XFM: Accelerated Software-Defined Far Memory

Neel Patel
University of Kansas
nmpatel@ku.edu

Amin Mamandipoor
University of Kansas
amin.mamandi@ku.edu

Derrick Quinn
University of Kansas
derrick.quinn@ku.edu

Mohammad Alian
University of Kansas
alian@ku.edu

ABSTRACT

DRAM constitutes over 50% of server cost and 75% of the embodied carbon footprint of a server. To mitigate DRAM cost, far memory architectures have emerged. They can be separated into two broad categories: software-defined far memory (SFM) and disaggregated far memory (DFM). In this work, we compare the cost of SFM and DFM in terms of their required capital investment, operational expense, and carbon footprint. We show that, for applications whose data sets are compressible and have predictable memory access patterns, it takes several years for a DFM to break even with an equivalent capacity SFM in terms of cost and sustainability. We then introduce XFM, a near-memory accelerated SFM architecture, which exploits the coldness of data during SFM-initiated swap ins and outs. XFM leverages refresh cycles to seamlessly switch the access control of DRAM between the CPU and near-memory accelerator. XFM parallelizes near-memory accelerator accesses with row refreshes and removes the memory interference caused by SFM swap ins and outs. We modify an open source far memory implementation to implement a full-stack, user-level XFM. Our experimental results use a combination of an FPGA implementation, simulation, and analytical modeling to show that XFM eliminates memory bandwidth utilization when performing compression and decompression operations with SFMs of capacities up to 1TB. The memory and cache utilization reductions translate to 5~27% improvement in the combined performance of co-running applications.

CCS CONCEPTS

• **Hardware**; • **Computer systems organization** → **Other architectures**;

KEYWORDS

Near-Memory Processing, Accelerator, Compression

ACM Reference Format:
Neel Patel, Amin Mamandipoor, Derrick Quinn, and Mohammad Alian. 2023. XFM: Accelerated Software-Defined Far Memory. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3623776

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623776

1 INTRODUCTION

DRAM memory capacity has stagnated due to the challenges in CMOS technology scaling [54, 58]. At the same time, DRAM manufacturing cost and carbon footprint exceed any other components in datacenter servers [15]. This is happening at a time when the memory capacity requirements for emerging workloads are exploding. One rescue path is the integration of a far memory tier with higher access latency than local memory into the memory hierarchy. At runtime, application data is dynamically swapped between regions based on their access frequency. Hyperscalars have already integrated far memory tiers into their servers [51, 77], adding elasticity to an expensive, often stranded, and highly contended resource [8].

Far memory can be implemented by pooling memory modules over an interconnect which is slower than traditional DDR channels. Such disaggregated far memory (DFM) implementations use PCIe [57], the datacenter network [6, 8, 18, 26, 36, 55, 70, 81], or CXL-based pooling [35, 56] to connect the far memory capacity to CPUs. On the other hand, a software-defined far memory (SFM) implementation dynamically allocates a portion of the local DRAM to the storage of compressed, cold data [51]. The cold data is only decompressed and moved to the local address space once it is accessed.

In comparison to DFM, SFM provides additional elasticity. Local and far memory capacities can be dynamically resized without additional hardware expenditures by re-partitioning the local and far memory address spaces. We develop a first-order analytical model which considers a DFM deployment using new DDR4 modules and running applications with compressible data sets and predictable memory access patterns. Our results show that it can take more than 8 years for a DFM to break even in terms of cost with an SFM counterpart when both provide an additional 512 GB memory capacity. Besides the higher initial cost of DFM, DRAM manufacturing has an order of magnitude higher carbon emission than logic manufacturing [15]. For this reason, SFM-related carbon emissions do not reach that of a DFM during the lifetime of the server (§3). Although DFM deployments come with numerous benefits, this cost and emission analysis motivates scavenging used DRAM for DFM implementations [56] and shows the importance of investment in SFM. SFM can be used to maximize the efficiency of precious DRAM resources, whether they are locally connected to the CPU or disaggregated over a system interconnect.

Current SFM implementations use the CPU to compress (swap out of local memory) and decompress (swap into local memory) candidate pages. Selected candidate pages should be cold and compressible, otherwise, the swap out to far memory will hurt application performance and may not provide tangible benefits to capacity.

Because both the candidate pages as well as the compressed pages in far memory are cold, SFM requires costly DRAM accesses on every swap in and out. For a 512GB SFM implementation, the memory bandwidth utilization for reading and writing data to memory can reach up to 34GBps. Although current multi-channel CPUs equipped with high bandwidth DDR5 DIMMs can accommodate this access bandwidth, the interference in the memory and cache hierarchy hurts the performance of co-running applications and also increases computational energy due to excessive data movement [39, 63].

Figure 1a illustrates the memory bandwidth utilization of current CPU-centric SFM implementations. A near-memory processing architecture is a natural fit to eliminate the data movement between CPU and memory while swapping data between local and far memory. SFM is especially interesting for near-memory acceleration because (1) far memory data is in the memory by definition, resolving the cache coherency issues common amongst near-memory processing architectures [16][17], (2) swap ins and outs from far memory take place at OS page granularity, simplifying the virtual memory implementation of near-memory processing [40], and (3) compression and decompression tasks are incrementally computable, simplifying memory channel interleaving complexities [25].

In this work, we implement Accelerated SFM (XFM), which captures the timely need for SFM acceleration. In contrast to prior works that are centered around maximizing the memory bandwidth between near-memory accelerators (NMAs) and DRAM, XFM is designed to implement an interface between NMAs and DRAM with just-enough bandwidth to accommodate SFM-related data movement. This enables XFM to leverage inevitable DRAM refresh cycles to implement a side-channel on each rank, granting the NMA access to DRAM. These accesses are transparent to the CPU memory controller. Fig.1b illustrates how the utilization of NMAs can reduce the memory bandwidth utilization of SFM in current servers with tens of DRAM ranks.

We prototype XFM on an FPGA-based near-memory processing platform. Additionally, we implement a full-stack implementation of XFM by modifying an application-integrated far memory framework (AIFM) [70] to include a compression/decompression engine. Our experimental results show that XFM can potentially eliminate the memory bandwidth utilization of software-defined far memories with up to 1TB capacity. The memory bandwidth savings translate to 5~27% higher performance for co-running applications. The benefits of XFM can be increased by improving the far memory controller's proficiency at predicting application memory access patterns.

2 BACKGROUND

2.1 Far Memory

The stagnation of DRAM capacity scaling has led hyperscalers to deploy a new tier in the established memory hierarchy. There now exists an intermediate level between DRAM and storage known as far memory. Far memory tends to have a larger capacity than local memory with higher access latency and lower bandwidth. In this paper, we consider DRAM as the memory technology that is used for both local and far memory. The current implementations

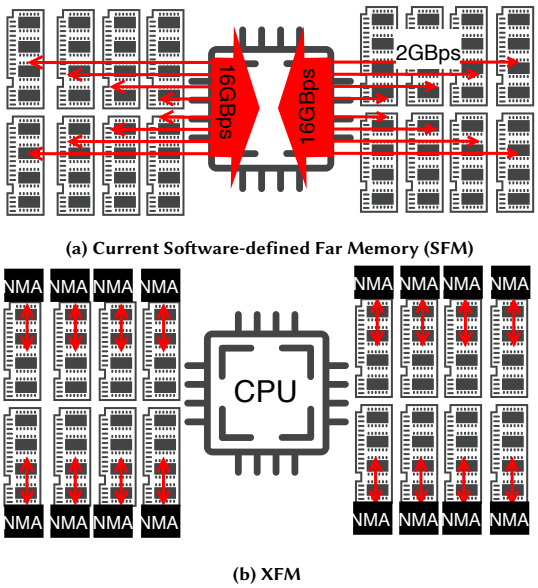


Fig. 1. Memory bandwidth utilization is the bottleneck in future SFM implementations with many DRAM ranks. XFM enables rank-level parallelism for performing SFM operations.

of far memory can be divided into two categories: **Disaggregated Far Memory (DFM)**, where extra DRAM modules are connected to the CPU over a high-speed serial interconnect such as PCIe, CXL, InfiniBand, or Ethernet, and **Software-defined Far Memory (SFM)** where a reconfigurable portion of local DRAM is used to store compressed pages. The access rate to far memory is quantified using a metric called *promotion rate* [51]. Promotion rate is the percentage of far memory that is accessed per minute. A 20% promotion rate for a 512GB far memory implies that 102GB of the far memory is accessed during a 60-second interval.

DFM aims at better utilizing existing memory capacity by disaggregating stranded local memory and exposing it as a shared pool to the applications. Network-accessible DRAM managed by the application [70, 81] or the OS [8] provides a means to reduce cluster-wide memory stranding and supplement the memory available to applications [57]. Additionally, the CXL protocol enables access to additional DRAM over the system bus, expanding the memory capacity of multi-socket servers. Although recent work shows the promise of CXL-based memory pooling when combined with intelligent provisioning and data placement [56], DFM suffers from static provisioning of DRAM resources and extra hardware requirements and support.

SFM is readily deployable in datacenter computing systems. Several hyperscalers realize cluster-wide DRAM savings by establishing page-swapping algorithms that utilize compressed page caching mechanisms within the OS [51, 77]. These smart paging mechanisms seek to move as much application memory to SFM while still meeting application level Service-Level Objectives (SLOs). Shifting cold pages to SFM makes room in local memory for other applications or frequently accessed pages, improving job throughput without compromising the access latency of local memory. Google's approach involves pre-emptively scanning for cold pages [51], while Meta utilizes pressure metrics exposed by the OS to respond to poor

resource utilization [77]. SFM is not new to other domains such as mobile and desktop systems [46].

