Quantifying Memory Underutilization in HPC Systems and Using it to Improve Performance via Architecture Support

Gagandeep Panwar*
Virginia Tech
Blacksburg, USA
gpanwar@vt.edu

Da Zhang*
Virginia Tech
Blacksburg, USA
daz3@vt.edu

Yihan Pang*
Virginia Tech
Blacksburg, USA
pyihan1@vt.edu

Mai Dahshan
Virginia Tech
Blacksburg, USA
mdahshan@vt.edu

Nathan DeBardeleben
Los Alamos National Laboratory
Los Alamos, USA
ndebard@lanl.gov

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

Xun Jian
Virginia Tech
Blacksburg, USA
xunj@vt.edu

ABSTRACT

A system's memory size is often dictated by worst-case workloads with highest memory requirements; this causes memory to be underutilized in the common case when the system is not running its worst-case workloads. Cognizant of this memory underutilization problem, many prior works have studied memory utilization and explored how to improve it in the context of cloud.

In this paper, we perform the first large-scale study of system-level memory utilization in the context of HPC systems; through seven million machine-hours of measurements across four HPC systems, we find memory underutilization in HPC systems is much more severe than in cloud. Subsequently, we also perform the first exploration of architectural techniques to improve memory utilization specifically for HPC systems. We propose exposing each compute node's currently unused memory to its CPU(s) via novel architectural support for OS. This can enable many new microarchitecture techniques that use the abundant free memory to boost microarchitecture performance transparently without requiring any user code modification or recompilation; we refer to them as Free-memory-aware Microarchitecture Techniques (FMTs). We then present a detailed example of an FMT – Free-memory-aware Memory Replication (FMR). On average across five HPC benchmark suites, FMR provides 13% performance and 8% system-level energy improvement compared to a highly optimized baseline representative of modern memory systems. To check the performance

benefits our simulation reports, we emulated FMR in a real system and found close corroboration between simulation results and real-system emulation results. The paper ends by discussing other possible FMTs and applicability to other types of systems.

CCS CONCEPTS

• **Hardware Dynamic memory**; • **Computer systems organization** *Grid computing*; • **Software and its engineering** Main memory.

KEYWORDS

Memory Architecture, Memory Management, HPC Systems, Operating Systems, DRAM, Supercomputing

1 INTRODUCTION

A system's physical memory size is often determined by the worst-case memory usage scenario with the highest memory requirement. However, memory usage varies by workload; much of the memory in a system is often not in use when running common-case workloads that use less memory. Many prior works [20, 24, 42, 55, 73] have quantified the memory underutilization problem in the context of cloud. Correspondingly, many prior works have explored techniques to mitigate memory underutilization in cloud; they propose intelligent ways to colocate heterogeneous workloads (e.g., memory-hungry database workloads and compute-intensive workloads) on the same machine to improve memory utilization. Examples include colocating both the memory and computations of different workloads on the same machine (e.g., Quasar [23], ElasticMem [76], VM Memory Overcommit [8]) and colocating different

workloads' program memory, but not computation (e.g., Disaggregated Memory [27, 52, 53], Infiniswap [34]).

In this paper, we perform the first large-scale study of HPC systems' system-level memory usage (i.e., each compute node's total physical memory usage, encompassing everything from memory used by the OS, by disk buffering/file caching, and user jobs themselves, etc). Our study spans four months of operation in four in-production HPC systems in Los Alamos National Laboratory and Virginia Tech, totaling seven million machine-hours of observation and ~three billion memory usage measurements. We find that the average system-level memory usage in active nodes running user jobs is only 24%; as such, memory utilization in HPC systems is much lower than in cloud, which is > 50% according to prior studies [20, 24, 42, 55, 73].

Memory utilization enhancement techniques that are effective for cloud, such as OS-directed file caching and workload co-location, are ineffective for HPC systems. HPC workloads are typically compute-intensive rather than storage-intensive (e.g., like database workloads); this minimizes the effectiveness of file caching. HPC workloads are also highly parallel; as a single workload often already has sufficient threads/processes to take up all cores in a compute node, workload colocation is unsuitable for HPC systems. In fact, many HPC systems (e.g., in all US national laboratories) deliberately disallow colocation of independent workloads on the same compute nodes to minimize the negative impact of increased thread/process-level performance variation on parallel workloads caused by the resultant inter-workload interference.

In this paper, we perform the first exploration of architectural techniques to improve memory utilization for HPC systems. We propose exposing each compute node's currently unused memory to its CPU(s) via novel architectural support for the operating system in a user-transparent manner (i.e., user code does not need to be modified or recompiled). When made free-memory aware, hardware can leverage HPC systems' abundant free memory in the common to record arbitrary data to boost microarchitecture performance. We present a detailed free-memory-aware microarchitecture technique – Free-memory-aware Memory Replication (FMR); FMR effectively hides many state-dependent memory latencies by using free memory locations to replicate logical memory blocks to allow CPU to fetch from the memory location with the faster state at the time of LLC miss. FMR can be readily deployed on commodity memory systems with commodity memory chips and commodity memory modules. We end the paper with discussing other use cases and applicability to other types of systems.

We make the following contributions in this paper:

- We perform the first large-scale study of system-level memory utilization for HPC systems. We find HPC systems suffer from more severe memory underutilization than cloud.
- We are first to explore architectural techniques to address memory underutilization in HPC systems. We propose exposing each compute node's OS-visible free memory to the node's CPU(s) to improve performance.
- We are first to propose the general concept of Free-memory-aware Microarchitecture Techniques (FMTs), which opportunistically records arbitrary data in free memory locations to boost microarchitecture performance.

- We present a detailed FMT – Free-memory-aware Memory Replication - that improves performance by 13% and system-level energy efficiency by 8%, on average across five HPC benchmark suites.
- To check the performance benefits our simulation reports, we emulated FMR in a real system and found close corroboration between simulation and real-system emulation results.

2 QUANTIFYING MEMORY UNDER-UTILIZATION IN HPC SYSTEMS

While many prior works [24, 50, 80] have studied HPC workload characteristics, they have only studied program-level memory usage. Our study quantifies system-level memory usage, which includes all memory usages (e.g., memory used by OS, by disk buffering/file caching, by user jobs themselves, etc.).

We studied for four months the memory utilization of four HPC systems – Grizzly, Badger, Snow at Los Alamos National Laboratory (LANL) and Cascade at Virginia Tech. Table 1 describes the studied HPC systems. We have made the raw measurement data for LANL systems publicly available at *https://usrc.lanl.gov/data/LA-UR-19-28211.php*. The biggest HPC system we studied – Grizzly - is a mid-range Top500 supercomputer [2]. All systems deploy the widely-used SLURM job scheduler. We collect every node's memory usage once every ten seconds by using the LDMS [5] monitoring tool. LDMS' "Meminfo/MemFree" output reports how much memory in the node is currently completely not in use (e.g., not by the user job, not by the OS, disk buffering/file caching, not by anything); we refer to such idle memory as *free memory*. Each compute node's system-level memory usage is calculated as the node's physical memory size minus its free memory.

Figure 1 shows the breakdown of how often active nodes use a given maximum amount of memory across one hour time intervals; for example, it shows that in Grizzly, the maximum memory usage of a node when active is less than 32GB in 79% of all one-hour intervals over the four-month study. We refer to a node running

Table 1: Description of the studied systems. "System utilization" is the fraction of time a system's compute node is running user job(s), on average across all nodes in the system.

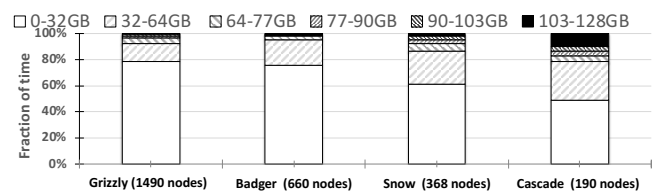| Cluster Name	Total Compute Nodes	Computing hardware per Node	Memory per Node	System Utilization	Age
Grizzly	1490	2x 18-core E5	128GB	78%	2 Years
Badger	660	2x 18-core E5	128GB	75%	1 Year
Snow	368	2x 18-core E5	128GB	83%	2 Years
Cascade	190	2x 18-core E5	128GB	71%	3 Years

Figure 1: Distribution of active nodes' hourly memory usage. A node's hourly memory usage is its maximum usage during the hour. 79% of hourly memory usages in active nodes in Grizzly are <= 32GB. All studied system have 128GB/node.

user job(s) as an active node; as such, Figure 1 filters out for each node all one-hour intervals in which the node did not run any job(s). The average node-level memory utilization of active nodes are 18%, 17%, 34%, and 26% for Grizzly, Badger, Snow, and Cascade respectively. Active nodes use on average < 50% of their memory for 88% of the time when equally weighing the studied systems.

Figure 2 shows every node's memory utilization; it shows for each node its maximum memory utilization observed during the study, the 90th percentile memory utilization (i.e., the node's memory utilization in a one-hour interval that is greater than the memory utilization of 90% of the node's one-hour intervals), and the 80th percentile memory utilization. Figure 2 also only considers active intervals in which a node has user job(s). Figure 2 shows there is a large gap between a node's maximum/worst-case memory utilization and common-case memory utilization (e.g., memory utilization for 90% or 80% of the time).

As Figures 1 and 2 report system-level memory usage, they account for all memory used by existing OS-level optimization. One important OS optimization is to opportunistically use free memory to transparently cache accessed files. The OS community has called this optimization by different names, such as disk buffering, caching, etc.; we call it the OS file cache to clearly distinguish it from CPU caches. One interesting question is why the OS file cache does not often expand to occupy all free memory over time, given that it can accumulate file pages accessed by all past jobs ran on the node. This question is particularly intriguing given that all studied systems are heavily utilized (see Table 1) and none of the studied systems enforce any size cap for the OS file cache as we have empirically verified through our measurements.

In HPC systems, a node's OS file cache typically grows slowly for two reasons. First, unlike data center workloads, which are storage-intensive (i.e., spend short time on computation after accessing the file system), HPC workloads tend to be compute-intensive (i.e., compute for a long time after reading input file(s)). Second, inputs to a node participating in a distributed job often come directly from a master process (e.g., an MPI master process) through message passing, not through the file system and, therefore, are often not inserted into many participating nodes' OS file caches. A major exception to the above is writing checkpoint files to provide fault tolerance. However, under network-based file systems (e.g., NFS,

HPFS, Lustre), which are commonly used by HPC systems, writing to files can cause evictions from client-side caches to help preserve cache coherence [72].

