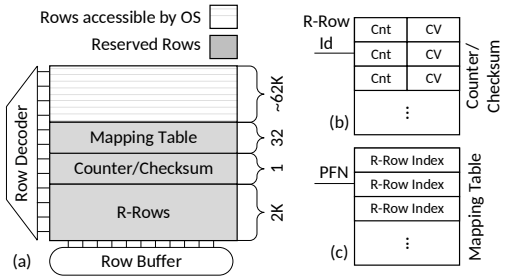


Fig. 5. (a) Overview of DSM metadata in each bank (b) Counter/Checksum array data structure. (c) Mapping Table data structure.

Figure 5(a) illustrates possible location of *R-Row*s in each bank. Please note that it is not required to keep *R-Row*s in any specific rows in memory – they could be in any row, as long as they are not accessible by OS.

*4.1.1 R-Row Counter.* A 1-byte counter is associated with every *R-Row* that captures the number of rows that have been mapped to this *R-Row* and, as a result, are not refreshed. Upon a write to a row that has been merged with an *R-Row*, the corresponding *R-Row* counter will be decremented. When the counter of an *R-Row* reaches zero, this *R-Row* is freed by memory controller and can be allocated to another row value. This counter is also used by *Replacement Procedure*, which will be discussed in Section 4.4.5.

*4.1.2 R-Row Checksum.* Values that are captured by *R-Row*s are perfect candidates for comparing against the rows that have not been merged and fetched by DSM for potential merging. Therefore, we keep a 1-byte checksum for every *R-Row*s and use it to determine the potential candidate rows that can be merged. As the *R-Row* content is not expected to be modified frequently, maintaining the checksum is quite useful since it reduces the need to calculate the checksum in every merging period for *R-Row*s.

R-Row Counter and Checksum arrays are located in the same data-structure and are kept off-chip to reduce the overhead of DSM, as shown in Figure 5(b). Each entry of this data-structure contains two 1-byte values: (i) Cnt: the number of rows that are mapped to this *R-Row*, and (ii) CV: Checksum Value of this *R-Row*. These arrays are cached for faster access by memory controllers.

## 4.2 Mapping Table

Mapping Table stores the index of the *R-Row* that each row has been mapped to, as depicted in Figure 5(c). If the row is not merged we use an reserved index indicating this case. Additionally, we reserve another index to indicate if the row contains zero value, *Zero Row*, without allocating an *R-Row* to this value. This data structure is necessary for the correctness of the approach, as we only refresh a subset of rows in the refresh intervals – allocated *R-Row*s and rows that are not merged. Using the information provided by the Mapping Table, memory controller is able to issue correct commands based on the state of the rows and guarantee that no data will be lost.

## 4.3 Caching DSM Metadata

Mapping Table has an entry for each row in memory and, as a result, it is not practical to keep it on-chip. In addition, keeping R-Row Counter and R-Row Checksum arrays on-chip is also expected to result in too much overhead. Observing this, DSM distributes different parts of the Mapping Table, *R-Row* Counter and *R-Row* Checksum array over different banks in memory, as illustrated

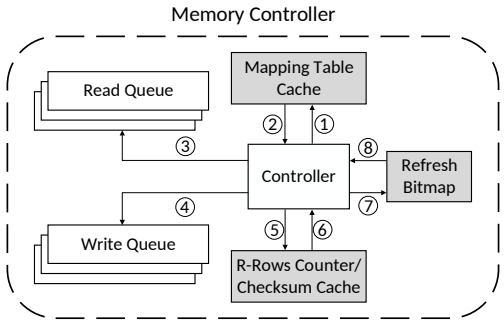


Fig. 6. DSM design overview, and interaction of DSM logic and other components of memory controller.

in Figure 5(a). On the other hand, the information captured by Mapping Table is crucial for serving data accesses. Accesses to rows that have been merged (and consequently not being refreshed) should be redirected to the corresponding *R-Row*; otherwise, the requests would be served by incorrect data (from the refreshed-skipped rows). Also, accessing off-chip Mapping Table results in a significant latency addition to on-demand memory requests. Since some rows are accessed more frequently than others (hot pages), we maintain an on-chip cache that holds the Mapping Table information in memory controller for faster redirection. Additionally, we preserve another on-chip cache for *R-Row* Counter and *R-Row* Checksum array entries, as shown in Figure 6.

Upon encountering a miss in Mapping Table Cache (MTC), additional memory access is required to bring the mapping information on-chip. As a result, the MTC's hit rate is an important performance indicator of DSM to avoid extra off-chip memory accesses. Thus, our proposed design also updates the *page walk procedure* in such a way that, after a TLB miss is resolved, DSM brings the corresponding Mapping Table entry for the page that caused the TLB miss on-chip. Such a proactive prefetching of the remapping table entry into the MTC upon a page walk results in reduced latency of fetching the Mapping Table entry to MTC. This approach is expected to improve the performance of the MTC significantly, as every page that is going to be accessed for the first time by a processor should access its page table entry first. DSM leverages this opportunity to bring the corresponding Mapping Table entry to MTC before the actual memory access.

### 4.4 Putting It All Together

In this subsection, we describe in detail how the different components of DSM interact with each other. Also, we discuss how memory controller handles accesses to memory and refresh operations in the presence of our hardware extension. Figure 6 illustrates the high-level overview of DSM as well as the interactions between different components.

*4.4.1 DRAM Read Operation in DSM.* DRAM read requests reside in a DRAM read queue. As the request is queued in the DRAM queue, MTC is accessed ① to retrieve the *true* row address corresponding to the read request ②. Memory controller changes the address of the read request ③ to the corresponding *R-Row* in case the row has been merged. If the mapping indicates that the row is merged with a *Zero Row*, the request can be serviced without accessing the DRAM, and the corresponding entry in the read queue will be removed.

*4.4.2 DRAM Write Operation in DSM.* Servicing write request also involves accessing MTC ① to obtain the mapping information and state of the accessing row ②. The write request for rows that are not merged will be serviced with no additional overhead since no update to any DSM metadata is required in this case. On the other hand, if the write request belongs to a merged

row, DSM has to copy the content of its *R-Row* ④ to this row before sending the actual write request. This operation is necessary to retrieve the previous value of the row since the refreshes to the merged rows are skipped. We use *RowClone* [72] to perform this row-copy operation in-memory, in an attempt to reduce the overhead of this operation. Memory controller does not issue a RowClone [72] command if the row has been merged into a *Zero Row*. As a result, no additional overhead would be incurred in this case. Once the write is done, the Mapping Table entry for the written row ① and the corresponding R-Row Counter ⑤ (for non-zero merged pages) need to be updated for future accesses.