SFM Control Plane. SFM implementations from Google and Meta and are built atop Linux zswap [43, 51, 77]. Zswap provides a mechanism to intercept pages being swapped out of main memory, compress them, and store them in a separate compressed memory pool (zpool) within DRAM. Although other memory allocators are available in zswap, the zsmalloc allocator is generally used [51, 77] as it makes the best use of a single physical page by inserting as many compressed pages as possible. This comes at the cost of intermittent compaction operations to address internal fragmentation resulting from swapping pages between near and far memory.

Although zswap is an OS-provided feature, hyperscalers have already moved the far memory control plane out of the OS. For example, Google implements a custom version of Linux's *kswapd* paired with an additional daemon for page compression, *kreclaimd* [51] and Meta uses a userspace program, *senpai*, to initiate reclaim based on OS-provided performance metrics [77].

Far Memory Compression Algorithms. Deploying a compression algorithm at scale requires meeting application SLOs and managing resource consumption. Finding the appropriate balance between compression ratio, speed, and CPU utilization has led to the development of new compression algorithms [23, 76] and reliance on algorithms that achieve high speeds at the cost of a lower compression ratio. The lzo [69] and zstd [22] compression algorithms are used at scale [51, 77] as they are capable of maintaining low enough CPU overhead while still achieving a good compression ratio. There are, however, hardware implementations of well-known algorithms, like deflate [30], which can achieve high-compression ratios. Support for deflate (de)compression using zswap is currently implemented in the form of on-chip accelerators within IBM's POWER9/z15 [5] and Intel's Sapphire Rapids CPUs [9].

2.2 DRAM Memory System

General DRAM Architecture. The DRAM main memory system can be viewed as a 5-dimensional hierarchy. Memory is first divided into multiple *channels*, with all memory in a single channel sharing the same address, command, and data busses to service host CPU accesses. A channel may contain multiple *ranks*. A rank is composed of multiple commodity DRAM devices which act in unison to serve memory reads/writes. A rank can also be viewed as a collection of *banks*, where each bank can service commands independently, but shares the same data and control paths. A bank is composed of many 2D arrays of DRAM cells, called mats. Each mat has its own row of sense amplifiers and row/column decoders. A collection of mats sharing the same wordlines compose a *subarray*. Cells in a subarray share the same row buffer and an entire row of data is always within the same subarray [48].

When data is to be read from DRAM, an entire row is brought into an array of sense-amplifiers, called the *row-buffer*. This row-buffer is *local* to a subarray. The global column decoder asserts the appropriate column select lines to drive a portion of the row buffer to the IO/bank periphery. In a DRAM device with a data width of 8 bits, 64 bits of data from the local row buffer will be driven onto the global bitlines and amplified by the global sense amplifiers, before it is read out of the bank periphery/IO.

DRAM Refresh Due to charge leakage, DRAM cells need to periodically be refreshed to prevent data loss. The memory controller sends periodic refresh commands to each DRAM rank. A *refresh counter* on each DRAM chip indicates which rows in DRAM banks to *sense*, *amplify*, and *write back* to restore the charges of the DRAM cells. Typically, every row of each DRAM bank needs to be refreshed every ~32ms [60]. In an all-bank refresh, no banks can be accessed. This causes read and write requests to be delayed by refresh operations. To reduce this latency, the memory controller spreads out the refreshes across the 32ms retention time. The memory controller typically sends 8192 individual REF commands every 32ms to each DRAM rank, initiating an auto-refresh operation.

The all-bank auto-refresh is optimized such that multiple rows in the same bank can be refreshed in parallel. This is accomplished by leveraging multiple local row buffers across different subarrays to refresh multiple rows at the same time [13]. Although recent DRAM chips support a selective bank refresh mode [60] to prevent the rank from being locked during each refresh cycle, the all bank mode is still the most efficient way of refreshing rows in a semi-parallel fashion [19].

The semantics of an auto-refresh command are equivalent to a series of Activate (ACT) and Precharge (PRE) commands [59], except that additional strain is placed on the power-delivery network from refreshing multiple rows in parallel. This necessitates waiting for a fixed interval t_{STAG} (~10ns in DDR4 devices) between initiating refresh operations in consecutive banks. Refreshing multiple rows in the *same bank* at the *same time* increases the duration of t_{RFC} beyond t_{RC}, the time to activate a row and precharge its corresponding bank [61].

Purposeful redirection of refresh operations is not unprecedented given its current applications in maintaining DRAM data integrity. To protect against the Rowhammer vulnerability in DRAM devices, DRAM manufacturers have added the capability to refresh a limited number of additional *victim* rows for each received REF command received. This mechanism is intended to refresh neighbors of rows that have been activated with high frequency, ensuring data reliability and mitigating attacks. Target Row Refresh (TRR) has already been implemented in commercial DDR4 devices [38], and the DDR5 specification further extends TRR through the refresh management command (RFM) [42].

3 DFM VS. SFM

3.1 First-Order Cost Model

In this section, we consider a CXL-based DFM implementation as nascent DFM technology [56] and compare it with SFM using a first-order analytical model. We show that an ideal, accelerated SFM can achieve lower TCO and environmental impact compared with DFM implementations. It is important to note, however, that the benefits provided by SFM are mainly orthogonal to DFM. A server equipped with DFM can additionally deploy SFM to increase the overall memory capacity by storing cold data more efficiently. In essence, DFM substitutes the compute cost of SFM with more DRAM capacity statically placed over CXL. Fig.2 illustrates these two configurations.

Our analytical model compares the capital and environmental cost of an SFM and DFM deployment over several years. We consider

two DFM implementations, one with DRAM and one with persistent memory (PMem). While it is possible to scavenge used DRAM modules to implement DFM, we cannot quantify the capital cost and emissions of such a deployment, and thus, we only consider newly manufactured DRAM in our modeling. We assume that the EXTRA Memory capacity is statically provisioned, and the system runtime/OS consistently populates it with cold application data. In practice, SFM implementations leverage more complex cold page identification mechanisms and heuristics. Our model establishes an upper bound on the TCO of SFM by eschewing these optimizations, compressing pages whenever space is available in the SFM memory region and decompressing at the predefined promotion rate.

In our model, we consider 64GB DRAM DIMMs, and 512GB PMem DIMMs (*DIMMSize* parameter in the following equations). For the remainder of this subsection, we estimate the total number of bytes compressed and decompressed every minute for a 512GB far memory implementation (i.e., *ExtraGB* parameter is set to 512GB in the following equations) as follows:

EQ1: $GBSwappedPerMin\ (GB/min) = ExtraGB \times PromotionRate$

Where *PromotionRate* is the percentage of far memory accessed every minute (as explained in Sec.2.1).

Capital Cost. We conservatively estimate the financial expenditures incurred by a new DFM investment to be equal to the upfront purchase of the additional memory modules at the current per-byte cost of DRAM and PMem. We also add 88pJ/byte (2.44×10^{-8} kWh/GB) energy cost for PCIe accesses [12], and 4 Watts of static power consumption for additional DIMMs (*IdleDIMMEnergy* parameter in EQ2.2). We consider an average cost of $0.12/kWh for electricity (*ElectricityCost*) [28].

EQ2: $DFMCost = ExtraGB \times MemoryCostPerGB\ + ((PCIeEnergy\ +\ IdleDIMMEnergy) \times ElectricityCost$

EQ2.1: $PCIeEnergy = 2.44 \times 10^{-8} kWh/GB \times GBSwappedPerMin \times TIME$

EQ2.2: $IdleDIMMEnergy = 0.24\ (kWh) \times ElectricityCost \times TIME \times (GBSwappedPerMin/DIMMSize)$

For modeling the financial expenditure incurred by SFM, we consider both the operational and upfront purchase costs of the extra CPU cores that need to be provisioned for running (de)compression operations. Using the TDP of an Intel Xeon E5 2670 (115 Watts), the clock rate (2.6GHz), and the average cycles required to (de)compress a byte using zstd [23] and lzo [69] algorithms (*CCPerGB* parameter in EQ3.4, which is 7.65×10^9 cycles on average), we calculate the energy consumed by the CPU for (de)compressing one GB of data (*EnergyPerGB* variable in EQ3). *CPUCost* includes the cost of provisioning extra CPU cores to run (de)compression operations.

EQ3: $SFMCost = EnergyPerGB \times GBSwappedPerMin \times ElectricituCost \times TIME\ +\ CPUCost$

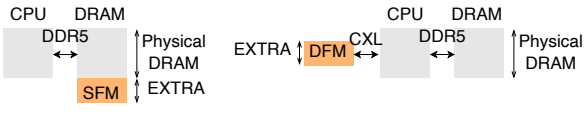


Fig. 2. Illustration of systems equipped with SFM and DFM of the same capacity.

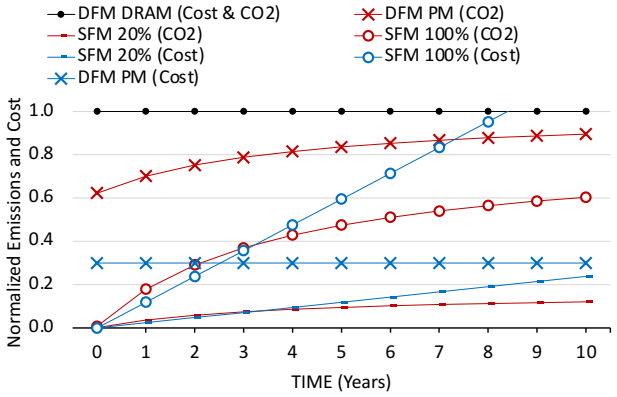


Fig. 3. Emission and capital cost comparison of SFM and DFM implementing the same far memory capacity. Values are normalized to that of DFM.