In HPC systems, a node's OS file cache often shrinks for multiple reasons. After a job ends, its input files are often compressed/archived and then deleted to preserve precious storage space on storage servers; deleting a file evicts its content from OS file cache. When running jobs that allocate a large amount of memory, OS also evicts cached file pages to make room for program memory; OS does not refetch evicted file pages when program ends to avoid costly storage and network access overheads. Even if OS were to refetch evicted file pages, it would only benefit accesses to few shared files, such as executables, but not individual users' files, because which nodes are available the next time the same user submits a job are often different in busy HPC systems.

In short, the slow growth rate of OS file cache coupled with the many factors shrinking OS file cache keeps it small in HPC systems.

A potential simple solution to address memory underutilization is to turn off unused memory. Off memory is still under utilized, however. Turning off memory can also reduce system performance by reducing memory rank-level and bank-level parallelism. Furthermore, memory voltage and, thus, power has also been steadily declining [39–41]; this has reduced memory's contribution to system power from ~30% in 2009 [9] down to ~18% in 2018 [10]. Another potential simple solution is to reduce memory system size. However, this reduces the maximum solvable problem size and, therefore, reduces the HPC system's capability, which is an important metric of merit for HPC systems.

Prior techniques that colocate heterogeneous workloads on the same node to improve memory utilization in cloud [8, 23, 34, 52, 76] are inadequate for HPC systems. Individual HPC workloads are highly parallel and, therefore, often occupy all cores in a node; deliberately spreading the threads/processes of individual workloads across more nodes than necessary to colocate different workloads' threads/processes on the same nodes increases network communication overheads and, therefore, reduces parallel performance. Also because they tend to be more parallel than cloud workloads, HPC workloads are more sensitive to performance variation; slowing down a single thread can significantly slow down the total execution time of parallel programs. By making multiple workloads share the same node's network access, CPU power budget, etc, workload colocation can cause substantial performance variability; to provide performance isolation, many HPC systems (e.g., in all US national laboratories) deliberately disallow workload colocation.

3 ARCHITECTURAL SUPPORT FOR OS TO EXPOSE FREE MEMORY TO CPU

Ideally, OS should be able to utilize HPC systems' abundant unused memory to effectively boost performance, just as OS can do so for data center (e.g., database) workloads via the OS file cache. To this end, we propose a novel architectural support for OS to expose a node's OS-visible free memory to its CPU(s) to enable Free-memory-aware Microarchitecture Techniques (*FMTs*), a new class of microarchitecture techniques that opportunistically record arbitrary data in free memory to boost microarchitecture performance. We observe through the Advanced Configuration and Power

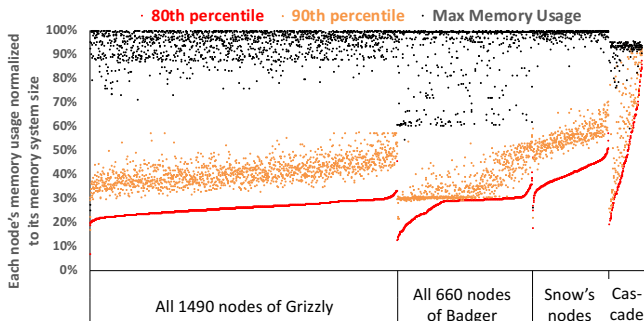Figure 2: The maximum, 90th percentile, and 80th percentile memory utilization of every node when active. Each vertical slice of three points belong to a distinct node. Nodes within each system are sorted by their 80th percentile utilization.

Interface (ACPI), OS in existing systems can already inform CPU of the underutilization of various hardware resources so that CPU can opportunistically boost microarchitecture performance. For example, OS can instruct CPU to power down cores/caches via ACPI; conceptually, this tells CPU which cores/caches are not in use and allows CPU to exploit this knowledge to opportunistically boost microarchitecture performance (e.g., to turbo-boost the frequency of the still in-use cores). Based on the above observation, we propose piggybacking on ACPI to enable OS to inform hardware which physical memory locations are currently not used by software.

Figure 3 provides an overview of how OS uses the proposed architectural support. OS maintains a variable-sized continuous free memory address range within its free list. OS communicates this large continuous free memory range to hardware by piggybacking on the existing OS-controlled ACPI interface. CPU then leverages the OS-exposed free memory to record arbitrary data "for free" to help boost microarchitecture performance.

The rest of this section describes the new architectural support and how OS uses it. Section 4 describes a detailed FMT it enables.

3.1 Architectural Support

ACPI defines for each processor several hardware registers for OS to write/set the processor's power states [4]. Similarly, we propose adding to each processor a hardware memory control register for OS to record a free continuous memory address range within the processor's physical memory address range. This register records the upper and lower addresses of the free continuous physical memory range. FMTs will be allowed to autonomously write arbitrary data in arbitrary locations within the free continuous physical memory range recorded in the register. We refer to this register as the CPU-visible Free Memory Register (*CVFMR*) register and refer to the physical memory region it records as the *CPU-visible free page*. OS can expand or shrink the CPU-visible free page by updating CVFMR via ACPI at runtime. A node's CPU-visible free page can expand up to 100s to 1000s of gigabytes (e.g., up to almost the entire memory system) when software-level memory usage is low.

To expand the CPU-visible free page, OS calls ACPI to write a smaller lower address and/or a greater upper address into CVFMR. This ACPI call can complete quickly because it simply sets the value of CVFMR. Writing to hardware ACPI registers is fast as they are used to manage CPU power modes, which can be updated within tens of microseconds [56]. Afterwards, CPU asynchronously initializes the pages added to the CPU-visible free page without interrupting any running programs by using spare bandwidth; initialization values depend on the FMT(s) in use. By initializing linearly, CPU can track the yet-uninitialized region via one register; this allows FMTs to continue to access initialized free memory in parallel. We note last-level cache (LLC) may evict dirty blocks with physical addresses that fall within the CPU-visible free page because a program can write to a page soon before freeing the page. To handle write requests LLC sends to MC for such dirty evictions, MC simply drops all write requests to addresses within CVFMR as free physical pages only store dead/freed virtual memory objects.

To shrink the CPU-visible free page (e.g., to allocate some of its physical memory to a requesting process), OS calls ACPI to write a greater lower address and/or a smaller upper address into CVFMR.
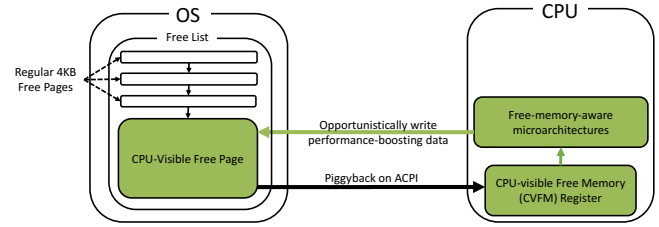

Figure 3: Overview of the proposed architectural support for OS to expose free memory to hardware.

This ACPI call can also complete quickly because it again simply sets the value of CVFMR. Afterwards, FMTs cease writing data to physical addresses outside of the updated address range in CVFMR. The contents of pages taken away from the CPU-visible free page do not need to be preserved because these pages hold opportunistically recorded data that did not exist in the first place without the CPU-visible free page. OS zeros out these physical pages before allocating them to programs, just as existing OS also zeros out physical pages before allocating them to enforce inter-process memory protection.

3.2 How OS Uses the Architectural Support

OS tracks the CPU-visible free page in its free list (see Figure 3). At runtime, OS expands the CPU-visible free page periodically; periodic expansion enables all nodes in an HPC system to expand their CPU-visible free pages at the same time to minimize OS jitter, which is a concern for HPC systems. We propose expanding the CPU-visible free page once an hour. At the end of each hour, OS first compacts[1] all free physical pages outside of the CPU-visible free page to one or both of its ends and then calls ACPI to expand the page by writing the widened continuous free memory range to CVFMR. For our measured systems with $128GB$/node, expanding the CPU-visible free page once an hour also limits the maximum amount of data migrated per node per hour to only $128GB$.[2] We pessimistically estimate compacting $128GB$ of free memory in an hour takes $128GB/(14GB/s) = 9$ seconds, where $14GB/s$ is the worse-case memory compaction throughput we observed via real-system experiments (see detail in Section 6.3)); this translates to a worst-case overhead of $9s/1hr = 0.3\%$. Common-case overheads are much lower (see evaluation in Section 6.3).

OS may shrink the CPU-visible free page for memory allocation requests. OS first uses other free pages in the free list to satisfy memory allocation requests. When the free list runs out of other free pages, OS decreases the size of the CPU-visible free page to use physical memory taken away from the CPU-visible free page to satisfy the memory allocation request. OS shrinks the CPU-visible free page by calling ACPI to write a smaller address range into CVFMR. OS takes away pages from the edges of the CPU-visible free page to maintain its contiguity. We note that after OS runs out of regular free pages in the free list, calling ACPI to shrink the CPU-visible free page for every page allocation is expensive. OS can effectively handle this overhead by reducing the size of the

---

[1]Memory compaction [21] is an existing OS feature to create huge (e.g., 2MB or 1GB) memory pages to improve TLB hit rate.
[2]Each program page is migrated once (e.g., shifted/dropped down once to compact it into lower physical addresses) regardless of whether it is a regular page or huge page; as such, the maximum data movement is 128GB.

CPU-visible free page by 256MB at a time and, therefore, call ACPI only once per $256MB/4KB = 65536$ page allocations. OS tracks the deducted 256MB as regular pages in the free list to quickly satisfy future allocation requests.

3.3 Discussion

For a node with multiple CPU sockets, the node's OS maintains a CPU-visible free page for each CPU. Each CPU is typically assigned its own contiguous physical memory address range [68]; the address range of a CPU's CPU-visible free page falls within the CPU's physical memory address range.

Another challenge with migrating pages to create a large continuous CPU-visible free page is that some virtual pages in the kernel are unmovable after boot up. Today's systems often map these pages to the lower physical addresses (e.g., within the first 4GB) [43]. Some X86 systems also reserve a physical address range below 4GB for memory-mapped I/O (MMIO) for backward compatibility with legacy 32-bit systems. To address this issue, OS may set each CPU's CPU-visible free page's upper address at a very high physical address (e.g., the CPU's maximum physical address) and grow the CPU-visible free page downward by decreasing its lower address.