*4.4.3 DRAM Refresh Operation in DSM.* DSM allows refreshing a *subset of rows* in each bank, based on the state of rows in the system. To be more specific, memory controller uses *Refresh Bitmap* to issue refresh commands only for rows that are not merged ⑧. The Refresh Bitmap is created from the contents of the Mapping Table on the fly before each refresh interval ⑦. Since DRAM refresh command is only issued once in a refresh interval of $t_{REFI}$, DSM can gather this bitmap in the background for the next refreshed rows without increasing the refresh operation duration. DSM collects this information from the corresponding entries of the Mapping Table for the bank that is going to be refreshed, and stores in a 4KB flip-flop in Refresh Bitmap.

The content of the Refresh Bitmap need to be updated if a write is incurred to one of the rows populated in the Refresh Bitmap in the background before the refresh is issued. During the refresh operation, refresh bitmap contents are written to the additional buffer at the refresh address register present in the DRAM device. This enables skipping the refreshes to the rows based on the bit-vector written by memory controller. The additional support we assume to write this bit-vector to the DRAM substrate is similar to that presented by researchers in [13]. Depending on the type of refresh employed by the DRAM, that is, either per-bank refresh or all-bank refresh [15, 43], a refresh command issued by memory controller can refresh DRAM rows within a single bank or all the banks in a rank. However, since DSM communicates the bit-vector per-bank, in the case of an all-bank refresh, multiple bit-vectors need to be written to separate private per-bank buffers in the DRAM device for the refresh module in DRAM to skip refreshes to DRAM rows successfully, similar to [13] proposal.

*4.4.4 Periodic Content Checking.* DSM fetches 400 rows that are not merged every five milliseconds for potential merging. DSM calculates the checksum value of each row that is fetched for performing the comparison. It, then, searches for a checksum match in *R-Row*s checksum cache in memory controller ⑤. If DSM finds an *R-Row* with the same checksum ⑥, it fetches the corresponding *R-Row* and checks whether they are same or not, also in memory controller. In case of a match, DSM proceeds to merge the rows, which involves an update to the Mapping Table ①, and R-Row Counter Cache ⑤. In case DSM could not find any match in *R-Row*s, it searches for a free *R-Row* to add this fetched row to *R-Row*s for potential merges in future. If DSM finds a free *R-Row*, it will copy the fetched row to the free *R-Row* and update the Mapping Table and Counter/Checksum Array accordingly. Otherwise, DSM fails to merge the fetched row and discards its value.

*4.4.5 Replacement Procedure.* *R-Row*s in our hardware are proposed to reduce the performance overhead of write accesses to *R-Row*s, and also to reduce the hardware overhead of each Mapping Table entry. However, since the number of *R-Row*s is limited, it is more efficient to keep values in the *R-Row*s that have higher sharing opportunities. In order to guarantee that *R-Row*s are not populated with values that are not frequent, we employ a *Replacement Procedure*. This procedure is invoked when the number of free *R-Row*s in a bank reaches a $T_{low}$ threshold and continues to free *R-Row*s that have low sharing patterns until the number of free *R-Row*s in the corresponding bank reaches $T_{high}$.

The victim candidates are chosen from the information that is extracted from the *R-Row* counter data structure ⑤. *R-Row*s with zero counter value are the best candidates for eviction since there is not any row that is mapped to them and there is no need to modify Mapping Table. Freeing an *R-Row* with a non-zero counter needs more caution as the rows(s) that are mapped to this *R-Row* have to be copied (via RowClone [72] command) before this *R-Row* is freed and the Mapping Table also needs to be updated. Although the counter provides enough information to choose the victim, we do not have enough information to identify the rows that have been mapped to this *R-Row*. As a result, we use refresh intervals to extract this information and perform *Replacement Procedure*, as we already fetch the Mapping Table to create the Refresh Bitmap.

## 5 DISCUSSION

In this section, we touch upon the interactions of DSM with the different software and hardware components of the overall system.

### 5.1 Interactions with Cache Coherence

All modern processors employ caches that might contain data in the modified state with respect to the content in memory. Since DSM architecture performs content similarity based on the data in DRAM, it is possible that DSM can aggressively merge a DRAM row with another though one of its cache lines is in the modified state in the cache. This is possible as DSM is unaware of the modified data in the cache hierarchy. In such scenarios, DSM architecture does not incur any issues with functional correctness since (i) any load instruction will be served from the recent value in cache hierarchy (ii) the modified cache line in the cache hierarchy will eventually be written back to DRAM upon eviction. As a result, if the row that contains the dirty cache block has been merged prior to the cache block eviction, DSM will unmerge it.

Hence, DSM can pro-actively skip refreshes to rows resulting in a performant and energy-efficient system without any adverse impact on cache coherence protocols. On the other hand, prior work [69, 74] have to take into account the data in the cache hierarchy by accessing the cache multiple times for every cache block in the rows that are being compared. Thus, they consume a lot of cache lookup bandwidth and energy.

### 5.2 Interactions with KSM

Kernel Same page Merging (KSM) is a software module implemented in hypervisors like VMWare ESXi and Qemu-KVM [53, 79] that targets de-duplicating memory pages based on the available free space in the system. Since DSM architecture already performs content similarity and merges the pages, KSM can potentially leverage the content similarity information from the DSM Mapping Table.

The potential interaction of KSM with DSM is a function of address mapping policies employed by the hardware to map a physical page to the corresponding memory channel/rank/bank etc. For example, since DSM maintains the content similarity information at a DRAM row granularity, if the underlying address mapping policy interleaves across DRAM rows at a page (4KB) granularity, KSM can readily leverage the content similarity information maintained by the DSM. Thus, KSM can promptly know which DRAM pages (rows) contain the same content values by reading the Mapping Table maintained by DSM. Consequently, DSM will have to unmerge the rows that KSM de-duplicates, which will potentially reduce the number of refreshes skipped by DSM. Thus, DSM can interact with KSM when the address interleaving across DRAM rows is at page granularity. However, when the address interleaving across DRAM rows is at a sub-page granularity, KSM cannot leverage the content similarity information managed by the DSM and will have to perform the page comparison to understand the content similarity across pages. Therefore, as explained

Table 1. Important architectural parameters of the modeled system.

| Component | Parameters |
|---|---|
| Processor | 4 cores, OoO 192-entry window, 8-wide issue, 32/32 LD/ST queue entries, 3GHz |
| L1I and L1D | 32KB per core, 64-byte cache line, 8-way associative |
| I-TLB and D-TLB | 64 entries per core |
| L2 | 256KB per core, 64-byte cache line, 16-way associative |
| Last-level Cache | 4MB shared, 64-byte cache line, 32-way associative |
| Memory Controller | One per channel, 64-entry read queue, 64-entry write queue |
| Main-Memory | 8GB DDR3-1600 (8-8-8), 2 channels, 2 rank per channel, 8 banks per rank 64K rows per bank, 4KB per row |
| Host and Guest | Ubuntu Server 16.04, QEMU-KVM [12] as hypervisor |
| DSM | 32K R-Rows in memory 32KB Mapping Table Cache (MTC) 4KB Counter Checksum Cache $T_{low} = 32, T_{high} = 64$ |

above, DSM will need to unmerge the DRAM rows de-duplicated by KSM, which reduces the number of skipped refreshes.