CPUCost is estimated by calculating the fraction of CPU cycles that are needed to perform (de)compression operations (*%CPUNeeded* variable in EQ3.2) and multiplying it by the purchase price of a CPU:

EQ3.1: $CPUCost = \%CPUNeeded \times CPUPurchasePrice$

EQ3.2: $\%CPUNeeded = CCNeededPerMin/CCAvailablePerMin$

EQ3.3: $CCAvailablePerMin = CPUFreq \times NumOfCores \times 60$

EQ3.4: $CCNeededPerMin = GBSwappedPerMin \times CCPerGB$

We assume that dynamic DRAM access energy consumption is the same for both SFM and DFM and factor it out of capital cost modeling.

As shown in Fig.3, even at a promotion rate of 100%, an SFM is more cost-effective than a DRAM-based DFM counterpart for implementing a far memory tier. It takes 8.5 years for SFM to break even with the cost of a DRAM-based DFM. A promotion rate of 100% means that every minute, all of the cold pages are accessed. This is an extreme case, but it establishes an upper bound on the cost of SFM. In Google's fleets, it was observed that classifying pages as cold after going 120 seconds without an access results in over 30% of memory being detected as cold and a 15% promotion rate [51]. As shown in Fig.3, at a 20% promotion rate, SFM may prove more cost-effective, even when compared to a PMem-based DFM.

Environmental Cost. Environmental cost comes from manufacturing (i.e., embodied energy used for production) and operational greenhouse emissions. We use processor and memory manufacturing emission data from Boavizta's model [15] for these analyses. We also consider the same emissions per silicon area for both DRAM and PMem [15, 37], with PMem having 2× higher bit density than DRAM [75]. Our analyses yield carbon dioxide equivalent (CO2eq) emissions of 1.01 kilograms per GB of DRAM, 0.62 kilograms per GB of PMem, and 0.625 kilograms per CPU core manufactured, respectively. We exclude the manufacturing emissions for local DRAM modules because the emissions are similar for both DFM and SFM configurations.

We consider the idle energy consumption of extra DIMMs (i.e., *IdleDIMMEnergy*) to include operational carbon emissions for DFM. We use pessimistic data from the Southwest Power Pool in the central US collected in 2022 for electricity emissions that report emissions of $479 gCO_2eq/kwh$ [27]:

EQ4: $DFMEmission = ExtraGB \times EmissionPerGB + OperationalEmission$

EQ4.1: $OperationalEmission = IdleDIMMEnergy \times ElectricityEmission$

EQ5: $SFMEmission = \%CPUNeeded \times EmissionPerCPU + EnergyPerGB \times GBSwappedPerMin \times ElectricityEmission \times TIME$

As illustrated in Fig.3, DRAM-based DFM and SFM never break even in terms of carbon emissions during the typical 5-year lifetime of a server. Even with PMem, it can take several years for SFM with a 20% promotion rate to break even in emissions. Using cleaner grids could further decrease CPU energy consumption when increasing the effective memory capacity of SFM. It should be noted that our model does not take the performance implications of using SFM and DFM into account and only discusses a first-order model of the cost and emissions of these far-memory implementations.

3.2 Opportunities for SFM Acceleration

In Sec. 3.1 we showed that an ideal SFM deployment can be more cost-effective when compared to a DFM solution. SFM is particularly attractive for applications that exhibit predictable access patterns and operate on a compressible data set. In this section, we discuss the opportunities for accelerating SFM to further reduce its cost and improve its performance. As shown in Fig.3, increasing the promotion rate results in higher costs for operating an SFM. The cost increases due to higher usage of CPU cycles for performing compression and decompression as the number of bytes needed to be (de)compressed increases proportionally to the promotion rate. Even though it is not shown in Fig.3, increasing the far memory capacity also proportionally increases the cost of operating SFM. The **O**verheads associated with a SFM can be classified into four categories: **(O1)** synchronous decompression operations stall application threads and increase effective memory access latency, **(O2)** CPU cycles spent on compression and decompression consume additional energy and take precious CPU cycles away from other applications, **(O3)** memory bandwidth is consumed by page-compression and decompression routines as both compression and decompression tasks read cold data from DRAM before performing the operation, and **(O4)** page-granular compression and decompression tasks pollute the cache hierarchy.

One solution to mitigate the cost of performing compression and decompression is to offload the operations to an on-chip accelerator. There are myriad works on hardware acceleration of compression and decompression [5, 24, 64, 68, 79] and the latest Intel Xeon CPUs integrate a compression accelerator [9] that can benefit the implementation of SFM. Using an on-chip hardware accelerator alleviates **O1** and **O2**. For SFM, compression is not on the critical path of execution and only de-compression latency can impact the application performance. Our experimental results show that a PCIe Intel QAT sustains 9.8GBps and 13.3GBps compression and decompression throughput, respectively. Assuming the same compression throughput for an integrated QAT accelerator, the host can offload all the compression required to operate a 512GB SFM with a 100% promotion rate. However, this comes at the cost of consuming a physical core to manage the offload operations. Using our cost model, we determine that an integrated hardware accelerator becomes beneficial when the average promotion rate is higher than 6% in a 512GB SFM. It also fails to address overheads **O3** and **O4**, due to its impact on the memory subsystem.

Although the average DRAM bandwidth requirements of SFM are unlikely to be a performance bottleneck[1], the page-granular memory accesses and bursty swap ins and outs result in extensive cache pollution and memory channel contention for co-running applications. To illustrate this, we co-run 8 LLC and memory sensitive SPEC workloads with two processes that continuously compress and decompress 4KB pages on an Intel Xeon E5-2658 CPU. We pin each SPEC workload as well as the antagonist processes to disjoint physical cores with SMT and Turbo boost disabled and the frequency set to 2.2GHz. The runtime increases by up to 7.5% with the antagonists' compression throughput degrading by more than 5.0%.

To alleviate **O3** and **O4**, a natural design decision is to use near-memory processing. Although performing the compression on the DRAM side addresses both **O3** and **O4**, offloading decompression to memory is not beneficial if: (1) near-memory decompression latency is higher than on-CPU decompression. This can happen if, due to the limited power budget, the near-memory accelerator has lower performance than CPU. The benefits are also lost when (2) the extra bytes read due to I/O amplification is less than the number of bytes used by the application after decompression of a page.

We define the I/O amplification ratio for accessing SFM as the ratio of compressed bytes accessed over the memory channel to the total number of decompressed bytes used by the application. The I/O amplification ratio for SFM is a function of application access patterns and the contention on the LLC. For example, if there is contention on the LLC or the use-distance of the decompressed bytes is long, then the I/O amplification ratio increases as the decompressed page is likely to be written back to DRAM before being used the application. One key benefit of offloading both compression and decompression to the memory is that the software control plane can now aggressively compress and decompress without hogging DDR bandwidth. This enables the control plane to implement better heuristics to further reduce the impact of higher far memory access latency on application performance. It also allows better management of the far memory space since application data can be quickly shifted between local and far memory. Note that a software controller cannot aggressively compress data into SFM without also aggressively decompressing data out of the SFM as the capacity of SFM is limited and, in a stable state, the rate of compression and decompression are the same.

3.3 Summary and Proposal

Here we summarize the main takeaways from the discussion in Sec.3.1 and Sec.3.2:

[1] 100% promotion rate in a 512GB SFM requires compressing and decompressing at a rate of 8.5GBps. Therefore the total DRAM read and write bandwidth utilization is 4×8.5GBps which is 1.3× of a single DDR5 channel.

- An SFM deployment can be a more cost-effective solution for applications that exhibit predictable access patterns and have a high compression ratio.
- On-chip acceleration of SFM compression operations is sub-optimal due to cold data accesses from DRAM without temporal locality.
- On-chip acceleration of SFM decompression operations is sub-optimal if the decompressed data is not used immediately.

In this paper we architect A**X**elerated software-defined **F**ar **M**emory (XFM) which implements offload capabilities for both compression and decompression in memory, and allows the software control plane to dynamically fall back to the CPU for decompression if deemed beneficial. The overriding design goals relevant to XFM's design are (**G1**) minimizing changes to the DRAM architecture, (**G2**) avoiding compromising host CPU DRAM access latency or bandwidth, (**G3**) and maintaining the flexible capacity provisioning provided by software-defined far memory. Since accesses to an SFM are made at the granularity of an OS page and the content of far memory is cold and resides exclusively in the DRAM, the virtual memory [40] and cache coherency [17] challenges of near-memory processing are not an issue for XFM. Designing a near-DRAM accelerator that satisfies **G1**∼**G3** is an open research question, however, and requires innovation to correctly manage concurrent accesses from the accelerator and host CPU to the local DRAM. Next we explain XFM's hardware and software architecture which leverages the inevitable refresh cycles in DRAM to implement a host-transparent communication channel between the (de)compression accelerator and DRAM banks, satisfying the above constraints.

4 REALIZATION OF XFM

4.1 Placement of the NMA

The very first design decision in architecting XFM is determining where to place the near-memory accelerator (NMA). We have two **O**ptions: (**O1**) integrate the NMA within the DRAM chip or (**O2**) place the NMA in the buffer device of a DIMM. Although **O1** offers higher bandwidth between NMA and DRAM banks, it suffers from high manufacturing cost, technology mismatch between logic and memory [11], and restrictions on the data mapping to localize computation [25]. Therefore, **O1** only makes sense when DDR bandwidth is the performance bottleneck in accelerating an application. As shown in Sec.3.2, in the worst case scenario (i.e., with 100% promotion rate), a 512GB SFM consumes ∼17GBps read bandwidth which is often accessed over multiple DDR channels. Although the DRAM accesses are costly for energy consumption, the DDR channel is not the performance bottleneck in SFM since the bandwidth of a DDR5 channel is 25GBps. With these insights, we opt to reduce the design cost and simplify the datapath by considering **O2** for XFM.