4 FREE-MEMORY-AWARE MICROARCHITECTURE

Due to the fundamental tradeoff between computation time and space, exposing free memory to hardware can enable many new microarchitecture techniques to boost performance. This section describes in detail one such new microarchitecture – Free-memory-aware Memory Replication (FMR). We observe DRAM access latency is heavily dependent on the state of the DRAM location at the time of LLC miss. For example, an LLC miss only incurs DRAM refresh latency if the memory location storing the requested block is currently under refresh. Using free memory locations to replicate the same memory bock across two different DRAM locations can hide state-dependent latencies by allowing CPU to read from the location with the faster state at the time of LLC miss. The rest of this section is organized as follows. Section 4.1 provides the background on the different state-dependent latencies in DRAM. Section 4.2 describes how to hide state-dependent latencies for read requests. Section 4.3 describes how to efficiently write to memory under FMR. Section 4.4 describes memory layout details.

4.1 Background: State-dependent Latencies

Refresh Latency (tRFC). DRAM requires periodic refresh because the charge stored in DRAM cells leaks over time. DDRx memory chips, which are used in HPC systems, are refreshed on a per-rank basis; a rank is a group of memory chips that are always accessed in lockstep. A rank cannot be accessed when it is refreshing.

Bus Turnaround Delay (tWTR/tRTW). After writing to a rank, CPU must reconfigure the rank's I/O circuitry back to read mode before it can read from the rank [74]; this is known as bus turnaround. Similarly, after reading a rank, CPU must reconfigure the rank to write mode before writing it. To prevent frequent stalls due to frequent bus turnaround, modern systems typically write in large batches [17, 19, 26]; for example, in our microarchitecture parameter design space exploration in Section 5, we find

a write batch size of ~100 maximizes the average performance for the baseline memory system. Note that in addition to minimizing bus turnaround, large write batch size also helps the scheduler improve write requests' row hit rate (i.e., how often a request accesses an already opened row). Unfortunately, writing in large batches requires stalling read requests for a long time.

Row-to-row Delay (tRRD) & Four-activation Window (tFAW). CPU can only activate (i.e., open a new DRAM row in) a bank in a rank after tRRD has passed since the last time it had activated a bank in the same rank. Similarly, CPU can only activate at most four banks in a rank within tFAW. tRRD and tFAW help to meet memory chip-level power constraint.

Row-to-column Delay (tRCD) & Precharge Delay (tRP). Prior to accessing a new DRAM row in a bank, CPU must first activate the DRAM row and wait row-to-column delay. Furthermore, before CPU activates a new DRAM row in a bank, the bank must be in the closed state; otherwise, CPU must first issue a precharge command and wait tRP to close the bank before activating a row in the bank.

4.2 Reading from Memory Under FMR

To hide state-dependent latencies, FMR stores a copy of a logical block in a free location in a different rank. This enables MC to fetch from the rank with the faster state at the time of LLC miss. To avoid inconsistency in the cache hierarchy, MC always inserts the block into the cache hierarchy using LLC misses' original physical addresses even when MC fetches from replicating locations.

Making Refresh Nonblocking for Read Requests. Existing systems typically refresh one rank at a time [12] in a memory channel, which is a group of one or more ranks that share the same I/O connections (called the memory bus) to the processor. When there are two copies of a memory block residing in two different ranks in the same channel, at most one of the two copies is inaccessible due to refresh at any given time. MC can *completely* hide all refresh latency for LLC misses by satisfying them using the copy residing in a rank that is not currently refreshing.

Making Large Write Batches Nonblocking for Read Requests. To prevent a batch of writes from blocking read requests, MC can write to only one rank in a channel at a time. When a rank is in write mode, MC can use another rank to satisfy *all* LLC misses as long as the latter has a copy of every block in the former.

Mitigating tRRD and tFAW. Having two copies of the same logical block in two different ranks gives MC the freedom to fetch from the rank where tRRD and tFAW constraints are already met to satisfy LLC misses sooner.

Mitigating tRCD and tRP. Having a copy of a logical block in a second rank and, therefore, bank can mitigate precharge-induced read stalls by allowing MC to read from a second bank that is currently closed instead of reading from an original bank that is currently open. When a pair of banks have identical content due to memory replication, MC can improve row hit rate and, thereby, mitigate activation-induced read stalls in two scenarios. First, if one of the banks in the pair is closed and the other is open, MC can purposefully cease to speculatively[3] close the open bank without suffering from increased row conflict rate (i.e., how often LLC misses

---

[3]Some row buffer policies, such as the closed page policy or timeout policy, predict whether the currently opened row in a bank is dead (i.e., it will not be accessed in the near future); if a row is predicted to be dead, MC speculatively closes the row early
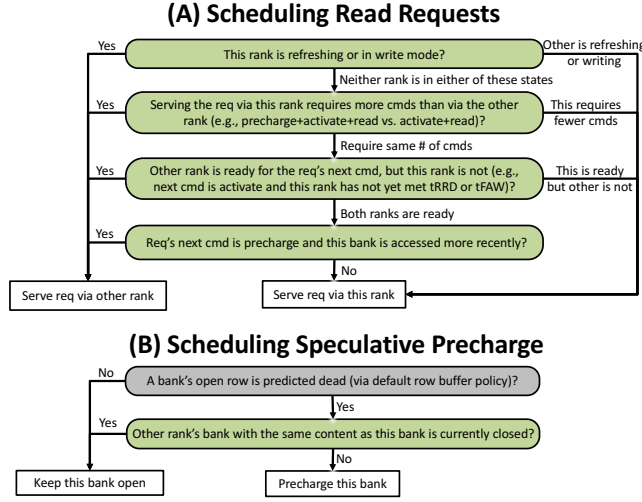
(A) Scheduling Read Requests

(B) Scheduling Speculative Precharge

Figure 4: FMR provides new scheduling choices (shown in green) for: (A) read requests and (B) row buffer policy.

access banks currently opened to wrong rows) because a future LLC miss requiring a new row can be served by the closed bank; minimizing how often open banks are closed speculatively strictly improves row hit rate. Second, if both banks in the pair currently have the wrong row open at the time of LLC miss, MC can close the less recently accessed bank; keeping more recently accessed row/bank open longer strictly improves row hit rate.

Figure 4 summarizes the new scheduling choices that FMR provides for read requests and speculative precharge operations (a.k.a, the page/row buffer policy).

4.3 Writing to Memory Under FMR

To hide refresh latency for write requests, we note that write requests are not on the critical path of program execution; as such, stalling write requests only slows down performance when the channel's write buffer is full, which can cause CPU to stall. To prevent CPU from stalling due to write requests to refreshing ranks clogging up the write buffer, we add a writeback cache to each channel between LLC and the channel's write buffer to cache write requests to the refreshing rank, similar to [62]. The writeback cache is only used as a storage buffer for write requests, unlike regular write buffer, which is also used by the memory scheduler to scan for writes to the same row to increase row hit rate. Write requests are always first placed into the writeback cache and then later drained to the write buffer. The writeback cache only drains a write request to the write buffer if the rank the request will go to is not currently refreshing. Because refresh can take a long time (e.g., 550ns for 16Gb DDR4 memory chips [41]), we organize the writeback cache as a large 16KB 64-way[4] set-associative writeback cache.

We also rely on the writeback cache to write to only one rank at a time to hide read stalls due to batching writes (see Section 4.2).

---

to speed up future accesses to different rows in the bank due to future LLC misses. Misprediction reduces row hit rate, however.

[4]Writeback cache has high associativity because write buffers are also highly associative (e.g., fully associative). The writeback cache is only accessed when accessing memory; its energy per access, as obtained from Cacti [35], is < 1% the energy per memory access as calculated from MICRON DDR4 datasheet [58].
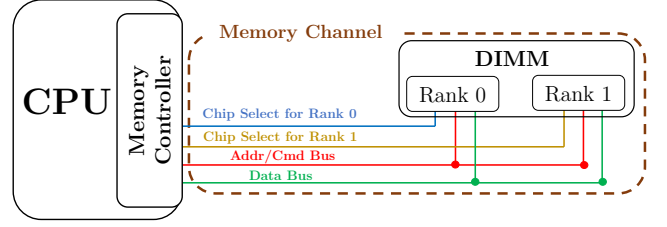

Figure 5: Organization of a memory channel today. Each rank has a dedicated chip select (CS) bit [57, 59, 61].

Writeback cache starts draining writes to the write buffer if one of the sets exceeds a high watermark of 75%. Writeback cache selects the rank with the highest occupancy in this set and then drains to the write buffer write requests belonging to the selected rank by visiting all sets in a round robin manner and draining one write when visiting the set. Writeback cache stops draining writes if it no longer has any write to the selected rank or the write buffer is full.

For write requests to logical memory blocks with a replica, MC must update both copies of the block to ensure consistency. To track which copy still needs to be written, we add two bits to each entry in the writeback cache to record which copy or copies still need to be updated/written to. The writeback buffer removes a write request only after both of its bits are currently false.

Finally, we note writing memory twice for each write request doubles memory bus utilization for writes; this can incur high performance overheads for write-intensive applications. To address the bandwidth overhead to write to both ranks, we observe that multiple ranks in a channel are connected to a shared bus and that the bus interconnection topology in general benefits from the unique message broadcasting capability that has long been exploited in on-chip networks to broadcast/multicast coherence messages. We propose exploiting the multicasting capability of the memory bus to simultaneously write/update both copies of the same logical block in a single memory bus transaction and, thereby, avoid bandwidth overheads for replication. Note that even in current server systems, CPU physically broadcasts every message to all ranks simultaneously over the shared bus and logically directs a message to the intended rank by asserting the intended rank's dedicated chip-select (CS) bit (see Figure 5); unintended ranks ignore all messages on the bus because their CS bits are deasserted. As such, to enhance CPU to multicast the same message to two ranks in a channel simply requires it to assert both ranks' CS bits simultaneously when transmitting the message. We confirmed with MICRON's chief technologist [65, 66] that while commodity memory buses are currently designed to write to one rank in a channel at a time, reusing them as is to write to two ranks at a time incurs minimal to no impact on bus signal integrity. Section 4.5 explains in detail.

To exploit memory bus multicasting to simultaneously update both copies of the same logical block in one bus transaction, MC must map both copies to identical DRAM locations (i.e., with same bank ID, row ID, and column ID) across two ranks in the same channel. Otherwise, CPU must issue two separate write commands, each communicating a different DRAM location over the bus; since the two write commands are spread out in time, the subsequent write data must also be spread out in time, resulting in two completely

separate writes. Furthermore, mapping both copies to identical DRAM locations, which includes identical bank ID, also creates pairs of banks with identical content, which are required by our optimization to mitigate tRCD and tRP in Section 4.2. We will describe how to map the original and its replica to identical DRAM locations across two ranks in the same channel in Section 4.4. We also note that before multicasting writes to two ranks, MC must first synchronize them by issuing a *precharge_all* command; this synchronization overhead is small, however, because it is amortized over the many writes in a batch.