### 5.3 Unallocated Pages

Initially, at bootup time of the system, all rows corresponding to unallocated DRAM rows contain zero value (for security reasons [44]) as well as Mapping Table. As a result, all rows will merge to *Zero row* at bootup, by default and after writes to memory, DSM will unmerge rows, gradually. In other words, a row in memory will not be refreshed until a write access targeting that row reaches memory controller. The reduced overhead of refreshing the unallocated rows in our design is a direct benefit of our design choice to initialize every row to be a *Zero row*.

Isen *et al.* [36] uses the semantic directives to remove the overhead of unallocated rows. However, it only can identify and remove the memory refresh for a subset of what DSM can detect and eliminate as DSM also detects the zero rows that are allocated, in addition to non-zero values that DSM identifies. Moreover, Eskimo [36] relies on an ISA modification to convey the information from the program to the hardware [36], requiring a hardware-software co-design.

## 6 METHODOLOGY

The server architecture that we model in this work has a 4-core processor and an 8GB main memory, as described in detail in Table 1. We use Ubuntu Server 16.04 [77] with QEMU-KVM [12] as the hypervisor in our experiments and run four VM instances. Each VM runs Linux kernel 4.15 and is pinned to a core. We set the size of memory for each VM to 2GB, similar to general-purpose Linux-based VM instances in Microsoft Azure [59].

Table 2. Evaluated workloads.

| Name | Benchmarks |
|---|---|
| Mix 1 | libquantum, mcf, img-dnn, xapian |
| Mix 2 | libquantum, mcf, moses, sphinx |
| Mix 3 | libquantum, img-dnn, masstree, moses |
| Mix 4 | libquantum, img-dnn, silo, sphinx |
| Mix 5 | libquantum, masstree, moses, silo |
| Mix 6 | mcf, img-dnn, sphinx, xapian |
| Mix 7 | mcf, masstree, silo, xapian |
| Mix 8 | mcf, moses, silo, xapian |
| Mix 9 | masstree, moses, img-dnn, sphinx |
| Mix 10 | masstree, silo, sphinx, xapian |

Table 3. Workloads description.

| Benchmark | Description | QPS |
|---|---|---|
| img-dnn | Handwriting recognition application | 500 |
| masstree | Scalable in-memory key-value store | 1000 |
| moses | Statistical machine translation | 100 |
| silo | In-memory transactional database | 2000 |
| sphinx | Speech recognition system | 1 |
| xapian | Online search engine | 500 |
| libquantum | Quantum computer simulator library | - |
| mcf | Vehicle scheduling in public transportation | - |

Moreover, in order to estimate the tail latency improvement of proposed approach, we employ BigHouse [58] simulator to simulate the queuing system with a confidence level of 95%; We use the calculated IPC in gem5 [14] to determine the service rate of an FCFS M/G/1 queuing system, similar to [60]. Our assumption about M/G/1 queuing system is aligned with prior work [39, 60, 82].

We also evaluated the area overhead and power consumption of our proposal with CACTI [10].

### 6.1 Experimental Setup

We implement DSM in gem5 [14] using the technique proposed in [71]. We setup gem5 using the X86KVMCPU model, which runs at native speed and we simulate 4 VMs. We wait till all the 4 VMs are booted. Then we simulate a combination of Tailbench [40] and SPEC2006 [32] benchmarks in each VM in X86KVMCPU mode until each core runs at least 250 Billion instructions to reach the region of interest, after which the caches are warmed up for 100 Million instructions using Out-of-Order cpus. We simulate and report the results of running 1 Billion instructions on each core. More specifically, we simulate ten mixes of workloads, described in Table 2 from Tailbench suite [40] and SPEC2006 [32] in three different configurations.

(i) Baseline: Memory controller refreshes all rows in DRAM.
(ii) DSM: Our proposed solution that reduces memory refresh overheads by detecting same-value rows and refreshing only the *R-Row*s and rows that are not merged in DRAM.

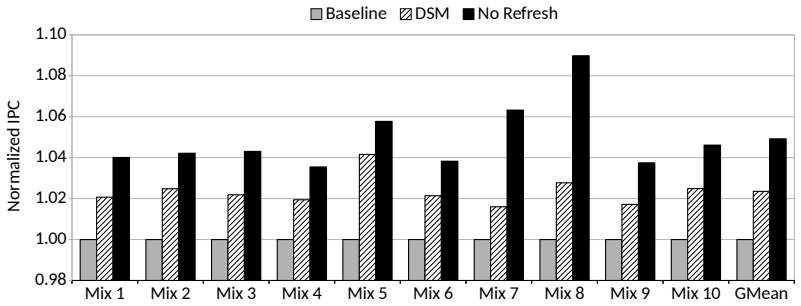


Fig. 7. Normalized performance results.

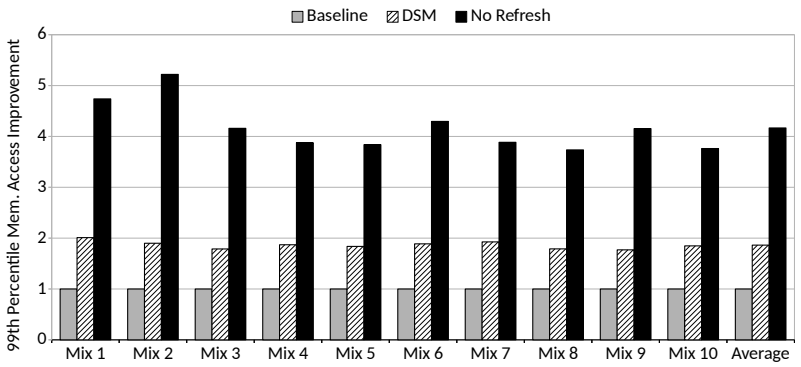


Fig. 8. $99^{th}$ percentile memory access latency improvement with DSM.