Fig.4 (left) shows an organization in which an NMA is integrated into the DIMM's Registering Clock Driver (RCD) chip. A RCD distributes the clock and Command/Address (C/A) signals to the DRAM chips on the DIMM [2]. The blue tracks shown in Fig. 4 (right) are the connections between the data buffers (DBs) and RCD which can be implemented using on-PCB links. Wilson et. al [78] implemented a 25Gbps serial link with 1.17pJ/bit with 80mm reach that is sufficient to connect DBs to RCD on a commodity DIMM form factor. Considering that the maximum PCB track length needs to travel half of the DIMM diagonal, the DB to RDC tracks are shorter than 68.5mm for a commodity DDR4 DIMM with 133.35×31.25mm diameters.

Regardless of the NMA's placement, a challenge for near-memory processing architectures is the support of error correction. The main ECC scheme that is implemented on commodity DDR DIMMs is side-band ECC SECDED (Single-bit Error Correction and Double-bit Error Detection) [71], in which the memory controller calculates the parity bits and stores them on separate DRAM chips on the DIMM. On reads, the memory controller re-calculates the error correction codes and compares them against those read from the DIMM to perform the error detection and correction. Side-band ECC provides protection against bit flips both inside the DRAM and the DDR channel. To improve the yield of DRAM chips in new technology nodes, DRAM manufacturers recently added on-die ECC support to their DRAM products [65]. Similar to prior work [53], XFM can leverage the on-die ECC which operates entirely within a DRAM chip to access error-free data from DRAM.

A DRAM with on-die ECC support has a parity engine that intercepts writes to DRAM, calculates a code-word and stores it in the banks. On reads, the code-word passes through the parity engine. If there are any single-bit flips, the error is corrected and sent out of the DRAM chip. If there is more than a single bit flip, the DRAM chip will notify the memory controller [49]. Since the near-memory accelerator (NMA) sits between the DRAM chips and the memory controller, it does not suffer from bit flips in the DDR channel. Therefore, the NMA does not need to implement error detection/correction for error-free data read from DRAM.

Even with on-die ECC support, DIMMs need to implement side-band ECC to protect against DDR channel errors. In this case, the NMA can ignore the ECC bits read from ECC DRAM chips. Although the NMA does not need to detect or correct errors using the parity bits, it should update the parity bits on the ECC DRAM chips so the memory controller can perform side-band ECC error detection and correction. To accomplish this, the NMA calculates the parity bits and stores them in the ECC DRAM chips, when writing back to DRAM chips.

4.2 Address Space Partitioning

A naive XFM implementation statically partitions the entire memory capacity between far memory and local memory and swaps pages between the two regions without fetching data to the CPU. The benefits of static partitioning are the simplicity of the design and the lack of changes to the CPU and DRAM architectures. This architecture lacks flexibility, however, as dynamically resizing far memory capacity is not possible. As static partitioning of the local and far memory capacities neglects one of SFM's key benefits, we only consider architectures that enable fine-grain sharing of DRAM space between local and far memory.

4.3 DRAM Interface to CPU and NMA

One of the advantages of near-memory acceleration is that the energy-hungry DDR channel can be bypassed, with low-energy, on-DIMM PCB tracks moving data between DRAM and the accelerator

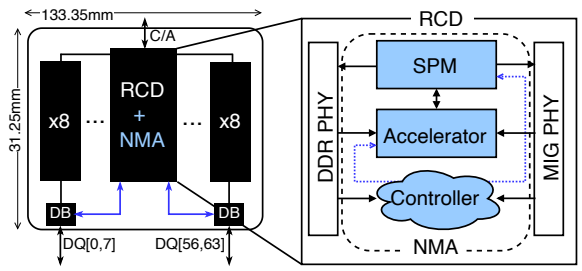


Fig. 4. XFM integrates a near-memory accelerator (NMA) into the buffer device of a commodity DIMM to offload (de)compression operations to the memory module. The NMA implements a ScratchPad Memory (SPM) as a staging buffer for the accelerator's output before it is written back to DRAM.

instead. This alteration cuts the overall data movement energy by 69%. Data travels across the DIMM PCB, between DRAM banks and the buffer device, instead of traversing the much longer DDR channel to the CPU. An important remaining research question is how to enable concurrent CPU and NMA accesses to the DRAM. We aim to design the interface between DRAM and NMA to enable fully transparent DRAM and near-memory accelerator communication. We leverage all-bank refresh cycles in which the entire DRAM rank is locked to implement an interface that matches the needs of SFM.

As explained in Sec.2, each DRAM row is refreshed every t_{REF} (32ms). Within this interval, 8192 REF commands are received and DRAM banks are locked for a duration of 8192×t_{RFC}. Assuming a t_{RFC} of 300ns and multi-bank refresh mode, the DRAM banks are locked ~2.46ms during a 32ms retention interval, which is ~8% of total DRAM access cycles. XFM leverages this locked period to move data between DRAM and the NMA completely transparent to the CPU.

This opportunity is enabled by two unique characteristics of SFM systems. First, in the common case, SFM has a low memory bandwidth requirement. Deploying a 512 GB SFM for a CPU — with four memory channels and two DIMMs per channel — requires 426 MBps access bandwidth between DRAM and the NMA. Second, compression and prefetch (i.e., early decompression due to predictable access pattern) operations are not latency critical and can be delayed. Thus we can postpone a compression offload command until the next refresh cycle, delaying the compression

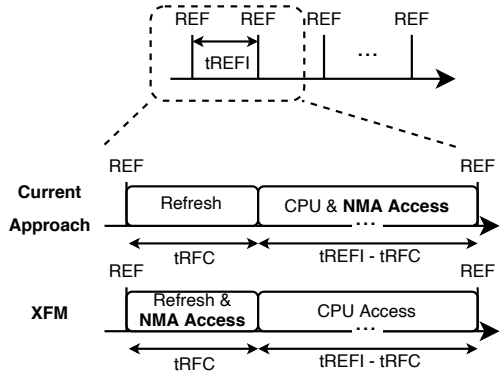


Fig. 5. Timing of NMA accesses in current approaches to concurrent CPU and NMA accesses compared with XFM.

and decompression operations by at least t_{REFI} (3.9μs for 32ms retention time).

XFM leverages the all-bank refresh intervals in which the entire rank inaccessible from the CPU to perform the NMA accesses. Fig. 5 compares the timing of NMA accesses amongst current proposals for arbitration of NMA and CPU accesses [20, 62] and XFM. XFM batches NMA accesses during a t_{REFI} interval and performs them at the end of the interval, in parallel with the all-bank refresh operations occurring within a rank. If all of the NMA accesses cannot be performed during t_{RFC} intervals, then the resulting structural hazard is resolved by the SFM controller which falls back to the CPU to perform any additional compression/decompression operations. In Sec.8 we show that such fall backs are rare when considering realistic promotion rates and a tuned SFM controller. In Sec.6 we explain how XFM can be integrated into an SFM system.

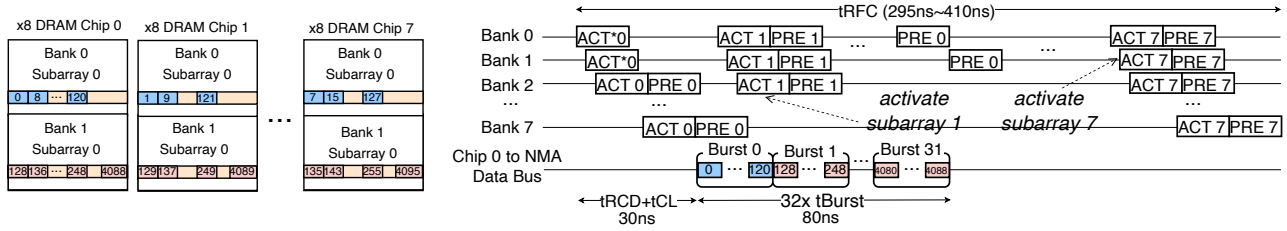
5 XFM HARDWARE ARCHITECTURE

One of the key benefits of confining NMA accesses to t_{RFC} is that the NMA side memory controller can access DRAM without perturbing the access state machine of the CPU side memory controller. At the end of each refresh cycle, all the DRAM banks are precharged and the CPU side memory controller starts fresh, so no modifications to the CPU's memory controller's state machine are necessary. Another benefit is that refresh cycles are no longer wasted since useful computation occurs within the DRAM rank during an all-bank refresh. Furthermore, less access energy is used since NMA accesses do not need to activate a page.

To confine NMA accesses within the refresh cycles, XFM batches all NMA accesses received during t_{REFI} interval and executes them during t_{RFC}. XFM supports two variants of accesses to DRAM: *conditional* accesses and *random* accesses. A *conditional* access requires that the DRAM row containing the accessed page is within the set of rows that are scheduled to be refreshed during a t_{RFC} interval. Row accesses performed outside of t_{RFC} are considered *random*.

To better understand required hardware changes to support conditional and random accesses, we must know the layout of a contiguous 4KB page when it is stored in a DRAM rank. Assuming the Intel Xeon Skylake architecture's physical address mapping [66], a 4KB page is interleaved between four DDR4 channels and two banks. The channel interleaving granularity for Skylake is 256B and bank interleaving granularity is 128B. Fig. 6a shows how a 4KB page is stored in a single DRAM rank assuming a single channel configuration. In the figure, We assume that the physical address of the page maps to a row in subarray 0 of bank 0 and bank 1. We also assume a burst length of 16B for each chip. The eight chips in a rank work in lockstep to prepare 8 bytes for read or write at each rising or falling edge of the clock (i.e., double-data rate).

