Quantifying Server Memory Frequency Margin and Using It to Improve Performance in HPC Systems

Da Zhang[§*], Gagandeep Panwar[§*], Jagadish B. Kotra[†], Nathan DeBardeleben[‡], Sean Blanchard[‡], Xun Jian[§]
[§]*Virginia Tech* [†]*AMD Research* [‡]*Los Alamos National Laboratory*
{daz3, gpanwar, xunj}@vt.edu Jagadish.Kotra@amd.com {ndebard, seanb}@lanl.gov

***Abstract*— To maintain strong reliability, memory manufacturers label server memories at much slower data rates than the highest data rates at which they can still operate correctly for most (e.g., 99.999%+ of) accesses; we refer to the gap between these two data rates as memory frequency margin. While many prior works have studied memory *latency* margins in a different context of consumer memories, none has publicly studied memory *frequency* margin (either for consumer or server memories).**

To close this knowledge gap in the public domain, we perform the first public study to characterize frequency margins in commodity server memory modules. Through our large-scale study, we find that under standard voltage and cooling, they can operate 27% faster, on average, without error(s) for 99.999%+ of accesses even at high temperatures.

The current practice of conservatively operating server memory is far from ideal; it slows down 99.999%+ of accesses to benefit the <0.001% of accesses that would be erroneous at a faster data rate. An ideal system should only pay this reliability tax for the <0.001% of accesses that actually need it.

Towards unleashing ideal performance, our second contribution is performing the first exploration on exploiting server memory frequency margin to maximize performance. We focus on High-Performance Computing (HPC) systems, where performance is paramount. We propose exploiting HPC systems' abundant free memory in the common case to store copies of every data block and operate the copies unreliably fast to speedup common-case accesses; we use the safely-operated original blocks for recovery when the unsafely-operated copies become corrupted. We refer to our idea as Heterogeneously-accessed Dual Module Redundancy (Hetero-DMR).

Hetero-DMR improves node-level performance by 18%, on average across two CPU memory hierarchies and six HPC benchmark suites, while weighted by different frequency margins and different levels of memory utilization. We also use a real system to emulate the speedup of Hetero-DMR over a conventional system; it closely matches simulation. Our system-wide simulations show applying Hetero-DMR to an HPC system provides 1.4x average speedup on job turnaround time. To facilitate adoption, Hetero-DMR also rigorously preserves system reliability and works for commodity DIMMs and CPU-memory interfaces.

***Index Terms*—Memory Frequency Margin, Memory System, Fault Tolerance, Reliability, Availability, HPC**

I. Introduction

To maintain strong reliability, memory manufacturers label server memories at much slower data rates than the highest data rates at which they can still correctly operate for most (e.g., 99.999%+) accesses; we refer to the gap between these two data rates as memory frequency margin.

Currently, no prior work in the public domain has characterized memory frequency margin; in fact, in the context of server systems, no prior work in the public domain has characterized memory margins of any kind (e.g., frequency margin or latency margin). Although many prior works [47], [50], [60], [62], [65] have studied memory margins, they only studied latency and voltage margins in the context of consumer systems (e.g., mobile systems and desktop systems).

In this paper, we perform the first public study to characterize frequency margin for server memory modules and close this knowledge gap in the public domain. The scale of our study is larger than any single prior work on characterizing memory *latency* margins for memories of consumer systems; in terms of the total number of chips studied, our study covers more than all such prior works combined. Our study shows that under standard voltage and cooling, commodity server memory modules can operate 27% faster, on average, without error(s) for 99.999%+ of accesses even at high temperatures.

The current server system design of conservatively operating memory is too wasteful; it slows down 99.999%+ of accesses to benefit the <0.001% of accesses that would be erroneous at a faster data rate. An ideal system should only pay this reliability tax for the <0.001% of accesses that need it.

Towards unleashing ideal performance, we also perform the first exploration on exploiting server memory frequency margin to improve server performance. We focus on HPC systems, where performance is the most important metric; HPC systems encompass traditional onsite supercomputers and the rapidly growing HPC in Cloud [2], [8], [24].

Exploiting memory frequency margin in HPC systems is challenging because doing so naively increases memory error rate and reduces reliability, which is also important for HPC systems [69], [74]. To facilitate adoption, an ideal solution should rigorously maintain the same system reliability as always abiding by specification.