(iii) Disabled Refresh: In this configuration, we disable DRAM refresh completely. This configuration gives the best case energy and performance improvements and hence serves as our limit study.

All gem5 simulations are executed on c8220 machines of CloudLab Servers [24].

*6.1.1 Benchmarks:* We study six applications from the TailBench suite [40] as they represent tail-sensitive applications that are usually targeted in servers, as well as mcf and libquantum, two memory-sensitive applications from SPEC2006 [32]. Table 3 gives a brief overview and Queries Per Second (QPS) of the applications that we used in our evaluation.

## 7 EVALUATION

### 7.1 Performance Improvement Results

DRAM refresh has a negative impact on the performance of the system as it delays servicing the on-demand memory accesses, thereby stalling the server processor. These stalls pose severe bottlenecks for latency-critical applications. Figure 7 shows the performance improvement of DSM over baseline. As can be observed, DSM outperforms the baseline in all benchmarks and improves the performance by 2.4% on average while the maximum improvements are as high as 4.2% in Mix 5.

The memory refresh operations stall on-demand accesses to DRAM, and this, in turn, leads to variable response time from memory. As a result, refresh operations can have a significant impact on $99^{th}$ percentile memory access latency. Figure 8 plots $99^{th}$ percentile memory access latencies

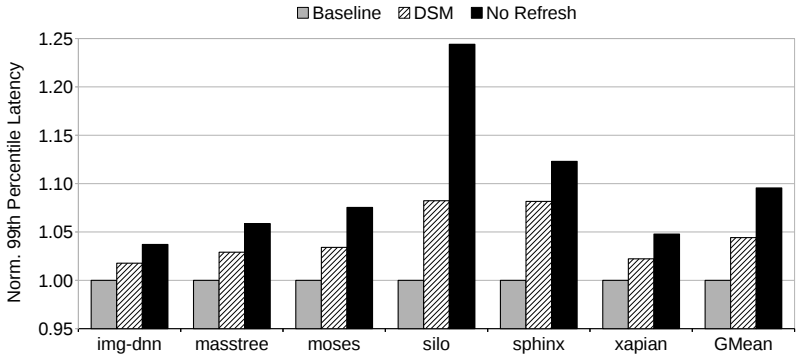


Fig. 9. $99^{th}$ percentile latency improvement with DSM.

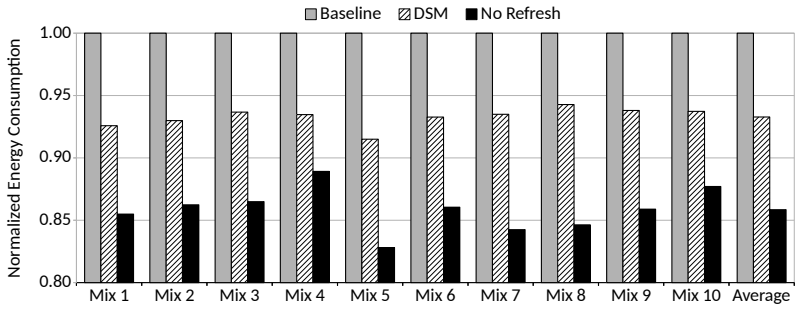


Fig. 10. Energy consumption results.

for the application programs in our hardware extension. As shown in this graph, our proposed approach can bring up to $2.01x$ reduction in $99^{th}$ percentile memory access latency. On average, our design improves $99^{th}$ percentile memory access latency by $1.86x$.

### 7.2 Tail Latency Improvement Results

Spikes in the latency of an application could be from different sources [19]. To illustrate the importance of memory latency and how DSM affects the application tail latency, we simulate a queuing system for Tailbench suite applications. Figure 9 shows that DSM can improve $99^{th}$ percentile latency by up to 8.2% in silo application and 4.4% on average across all tail-latency sensitive applications.

### 7.3 Energy Improvement Results

Figure 10 presents the energy consumption results for baseline, DSM and, No Refresh (limit study) approaches. It can be observed from these results that DSM can reduce the total energy consumption by up to 8.5%, and on average by 6.7%. These energy savings are going to be more significant [68] in high-density DRAMs as more cycles are spent refreshing rows as shown in Figure 10.

### 7.4 Bandwidth Consumption

DSM periodically fetches different rows of memory for page comparison and merges them in case of a match. As opposed to prior work [53, 69, 74] that drastically consume bandwidth without

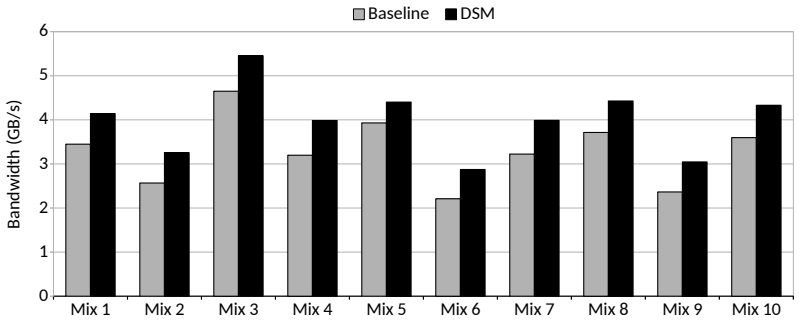


Fig. 11. Bandwidth consumption of DSM.

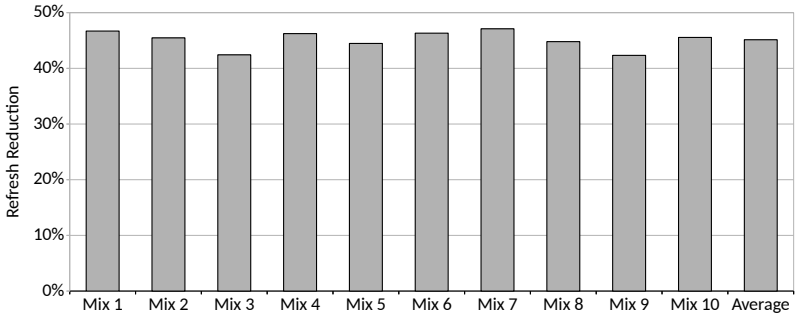


Fig. 12. Percentage of rows DSM skips refreshing.

increasing the throughput, DSM slightly increases the bandwidth consumption by up to 0.8 GB/s in Mix 3 without occupying any of cores in the processor, as illustrated in Figure 11.