Unfortunately, at the same time, MC cannot both multicast writes to eliminate write bandwidth overhead and write to only one rank at time to make batched writes nonblocking for reads. We note the benefit of reducing bandwidth overhead is only higher than the benefit of making writes nonblocking when bandwidth utilization is high. As such, we propose dynamically switching between the two write modes according to the instantaneous bandwidth utilization. MC switches from nonblocking writes to multicast writes when the writeback cache becomes so full that it cannot hold an incoming write request from LLC. MC switches from multicast writes back to nonblocking writes if the writeback cache can always hold incoming write requests for a continuous period of $100\mu s$.

4.4 Memory Layout Details

Because a CPU's CPU-visible free page is located in the CPU's higher physical address range (see Section 3.3), we assign physical address $O(p) = p + \frac{S}{2}$ (i.e., when it is free) to replicate the logical block stored at physical address p; S stands for a CPU's installed physical memory size. This simple and fast assignment function $O(p)$ works for both single-socket and multiple-socket systems. A CPU's MC can quickly tell whether a block at p is replicated by checking whether $O(p)$ is within the address range recorded in the CPU's CVFM register; each MC keeps a local copy of the CVFM register value for fast lookup. After OS calls ACPI to expand the CPU-visible free page, MC must initialize the physical memory newly added to the CPU-visible free page; MC copies the value at p to $O(p)$ for every uninitialized $O(p)$ in the CPU-visible free page. MC can perform this asynchronously in the background whenever there is slack in memory bandwidth utilization.

To reap the performance benefits described in Section 4.2, MC must map p and $O(p)$ to different ranks. Furthermore, as discussed in Section 4.3, multicasting writes requires MC to map p and $O(p)$ to identical DRAM locations across two different ranks in the same channel. Finally, we note that simultaneously writing to two ranks in the same memory module may exceed the power/thermal budget for corner-case power-limited memory modules (e.g., DIMMs with many ranks). As such, MC should preferably map p and $O(p)$ to two ranks that differ by $N/2$ in their rank IDs, where N is the number of ranks per channel, to map them to different DIMMs to preserve each DIMM's original power and thermal profile when there are multiple DIMMs per channel.[5]

---

[5] In systems with only one DIMM per channel, multicasting writes still works for many DIMM configurations because the number of power pins in a JEDEC-compliant DIMM socket is designed for worst-case DIMM configuration with maximum current draw. For example, 2-rank R-DIMMs and 8-rank R-DIMMs have identical DIMM-level pin count, even though the two R-DIMMs differ by 5X in their peak current draw (see [61] and [60]). We also confirmed that simultaneously writing to two ranks in the same commodity DIMM is feasible through MICRON's Chief Technologist [66].

Before describing our proposed mapping to satisfy the above requirements, we first define address mapping terminologies. In current systems, MC translates the physical address p to its DRAM location d using a simple static direct-mapped (i.e., one-to-one) function $d = f_{P \to D}(p)$, where P is the set of all physical addresses in a system and D is the set of all DRAM addresses. $f_{P \to D}(p)$ is often a collection of functions (i.e., $f_{P \to channelID}(p)$, $f_{P \to rankID}(p)$, $f_{P \to}(bankID)$, $f_{P \to columnID}(p)$, and $f_{P \to rowID}(p)$) that compute the memory channel ID, rank ID, bank ID, row ID, and column ID, which collectively make up a DRAM address. For brevity, we refer to all functions under $f_{P \to D}(p)$ other than $f_{P \to rankID}(p)$ collectively as $f_{P \to OtherIDs}(p)$.

We propose a general transformation to derive from an arbitrary $f_{P \to D}(p)$ a $f'_{P \to D}(p)$ such that $f'_{P \to rankID}(p)$ differs from $f'_{P \to rankID}(O(p))$ by $N/2$ and $f'_{P \to OtherIDs}(p) = f'_{P \to OtherIDs}(O(p))$. Our proposed transformation only has two restrictions. First, N (i.e., the number of ranks per channel) is even. Second, the original $f_{P \to rankID}(p)$ is periodic; a good example is the well-known $f_{P \to rankID}(p) = (p/L) \bmod (N)$ function, which interleaves across different ranks adjacent memory segments of size L, where L can be any multiple of 64B.

For clarity, we will describe $f'_{P \to D}(p)$ for single-socket systems. One can easily adapt $f'_{P \to D}(p)$ designed for single-socket systems to multi-socket systems as CPUs can preserve the appearance of single-socket systems for the purpose of calculating physical to DRAM address mapping within the socket; this adaptation requires MC to subtract all physical addresses by Min_{socket} before giving them as input to $f_{P \to D}(p)$, where Min_{socket} is lowest physical memory address assigned to the socket.

To derive $f'_{P \to rankID}(p)$, we note that one can visualize the periodic sequences $\{f_{P \to rankID}(0L), f_{P \to rankID}(1L),..., f_{P \to rankID}(((\frac{S}{2} - 1)/L) \cdot L)\}$ and $\{f_{P \to rankID}(O(0L)) = f_{P \to rankID}(\frac{S}{2} + 0L), f_{P \to rankID}(O(1L)) = f_{P \to rankID}(\frac{S}{2} + 1L),..., f_{P \to rankID}(O(((\frac{S}{2}-1)/L)\cdot L)) = f_{P \to rankID}(((S-1)/L)\cdot L)\}$ as the motions of two circular clocks starting with different initial clock positions. As such, to derive $f'_{P \to rankID}(p)$ such that $f'_{P \to rankID}(p)$ and $f'_{P \to rankID}(O(p))$ always differ by $N/2$, one simply needs to apply a constant offset to initialize the starting position of the second periodic/clock sequence to differ by $N/2$ compared to the starting position of the first periodic/clock sequence. Concisely,

$$f'_{P \to rankID}(p) = \begin{cases} f_{P \to rankID}(p), & \text{if } p < \frac{S}{2} \\ f_{P \to rankID}(p + C), & \text{if } p >= \frac{S}{2} \end{cases}$$

The constant C is different for different original $f_{P \to rankID}(p)$ functions; for the example of $f_{P \to rankID}(p) = (p/L)\bmod(N)$, $C = L \cdot \lceil N/2 \rceil - (\frac{S}{2}/L)\bmod(N)$.

After transforming $f_{P \to rankID}(p)$ to $f'_{P \to rankID}(p)$, one can obtain $f'_{P \to otherIDs}(p)$ as follows:
$f'_{P \to otherIDs}(p) =$

$$\begin{cases} f_{P \to otherIDs}(p \bmod(S)), & \text{if } f'_{P \to rankID}(p) < \frac{N}{2} \\ f_{P \to otherIDs}(O(p)\bmod(S)), & \text{otherwise} \end{cases}$$

Proof. Proving $f'_{P \to otherIDs}(p) = f'_{P \to otherIDs}(O(p))$ requires proving the equality holds for both any p_1 such that $f'_{P \to rankID}(p_1)$
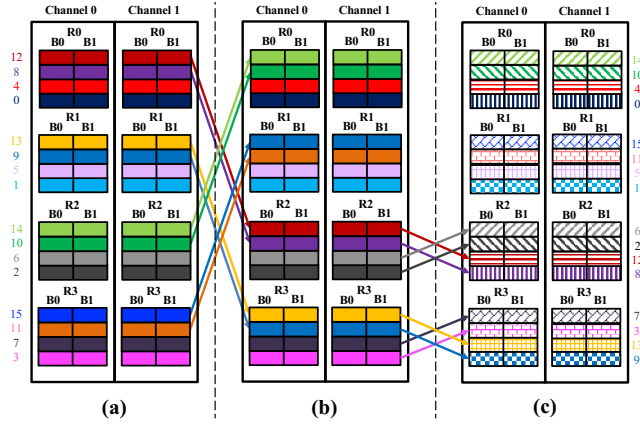
Cores	16 cores, 2.8 Ghz, 4-wide OoO, 2K TLB entries, 192-entry ROB
L1\$	Split Data-Instruction, 64 kB, 4-way assoc, 2-cycle latency
L2\$	256 kB per core, 16-way assoc, 12-cycle latency
L3\$	32 MB shared, 16-way assoc, 16ns (on top of L1&L2 Latency), degree-4 stride prefetcher, prefetch on both hit and miss
MC	8ns round trip latency between MC and LLC, DDR4-3200, 4 channels, 2 ranks/channel, 16-banks/rank, XOR-based address mapping for banks, 256-entry read queue/chan, 128-entry write buffer/chan, 0.5KB interleaving across banks & channels, FR-FCFS memory scheduling policy, Timeout policy: close a row after 200 DRAM cycles (i.e., 125ns)

Table 2: Evaluated Baseline.

Figure 6: (a) Original physical-to-DRAM address mapping (i.e., $f_{P \rightarrow D}(p)$). B is Bank. R is Rank. 16 different colors refer to 16 different memory segments. (b) Physical-to-DRAM address mapping after applying the rank ID transformation (i.e., $f'_{P \rightarrow rankID}(p)$). (c) Physical-to-DRAM address mapping after applying the transformation for other DRAM location IDs (i.e., $f'_{P \rightarrow otherIDs}(p)$). Segments with the same hash patterns are p and $O(p)$ pairs; they now differ by $\frac{N}{2} = \frac{4}{2} = 2$ in rank ID and have identical row, bank, and channel IDs.

$< \frac{N}{2}$ and for any p_2 such that $f'_{P \rightarrow rankID}(p_2) >= \frac{N}{2}$. Given any p_1, $f'_{P \rightarrow otherIDs}(p_1) = f_{P \rightarrow otherIDs}(p_1 mod(S)) = f_{P \rightarrow otherIDs}(p_1)$. Because $f'_{P \rightarrow rankID}(p)$ and $f'_{P \rightarrow rankID}(O(p))$ always differ by N/2 for any p, $f'_{P \rightarrow rankID}(O(p_1)) >= \frac{N}{2}$; as such, $f'_{P \rightarrow otherIDs}(O(p_1)) = f_{P \rightarrow otherIDs}(O(O(p_1))mod(S)) = f_{P \rightarrow otherIDs}((p_1 + \frac{S}{2} + \frac{S}{2})mod(S)) = f_{P \rightarrow otherIDs}(p_1)$. Therefore, $f'_{P \rightarrow otherIDs}(p_1) = f'_{P \rightarrow otherIDs}(O(p_1))$. Similarly, given any p_2, $f'_{P \rightarrow otherIDs}(p_2) = f_{P \rightarrow otherIDs}(O(p_2)mod(S))$. Again because $f'_{P \rightarrow rankID}(p)$ and $f'_{P \rightarrow rankID}(O(p))$ always differ by N/2 for any p, $f'_{P \rightarrow rankID}(O(p_2)) < \frac{N}{2}$; as such, $f'_{P \rightarrow otherIDs}(O(p_2)) = f_{P \rightarrow otherIDs}(O(p_2)mod(S))$. Therefore, $f'_{P \rightarrow otherIDs}(p_2) = f'_{P \rightarrow otherIDs}(O(p_2))$.