We observe errors due to operating modules faster than specification, by definition, do not occur in modules that consistently abide by specification. As such, we propose replicating every block to a second module and operating the two modules heterogeneously: Operate **ONLY** the module with copies unsafely fast and read it commonly to boost performance for the 99.999%+ of reads; operate the module with original blocks reliably **ALWAYS** according to specification and read it uncommonly to correct errors for the <0.001% of accesses. As such, our proposed design can

[*] The student authors are ordered by their first names.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00064

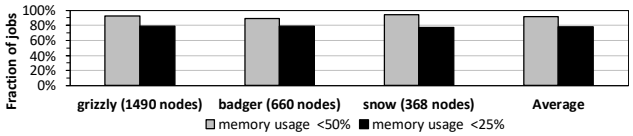


Fig. 1: Fraction of jobs in which every node the job occupies has <50% and <25% memory utilization (i.e., all included, including OS file cache) throughout the job's lifetime.

achieve similar performance as the ideal system; furthermore, under our design, regardless of what imaginable error rate or error pattern or specific error model may occur in the modules holding copies due to operating them unsafely, the modules holding original blocks are unaffected and thus can maintain correctness. We refer to our proposal as *Heterogeneously-accessed Dual Module Redundancy (Hetero-DMR)*.

Naively replicating data can incur 100% memory capacity overhead. Recent prior works - FMR [64] and CLR-DRAM [63] - solve this problem by storing copies only in memory that are currently not used by software; by dynamically releasing memory when software needs more memory, these prior works maintain the same software-usable memory capacity as conventional systems. We refer to these prior works as Free-memory-aware architectures. Like prior works on Free-memory-aware architectures, Hetero-DMR stores copies only in free memory to maintain the same software-usable capacity as without Hetero-DMR. Moreover, Hetero-DMR reuses FMR's broadcasting write design to eliminate write bandwidth overhead for updating data copies. Unlike FMR and CLR-DRAM, which exploit free memory to only improve latency, Hetero-DMR improves both latency and frequency; as such, Hetero-DMR must address the new memory reliability challenge of how to safely exploit memory frequency margin.

Many studies [61], [64], [66], [77] find that active nodes in HPC systems have abundant free memory. We analyze 3,000,000,000 memory measurements, spanning 7,000,000 machine-hours, recently released by Los Alamos National Lab (LANL) [29] and reach similar conclusion (see Figure 1). Prior study [64] contributes low memory utilization to HPC workload and runtime behaviors: First, because HPC workloads are highly parallel, just one workload usually occupies all cores in a node; this makes it difficult to collocate/consolidate multiple workloads on each node to improve memory utilization. Second, an HPC node's data input usually comes from message passing (MPI) [19] transactions, which cannot be cached in OS file cache; this keeps memory used by OS file cache small. Third, HPC workloads are typically compute-intensive rather than storage-intensive; this further limits memory usage by OS file cache. Fourth, HPC improves performance by dividing large problems into smaller problems and distributing them to many nodes to compute in parallel; this keeps per-node memory utilization low even when the problem size is large.

While memory utilization for HPC in Cloud are not publicly available, it is likely similar to onsite HPC due to similar software and hardware. HPC in Cloud runs many of the same workloads as onsite HPC [2], [11]. HPC nodes in Cloud have similar memory-to-core ratio as onsite HPC; for example, HPC nodes in Azure [25] and AWS [1] have the same or higher 4GB per physical core ratio as those in Figure 1.

We make the following contributions in this paper:

- We perform the first study to characterize frequency margins for server memory modules in the public domain.
- We are first to explore how to rigorously maintain system reliability when operating memory faster than spec.
- Our simulations show applying Hetero-DMR to a Commercial Baseline and FMR improves node-level performance by 18% and 15%, respectively, without degrading average system-level energy efficiency.
- System-wide simulations show Hetero-DMR provides 1.4x average speedup on job turnaround time over a conventional HPC system.
- We also emulate Hetero-DMR on a real system and find closely matching performance benefit as our simulations.