### 7.5 Memory Refresh Skipping Results

As explained previously, DSM leverages content similarity across rows and skips refreshing the corresponding rows. Figure 12 demonstrates the efficacy of DSM as it presents the percentage of rows that are skipped refreshing. As can be observed, DSM skips refreshing up to 47.1% of the DRAM rows in masstree application, whereas, it skips refreshing 45.1% across all the workloads, on average.

### 7.6 DSM Content Similarity Results

The number of *R-Row*s that are allocated in DSM contributes to the required area overhead of DSM, as it affects the bits required in the Mapping Table entries. Figure 13 plots the number of rows that a value is repeated in memory. For example, it shows that there are around 10K row values in memory that are repeated twice in all benchmarks (the second data point in each mix). Moreover, it also indicates that there is a value in memory that is repeated more than 2000 times in all mixes (the last data point). The number of row values that are repeated more than once is from 29K to 40K in each mix (sum of all data points except the first one). This figure also shows that the number of rows whose values are repeated more than 255 times is 4 out of 29K. As a result, we set the length of each R-Row Counter to 8-bits and do not map more than 255 rows to any given *R-Row*. In case of an overflow, DSM allocates another *R-Row* for this value to avoid losing the merge opportunity.

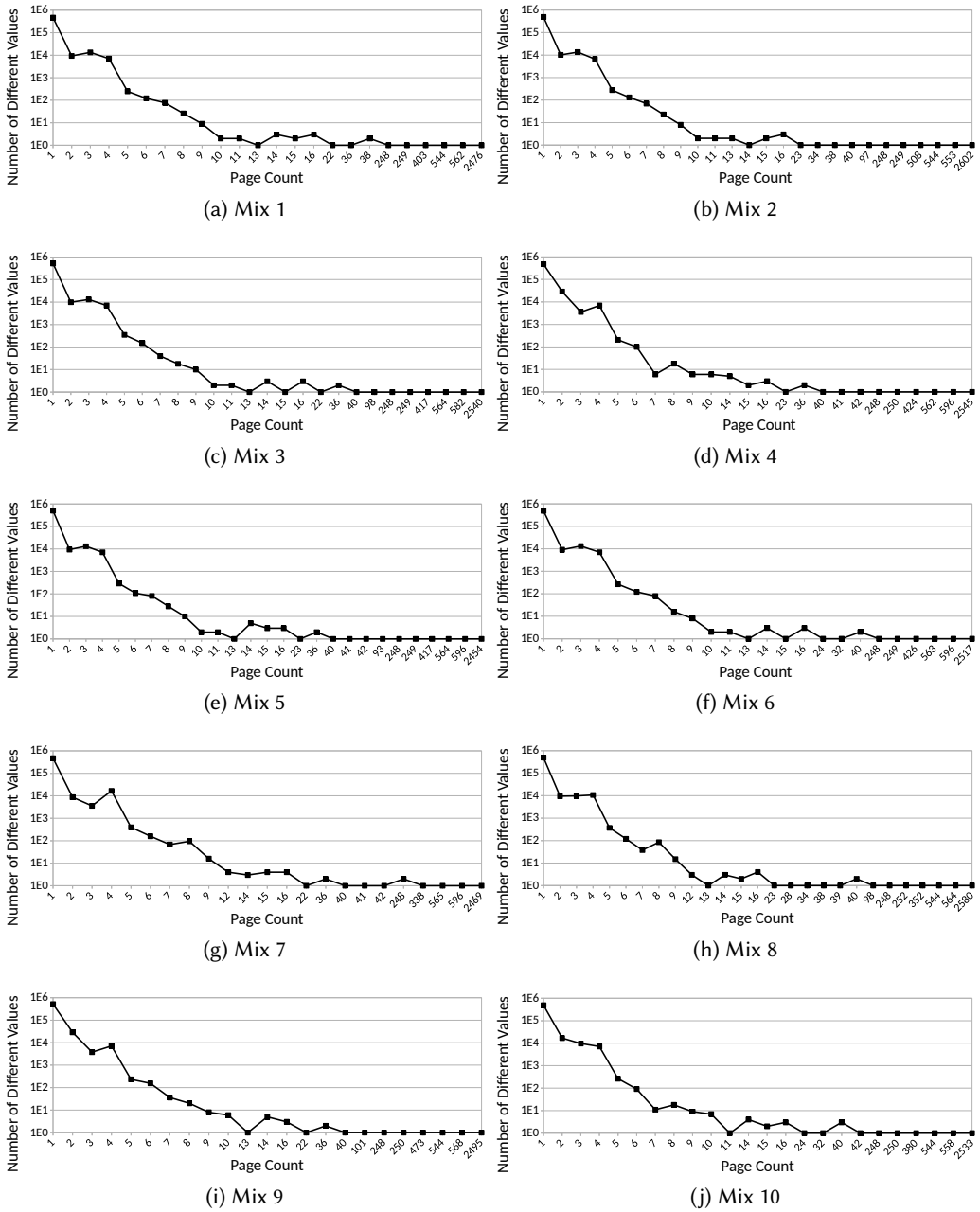


Fig. 13. Distribution of number of times a value is repeated in memory.

Finally, we decided to allocate 32K rows for this purpose to leverage the same-value opportunity in DSM.

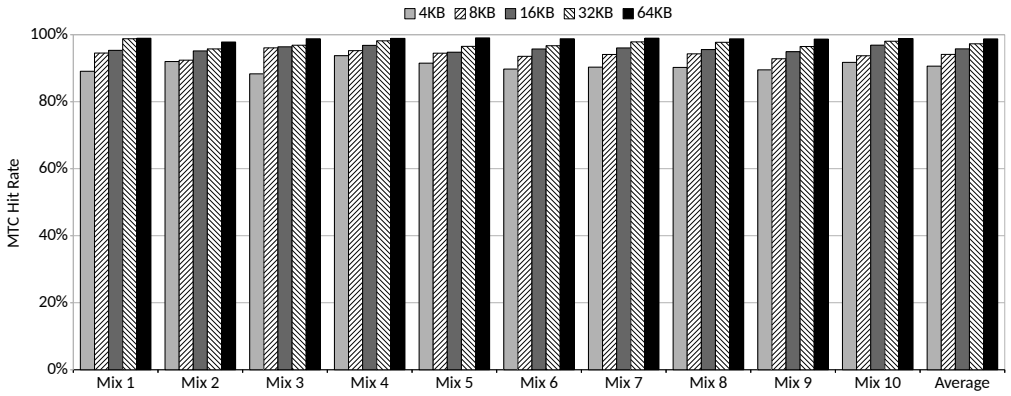


Fig. 14. Mapping Table Cache hit rate results with varying sizes.