Figure 6 graphically illustrates the transformed $f'_{P \rightarrow D}(p)$ for a simple $f_{P \rightarrow D}(p)$ mapping scheme that interleaves adjacent addresses first across column, then channels, banks, ranks, and finally rows. To clarify the figure, the proposed address mapping function is static (i.e., a CPU with FMR always uses $f'_{P \rightarrow D}(p)$ and never switches back to $f_{P \rightarrow D}(p)$).

4.5 Discussion on Memory Bus Signal Integrity

Simultaneously writing to two ranks in a channel has no impact on the signal integrity of the memory command bus. Current systems already broadcast every command over the command bus to all DRAM chips in all ranks in a memory channel [6]; unintended ranks know not to act upon the broadcasted command because their chip select bits are deasserted.

Simultaneously writing to two ranks in a channel also has no or little impact on signals over the data bus. First, due to modular chip design, the external data bus is insulated from what the second rank's DRAM chip's internal digital logic sitting behind its IO receiver decides to do with the received bus signal (e.g., ignore it

in the case of conventional single-rank write or write it to DRAM array in the case of multicasting write). Second, in existing systems, all ranks in a channel are allowed to keep their data pins' IO receivers active at the same time, as power gating all or part of an IO receiver whenever a rank is idle can incur costly latency and energy overheads for some workloads. Third, while changing a rank's data pins' on-die termination (ODT) impedance values can affect signals over the bus, writing to a rank does not require changing its ODT values except when using Dynamic ODT, which requires setting a rank's ODT to a special $RttWr$ value (see Table 70 in [58]). This exception can be handled by tuning $RttWr$ [66], which is reconfigurable [41]. Note that it is not uncommon to not use Dynamic ODT, which is designed for "certain applications cases" [41, 58]; for example, it is disabled in most of the memory configurations used in a recent memory study on AMD Ryzen systems [15].

5 METHODOLOGY

We simulate a 16-core CPU using Gem5 [13] in full system mode. LLC latency is set to 20ns, which is reported by a recent real-system study [32] for servers. TLB entry count is set to 2000; this is similar to the 1.5K TLB entry count in Skylake processors [78]. We simulate a DDR4 memory system by incorporating Ramulator [47] into Gem5; we use DRAM timing and current parameters from MICRON DDR4 datasheet [58] and JEDEC's 550ns tRFC [41], which represents the refresh latency of latest-generation and future DRAM chips. We simulate four channels and two ranks per channel to match the memory configuration of our measured HPC systems. We interleave 512B of adjacent physical addresses across different channels, ranks, and then banks and use XOR-based mapping for channels and banks. These choices model after real-system studies of modern processors [69]. Table 2 summarizes the simulated microarchitecture parameters.

We evaluate five HPC benchmark suites – NASA Parallel Benchmarks (NPB) [7], Graph500 [1], HPCG [25], Linpack [3], and GAP [11]. We evaluate all benchmarks under the five suites, except for NPB's EP, which is tiny; for GAP, we evaluated all but one input set - the RANDOM input set. We note that because many HPC benchmarks run for several hours in a real system, identifying representative simulation points is important. For each benchmark, we use Gem5's KVM CPU, which runs at native execution speed, to take three checkpoints at ~1/4, ~1/2, and ~3/4 of the benchmark's total program execution; subsequently, we simulate from the three
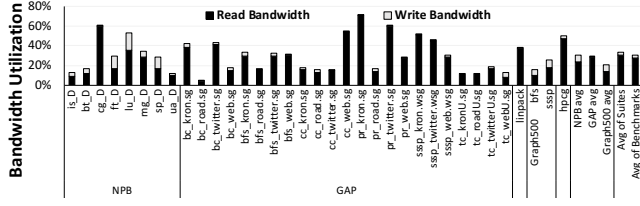
Figure 7: Workload characterization: bandwidth utilization breakdown between reads (including prefetch) and writes under the primary baseline.
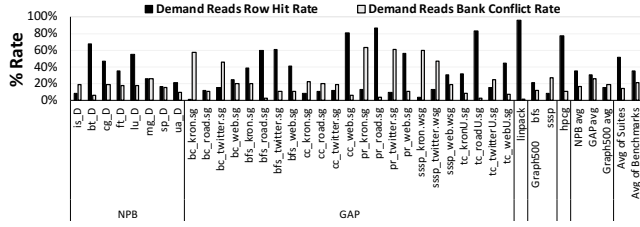

Figure 8: Workload characterization: row hit rate and bank conflict rate of demand read requests under the baseline.

checkpoints to measure the benchmark's memory bandwidth utilization[6] at these three different points in program execution. We then pick the representative simulation point as the checkpoint[7] that exhibits the median bandwidth utilization across all three checkpoints. For each cycle-accurate simulation, we first warm up the cache via 10ms of atomic simulation, then warm up the branch predictor and prefetcher via 10ms of cycle-accurate simulation,[8] and report the performance observed during the next 10ms of cycle-accurate simulation. Because all benchmarks are parallel (i.e., 16 threads or 16 MPI processes), we use FLOPs (floating operations per second) to measure performance for floating point benchmarks and use committed store instructions for the rest. Figure 7 characterizes the memory bandwidth utilization of each benchmark at its chosen checkpoint under the baseline memory system. Finally, we model CPU power via McPAT [51] and use its 22nm power model; we scaled down the output linearly to model 14nm. We modeled memory power via DRAMPower [16] by setting 18 chips/rank.

We optimize our primary baseline by using cheaper 4-core simulations to perform multi-dimensional space exploration to determine the best prefetch degree, best page policy (e.g., among open policy, closed policy, and timeout policies of different thresholds), and best write buffer size (and thus write batch size) that yield the maximum average baseline performance for the evaluated benchmarks. We also implement in the baseline the optimization in [17] that switches a channel to write mode when all pending read requests in the channel are blocked due to refresh. Finally, since FMR

requires an additional 16KB writeback cache per channel, we also add a 16KB writeback cache to the baseline for fair comparison; this improves baseline's average performance by 0.5%.

We optimistically model a prior work – Nonblocking Memory Refresh [62, 63] - by setting refresh latency to 0; all other settings are identical with the primary baseline.

We model a second prior work, Duplicon Cache [54], which statically reserves 64MB of off-chip DRAM for each channel to replicate data in DRAM. Because the simulated CPU has four channels, we model a 256MB Duplicon Cache.

To evaluate FMR, we assume the CPU-visible free page can fully replicate all in-use memory; Section 6.3 discusses the validity of this assumption. FMR uses the same setting as the primary baseline (e.g., same prefetcher setting, page timeout setting, write buffer size, etc.) wherever possible.

6   RESULTS
Figure 9 shows the performance of FMR normalized to the primary baseline. FMR improves performance by 13.7% when weighing each of the five benchmark suite equally. When weighing each of the 35 benchmarks equally, FMR improves performance by 12.2%. On average across both types of averages, the average benefit is 13%. Benchmarks that benefit the most from FMR are *hpcg*, *cg_D*, *bc_kron.sg*, *bfs_kron.sg*; these benchmarks are characterized by high memory bandwidth utilization (see Figure 7) and low spatial locality (see Figure 8). Benchmarks that benefit the least from FMR are *linpack*, *bfs_road.sg*, *bc_road.sg*, *tc_kronU.sg*, *tc_webU.sg*; these benchmarks are characterized by low memory bandwidth utilization (see Figure 7) or very high spatial locality (see Figure 8).

Figure 9 also shows the performance of Nonblocking Memory Refresh normalized to the primary baseline; compared to Nonblocking Memory Refresh, FMR provides an additional 6.3% and 4% average performance improvement over the primary baseline when averaging equally across benchmark suites and averaging equally across benchmarks, respectively. FMR also provides another important benefit over Nonblocking Memory Refresh - FMR can be deployed on commodity memory systems with commodity memory chips and modules, whereas Nonblocking Memory Refresh modifies processor-memory interface/protocol [62].

To evaluate the Duplicon Cache baseline, we warmed up a 256MB Duplicon Cache for 200 simulated milliseconds in Gem5's atomic CPU mode to eliminate cold misses. Figure 10 shows the demand
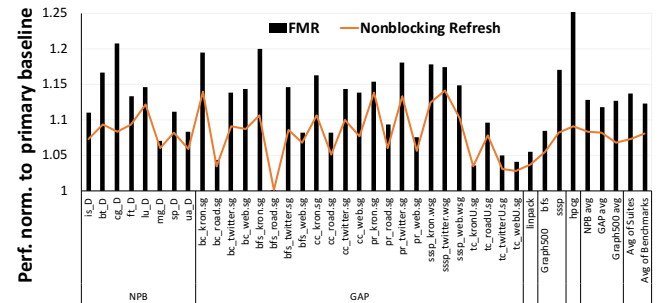
---

[6]Our evaluation shows that FMR's performance benefit depends significantly on bandwidth utilization; in general, FMR provides more performance benefit when bandwidth utilization is higher.

[7]We did not use Simpoint [67] because it is designed for serial benchmarks. If you want to use these checkpoints either to verify this research or for your own research, contact us and we will share the checkpoints.

[8]We ensure all schemes execute the same number of instructions during cycle-accurate warmup by making DRAM latency constant during this period.