II. Real-system Characterizations

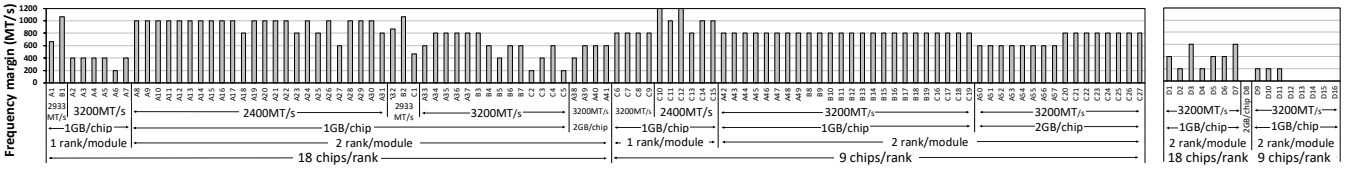
In this paper, we perform the first public study to characterize frequency margin of server memory modules. While many prior works [47], [50], [60], [62], [65] study memory latency and voltage margins, none study frequency margin. While consumer computing enthusiasts talk about overclocking memory on forums and blogs, they usually report on one or two personal desktop modules. Unlike these user reports, our study quantifies memory frequency margins for a broad range of server DIMMs. Table I shows the scale of our study relative to prior works that characterize other aspects of DRAM.

Manufacturers often "design significant margin into their products" to maintain strong reliability [22]. For those unfamiliar with design margins, a general example is that roads are designed to be much wider than cars that drive on it. A closer example is CPU margins; CPU has significant margins to protect against different (e.g., voltage, temperature, manufacturing, process, etc.) variations to maintain correct operations under worst-case conditions [55]. Memory manufactures also stress test memories to reject 'weak' parts without sufficient margins (e.g., voltage, temperature margins [18]) to increase the likelihood that their products are reliable in the field.

Specifically, memory manufacturers design significant frequency margins into their products to protect against non-deterministic frequency variations in the field, such as clock jitters [5], [15], [68]. For example, JEDEC DDR5 standard [15] stipulates that DRAM "must pass" stress tests with 10^{-9} bit error rate; these stress tests are specified with artificially-created, aggressive clock jitters and transmission

	DRAM type	# of modules	# of chips	Margin Studied
This Paper	DDR4 RDIMM	119	3006	frequency
Prior Work [60]	DDR3 SO-DIMM	96	768	latency
Prior Work [56]	DDR3 SO-DIMM	32	416	latency
Prior Work [47]	DDR3 SO-DIMM	30	240	latency
Prior Work [65]	LPDDR4	N/A	368	latency
Prior Work [62]	DDR3 SO-DIMM	34	248	latency
Prior Work [50]	DDR3 UDIMM	8	64	voltage

TABLE I: Scale of our study compared to prior works.



(a) Brands A, B, and C. They are major memory chip manufacturers, unlike D, which manufactures only modules but not chips. (b) Brand D.

Fig. 2: Memory frequency margins across 119 server modules of brands A, B, C, and D.

noises (e.g., crosstalk). Because memories are designed according to stress tests, they end up with a memory frequency margin that may be opportunistically exploited in the field, where the stress test conditions occur much less frequently.

A. Characterizing Memory Frequency Margin

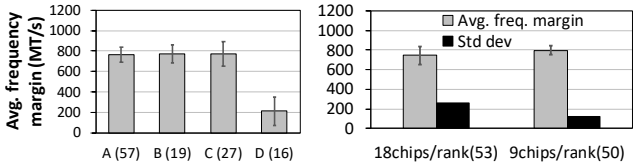
We use an Intel Xeon W-3175X CPU [13] on a GIGABYTE C621 AORUS Xtreme motherboard [7] to measure frequency margins of RDIMMs. To characterize a given module, we install just that module by itself on our test machine. We determine a module's frequency margin by measuring the highest data rate at which the module can still correctly carry out 99.999%+ of accesses. Due to BIOS limitation, we use the step size of 200MT/s to scale memory data rate.

All experiments comply with the DDR4 standard voltage of 1.2V as high voltages can damage hardware. All experiments use standard DDR4 server modules without any cooling support (e.g., heat sink). While increasing data rate could theoretically damage memory due to increasing temperature, we do not expect this to be a problem for memory for two reasons: First, memories are designed to operate up to $95\,^{\circ}\mathrm{C}$ [14], [15]. Second, memory chips consume little power (e.g., 0.3W/chip at full utilization [21], [23]), unlike CPUs, which consume much higher power (e.g., 255W for Intel Xeon W-3175X [13]); in our test machine, on-DIMM sensors report $<1\,^{\circ}\mathrm{C}$ average temperature difference between operating at specified data rate and at maximum bootable data rate after stress testing until temperature stabilizes.