Table 1 shows the number of rows that are refreshed in each bank during the t_{RFC} interval for three DDR5 chips with different capacities. As the capacity increases, t_{RFC} also increases because more cells need to be refreshed. For a 32Gb chip, sixteen rows per bank are refreshed with each REF command. Assuming that each subarray contains 512 rows of DRAM cells [63], each bank of the 32Gb chip consists of 256 subarrays. Because the number of rows refreshed per t_{RFC} in each bank is much lower than the number



(a) Single channel 4KiB data mapping.

(b) Conditional read during a refresh cycle.

Fig. 6. Utilizing a conditional read to transfer a 4KiB page from a DDR5 3200MHz rank to the NMA. Each 128Byte consecutive physical address is interleaved between two banks. The DRAM timings are not drawn to scale.

of subarrays per bank, it is safe to assume that the rows refreshed within a bank each belong to a different subarray.

To refresh a row, the row needs to be activated and precharged. During a conditional access, XFM keeps a given row activated while reading or writing its data instead of immediately precharging it. Fig.6b illustrates a conditional read cycle of 4KB data from chip 0 of the rank in Fig.6a. Because the row is interleaved between subarray 0 of bank 0 and bank 1, the conditional read keeps the rows in both banks activated and alternates between the two, bursting data from both out through the bank and chip IO to the NMA. If we consider a 3200 MT/s DDR5 [60] transfer rate and a burst length of 16 bytes, it would take 110ns to send all the data out of the chip to the NMA (t_{RCD} + t_{CL} + 32×t_{BURST}). Once the last burst is read, the rows in subarray 0 of bank 0 and 1 can be precharged. While bursting out the data, other rows in subarray 0 of bank 0 and 1 can be refreshed. Considering that t_{RCD} + t_{CL} for subsequent accesses can be overlapped with the tail of the previous burst, the maximum number of 4KB conditional accesses are 4, 3, and 2 for 32Gb, 16Gb, and 8Gb chips.

If a given row must be accessed by the NMA, but is not being refreshed during the t_{RFC} interval, it is still possible perform the access, while other rows in the bank are being refreshed. As stated earlier, we call this a *random* access. During a random access, XFM ensures that there are no subarray conflicts by reordering the pending row accesses if the target row maps to the same subarray as

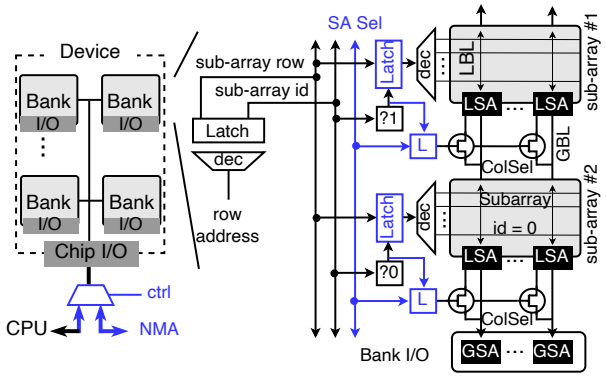


Fig. 7. The required changes to a DRAM bank to enable simultaneous refresh and page access from the same bank. Modifications and added components are highlighted (blue).

Device	8Gb	16Gb	32Gb
#Rows per bank	64K	64K	128K
# Banks per chip	16	32	32
tRFC (all bank refresh)	195	295	410
#Rows of a bank ref during tRFC	8	8	16
#Subarrays per bank	128	128	256

Table 1. DDR5 device configuration [60]

another refresh candidate. As explained in Sec. 2.2, extra TRR operations may be performed to maintain data integrity in commodity DRAM devices. Prior work shows that TRR cycles are only utilized if the number of accesses to neighbouring rows surpass a threshold which is not frequently seen in real scenarios [32]. These unused refreshes can be utilized by XFM to perform random accesses.

Figure 7 illustrates the changes required in the DRAM bank to enable parallel refresh and access to different subarrays within the same bank. Similar to prior work [19], we overlap accesses to rows within one subarray with refreshes taking place in another by propagating the global row address to a row decoder latch which is added to each subarray. It is important to note that the global bit line (GBL) is shared between all the subarrays. There is only one global column select connecting the local bit lines (LBLs) of a given column amongst all subarrays to the GBL. This means it is not possible to have two rows in two separate subarrays activated while only accessing one of them. To overcome this limitation, we also need a mechanism to isolate each subarray's LBLs from the GBL. This is done by having a sub-array select signal and a single bit latch (L) that keeps the LBL of the target subarray connected to the GBL.

6 XFM SYSTEM INTEGRATION

High-Level Overview and System Components. A system utilizing XFM is composed of (1) a `SFM_Controller`, (2) a `SFM_Backend`, and (3) an `XFM_Driver`. The `SFM_Controller` encapsulates the SFM control plane and is responsible for cold page selection. An `SFM_Controller` may be implemented as a userspace program [77] or a kernel daemon [51]. The `SFM_Backend` handles SFM region management and initiation of (de)compression operations. The `XFM_Driver` handles communication with XFM memory modules and initiating SFM operation offloads. Integration of XFM into the control path of an SFM stack requires providing the `SFM_Backend` with access to the: (1) `SP_Capacity_Register`, (2) `Compress_Request_Queue`, and (3) `xfm_compress()` and `xfm_decompress()` functions.

Flow of Control. We first step through the required control sequence to store a cold page in SFM for the baseline CPU implementation. Next, we explain how XFM integrates into this control flow using a modified SFM_Backend.

First, the SFM_Controller selects a cold page based on an algorithm or set of heuristics and passes control to the SFM_Backend using its exposed swapOut() API. swapOut() determines whether the SFM region has space to accommodate any incoming data and initiates an internal compaction operation if the SFM capacity limit is hit. The zswap SFM_Backend [43], implemented in the Linux kernel, uses the zsmalloc memory allocator. This allocator performs compaction by shifting compressed pages via memcpys to one end of the encapsulating OS page. Next, swapOut() locates a free entry with a corresponding virtual address in the SFM region. Upon finding a free entry, the cold page is compressed and copied to a destination address inside an OS page within the SFM region. If the entry is successfully acquired, the cold page data is first compressed using compress() which internally runs a compression algorithm on a source buffer before copying its output to the destination address.

A symmetric process occurs when a cold page is promoted out of SFM. This could be a preemptive promotion incited by the SFM_Controller, or a required swap-in incurred by an application access to non-local memory. In either case, SFM_Backend's swapIn() function determines the SFM_Entry corresponding to the faulting page and uses its virtual address as the destination address in a call to decompress(). After decompress() has completed its internal decompression operations and memory copy, control is returned to the faulting application.

XFM mirrors this flow of control by implementing an SFM_Backend with modified swap-in/out functions: xfm_swap_in() and xfm_swap_out(). For clarity, we distinguish between the baseline SFM_Backend and the modified SFM_Backend with XFM support by referring to the latter as an XFM_Backend.

For the swap-out path, the same flow of control in which the SFM_Controller selects a page and passes control to the XFM_Backend is followed, but the xfm_swap_in() and xfm_swap_out() APIs are called instead of the baseline swapping functions. Similar to the baseline, xfm_swap_in() first checks for available space in SFM with an additional check for capacity in XFM's ScratchPad Memory (SPM). SPM is a staging buffer inside the NMA to store the accelerator's output before it gets written back to DRAM (Fig.4). These checks are performed lazily and do not require synchronization with hardware in the common case. This is because the SFM_Backend can track the number of compression requests made to XFM and maintain knowledge on the upper bound of the SPM's consumed memory.

When a 100% occupancy of the SPM is inferred, an MMIO read is issued to the SP_Capacity_Register to check for the actual available resources. If there is truly no available room for submissions to the Compress_Request_Queue, a CPU_Fallback function is called allowing the host to assume responsibility for compression operations. In the common case, spare capacity will be found since SPM data is written back to DRAM at regular intervals. The SFM_Backend will then simply synchronize with the returned value. Next, xfm_compress() is called, which pushes SFM_Controller's selected page into the Compress_Request_Queue using an MMIO write.

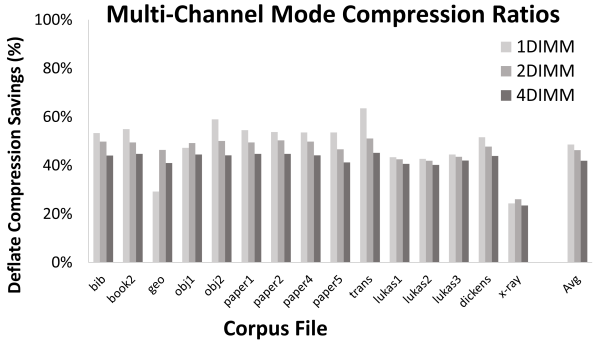


Fig. 8. Average compression ratios of page-divided corpuses compressed at memory-channel interleave granularity using XFM's out of order compressed data layout. Compressing interleaved data shows negligible decreases in space savings. Losses due to the decreased compression window are also minimal, even down the the 1KB window used in the 4-DIMM configuration.

A similar procedure occurs on xfm_swap_out() calls. xfm_swap_out() simply performs a lookup in an internal red-black tree to find the associated physical address of the compressed page entry. Using a reference to the destination page frame, it then calls xfm_decompress() to offload decompression operations to the NMA after performing the same synchronization steps mentioned previously. The key difference with xfm_swap_out() is that CPU_Fallback is called by default unless the do_offload parameter is asserted as applications may be sensitive to the decompression latencies incurred by XFM's datapath. It is up to the SFM_Controller to determine whether to issue an xfm_swap_out() with offloading enabled (e.g., during a page prefetching operation).

Other XFM Backend Operations.

Initialization: To begin operation, XFM needs knowledge of the desired SFM region size and starting offset in physical memory. This is done using the xfm_paramset() function which internally passes this information using ioctl() interfaces that perform MMIO writes to internal configuration registers.