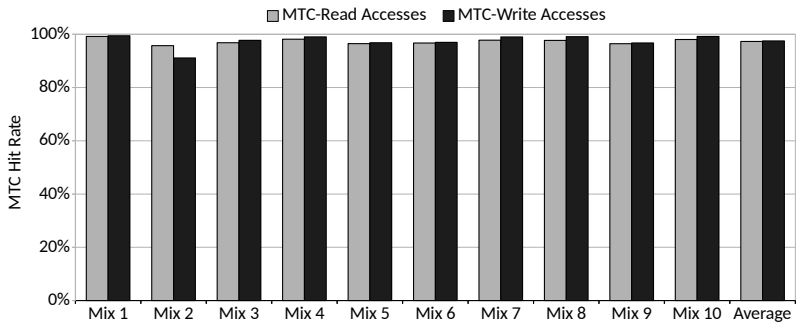


Fig. 15. Mapping Table Cache hit rate for DRAM reads vs writes.

### 7.7 Mapping Table Cache Hit Rate Results

The Mapping Table is stored in memory since the size of the Mapping Table is in order of several Megabytes. To reduce the additional DRAM traffic incurred to fetch the mapping information from DRAM, DSM design caches the recently accessed Mapping Table entries in a small on-chip hardware structure referred to as MTC. The size of this MTC plays a crucial role in the overall performance of our architecture as it governs the additional DRAM traffic incurred to fetch the Mapping Table information. Figure 14 shows the average on-chip hit rate incurred as a function of the Mapping Table Cache employed in DSM architecture. As can be observed, on average a 32KB MTC per channel incurs a hit rate as high as 97.3%. Consequently, we choose 32KB for the MTC capacity. Furthermore, Figure 15 shows the hit rates in MTC for read vs. write memory requests. As can be observed, the MTC hit rate is substantially high for both reads and writes, thereby reducing the mapping traffic to DRAM.

Summarizing the results from Figures 14 and 15, DSM architecture does not incur additional metadata mapping traffic in the quest of reducing DRAM refreshes.

### 7.8 DRAM Capacity Sensitivity Results

Figure 16 shows the performance benefit of removing the DRAM refresh overhead for larger memory capacities in baseline, DSM, and no refresh configurations. DSM architecture achieves 5.5% and 16.5% on an average for 16GB and 32GB memory sizes, respectively. It can be observed that

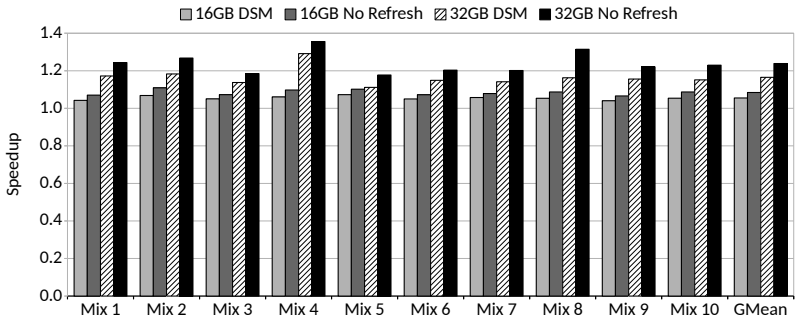


Fig. 16. Performance speedup results with varying DRAM capacities.

skipping the refresh of all the rows improves the performance by 35.6% in a server with 32GB memory capacity. Thus, DSM incurs more improvements in performance and energy with increased memory capacity as it eliminates refreshes to both non-allocated and similar value content rows [38, 55, 68].

### 7.9 Design Characteristics

The DSM's hardware overheads comprise of two major components, viz., (1) On-chip SRAM structures, and, (2) Off-chip Storage Overheads.

*7.9.1 DSM On-chip Overheads:* As explained in Section 4, our DSM architecture employs the following additional hardware structures at every memory controller: (1) 32KB Mapping Table Cache (MTC) (2) a 4KB R-Row counter/checksum array (3) a 4KB Refresh Bitmap. Since these are per-channel, the hardware on-chip structures required by our DSM architecture is negligible.

*7.9.2 DSM Off-chip Overheads:* DSM contains 32K *R-Row*s (details in Section 7.6) that results into 128MB hardware overhead in DRAM. Also, each Mapping Table entry needs 15 bits to maintain the index of the corresponding *R-Row*. Consequently, the Mapping Table amounts to 1.875MB hardware overhead for a DRAM channel with 4GB capacity containing 1M rows. In addition, for each R-Row, DSM needs a 1-byte counter and a 1-byte checksum that sums up to 64KB. Hence, the off-chip overhead of DSM in memory is 130MB, which is about 3% of the total DRAM capacity per channel.

Table 4 illustrates the area overhead and power consumption of different on-chip components of DSM. As it is indicated in this table, our evaluations with CACTI [10] show that DSM amounts to 0.236 $mm^2$ and 0.023 $W$ area and power overheads, respectively, in high-speed 22$nm$ technology devices. For comparing rows and checksum calculation, we employ an ALU, similar to PageForge [74].

## 8 RELATED WORK

We categorize the related work into two areas: (1) Refresh-based Optimization proposals, and, (2) KSM Optimization proposals.

### 8.1 Refresh Optimization Proposals

DRAM memory refresh contributes to a considerable portion of DRAM overheads, and there exists a wide range of prior works that address this overhead. In this section, we categorize some of these proposals.

Table 4. Design characteristics of DSM.

| Component | Area ($mm^2$) | Power (W) |
|---|---|---|
| Mapping Table Cache (MTC) | 0.198 | 0.012 |
| R-Row Counter/Checksum Cache | 0.019 | 0.002 |
| ALU [74] | 0.019 | 0.009 |
| Total | 0.236 | 0.023 |

*8.1.1 Refresh Skipping:* This type of memory refresh optimizations tries to remove the refresh of some rows in memory by leveraging the fact that some regions of memory are not allocated or the application is error-tolerant.

Isen *et al.* [36] proposed a scheme that removes the memory refresh for unallocated/freed rows in DRAM by exploiting program semantics and adding new instructions to ISA to pass the required information about the region that is allocated or freed from the program to the architecture. DSM also targets skipping the refresh of non-allocated rows in addition to the same content rows other than zero without any costly modification to ISA or program.

Researchers in [56] proposed a criticality-aware DRAM refresh skipping mechanism where refreshes to the non-critical data are skipped, introducing errors in the execution, as a side effect. Hence, their scheme is confined to applications that can tolerate errors in the output. This approach differs from our proposed DSM in that we do not introduce any errors in application execution due to the lost data integrity caused by refresh skipping. The effectiveness of the flikker [56] is limited by the amount of non-critical data present in the application program.