We conduct our experiments in an ambient temperature of $23\,^{\circ}\mathrm{C}$. When operating at manufacturer specification, the on-DIMM temperature sensors report $43\,^{\circ}\mathrm{C}$, on average, if the system is idle and read $53\,^{\circ}\mathrm{C}$, on average, under memory stress test [31]. To put these temperatures in perspective, we analyzed three million on-DIMM temperature sensor measurements [16] taken from Trinitite system at LANL. The minimum is $16\,^{\circ}\mathrm{C}$, which suggests the ambient temperature is $16\,^{\circ}\mathrm{C}$. The $43\,^{\circ}\mathrm{C}$ idle and $53\,^{\circ}\mathrm{C}$ active DIMM temperatures in our test machine are higher than 99% and 99.85%, respectively, of all temperature measurements from Trinitite. The cooler DIMM temperatures in Trinitite are likely due to better cooling [75].

Figures 2a and 2b summarize our frequency margin measurements across 119 server modules. Below, we analyze the impact of brand, DIMM organization, chip density, manufacturing date, and system configuration on frequency margin.

Impact of Brand. We find that modules manufactured by the largest memory brands have higher frequency margins. The 119 memory modules in our study are manufactured by four companies: Brands A to C are three major memory chip manufacturers; Brand D is a much smaller memory company that manufactures only modules but not chips. Figure 3a shows the average frequency margin and 99% Confidence Interval (CI); we use the normal distribution to calculate CI similar to a prior work [60]. The average frequency margin across the three largest brands is 770MT/s (i.e., 27% when normalized to each module's manufacturer specified data rate); it is 2.6x higher than the 213MT/s average frequency margin of modules of brand D. However, among brands A to C, their average frequency margins are similar to each other. Because brands A to C are mainstream server memory brands commonly used in HPC systems, we will focus on brands A to C in the rest of this paper and ignore brand D.

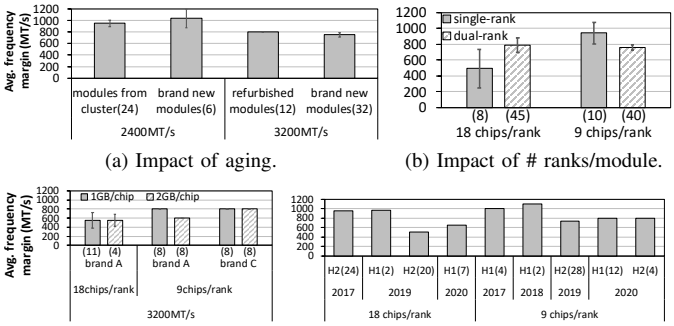


(a) Impact of brand. (b) Impact of # chips/rank.

Fig. 3: Impact of memory module factors. The numbers within the parentheses indicate the number of modules.

Impact of # Chips/Rank. We find that modules with fewer chips per rank have consistently high frequency margins. A rank is a group of memory chips that operate in lockstep synchronously. Figure 3b shows the average frequency margins and standard deviations (STDev) of modules with 18 chips/rank and modules with 9 chips/rank. Modules with 9 chips/rank have a much lower variation on frequency margins with an STDev of 124MT/s; the minimum frequency margin is 600MT/s. The STDev of modules with 18 chips/rank is 2.1x in comparison. We hypothesize that modules with 9 chips/rank have consistently high frequency margin with lower variation because, intuitively, synchronizing fewer chips is easier than more chips when operating at high frequency.

Impact of Manufacturer-specified Data Rate. A module's manufacturer-specified data rate seems to impact its frequency margin. 2400MT/s modules have 967MT/s average margin; 3200MT/s modules have 679MT/s average margin in comparison. But we are not confident about this conclusion; evidences suggest system-level factors external to the memory modules may be capping memory data rate to 4000MT/s, which in turn limits the maximum observable memory frequency margin of the 3200MT/s modules to 4000-3200=800MT/s. The initial evidence that caught our attention is that while most (i.e.,



(a) Impact of aging. (b) Impact of # ranks/module.

(c) Impact of chip density. (d) Impact of manufacturing date.

Fig. 4: Impact of other memory module factors. The numbers within the parentheses indicate the number of modules.

36 out of 44) of the 3200MT/s modules with 9 chips per rank can operate at 4000MT/s, not one of them operate faster than 4000MT/s in our test machine. To investigate whether our test system may have a system-level memory data rate cap of 4000MT/s, we conduct more experiments to check whether data rate can go beyond 4000MT/s by increasing memory voltage from the standard 1.2V to 1.35V. We again find that no 3200MT/s module that can already operate at 4000MT/s at 1.2V manifest higher data rate at 1.35V; in comparison, out of the 27 3200MT/s modules that cannot operate at 4000MT/s at 1.2V, 22 of them can operate at higher frequency at 1.35V.