SFM Compaction: One challenge with SFM region management is the need to resolve internal fragmentation resulting from holes left behind by pages promoted out of SFM. The baseline SFM_Backend, intermittently shifts compressed pages to one end of the encapsulating OS pages to make room for additional entries. An SFM_Controller may opt to manually initiate compaction to avoid unpredictable overheads. This is supported by XFM using the xfm_compact() interface, which will shift pages in SFM using memcpys.

Multi-Channel Mode.

We ensure XFM-augmented memory modules are compatible with commodity servers which utilize memory channel interleaving. This is accomplished through codesign of the xfm_compress()/xfm_decompress() functions and the data layout of compressed pages on XFM-enabled DIMMs. Similar to previous works which maintain host control over NMA-accessible regions [20], we assume the OS/runtime is aware of the virtual-to-physical address mapping and avoid DIMM-side address translation by constraining SFM-related,

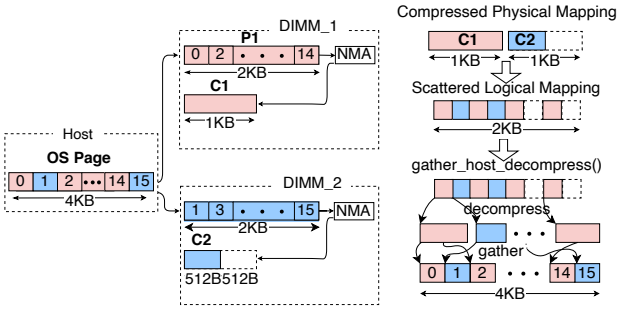
NMA accesses to memory regions that are contiguous within the virtual address space. We first describe and evaluate the compressed page layout within each DIMM's SFM region. Averaging across all compression tests, we find that 86.2% of the compression ratio of an in-order mapping is maintained for a quad memory channel configuration. We also describe the `CPU_Fallback` functions utilized by the `XFM_Driver` to allow the host to perform (de)compressions when a prefetch-enabled `xfm_swap_in()` is not possible or accelerator resources are constrained.

For the remainder of this section, we consider the 256B channel interleaving granularity used by Intel's Skylake architecture [66], although XFM could be configured for compatibility with any address mapping.

The main mechanism used in multi-channel mode is to place cold data from an `xfm_swap_out()` in the same position of each SFM region on each DIMM. Considering that compression ratios are different across XFM memory modules, this comes at the cost of some internal fragmentation. We find that this design decision is justified as design complexity is greatly reduced, and transparent host interaction with the SFM address space is made possible by this approach.

One potential concern in utilizing multi-channel mode is that partitioning uncompressed data across multiple DIMMs can cause compression ratios to drop. We compare the space savings of XFM's multi-channel modes in 1, 2, and 4-DIMM configurations where the 1-DIMM configuration is equivalent to compressing data in host-logical order at 4KiB granularity. The compression ratios of 4KiB pages created using various corpora are shown in Fig.8. 2-DIMM and 4-DIMM configurations divide buffers across compressed memory regions and perform compression on the reordered data (see Fig.9b). We hypothesize that compression ratios are mostly conserved as per-DIMM dictionaries are still able to find ample matches in spite of the interleaved data. Additionally, compression windows are still sized greater than 1KB. We postulate that any lost compression savings are due to the lack of a shared dictionary between DIMMs and the separation of spatially correlated application data, reducing the total number of potential matches in LZ77 encoding.

In the case of insufficient NMA resources, `CPU_Fallback` functions will handle the required (de)compression. As depicted in Fig.9b, the specialized decompression function handles both decompression and gathering operations without additional memory copies, allowing transparent translation between XFM's compressed data layout (shown in Fig.6a) and the host's view of logically contiguous pages in virtual memory.



(a) Interleaved Compression of a 4KiB page (b) Gather decompression

Fig. 9. XFM operating in multi-channel mode.

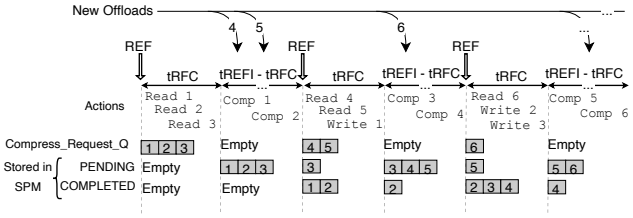


Fig. 10. Asynchronous XFM operations take place on the DIMM, transparent to the CPU. The minimum latency for an XFM operation is $2\times t_{REFI}$.

XFM_Driver. The `XFM_Driver` consists of primitives for interacting with XFM hardware via MMIO operations to internal registers. These low-level functions serve as the interface between `xfm_swap_out()` and `xfm_swap_in()` and relevant NMA resource information exposed via the `SP_Capacity_Register` and `Compress_Request_Queue`. In Linux, the driver functions are exposed using the ioctl() function to interact with an exposed character device exposing the `XFM_Driver`'s functions.

Putting it All Together. Figure 10 illustrates how XFM leverages the t_{RFC} interval to asynchronously access DRAM and perform compression. As new offload requests are submitted to the `Compress_Request_Queue`, XFM schedules batches of read accesses during t_{RFC}. In Fig. 10 we assume that a total of 3 conditional or random accesses can be accommodated during each t_{RFC} interval. The pages read from the `Compress_Request_Queue` will be stored in the SPM with a *PENDING* tag when the compression operation is underway, and with a *COMPLETED* tag when compression is finished. The *COMPLETED* pages will be written back to DRAM in a subsequent t_{RFC} interval. If XFM runs out of SPM space, the contention back propagates to the `Compress_Request_Queue` prompting the `XFM_Driver` to stop submitting new offload requests and instead fall back to the CPU for compression.

7 METHODOLOGY

We implement XFM hardware on Samsung's AxDIMM [47], which integrates a Xilinx UltraScale+ series FPGA into the buffer device of a DDR4 DIMM. Our design includes an open source compression an decompression accelerator customized for memory [64], a 2MB ScratchPad Memory, and controller logic. The compression and decompression accelerators achieve 14.8GBps and 17.2GBps throughput respectively. Since AxDIMM does not provide control over the refresh controller within DRAM ranks, we instead implemented an XFM prototype that monitors read and write accesses from the CPU inside the buffer device. The NMA accesses local DRAM by intercepting data from RD commands or replacing data within WR commands which are transmitted over the DDR data bus to the DRAM chips. This implementation introduces cache pollution and does not save memory channel bandwidth when compressing and decompressing pages. We use the prototype as a proof

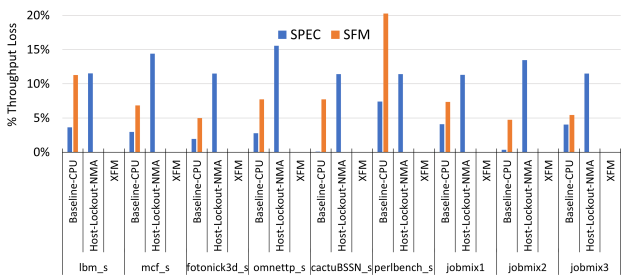


Fig. 11. Interference between SPEC and SFM operations. XFM eliminates the interference in the memory hierarchy.

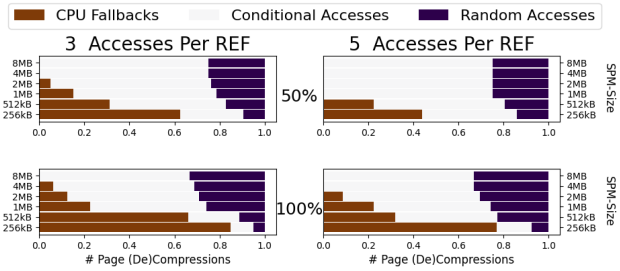


Fig. 12. CPU fallbacks for different SPM sizes implementing a 512GB SFM. The top figures are for a 50% promotion rate, and the bottom ones are for a 100% promotion rate.

of concept to test the software stack functionality and estimate XFM power and area overheads.

A real implementation of XFM operates transparently to CPU and eliminates CPU and NMA access interference. This motivates our performance comparison for which we implement an XFM emulator that skips compression and decompression tasks while implementing the complete software stack, including the XFM kernel driver to support userspace offloads, page registration, and MMIO communication with DIMMs. The emulator feeds swap-in/out data into a cycle approximate DRAM timing model that implements XFM timing based on gem5's [14] DDR4-2400 DRAM interface. Swap-in/out traces are generated using the AIFM userspace far memory framework [70] when running a synthetic web front-end application [1]. Objects are allocated at the traditional page-size granularity to represent swap-in/outs in production operating systems that use paging. We integrate the emulation framework as well as a fully-functional SFM backend into AIFM. More specifically, we develop two compressed far memory back-ends for the AIFM framework: one for baseline CPU (de)compression and one for emulating XFM.

Both the CPU (de)compression and XFM emulator backends use the zstd [22] algorithm. The experiments are performed on servers equipped with Intel Xeon Scalable Gold 6242 CPUs and 6× DIMMs of 16 GB memory at 3200 MHz (96GiB). We set the retention time of DRAM to 32ms, t_{RFC} to 410ns, t_{BURST} to 2.5ns, and assume that only one random access can be performed during a t_{RFC}. We utilize hardware counters exposed by turbostat [4] as well as Intel RAPL [3] for profiling package and memory power consumption. The Xilinx Vivado Suite was used to synthesize and implement XFM.