Researchers in [29] proposed skipping refreshes to certain rows in stacked DRAM that caches content from off-chip DRAM. Thus, they targeted the heterogeneous memory system and proposed balancing accesses between stacked and off-chip DRAMs by invalidating and skipping refreshes to data cached in the stacked DRAM. Their proposal of skipping refreshes to stacked DRAM is based on the fact that stacked DRAM is used as a cache, which implies that off-chip DRAM contains a copy of valid data. Hence, their scheme does not require any remapping of DRAM rows to other rows. Also, in their proposal, in scenarios where the refresh skipped (invalidated) data in stacked DRAM is in the modified state, they incur additional writebacks to off-chip DRAM to ensure the data integrity in the off-chip DRAM. Our DSM proposal is much different compared to their scheme as we do not target a heterogeneous memory system.

*8.1.2 Access-pattern-aware Refresh:* There is plenty of work on refresh-aware scheduling in order to improve the throughput by reducing the conflicts between refresh operations and on-demand read requests.

Kevin *et al.* [15] proposed a per-bank refresh scheduling policy that is aware of the requests waiting to be served by memory. Based on the requests waiting for the data to be fetched from memory, memory controller issues refreshes to banks that do not have any data waiting to be fetched. This request-aware per-bank refresh scheduling policy will avert on-demand DRAM request stalls due to DRAM refreshes.

Kotra *et al.* [43] proposed a hardware-software co-design scheme which involves changing the default per-bank refresh scheduling in hardware and exposing it to the system software. In this work, cognizant of the refresh scheduling policy, the system software performs best-effort scheduling of applications on the computing cores such that none of the on-demand requests are stalled by DRAM refresh.

*8.1.3 Retention-time-aware Refresh:* Liu *et al.* [55] observed that different portions of the DRAM have different retention times. Consequently, they argued that the number of cells that requires exact 64ms retention time and lose their data with higher refresh cycles are very small, and this phenomenon is also true for 256ms retention time. As a result, RAIDR [55] tries to apply different refresh cycles to different memory rows in the system and reduces the DRAM refresh overheads by significantly reducing the number of required refresh commands.

Researchers in [68] argued that detecting the accurate retention time for DRAM cells statically is not feasible. This is due to their observation that some DRAM cells experience variable retention-times, and RAIDR [55] cannot detect these memory cells and data might be lost in such cases. Qureshi *et al.* [68] propose AVATAR to detect such memory cells with variable retention-time before they incur a hard-error and refresh them faster to avoid failure.

We want to emphasize that, as opposed to our proposal, none of these previously proposed refresh optimizations perform "same data-aware" refresh skipping to DRAM rows. Our scheme is complementary to most of the approaches discussed above and is expected to improve both performance and energy-efficiency further when applied in tandem with these proposals.

### 8.2 KSM Optimization Proposals

KSM optimizations proposed in the past can be classified into hardware-based, software-based, and hardware-software co-design-based hybrid approaches.

*8.2.1 Hardware-based Approaches:* There exist multiple hardware-based approaches to exploit similar values during execution. Tian *et al.* [75] proposed and evaluated an LLC de-duplication scheme that merges cache lines with the same value in the LLC and hence, increases the effective size of the LLC and improves the performance of the system. In comparison, Cheriton *et al.* [17] introduced a content-addressable memory system where memory is accessed with "values" instead of "addresses". This approach needs a new programming model to access memory locations. For accessing memory, processor sends a value to memory and checks whether this value exists in memory. The paper indicates that the number of different values in memory is lower than the required capacity of memory in servers so that they can de-duplicate all pages with the same value to one line.

*8.2.2 Software-based Approaches:* Xia *et al.* [80] improved the existing KSM module in the kernel by organizing memory into regions with the possibility of same-value pages in each region and prioritizing regions with a higher probability of same-value pages for KSM lookups. Their approach also looks up the whole memory for de-duplication, instead of only VMs allocated memory, increasing the pressure on TLB even more.

*8.2.3 Hybrid Approaches:* Lin *et al.* [52] improved the Linux KSM approach by using a GPU to accelerate the checksum calculation for detecting these pages and changing the key of the tree traversal of KSM to the checksum that is calculated by GPU, instead of value. Skarlatos *et al.* [74] observed that the KSM is a CPU-consuming module because it reads all pages of VMs and compares them together (this requires lots of CPU cycles). As a result, they forward this computation to memory controller, and OS asks memory controller to check the potential pages for de-duplication. Then, memory controller looks at these pages, and if it detects a pair of pages with the same value, it informs the OS, which in turn changes the page tables accordingly. Raoufi *et al.* [69] went one step further and compared pages in-memory and removed the need to bring a page for comparison from memory to the processor. We want to point out that all such variants of KSM have the performance overhead caused by TLB shoot-downs.

We want to emphasize that our work is orthogonal to these prior approaches. If desired, it can be used in conjunction with any of them, except [17], which radically changes the underlying DRAM substrate.

## 9 CONCLUSION

Targeting data center virtualized environments, in this paper, we address the scaling problem of DRAM memories and the high DRAM refresh overhead in dense DRAM devices. To improve the performance and energy efficiency of virtualized environments, we propose DSM, a hardware extension in memory controller that detects the DRAM rows with the same content and only refreshes one row per each group of rows that contain similar content. DSM maintains only one copy of valid DRAM row per value in a content representative rows called R-rows. DSM ensures fetching correct data by redirecting the accesses to these representative rows by managing remapping of DRAM rows. DSM leverages these remappings to skip refreshes to rows that contain similar content. Our evaluation on a four-core processor simulating latency-sensitive Tailbench and SPEC2006 benchmarks used in virtualized environments show that DSM reduces the $99^{th}$ percentile memory access latency by up to 2.01x and reduces the energy consumption of memory by up to 8.5%.

## ACKNOWLEDGMENTS

We thank Murali Annavaram for shepherding our paper. We also thank the AMD internal reviewer Mike Ignatowski and the anonymous reviewers for their constructive feedback. This research is supported in part by NSF grants #1763681, #1908793, #1931531, #1822923, and #1629129. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

## REFERENCES

[1] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 336–348.

[2] Amazon. 2020. Amazon AWS EC2. https://aws.amazon.com/ec2/

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, Montreal, Canada, 19–28.

[4] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA.

[5] Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Reducing Writebacks Through In-Cache Displacement. *ACM Trans. Des. Autom. Electron. Syst.* 24, 2, Article Article 16 (Jan. 2019), 21 pages.