A possible limiting factor external to memory may be our CPU. While the 3200MT/s memory modules are top of the line, our test CPU is not; it is advertised to run memory at 2666MT/s. As such, the CPU may easily handle 2400MT/s plus 2400MT/s modules' margin, but may have some difficulty handling 3200MT/s plus 3200MT/s modules' full margin (which may be much higher than the observed 679MT/s). We use this CPU because it is the only unlocked server CPU currently on the market to the best of our knowledge.

Impact of Other Memory Module Factors. We find that aging has little impact on frequency margin; Figure 4a shows the characterization of brand new modules compared to modules extracted from a three-years-old in-production cluster and refurbished modules. We also find that other memory module factors, including number of ranks/module, chip density, and manufacturing date have little impact on frequency margin as shown in Figure 4b, 4c, and 4d.

Impact of Different CPUs. We find that different CPUs of the same model have little impact on memory frequency margin. We measure frequency margins for all modules using a second CPU of the same model as the CPU used to generate Figure 2. The memory frequency margins of all modules remained the same under the second CPU.

Impact of Exploiting Memory Latency Margin. We explore the impact of exploiting latency margin on exploiting frequency margin. We explore the margins of four most important latency parameters: tRCD, tRP, tRAS, and tREFI. Determining the ideal latency margin combination for one module requires $(\#\ param.*permutations\ of\ param.*\#\ tests\ per\ param.)$ tests; for 119 modules, four parameters, and multiple (e.g., five) tests per parameter , this translates to $(119*4*4!*5) =$ 52320 tests, which is intractable. As such, we test just one permutation - $< tRCD, tRP, tRAS, tREFI >$ - and use the latency margin combination measured for the previous module as the starting point to measure that of the next module. The end result is the conservative latency margin combination that works for all 119 modules; it is $< 16\%, 16\%, 9\%, 92\% >$. We then measure the frequency margins for every module when operating under this conservative latency margin combination; we find that every module has the same frequency margin as when operating under the manufacturer specified latency.

	Data Rate	tRCD	tRP	tRAS	tREFI
Manufacturer-specified Setting	3200MT/s	13.75ns	13.75ns	32.5ns	7.8us
Setting to Exploit Latency Margin	3200MT/s	11.5ns	11ns	29.5ns	15us
Setting to Exploit Frequency Margin	4000MT/s	13.75ns	13.75ns	32.5ns	7.8us
Setting to Exploit Freq+Lat Margins	4000MT/s	11.5ns	11ns	29.5ns	15us

TABLE II: Memory setting for exploiting memory margins.

	Memory Hierarchy1	Memory Hierarchy2
L2$+L3$ per core	4.5MB / core	2.375MB / core
Cores	8 cores	16 cores
Memory Channels	1 channel, 2modules/channel, 2ranks/module	4 channels, 2modules/channel, 2ranks/module

TABLE III: Real System Configurations.

B. Characterizing Performance Loss due to Current Practice of Conservatively Operating Memory

Intuitively, the current practice of slowing down ALL accesses just for the <0.001% of accesses that need to loses out on substantial performance. To quantify this loss, we use our test machine to measure the four memory settings listed in Table II. All experiments use state-of-the-art DDR4 RDIMMs with a labelled data rate of 3200MT/s - the maximum rate under DDR4 JEDEC [14]. Our experiments use modules with 9 chips/rank, instead of 18 chips/rank, because they better resemble upcoming DDR5 server modules; DDR5 only supports up to 10 chips/rank [39]. Lastly, our experiments use modules with 800MT/s memory frequency margin, which is the most common frequency margin among the 119 modules (see Figure 2). We cherry-pick modules that correctly run the benchmarks while exploiting these memory margins; **Hetero-DMR eliminates the need for cherry-picking**.

We evaluate two memory hierarchies (see Table III); we use Intel Cache Allocation Technology [12] to enforce the cache-to-core ratios in Table III. We disable Hyper-Threading. We evaluate six HPC benchmark suites – Linpack [17], HPCG [53], Graph500 [9], CORAL2 [6], LULESH [57], and NASA Parallel Benchmarks (NPB) [41]. We evaluate all benchmarks

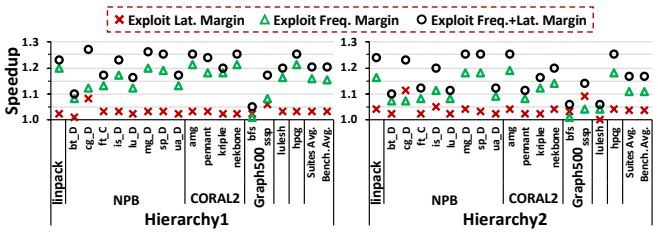


Fig. 5: Real-system speedup due to exploiting memory margins. Speedup = execution time under manufacturer specification / execution time when operating faster than specification.