8 EVALUATION

Memory & Cache Contention Mitigation. To evaluate the impact of SFM operations on job throughput, we co-run a set of memory-intensive SPEC benchmarks with several antagonist processes running SFM swap ins and outs (compression and decompression). The antagonist implements an SFM with an extra capacity of 512GB and a moderate promotion rate of 14%. We isolate antagonist processes and run them on separate cores to isolate the impact of memory and shared cache interference on the co-running applications. We compare XFM against Baseline (CPU) and Host-Lockout-NMA configurations. Host-Lockout-NMA implements an NMA-DRAM interface that locks host accesses when the NMA accesses are in progress, mimicking Boroumand et al.'s design [16].

As shown in Fig. 11, although Host-Lockout-NMA does not see any performance degradation for SFM, it suffers a higher runtime penalty for SPEC when co-running. This is because the low per-rank memory bandwidth requirement of SFM does not justify the lockout interface implemented by Host-Lockout-NMA. Host-Lockout-NMA would be beneficial for coarse-grain near-memory offloads or instances when near-memory offloading could eliminate all the CPU accesses to DRAM. In the Baseline-CPU configuration, the SFM throughput degrades by 5∼20%, while the SPEC workloads see up to 8% and 15% performance degradation for Baseline-CPU and Host-Lockout-NMA, respectively. The job mix configurations include multiple SPEC applications co-running on separate CPUs. We note that the SFM throughput degradation could have a multiplicative impact on job throughput in production as the reduction in extra memory capacity would limit additional jobs that could be scheduled on a server.

CPU Fall Backs. When the hardware resources on XFM DIMMs are exhausted (i.e., the SPM gets full), the XFM driver falls back to the CPU to perform compression and decompression operations. Fig.12 displays the sensitivity of CPU fall back rate to SPM size and the number of accesses that can be accommodated in each t_{RFC} assuming a 32ms retention time. Because t_{RFC} is correlated with the number of rows that are refreshed every t_{REFI}, the number of NMA accesses per REF command is a function of t_{RFC} and, by extension, device capacity and DRAM manufacturing technology node. As shown in Fig.12, regardless of the promotion rate, an 8MB SPM can eliminate all CPU fall backs for an XFM implementation that accommodates 3 NMA accesses per REF command. Fig.12 also shows the breakdown of conditional and random accesses. The rate of random accesses is shown to scale with the promotion rate, however, the majority of accesses can be accommodated with conditional accesses. On average, the conditional accesses enable XFM to reduce the NMA access energy by 10.1% across various promotion rates and DRAM configurations.

Compression Ratio in Multi-Channel Mode. We choose 16 corpus files and measure the compression ratio of XFM when running in single-channel, 2-channel, and 4 channels modes. On average, 2- and 4-channel modes reduce the memory savings from compression by 5% and 14%, respectively. This reduction in memory saving comes from the lower compression ratio of Deflate operating

in multi-channel mode and the internal fragmentation introduced by aligning compressed data to the same offset across separate DIMMs (§6). More sophisticated memory management and support for larger offload sizes (instead of fixed 4KB offloads) could improve the savings in multi-channel mode. We leave this investigation to future work.

Area and Power Overhead. The resource utilization of the FPGA implementing XFM is shown in Table 2. The reason for the high LUT utilization of our XFM implementation is the complexity of the compression and decompression logic functions. The FPGA implementation of the open-source Deflate accelerator sustains around 1.4 GBps compression and 1.7 GBps decompression throughput, which is highly overprovisioned for XFM. The theoretical memory bandwidth available to the NMA is less than 1 GBps (when leveraging parallel refresh accesses), therefore the compression and decompression units are mostly underutilized in our prototype. As shown in Table 3, an XFM FPGA implementation consumes 5.7 W and 1.3 W of dynamic and static power respectively.

Our CACTI simulation, modeling a 8Gb DDR-4 DRAM chip in 22nm technology, shows that the required changes to DRAM banks to enable parallel refresh and subarray accesses incur ~0.15% area and ~0.002% power overhead, corroborating prior work [48].

9 RELATED WORKS

There are a myriad of prior works which enable far memory using various systems and interconnects [6, 8, 18, 26, 35, 36, 55–57, 70, 81]. To our knowledge, XFM is the is the first work to use near-memory processing to accelerate SFM. There are also many prior works which accelerating compression and decompression of lossless compression algorithms [5, 21, 31, 33, 52, 64, 67]. The goal of our work on XFM is not to design a compression accelerator, but to instead leverage existing accelerators within a near-memory processing architecture which is constructed specifically for memory compression.

Many works have proposed novel near-memory processing architectures [7, 10, 29, 34, 41, 44, 45, 50, 72–74, 80, 82]. Boroumand et al. [16] identified compression as a lucrative workload for near-memory processing on consumer devices as the uncompressed data is cold and sits inside the memory. Motivated by the same observation, XFM offloads SFM operations in commodity DRAM devices. Integration of NMA into the RCD of commodity DIMMs enables XFM to achieve higher compression ratios and construct a novel data layout and access scheme for performing SFM operations across multiple DDR channels. Most importantly, XFM enables NMAs to transparently access DRAM without compromising CPU accesses. In contrast to prior works on near-memory acceleration that are centered around maximizing the memory bandwidth between the near-memory accelerators (NMAs) and DRAM, XFM is designed to implement an interface between NMAs and DRAM with just-enough bandwidth for its target workload, SFM. This enables XFM to leverage the inevitable refresh cycles to implement a side-channel on each rank enabling the NMA to access DRAM completely transparent to the CPU memory controller.

Chang et al. evaluate an out-of-order per-bank refresh mechanism and modifications to DRAM banks which allow parallelization of refreshes and accesses to different subarrays in the same bank [19]. XFM uses a similar technique to overlap refresh and NMA accesses.

Table 2. FPGA resource utilization of XFM

Resource	Used	Total	Percent
LUTs	435467	522720	83.30%
FFs	94135	1045440	9.00%
BRAM	51	984	5.18%

Table 3. Power consumption breakdown of XFM

Power consumption	Dynamic	%	Static	%
Total = 7.024 Watts	5.718	81	1.306	19

10 CONCLUSION

In this work, we first compare the capital and environmental cost of operating the same capacity disaggregated far memory (DFM) and software-managed far memory (SFM) and showed that it can take several years for DFM to break even in terms of both capital and environmental costs with SFM. We then discuss opportunities for accelerating SFM using near-memory processing. Leveraging the unique characteristics of SFM, we designed XFM which implements a transparent side channel between near-memory accelerators (NMA) and DRAM to perform NMA accesses concurrently with the CPU. We also explain how XFM integrates into the system and operates in multi-channel mode. Finally, we integrate XFM into an application-level far memory implementation and prototype it on an FPGA platform.

Our results show the promise of XFM as well as the concurrent NMA-CPU access mechanism it utilizes. They also show a significant reduction in memory bandwidth utilization while performing SFM-related operations. XFM is capable of eliminating memory bandwidth utilization of compression and decompression operations for SFMs with capacities of up to 1TB. Additionally, its memory and cache utilization reductions translate to 5~27% improvement in the performance of co-running SPEC applications.

ACKNOWLEDGMENTS

This work was supported in part by grants from National Science Foundation (CCF-2239020 and DGE-1565570), Samsung's Open Innovation Contest, and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank NVIDIA Academic Hardware Grant Program and Ampere Computing for their hardware donations.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to use the provided scripts to reproduce the sensitivity analysis in Section 6 and evaluations in Section 8 of this paper. We will utilize publicly available compression corpora and software compression algorithms as well as the lzbench and SPEC 2017 benchmarks. Results reproduced will include compression ratios achievable by XFM for different memory configurations, and an evaluation of contention between SPEC workloads and corunning (de)compression tasks.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Model of the XFM Memory Module and Sensitivity Analyses
- **Program:** SPEC 2017, Licensed Benchmark Suite
- **Compilation:** gcc 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)
- **Run-time environment:** Tested on Ubuntu 18.04
- **Hardware:** Intel c6250 Server
- **Output: Numerical results and plots corresponding to Figures 8, 11, and 12.**
- **Experiments:** As described in Section 6 (Multi-Channel Mode) and Section 8 (Evaluation).
- **How much time is needed to prepare workflow (approximately)?:** Less than 10 minutes. Building SPEC 2017 is the most time consuming part.
- **How much time is needed to complete experiments (approximately)?:** Less than 1 hour for XFM performance profiling tests (figures 8 and 12). Between 2-3 hours to regenerate SPEC (de)compression contention results from figure 11.
- **Publicly available?:** https://github.com/architecture-research-group/XFM-Artifacts.git
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?: https://doi.org/10.5281/zenodo.8353767**

A.3 Description

A.3.1 How to access. See github link above.

A.3.2 Hardware dependencies. Some experiments depend on access to a server which can be a multi-core server which can be allocated as a single Intel server (c6220 node) available through cloudlab. This hardware is sufficient to reproduce all results.

A.3.3 Software dependencies. See github link above.

A.4 Installation

Obtain the code and sub-modules from github and run corresponding fetch and run scripts. Instructions for each experiment are provided in each subdirectory of the repository. Scripts for running each experiment are provided as bash shell scripts.

A.5 Experiment workflow and expected results

Use shell scripts (e.g., fetch.sh and run.sh) to acquire any required dependencies and corpus files and reproduce results. Plot them using python scripts (Experiment-specific instructions provided in the Github repository provided above).