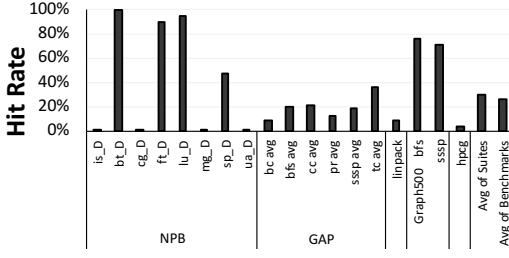**Figure 9: Performance normalized to the primary baseline.**

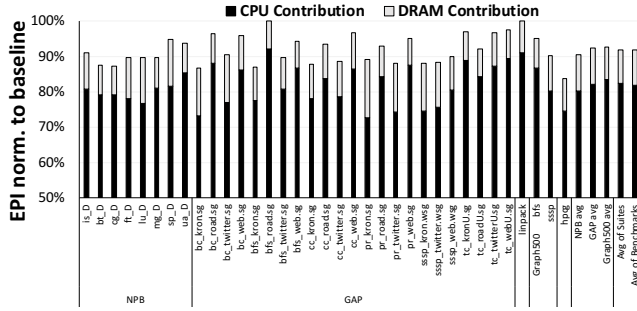**Figure 10: Duplicon Cache demand read hit rate.**

Figure 11: FMR's Energy per instruction (EPI) normalized to baseline. Each stacked bar shows the total EPI. The two segments in each stacked bar show CPU's and DRAM's contribution to the total EPI.
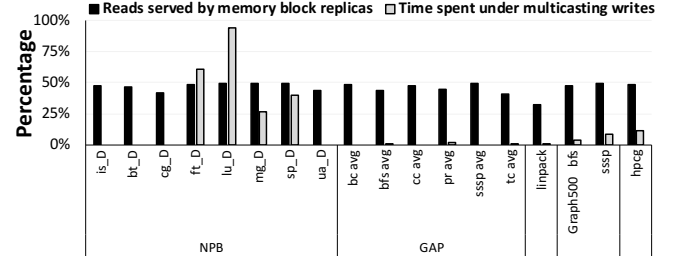

Figure 12: (A) Fraction of read requests that are satisfied using a memory block copy in a free memory location. (B) Fraction of time spent under multicasting writes mode.
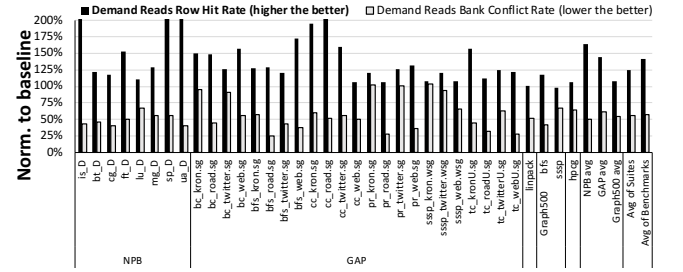

Figure 13: Row hit rate and bank conflict rate of demand reads in FMR normalized to the primary baseline.

read hit rate of Duplicon Cache during 20ms of simulated time after warmup; it is only 26% when averaging across benchmarks equally and 29% when averaging across suites equally. The read hit rate is low because the average memory footprint of our HPC benchmarks is 20GB; in comparison, the average footprint of the benchmarks the Duplicon Cache paper evaluates is only ~2GB as they are mostly desktop workloads. While making Duplicon Cache bigger can improve its hit rate, this is expensive because (i) memory Duplicon Cache uses are always hidden from and thus unusable by OS and (ii) Duplicon Cache tracks replicated blocks in memory via a large on-chip SRAM tag that increases linearly with Duplicon Cache size. As such, Duplicon Cache can only provide 26% to 29% the performance benefit of FMR in the *ideal* case; the actual benefit should be much lower for several reasons. First, we find that nearly 100% of writebacks hit in Duplicon Cache because dirty blocks evicted from the 32MB LLC almost always hit in the much bigger 256MB Duplicon Cache; as such, writes to memory require nearly double bandwidth, as Duplicon Cache does not and cannot multicast writes. Second, Duplicon incurs an overhead memory write request even for read requests that miss in Duplicon Cache to insert the new block to Duplicon Cache. Third, Duplicon Cache, as described in its paper, only replicates blocks across banks in the same rank; as such, it does not mitigate rank-level latencies as does FMR.

We note that the 13% node-level speedup FMR provides may not translate directly to 13% speedup for distributed workloads running across multiple nodes. However, making each node 13% faster can allow a distributed workload to complete in same (or less, as fewer nodes means less network communication overheads) amount of

time while occupying 13% fewer nodes. Achieving same or higher workload-level performance while occupying 13% fewer nodes can improve workload-level energy efficiency by >=13%; this also frees up 13% of nodes to run other workloads and, therefore, help improve HPC-system-wide throughput by 13%.

In terms of system-level (i.e., CPU+DRAM) energy efficiency, FMR reduces energy per instruction (EPI) by 8% compared to the primary baseline. Although FMR increases DRAM write power (but not idle power) due to performing two DRAM writes for each memory write request, FMR still reduces system-level EPI because CPU power is much higher than DRAM power. In addition, CPU idle power dominates dynamic power according to McPat. As such, the reduction in CPU idle energy due to improving performance outweighs the energy overheads due to doubling writes.

6.1 Memory Behavior Analysis

Figure 12 shows the fraction of read requests that are satisfied using duplicate memory blocks under FMR. On average, 45.5% of all read requests are satisfied using duplicate memory blocks residing in free memory locations; this is to be expected, as each rank has ~50% chance of being faster than another rank for an incoming read request. Figure 12 also shows the fraction of time spent under multicasting writes (as opposed to nonblocking writes); it is high for benchmarks with high write bandwidth utilization, such as ft_D, lu_D, mg_D, and sp_D (See Figure 7).

Figure 13 shows the row hit rate and bank conflict rate of demand read requests (i.e., excluding prefetch requests) under FMR normalized to those of the primary baseline. On average across all
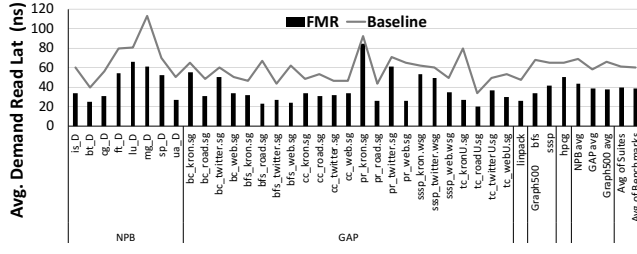
Figure 14: Average DRAM Demand Read Latency (ADDRL) of FMR and primary baseline.

Figure 15: Performance for normalized to the primary baseline (quad-rank configuration).

benchmarks, FMR increases row hit rate by 40% while reducing bank conflict rate by 43%.

Figure 14 shows the average DRAM latency of L3 demand misses under the primary baseline and under FMR; for simplicity, we refer to this average latency as Average DRAM Demand Read Latency or ADDRL. The primary baseline's average DRAM latency for L3 demand miss is 59ns. FMR reduces ADDRL by 30%. Note that adding the baseline's 59ns ADDRL to the L1,L2,L3 hit latencies and 8ns LLC-to-MC latency in Table2 gives 87ns per memory access; this closely matches the 92ns loaded memory latency a prior real-system study reports for servers [32]. We note that for some workloads, while FMR substantially improves performance, its ADDRL can be very similar to baseline's ADDRL. Consider for example *pr_kron.sg*; while FMR improves performance by 16%, ADDRL under FMR is 92% normalized to that of the baseline. In other word, FMR improves performance by a substantially greater amount compared to how much FMR reduces ADDRL. This seemingly counter-intuitive phenomenon likely occurs because improving an application's performance increases memory access rate and thus memory queuing delay, which in turn increases ADDRL. We believe FMR would show much more significant ADDRL reduction if FMR's performance/memory access rate were somehow artificially throttled to match that of the baseline.

6.2 Sensitivity Analysis

We evaluate the sensitivity of FMR with memory system configuration of 4 ranks per channel. Figure 15 shows the performance of FMR and Nonblocking Memory Refresh normalized to the primary baseline for the quad-rank configuration. FMR improves performance by 13.5% when weighing each of the five benchmark suite equally, and by 14.1% when weighing each of the 35 benchmarks equally. The average differences in the performance benefit that FMR provides under different memory system configurations – dual-rank and quad-rank are small; the average differences are 0.2% and 1.9%, when weighing benchmark suites equally and weighing benchmarks equally, respectively. As such, FMR provides robust benefits across different memory system configurations.

6.3 Real-system Emulation

Memory Compaction Overhead. Expanding the CPU-visible free page requires memory compaction, which can be expensive (See Section 3.3). To estimate this overhead, we measured the throughput of memory compaction on a real node. First, we created a simple microbenchmark to fragment free memory in the node; it
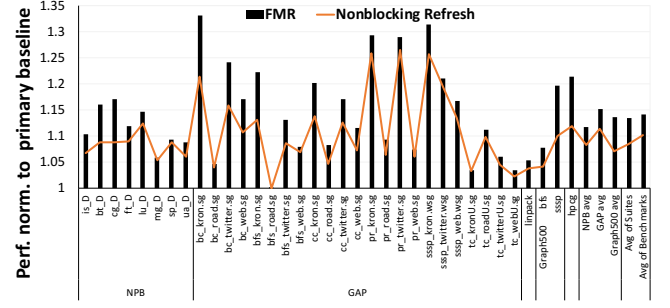
repeatedly calls mmap() at 4KB granularity to consume nearly all of the node's free memory, touches the requested pages, then frees a controlled fraction of random/non-contiguous requested pages, and finally goes into sleep. Next, we invoke Linux's built-in memory compaction routine via the "echo 1 > /proc/sys/vm/compact_memory" commandline [31] and examine the compaction latency reported in Linux's event tracing utility. Using compaction latencies measured for different controlled amount of fragmented free memory in the system, we observed a worst-case (i.e., slowest) memory compaction throughput of compacting 14GB of free memory per second. Memory compaction throughput steadily increases (by up to 2X) when there are more free memory.

CPU-Visible Free Page Expansion Policy. Using the measured worst-case memory compaction throughput and the memory usage measurements collected in our study in Section 2, we can emulate the performance overhead for the proposed hourly expansion. We estimate the amount of free memory to compact each hour in a node as the positive increase in the node's free memory at the end of an hour compared to its free memory at the beginning of the hour; we define a node's free memory in an hour as the node's minimum free memory during the hour. Figure 16 (A) shows the average amount of free memory to compact per node per hour for the studied HPC systems. We estimate the average performance overhead as the average amount of free memory to compact per hour divided by the 14GB/s compaction throughput divided by one hour. Figure 16 (B) shows the estimated average performance overhead due to page migration; they are small (e.g., <0.005%).