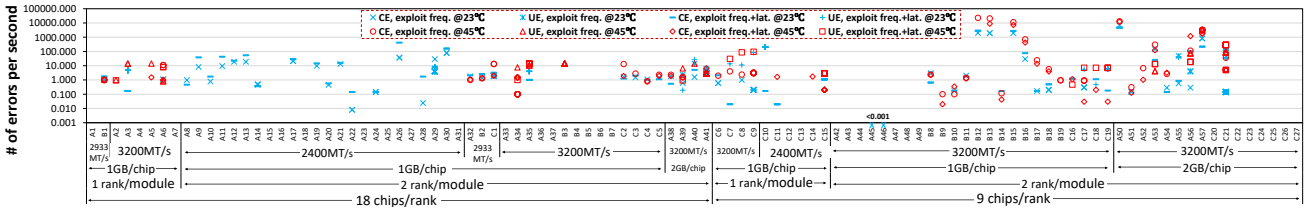


Fig. 6: **Memory error rate when exploiting each module's margins at** 23 °C **and** 45 °C **ambient temperatures.** Tests (e.g., tests for C22-C27) that had zero errors are not plotted in the figure. Modules A3, A40, A55, B12, B19, C3, C6, C10, and C12 do not have some error rate numbers in the high ambient temperature due to failing to boot. We did not test modules A8-A31 in the thermal chamber because they were borrowed from an in-production cluster.

under all suites, except for CORAL2, where we evaluate four benchmarks. We note our single-node experiment may under-represent communication time. To address this, we run all benchmarks under MPI, instead of OpenMP, to increase communication. We also use benchmarks' small standard input sizes; it is well-known that communication time increases relative to computation time as problem size decreases. Our benchmarks spend 13% of core-hours on MPI communication under Hierarchy1, on average. To put this in perspective, a recent study [49] reports a 786K-core supercomputer [26] spends 34% of its core-hours on MPI communication.

Figure 5 shows the speedup from exploiting memory margins. Exploiting both memory frequency and latency margins provides 1.19x speedup on average across six HPC benchmark suites[1] and on average across both memory hierarchies. For Linpack, the de facto standard benchmark for ranking HPC systems world-wide, it provides 1.24x average speedup.

C. Characterizing Memory Error Rate

While memory manufacturers design margins into their memory, they cannot guarantee reliability beyond their specifications [22]. As such, exploiting frequency margin faces the problem of higher error rate, similar to when driving cars over road/lane margins faces the problem of higher crash rate.

To study a module's error rate when exploiting its frequency margin, we install just that module by itself and run a one-hour memory reliability stress test [31]; all tests comply with the DDR4 standard voltage of 1.2V. We record the Corrected Errors (CEs) and Uncorrected Errors (UEs) encountered during the stress test. By default, Linux crashes when user programs encounter UEs; we configure Linux to not do so by setting its tolerance level [34]. Figure 6 shows each module's error rate when exploiting its frequency margin in an ambient temperature of 23 °C.

To characterize memory modules' error rate due to occasional unintended spikes in ambient temperature, we use a thermal chamber [33] to emulate an ambient temperature of 45 °C, and repeat the characterization of all modules of Brand A, B, C. We put the whole test machine in the thermal chamber and heat it for at least 30 minutes. At 45 °C, the active DIMM temperature is 60 °C on average, which is higher than 99.991% of all temperature measurements from the HPC system discussed in Section II-A. Five of 103 modules show reduced frequency margin. Figure 6 also shows each module's error rate when operating at its highest bootable data rate. Error rate is 4X higher than in 23 °C ambient temperature.

To study the error rate due to simultaneously exploiting both memory frequency and latency margins, we use the same latency margin as in Section II-B when operating a module at its highest bootable data rate and repeat the characterization, as described above; figure 6 shows the error rate. At 45 °C, nine of 103 modules show reduced frequency margin. Compared to the error rate at 23 °C, the error rate at 45 °C is 2X higher.