A.6 Experiment customization

Different SPEC workloads can be tested by modifying the shared.sh file provided in the `spec_workload_experiment` subdirectory. Different memory channel interleave granularities can be testedu is

REFERENCES

[1] [n. d.]. *DataFrame Documentation / Code Samples.* https://github.com/hosseinmoein/DataFrame

[2] [n. d.]. DDR5 DIMM Chipset. https://www.rambus.com/memory-and-interfaces/server-dimm-chipsets/ddr5-dimm-chipset/

[3] [n. d.]. Reading RAPL energy measurements from Linux. https://web.eece.maine.edu/~vweaver/projects/rapl/

[4] [n. d.]. turbostat - Report processor frequency and idle statistics at Linux.org. https://www.linux.org/docs/man8/turbostat.html

[5] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[6] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[7] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. 2018. Application-Transparent Near-Memory Processing Architecture with Memory Channel Network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 802–814. https://doi.org/10.1109/MICRO.2018.00070

[8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[9] Vinodh Gopal Wajdi Feghali Mahesh Wagh Alberto Villarreal Arijit Biswas, Ruchira Sasanka and Dan Zimmerman. 2022. Technical Overview Of The 4th Gen Intel® Xeon® Scalable processor family. https://doi.org/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.r5l2ms

[10] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 908–920. https://doi.org/10.1109/HPCA51647.2021.00080 ISSN: 2378-203X.

[11] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014), 36–42. https://doi.org/10.1109/MM.2014.55

[12] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 2018. 'Zeppelin': An SoC for multichip architectures. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. 40–42. https://doi.org/10.1109/ISSCC.2018.8310173

[13] Ishwar Bhati, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. 2015. Flexible auto-refresh: Enabling scalable and energy-efficient DRAM refresh reductions. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 235–246. https://doi.org/10.1145/2749469.2750408

[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. https://doi.org/10.1145/2024716.2024718

[15] BOAVIZTA. 2021. Digital & environment : How to evaluate server manufacturing footprint, beyond greenhouse gas emissions? https://boavizta.org/en/blog/empreinte-de-la-fabrication-d-un-serveur.

[16] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[17] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for near-Data Accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 629–642. https://doi.org/10.1145/3307650.3322266

[18] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[19] 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)*. 356–367. https://doi.org/10.1109/HPCA.2014.6835946

[20] Benjamin Y Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near data acceleration with concurrent host access. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 818–831.

[21] Seungdo Choi, Youngil Kim, Daeyong Lee, Sangjin Lee, Kibin Park, Yun Heub Song, and Yong Ho Song. 2019. Design of FPGA-Based LZ77 Compressor With Runtime Configurable Compression Ratio and Throughput. *IEEE Access* 7 (2019), 149583–149594. https://doi.org/10.1109/ACCESS.2019.2947273 Conference Name: IEEE Access.

[22] Yann Collet and Murray Kucherawy. 2021. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878. https://doi.org/10.17487/RFC8878

[23] Yann Collet and Chip Turner. 2016. Smaller and faster data compression with Zstandard. https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/

[24] David J Craft. 1998. A fast hardware data compression algorithm and some algorithmic extensions. *IBM Journal of Research and Development* 42, 6 (1998), 733–746.

[25] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or Not for Emerging General Purpose Processing in DDR Memory Systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 231–244. https://doi.org/10.1145/3470496.3527431

[26] 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. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[27] ELECTRICITY MAP. 2021. ELECTRICITY MAP. https://app.electricitymaps.com/map.

[28] EnergyBot. 2023. EnergyBot. https://www.energybot.com/electricity-rates-by-state.html.

[29] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295. https://doi.org/10.1109/HPCA.2015.7056040

[30] Antaeus Feldspar. 2002. An Explanation of the Deflate Algorithm. https://zlib.net/feldspar.html

[31] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 52–59. https://doi.org/10.1109/FCCM.2015.46

[32] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. 747–762. https://doi.org/10.1109/SP40000.2020.00090

[33] Ruihao Gao, Xueqi Li, Yewen Li, Xun Wang, and Guangming Tan. 2022. MetaZip: A High-Throughput and Efficient Accelerator for DEFLATE. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 319–324. https://doi.org/10.1145/3489517.3530450

[34] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2021. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 263–276. https://doi.org/10.1109/HPCA51647.2021.00031 ISSN: 2378-203X.

[35] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[36] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap.. In *NSDI*. 649–667.

[37] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2022. ACT: Designing Sustainable Computer Systems with an Architectural Carbon Modeling Tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 784–799. https://doi.org/10.1145/3470496.3527408

[38] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. https://doi.org/10.48550/ARXIV.2110.10603

[39] Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[40] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 25–32. https://doi.org/10.1109/ICCD.2016.7753257

[41] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2019. Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 726–739. https://doi.org/10.1145/3352460.3358297

[42] JEDEC. 2022. *DDR5 SDRAM*. Technical Report.

[43] Seth Jennings. 2013. The zswap compressed swap cache. https://doi.org/Articles/537422/

[44] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[45] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. 2022. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM. *IEEE Micro* 42, 1 (Jan. 2022), 116–127. https://doi.org/10.1109/MM.2021.3097700 Conference Name: IEEE Micro.

[46] Jongseok Kim, Cheolgi Kim, and Euiseong Seo. 2019. $ezswap$: Enhanced Compressed Swap Scheme for Mobile Devices. *IEEE Access* 7 (2019), 139678–139691. https://doi.org/10.1109/ACCESS.2019.2942362

[47] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. 2022. Aquabolt-XL HBM2-PIM, LPDDR5-PIM With In-Memory Processing, and AXDIMM With Acceleration Buffer. *IEEE Micro* 42, 3 (2022), 20–30. https://doi.org/10.1109/MM.2022.3164651

[48] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 368–379. https://doi.org/10.1109/ISCA.2012.6237032

[49] Sanghyuk Kwon, Young Hoon Son, and Jung Ho Ahn. 2014. Understanding DDR4 in pursuit of In-DRAM ECC. In *2014 International SoC Design Conference (ISOCC)*. 276–277. https://doi.org/10.1109/ISOCC.2014.7087646

[50] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 740–753. https://doi.org/10.1145/3352460.3358284

[51] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. http://doi.acm.org/10.1145/3297858.3304053

[52] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. 2020. High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis. *IEEE Access* 8 (2020), 62207–62217. https://doi.org/10.1109/ACCESS.2020.2984191 Conference Name: IEEE Access.

[53] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021.

Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[54] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*. 1.1.1–1.1.8. https://doi.org/10.1109/IEDM.2016.7838026

[55] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. 2019. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation. *arXiv preprint arXiv:1910.09727* (2019).

[56] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[57] 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. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[58] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. 2002. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development* 46, 2.3 (2002), 187–212. https://doi.org/10.1147/rd.462.0187

[59] Deepak M. Mathew, Éder F. Zulian, Matthias Jung, Kira Kraft, Christian Weis, Bruce Jacob, and Norbert Wehn. 2017. Using Run-Time Reverse-Engineering to Optimize DRAM Refresh. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia) *(MEMSYS '17)*. Association for Computing Machinery, New York, NY, USA, 115–124. https://doi.org/10.1145/3132402.3132419

[60] Micron. 2020. dDR5SDRAM. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf.

[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* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/2485922.2485927

[62] Anirban Nag and Rajeev Balasubramonian. 2021. OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 298–310. https://doi.org/10.1145/3466752.3480103

[63] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3123939.3124545

[64] Gagandeep Panwar, Muhammad Laghari, David Bears, Yuqing Liu, Chandler Jearls, Esha Choukse, Kirk W. Cameron, Ali R. Butt, and Xun Jian. 2022. Translation-optimized Memory Compression for Capacity. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 992–1011. https://doi.org/10.1109/MICRO56248.2022.00073

[65] Minesh Patel, Jeremie S. Kim, Hasan Hassan, and Onur Mutlu. 2019. Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 13–25. https://doi.org/10.1109/DSN.2019.00017

[66] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[67] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015 ISSN: 2576-2621.

[68] Suzanne Rigler, William Bishop, and Andrew Kennings. 2007. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *2007 Canadian conference on electrical and computer engineering*. IEEE, 1235–1238.

[69] Dave Rodgman and Willy Tarreau. [n. d.]. *LZO stream format as understood by Linux's LZO decompressor*. The kernel development community. https://docs.kernel.org/staging/lzo.html

[70] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 18, 18 pages.

[71] Vadhiraj Sankaranarayanan. 2023. Error Correction Code (ECC) in DDR Memories. https://www.synopsys.com/designware-ip/technical-bulletin/error-correction-code-ddr.html

[72] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications. *IEEE Micro* 41, 4 (July 2021), 39–48. https://doi.org/10.1109/MM.2021.3088396 Conference Name: IEEE Micro.

[73] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F. Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2019. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3316781.3317867

[74] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 237–250. https://doi.org/10.1109/ISCA52012.2021.00027 ISSN: 2575-713X.

[75] Kosuke Suzuki. 2015. A survey of trends in non-volatile memory technologies: 2000-2014. *IEEE International Memory Workshop (IMW)* (2015).

[76] Zoltan Szabadka. 2016. Introducing Brotli: a new compression algorithm for the internet. https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html

[77] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[78] John M. Wilson, Walker J. Turner, John W. Poulton, Brian Zimmer, Xi Chen, Sudhir S. Kudva, Sanquan Song, Stephen G. Tell, Nikola Nedovic, Wenxu Zhao, Sunil R. Sudhakaran, C. Thomas Gray, and William J. Dally. 2018. A 1.17pJ/b 25Gb/s/pin ground-referenced single-ended serial link for off- and on-package communication in 16nm CMOS using a process- and temperature-adaptive voltage regulator. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. 276–278. https://doi.org/10.1109/ISSCC.2018.8310291

[79] Francis G Wolff and Chris Papachristou. 2002. Multiscan-based test compression and hardware decompression using LZ77. In *Proceedings. International Test Conference*. IEEE, 331–339.

[80] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583. https://doi.org/10.1109/HPCA51647.2021.00055 ISSN: 2378-203X.

[81] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

[82] Zhe Zhou, Cong Li, Fan Yang, and Guangyu Sun. 2023. DIMM-Link: Enabling Efficient Inter-DIMM Communication for Near-Memory Processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 302–316. https://doi.org/10.1109/HPCA56546.2023.10071005 ISSN: 2378-203X.