While infrequently expanding the CPU-visible page (i.e., once an hour) effectively minimizes the performance overhead due to page migration, it can reduce the size of the CPU-visible free page. Using the memory measurements collected in our study (see Section 2), we emulated the hourly expansion policy to calculate the fraction of jobs for which the CPU-visible free memory are always at least half the memory system size on all nodes the job occupy throughout the job's lifetime. Figure 16 (C) shows that FMR can fully replicate all in-use memory across all nodes occupied by a job for 87% to 90% of jobs, on average across Grizzly, Badger, and Snow.[9]

Checking Simulation Results in Silicon. To check our simulated performance improvement, we used a real system to emulate the performance impact of reducing ADDRL. One can reduce a real system's ADDRL in unit of CPU clock cycles by reducing CPU

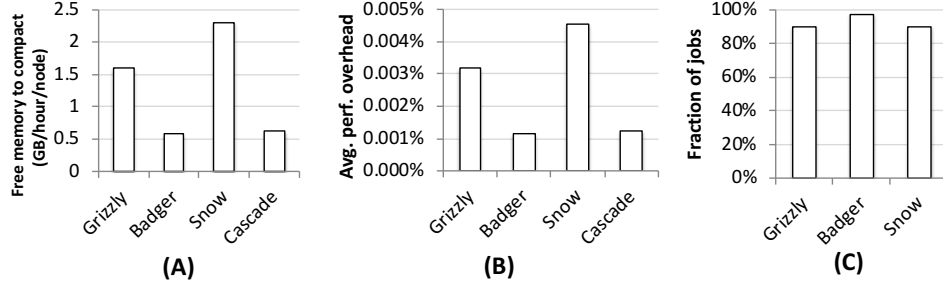
[9]Cascade did not record job IDs.

Figure 16: Emulation study of the proposed hourly expansion policy for the CPU-visible free page. (A) Average amount of free memory to compact per hour per node. (B) Average performance overhead. (C) Fraction of jobs in which CPU-visible free memory can fully replicate all in-use memory across all nodes a job occupies during its entire lifetime.
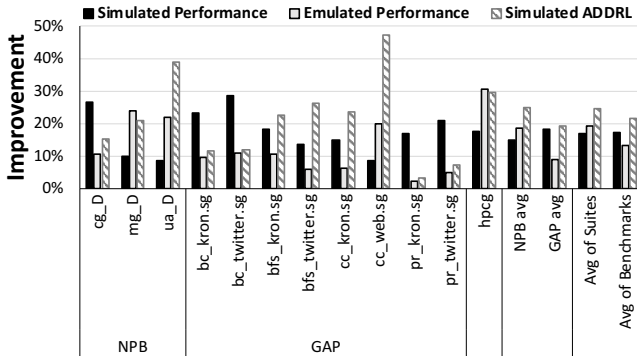

Figure 17: Comparison of simulated and emulated performance improvement for a sampling of benchmarks.

clock frequency. As such, it is possible to emulate the performance impact of reducing ADDRL on a real system by comparing the total execution time in unit of total CPU clock cycles; in other words, emulated speedup = (execution time in total CPU clock cycles when executing a benchmark in the real system under its normal CPU frequency) divided by (execution time in clock cycles when executing the benchmark under reduced CPU frequency for emulating ADDRL reduction). To emulate the performance benefit of FMR for a benchmark on a real system, we reduce the real system's CPU frequency by the same factor that FMR reduces ADDRL as reported by simulation for the benchmark. For the real-system emulation to be meaningful, the real system' cache latency must remain constant in unit of CPU clock cycles when CPU frequency changes; through extensive microbenchmark testing, we identified a 6-core Intel Core i7 PC[10] satisfying this criteria. We re-simulated a third of the benchmarks using six threads/processes per benchmark; these new simulations adopt the evaluated PC's microarchitecture parameters (e.g., the PC's 2666MHz DRAM frequency, the PC's 8Gb DRAM chip's 350ns tRFC, two channels, two ranks/channel, and 12ns L3 hit latency as measured by running the X-mem tool [32]). Figure 17 also shows FMR's ADDRL improvement (i.e., 1-FMR's ADDRL/baseline's ADDRL) according to simulation; when emulating FMR's performance benefit for an individual benchmark on the

---

[10]Unfortunately, we could not find a suitable server-class machine as changing a machine's CPU frequency requires root access to the machine and gaining root accesses to server clusters is difficult.

PC, we multiplied the PC's CPU frequency by (the benchmark's ADDRL under FMR / the benchmark's ADDRL under baseline).

Figure 17 shows the performance improvement reported by simulation and the performance improvement reported by real-system emulation (i.e., emulated speedup minus one). The average performance benefits are similar across both sets of experiments; they are within 5% of each other. However, for individual benchmarks, the performance improvements reported by simulation and emulation can vary significantly. *pr_kron.sg* exhibits the greatest mismatch: FMR improves performance by 17% according to simulation, but by only 2% according to emulation. Close inspection of *pr_kron.sg* reveals that while FMR's performance is 117% normalized to baseline, FMR's ADDRL is 96.7% of the baseline (see Figure 17); the discussion surrounding Figure 14 describes why this is possible. Ideally, the real-system emulation experiments should have used the ADDRL FMR can hypothetically achieve *had its performance/memory access rate been throttled to match that of the baseline.* This is difficult to measure, however; as such, our best-effort real-system emulation experiment underestimates the performance benefit for *pr_kron.sg*. For the same reason, the emulated performance improvement is lower than simulated performance improvement for *cg_D*, *bc_kron.sg*, *bc_twitter.sg*, and *pr_twitter.sg*.

7 GENERALITY

7.1 Other Potential FMTs

Beside hiding DRAM's dynamic latencies, making microarchitecture free-memory aware can enable other potential use cases to improve performance. One use case is to enable effective correlated prefetching [77] for very large applications. Correlated prefetching requires recording a large amount of past address sequences to predict future memory addresses. This memory overhead is particularly high for large applications with tens to hundreds of gigabytes of memory footprint. Making microarchitecture free-memory aware can enable multi-gigabyte correlation history tables to enable effective correlated prefetching for large-memory applications. Such large tables are impossible for existing correlated prefetchers, which all use static memory locations to store their metadata.

Another use case is to reduce page walk overhead. A TLB miss requires an expensive page walk that takes up to five memory accesses [37]. Naively reducing this overhead by adding more TLB

entries in the CPU incurs high static area overhead. Making microarchitecture free-memory aware can opportunistically enable large gigabyte-sized off-chip L3 TLBs that can potentially eliminate all page walks. While accessing the off-chip L3 TLB still incurs one overhead memory access, it is much faster than the up to five overhead memory accesses under a page walk.

Yet another use case is to improve hardware-based memoization. Recent studies show that hardware-based memoization can provide up to several times speedup. Hardware-based memoization can either be fully transparent [75] or provide new instructions (e.g., *memo_lookup* and *memo_update* [79]) to accelerate memoization [28, 79]. Because memoization records past computation results, the more space available, the more effective memoization can become. Making microarchitecture free-memory aware can enable very large and fast hardware-managed memoization tables to maximize the effectiveness of hardware-based memoization.

7.2 Applicability to Other Systems

Making microarchitecture free-memory aware may also benefit other systems such as cloud and PCs. Prior studies [20, 24, 42, 55, 73] report cloud also experiences memory underutilization; as such, FMTs can potentially also utilize the unused memory in cloud to improve performance. We also note that existing systems' physical memory size may be limited by the fact that the return of adding more memory can diminish quickly once there is already sufficient physical memory to hold common-case workloads' program memory and to cache frequently accessed files. By enabling CPUs to extract more performance gains from additional memory, FMTs may encourage cloud service providers to increase memory size per node. For the same reason, PC users may be willing to buy more memory to improve performance via FMTs.

8 RELATED WORK

Servers support memory mirroring across two memory channels to improve memory reliability [36]. Memory mirroring improves reliability at the cost of performance; for example, Xeon E7 processors are reported to suffer up to 50% slowdown under memory mirroring [18]. Also, memory mirroring is only enabled or disabled at boot-time [22], unlike FMR, which adjusts dynamically at runtime.

Existing OS on multi-socket systems replicate read-only kernel text across different CPUs' physical memory to hide the latency and bandwidth overheads of accessing kernel text from remote memory [14]. The free memory measured in our study already accounts for all memory usages due to to existing OS optimization. Also to reduce the overhead of remote memory accesses, a prior work purely in the OS domain [29] replicates mostly-read (e.g., 95% read) program memory pages across multiple sockets; this restriction is because every write to a replicated page incurs an expensive soft page fault. However, their evaluations show no benefit for NPB due to only replicating mostly-read pages. In comparison, FMR provides substantial benefits for NPB due to efficiently replicating even write-heavy pages.

To mitigate the performance overhead due to refresh, many recent prior works propose refreshing DRAM cells less frequently [12, 45, 49, 64, 71]; however, reducing the refresh rate of DRAM cells causes memory security and reliability problems [33, 38, 44, 46,

48] and is, therefore, undesirable for many important application scenarios. Nonblocking Memory Refresh [62, 63], the state-of-art prior work we compare against in Section 6, maintains JEDEC-compliant DRAM refresh frequency.

For multi-level cell (MLC) technologies, such as MLC Flash and MLC PCM, both existing products [30] and prior works [70] have explored how to dynamically switch between the slower MLC and the faster single-level cell (SLC) storage modes depending on the current level of usage. Switching from MLC to SLC in hardware can at most be viewed as allowing hardware to write a very restricted constant value (i.e., 0) to unused locations. The proposed architectural support, however, allows hardware to autonomously write arbitrary values to free memory locations to enable many more ways to boost microarchitecture performance.

9 CONCLUSION

In this work, we perform the first large-scale study of system-level memory utilization in HPC systems. Through seven million machine-hours of measurements across four in-production systems, we conclude that memory underutilization in HPC systems is much more severe than in cloud. Correspondingly, we perform the first exploration of architectural techniques to improve memory utilization in HPC systems. We propose exposing each compute node's currently unused memory to its CPU(s) via novel architectural support for OS. This can enable many new microarchitecture techniques that use the abundant free memory to boost microarchitecture performance transparently without requiring any user code modification or recompilation; we refer to them as Free-memory-aware Microarchitecture Techniques (FMTs). This paper presents a detailed example of an FMT – Free-memory-aware Memory Replication (FMR). Our evaluation shows FMR improves performance by 13% and system-level energy efficiency by 8%, on average across five HPC benchmark suites.

10 ACKNOWLEDGEMENTS

REFERENCES
[1] [n. d.]. Graph500. https://graph500.org/.

[2] [n. d.]. LANL CTS-1 Grizzly - Tundra Extreme Scale, Xeon E5-2695v4 18C 2.1GHz, Intel Omni-Path. https://www.top500.org/system/178972.

[3] [n. d.]. LINPACK. http://www.netlib.org/linpack/.

[4] 2017. Advanced Configuration and Power Interface Specification Version 6.2. https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[5] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. 2014. Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Proc. IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis (SC14)*. IEEE/ACM.