We also measure full-system-level memory error rate. We populate all channels and all slots per channel in our test machine with 3200MT/s dual-rank modules with 800MT/s margin per module; we find that the memory system as a whole has 800MT/s margin. We measure error rates for the Setting to Exploit Freq+Lat Margins from Table II at both 23 °C and 45 °C ambient temperatures. The memory system's error rate is roughly half the sum of all installed module's error rates. We theorize it is half because each module is accessed half as frequently due to having two modules per channel when fully populating our test machine's memory system.

III. Hetero-DMR

While prior works [47], [60], [62], [65] have explored exploiting memory latency margin, none has explored exploiting memory frequency margin to boost performance. Exploiting only latency margin loses out on the many times higher benefit from ALSO exploiting frequency margin (see Figure 5). These prior works are also in the context of mobile and desktop systems, where reliability is not as important as our context of server systems; as such, their solutions need not and do not rigorously maintain system-level reliability (see Section V).

While some prior works [50]–[52] explore memory frequency scaling, they only reduce frequency to save energy, instead of increasing frequency beyond manufacturer specification to improve performance.

In this paper, we perform the FIRST exploration on exploiting memory frequency margin. This is also the first work to exploit memory margins of any kind in the context of servers, where reliability should be preserved. When operating memory outside of specification, many kinds of errors could happen (e.g., full block errors due to IO errors or losing all blocks due to misinterpreting a command as a DRAM reset command).

[1]"Average across six HPC benchmark suites" means weighing every suite equally in the average. This definition is used throughout the paper.

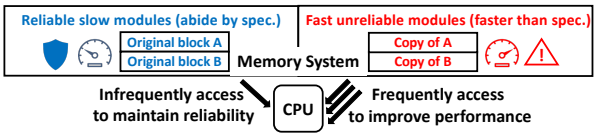


Fig. 7: Hetero-DMR overview.

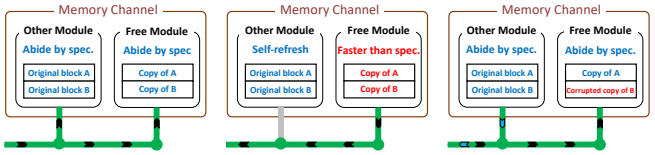


Fig. 8: (a) Write Mode under Hetero-DMR. (b) Read Mode under Hetero-DMR. (c) Error correction under Hetero-DMR.

Goal: Our goal is to enhance server memory systems to transparently operate faster than memory manufacturer specification while *rigorously maintaining the same level of reliability as a system that always abides by specification*. This means being able to protect against any imaginable error rate, pattern, or error model that may arise from operating memory beyond its specification. No prior work claims or achieves this goal; Section V discusses prior works in more detail.

The reason for this goal is to facilitate adoption. System designers and decision-makers often hesitate to adopt techniques without knowing the reliability risks [45]. If our solution cannot rigorously protect against all imaginable errors that may arise from operating memory out-of-spec, its reliability impact has to be empirically quantified prior to adoption; doing so for large-scale systems with 1000s of nodes (e.g., in an HPC system) is costly and difficult. Also, to facilitate adoption, we target commodity memory modules and interfaces.

Overview: We observe errors due to operating modules faster than specification, by definition, do not occur in modules that consistently abide by specification. As such, we propose opportunistically replicating every block to a second module when it is free (i.e., not being used by any software) and operate the two modules heterogeneously: Operate **ONLY** the Free Module unsafely fast and use the copies it stores to satisfy 99.999%+ of all cache misses at improved bandwidth and latency; operate the module holding original blocks reliably **ALWAYS** according to specification and use it to correct errors for the <0.001% of accesses. Due to the heterogeneous operations, regardless of what imaginable error rate or error pattern or specific error model may occur in the copies in the unsafely fast Free Module, the original blocks remain intact to maintain correctness. We refer to our proposal as *Heterogeneously-accessed Dual Module Redundancy (Hetero-DMR)*. Figure 7 graphically illustrates Hetero-DMR.

The rest of this section is organized as follows. Section III-A describes how to operate the original blocks and their copies heterogeneously. Sections III-B and III-C describe how to detect and correct errors. Section III-D describes how to handle variability in memory frequency margin. Section III-E provides implementation details. Section III-F discusses generality and impact on aging.

A. Challenge 1: How to Efficiently Keep Original Blocks Safe While Operating Copies Unsafely Fast?

One solution is to store the original blocks and their copies in different channels and heterogeneously operate the channels; operate channels containing copies faster than spec and operate channels containing original blocks consistently according to spec. This naive solution operates only half of the channels faster than spec, which halves the performance benefit compared to an ideal design that operates all channels faster than spec. For each writeback from LLC, this solution also must write to two channels to update the original block and its copy; this incurs 100% write bandwidth overhead.