[6] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, Abbas Mazloumi, Farid Samandi, Mahmood Naderan-Tahan, Mehdi Modarressi, and Hamid Sarbazi-Azad. 2018. Fast data delivery for many-core processors. *IEEE Trans. Comput.* 67, 10 (2018), 1416–1429.

[7] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Vienna, Austria, 131–142.

[8] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Washington, DC, USA, 399–411.

[9] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Evaluation of Hardware Data Prefetchers on Server Processors. *ACM Comput. Surv.* 52, 3, Article Article 52 (June 2019), 29 pages.

[10] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article Article 14 (June 2017), 25 pages.

[11] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[12] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41.

[13] Ishwar Bhati, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. 2015. Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 235–246.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[15] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Orlando, FL, USA, 356–367.

[16] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. Association for Computing Machinery, New York, NY, USA, Article Article 3, 10 pages.

[17] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 287–300.

[18] Winter Corp. 2020. WinterCorp. Big Data and Data Warehousing. http://www.wintercorp.com/

[19] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220.

[21] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144.

[22] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active Low-Power Modes for Main Memory. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 225–238.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14.

[25] Duncan Elliott, Michael Stumm, W. Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. 1999. Computational RAM: Implementing Processors in Memory. *IEEE Des. Test* 16, 1 (Jan. 1999), 32–41.

[26] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 100–113.

[27] Sukhpal Singh Gill and Rajkumar Buyya. 2018. A Taxonomy and Future Directions for Sustainable Cloud Computing: 360 Degree View. *ACM Comput. Surv.* 51, 5, Article Article 104 (Dec. 2018), 33 pages.

[28] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer* 28, 4 (April 1995), 23–31.

[29] Nagendra Gulur, R. Govindarajan, and Mahesh Mehendale. 2016. MicroRefresh: Minimizing Refresh Overhead in DRAM Caches. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 350–361.

[30] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322.

[31] Hasan Hassan, Minesh Patel, Jeremie S. Kim, A. Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. 2019. CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 129–142.

[32] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[33] IBM. 2020. IBM Cloud Services. https://www.ibm.com/services/cloud

[34] Intel. 2017. Intel Optane Memory. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-memory-brief.pdf

[35] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F. Coutinho, and Mark Stillwell. 2016. High Performance in the Cloud with FPGA Groups. In *Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC '16)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[36] Ciji Isen and Lizy John. 2009. ESKIMO: Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 337–346.

[37] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2017. Computing in memory with spin-transfer torque magnetic ram. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (2017), 470–483.

[38] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. 2014. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The Memory Forum*.

[39] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-Critical Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 598–610.

[40] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Providence, RI, USA, 1–10.

[41] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R. Alameldeen, Chris Wilkerson, and Onur Mutlu. 2014. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*. Association for Computing Machinery, New York, NY, USA, 519–532.

[42] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 380–392.

[43] Jagadish B. Kotra, Narges Shahidi, Zeshan A. Chishti, and Mahmut T. Kandemir. 2017. Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 723–736.

[44] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, Fukuoka, Japan, 533–545.

[45] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[46] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721.

[47] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[48] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 173, 6 pages.

[49] Zheng Li, Selome Tesfatsion, Saeed Bastani, Ahmed Ali-Eldin, Erik Elmroth, Maria Kihl, and Rajiv Ranjan. 2017. A survey on modeling energy consumption of cloud applications: deconstruction, state of the art, and trade-off debates. *IEEE Transactions on Sustainable Computing* 2, 3 (July 2017), 255–274.

[50] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 267–278.

[51] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-Level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, USA, 1–12.

[52] Wei-Cheng Lin, Chia-Heng Tu, Chih-Wei Yeh, and Shih-Hao Hung. 2017. GPU acceleration for kernel samepage merging. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, IEEE, Hsinchu, Taiwan, 1–6.

[53] Linux. 2009. Linux KSM - Kernel Samepage Merging (KSM). https://www.linux-kvm.org/page/KSM

[54] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. 2013. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 60–71.

[55] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. 2012. RAIDR: Retention-Aware Intelligent DRAM Refresh. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 1–12.

[56] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 213–224.

[57] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 417–433.

[58] David Meisner, Junjie Wu, and Thomas F. Wenisch. 2012. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12)*. IEEE Computer Society, USA, 35–45.

[59] Microsoft. 2020. Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

[60] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F Wenisch. 2019. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Washington, DC, USA, 185–198.

[61] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. 2013. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 48–59.

[62] Kate Nguyen, Kehan Lyu, Xianze Meng, Vilas Sridharan, and Xun Jian. 2018. Nonblocking Memory Refresh. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Los Angeles, California, 588–599.

[63] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *SIGPLAN Not.* 49, 4 (Feb. 2014), 3–18.

[64] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. 2012. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12)*. IEEE Computer Society, USA, 1207–1214.

[65] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. *SIGPLAN Not.* 53, 2 (March 2018), 679–692.

[66] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44.

[67] Nathan Pemberton, John D Kubiatowicz, and Randy H Katz. 2018. *Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters*. Ph.D. Dissertation. Master's thesis, University of California at Berkeley, Berkeley, CA.

[68] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. 2015. AVATAR: A variable-retention-time (VRT) aware refresh for DRAM systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Rio de Janeiro, Brazil, 427–437.

[69] Mehrnoosh Raoufi, Quan Deng, Youtao Zhang, and Jun Yang. 2019. PageCmp: Bandwidth Efficient Page Deduplication through In-memory Page Comparison. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, Miami, FL, USA, 82–87.

[70] Samsung. 2020. Samsung DDR4. https://www.samsung.com/semiconductor/dram/ddr4/

[71] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, Atlanta, GA, USA, 183–192.

[72] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and et al. 2013. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 185–197.

[73] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287.

[74] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: A Near-memory Content-aware Page-merging Architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 302–314.

[75] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 53–62.

[76] Josep Torrellas. 2012. FlexRAM: Toward an Advanced Intelligent Memory System: A Retrospective Paper. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012) (ICCD '12)*. IEEE Computer Society, USA, 3–4.

[77] Ubuntu. 2020. Ubuntu Server. https://www.ubuntu.com/server

[78] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low Latency via Redundancy. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA, 283–294.

[79] Carl A. Waldspurger. 2003. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2003), 181–194.

[80] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. 2018. UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 325–340.

[81] Dongli Zhang, Moussa Ehsan, Michael Ferdman, and Radu Sion. 2014. DIMMer: A Case for Turning off DIMMs in Clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–8.

[82] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, USA, 406–418.

[83] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Seoul, Republic of Korea, 456–468.

Received January 2020; revised February 2020; accepted March 2020