[6] Jung Ho Ahn, Norman P. Jouppi, Christos Kozyrakis, Jacob Leverich, and Robert S. Schreiber. 2012. Improving System Energy Efficiency with Memory Rank Sub-setting. *ACM Trans. Archit. Code Optim.* 9, 1, Article 4 (March 2012), 28 pages. https://doi.org/10.1145/2133382.2133386

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Bench-marks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[8] Ishan Banerjee, Fei Guo, Kiran Tati, and Rajesh Venkatasubramanian. 2013. Mem-ory Overcommitment in the ESX Server. *VMWare Technical Journal* (2013). https://labs.vmware.com/vmtj/memory-overcommitment-in-the-esx-server.

[9] Luiz AndrĂl Barroso and Urs HĂŭlzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Synthesis Lectures on Computer Architecture.* Morgan&ClayPool Publishers.

[10] Luiz AndrĂl Barroso, Urs HĂŭlzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. Synthesis Lectures on Computer Architecture.* Morgan&ClayPool Publishers. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[11] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[12] I. Bhati, Z. Chishti, S. L. Lu, and B. 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

[13] 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

[14] John Blackwood. 2012. An Overview of Kernel Text Page Replication in RedHawk Linux 6.3. https://www.concurrent-rt.com/wp-content/uploads/2016/11/kernel-page-replication.pdf.

[15] Yuri Bubly. 2019. AMD Ryzen Memory Tweaking & Overclocking Guide. https://www.techpowerup.com/review/amd-ryzen-memory-tweaking-overclocking-guide/9.html.

[16] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. [n. d.]. DRAMPower: Open-source DRAM Power & Energy Estimation Tool. URL: http://www.drampower.info.

[17] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with ac-cesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 356–367. https://doi.org/10.1109/HPCA.2014.6835946

[18] Sylvester Cash Charles Stephan, Alicia Boozer. 2016. Optimizing Mem-ory Performance of Lenovo Servers Based on Intel Xeon E7 v3 Processors. https://lenovopress.com/lp0048.pdf.

[19] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. 2012. Staged Reads: Mitigating the impact of DRAM writes on DRAM reads. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. https://doi.org/10.1109/HPCA.2012.6168943

[20] W. Chen, K. Ye, Y. Wang, G. Xu, and C. Xu. 2018. How Does the Workload Look Like in Production Cloud? Analysis and Clustering of Workloads on Alibaba Clus-ter Trace. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 102–109. https://doi.org/10.1109/PADSW.2018.8644579

[21] Jonathan Corbet. 2010. Memory compaction. https://lwn.net/Articles/368869/.

[22] Neo Cui. 2017. Demonstrating the Memory RAS Features of Lenovo ThinkSystem Servers. https://lenovopress.com/lp0778.pdf.

[23] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[24] S. Di, D. Kondo, and W. Cirne. 2012. Characterization and Comparison of Cloud versus Grid Workloads. In *2012 IEEE International Conference on Cluster Comput-ing*. 230–238. https://doi.org/10.1109/CLUSTER.2012.35

[25] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance Conjugate-gradient Benchmark. *Int. J. High Perform. Comput. Appl.* 30, 1 (Feb. 2016), 3–10. https://doi.org/10.1177/1094342015593158

[26] L. Ecco and R. Ernst. 2017. Tackling the Bus Turnaround Overhead in Real-Time SDRAM Controllers. *IEEE Trans. Comput.* 66, 11 (Nov 2017), 1961–1974. https://doi.org/10.1109/TC.2017.2714672

[27] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/224056.224072

[28] Adi Fuchs and David Wentzlaff. 2018. Scaling Datacenter Accelerators with Compute-reuse Architectures. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 353–366. https://doi.org/10.1109/ISCA.2018.00038

[29] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (Nov. 2015), 59–66. https://doi.org/10.1145/2814328

[30] Dave Glen. 2014. Optimized Client Computing With Dynamic Write Accel-eration. https://www.micron.com/~/media/documents/products/technical-marketing-brief/brief_ssd_dynamic_write_accel.pdf.

[31] Mel Gorman. 2010. Memory Compaction v1. https://lwn.net/Articles/368854/.

[32] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta. 2016. X-Mem: A cross-platform and extensible memory characterization tool for the cloud. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 263–273. https://doi.org/10.1109/ISPASS.2016.7482101

[33] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 300–321. https://doi.org/10.1007/978-3-319-40667-1_15

[34] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[35] HP. [n. d.]. CACTI. https://www.hpl.hp.com/research/cacti/.

[36] Intel. [n. d.]. Intel Xeon Processor E7 Family: Reliability, Availability, and Service-ability: Advanced data integrity and resiliency support for mission-critical de-ployments. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-e7-family-ras-server-paper.pdf.

[37] Intel. 2017. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[38] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)* (October 2017).

[39] JEDEC. 2009. DDR2 SDRAM SPECIFICATION. https://www.jedec.org/system/files/docs/JESD79-2F.pdf.

[40] JEDEC. 2012. DDR3 SDRAM SPECIFICATION. https://www.jedec.org/sites/default/files/docs/JESD79-3F.pdf.

[41] JEDEC. 2017. JEDEC STANDARD DDR4 SDRAM JESD79-4B. https://www.jedec.org/standards-documents/docs/jesd79-4a.

[42] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan. 2019. Characteristics of Co-Allocated Online Services and Batch Jobs in Internet Data Centers: A Case Study From Alibaba Cloud. *IEEE Access* 7 (2019), 22495–22508. https://doi.org/10.1109/ACCESS.2019.2897898

[43] Vincent J. Zimmer Jiewen Yao. 2015. A Tour be-yond BIOS Memory Map Design in UEFI BIOS. https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in%20UEFI_BIOS.pdf.

[44] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R. Alameldeen, Chris Wilkerson, and Onur Mutlu. 2014. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. *SIGMETRICS Perform. Eval. Rev.* 42, 1 (June 2014), 519–532. https://doi.org/10.1145/2637364.2592000

[45] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. 2017. Detecting and Mitigating Data-dependent DRAM Fail-ures by Exploiting Current Memory Content. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3123939.3123945

[46] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. 2014. Flipping bits in memory without accessing them: An experimental

study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[47] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[48] Mark Lanteigne. 2016. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. (march 2016). http://www.thirdio.com/rowhammer.

[49] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. 2015. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 489–501. https://doi.org/10.1109/HPCA.2015.7056057

[50] H. Li and L. Wolters. 2007. Towards A Better Understanding of Workload Dynamics on Data-Intensive Clusters and Grids. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–10. https://doi.org/10.1109/IPDPS.2007.370250

[51] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[52] 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. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 267–278. https://doi.org/10.1145/1555754.1555789

[53] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. https://doi.org/10.1109/HPCA.2012.6168955

[54] Ben Lin, Michael Healy, Rustam Miftakhutdinov, Phil Emma, and Yale Patt. 2018. Mitigating Off-Chip Memory Bank and Bank Group Conflicts via Data Duplication. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*.

[55] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. 2884–2892. https://doi.org/10.1109/BigData.2017.8258257

[56] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. 2014. Evaluation of CPU frequency transition latency. *Computer Science - Research and Development* 29, 3 (01 Aug 2014), 187–195. https://doi.org/10.1007/s00450-013-0240-x

[57] MICRON. 2015. DDR4 SDRAM LRDIMM MTA72ASS4G72LZ - 32GB. https://www.micron.com/-/media/documents/products/datasheet/modules/lrdimm/ddr4/ass72c4gx72lz.pdf.

[58] MICRON. 2017. 8Gb: x4, x8, x16 DDR4 SDRAM. https://classes.engineering.wustl.edu/permanant/cse260m/images/0/0c/8Gb_DDR4_SDRAM.pdf.

[59] MICRON. 2017. DDR4 SDRAM UDIMM MTA18ASF2G72AZ - 16GB. https://www.micron.com/-/media/documents/products/datasheet/modules/unbuffered_dimm/ddr4/asf18c2gx72az.pdf.

[60] MICRON. 2019. DDR4 SDRAM RDIMM: MTA144ASQ16G72PSZ - 128GB. https://www.micron.com/-/media/client/global/documents/products/datasheet/modules/lrdimm/ddr4/asq144c16gx72psz.pdf.

[61] MICRON. 2019. DDR4 SDRAM RDIMM: MTA36ASF4G72PZ - 32GB. https://www.micron.com/-/media/documents/products/datasheet/modules/rdimm/ddr4/asf36c4gx72pz.pdf.

[62] K. Nguyen, K. Lyu, X. Meng, V. Sridharan, and X. Jian. 2018. Nonblocking Memory Refresh. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 588–599. https://doi.org/10.1109/ISCA.2018.00055

[63] K. Nguyen, K. Lyu, X. Meng, V. Sridharan, and X. Jian. 2019. Nonblocking DRAM Refresh. *IEEE Micro* 39, 3 (May 2019), 103–109. https://doi.org/10.1109/MM.2019.2907486

[64] Minesh Patel, Jeremie S. Kim, and Onur Mutlu. 2017. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/3079856.3080242

[65] J. Thomas Pawlowski. 2018. In-person Interview.

[66] J. Thomas Pawlowski. 2019. Email Interview.

[67] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '03)*. ACM, New York, NY, USA, 318–319. https://doi.org/10.1145/781027.781076

[68] 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

[69] 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

[70] Moinuddin K. Qureshi, Michele M. Franceschini, Luis A. Lastras-Montaño, and John P. Karidis. 2010. Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 153–162. https://doi.org/10.1145/1816038.1815981

[71] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu. 2015. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 427–437. https://doi.org/10.1109/DSN.2015.58

[72] redhat. 2019. CACHE LIMITATIONS WITH NFS. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/fscachelimitnfs.

[73] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/2391229.2391236

[74] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 72–82. https://doi.org/10.1145/1816038.1815972

[75] G. Tziantzioulis, N. Hardavellas, and S. Campanoni. 2018. Temporal Approximate Function Memoization. *IEEE Micro* 38, 4 (Jul 2018), 60–70. https://doi.org/10.1109/MM.2018.043191126

[76] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 745–758. http://dl.acm.org/citation.cfm?id=3154690.3154760

[77] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. 2010. Making Address-Correlated Prefetching Practical. *IEEE Micro* 30, 1 (Jan 2010), 50–59. https://doi.org/10.1109/MM.2010.21

[78] Wikichip. [n. d.]. Skylake (server) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server).

[79] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 59–63. https://doi.org/10.1109/LCA.2017.2762308

[80] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article 3 (March 2017), 26 pages. https://doi.org/10.1145/3023362