Hetero-DMR stores a copy in the same channel as its original block; the copy and its original block are in different modules in the same channel. Under this design, all channels store some copies and, therefore, all have the opportunity to operate faster than spec. **This design can also eliminate all write bandwidth overhead to update copies** by enabling a single memory bus transaction to update both an original and its copy; this is because CPU connects to all ranks in a channel via the bus interconnection topology, which is well-known to support broadcasting of identical command and data values to multiple components on the bus in a single bus transaction (for more details, see paragraphs 4-6 of Section 4.3 in FMR [64]). The only restriction is that the original block and its copy must reside in the same location across different ranks in a channel because a broadcasted write command can only send the same address field to all ranks; as such, for an original block at location *i* in a module, Hetero-DMR stores its copy at location *i* in a Free Module in the same channel.

Sections III-A1 and III-A2 address the unique challenges of how to efficiently keep the original blocks safe when storing both originals and their copies in the same channel.

1) How To Keep Original Blocks Safe When a Channel is in Write Mode: Today's DRAM chips take a long time to switch between read and write; this round-trip latency from read to write and back takes ∼20ns [20]. To mitigate the corresponding performance overhead, modern memory channels minimize the round-trip latency per write by buffering many writes (e.g., 128) and performing them together in a large batch without reading in-between [46], [54]; we say a channel is in write mode when it is performing a batch of writes.

When a channel is in write mode, Hetero-DMR safely operates all modules in the channel (see Figure 8a) to safely update the originally blocks (i.e., before entering write mode, Hetero-DMR slows down the channel's frequency and latency to specification.). Figures 9 illustrate how Hetero-DMR scales down frequency in a JEDEC-compliant manner; all together, the steps take 1us [20], which effectively increases the round-trip latency from read to write and back by 100x compared to that of today's systems. As such, to maintain the same read-write switching overhead as today's systems, Hetero-DMR must switch between reads and writes 100x less often, which in turn requires Hetero-DMR to increase the write batch size by 100x (e.g., 128*100=12800 writes per batch). Hetero-DMR

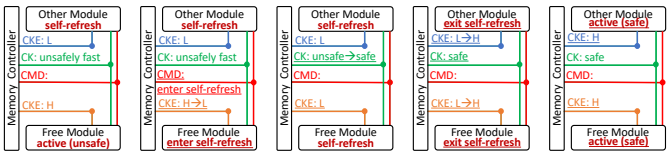


Fig. 9: Decreasing frequency under Hetero-DMR: (a) Unsafely fast state. (b) Transition state: prepare to reduce frequency. (c) Transition state: reduce frequency. (d) Transition state: synchronize memory to new frequency. (e) Safe state.

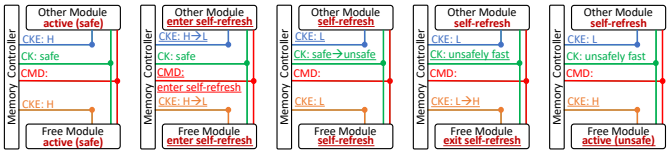


Fig. 10: Increasing frequency under Hetero-DMR: (a) Safe state. (b) Transition state: prepare to increase frequency. (c) Transition state: increase frequency. (d) Transition state: synchronize memory to new frequency. (e) Unsafely fast state.

implements this large batch size by proactively cleaning some dirty blocks from last-level cache (LLC) each time a channel enters write mode (see Section III-E).

During write mode, Hetero-DMR also writes to the copies; updating copies does not cause write bandwidth overhead (see the second Paragraph of Section III-A).

Because Hetero-DMR safely operates all modules when a channel is in write mode, there is no benefit for writes. However, writes only account for a small fraction of memory accesses (e.g., only 15%, see Figure 15); as such, Hetero-DMR can still achieve similar performance as an ideal system.

2) How to Keep Original Blocks Safe When a Channel is in Read Mode: After ending write mode, a channel switches to read mode. At the beginning of read mode, Hetero-DMR speeds up the channel's frequency and latency beyond specification; Figure 10 shows how to scale up frequency.

When a channel operates a module faster than specification, even reads from the module can corrupt the module's content. When reading, today's DRAM chip first extracts the data to a buffer (i.e., row buffer) via a destructive process [20] that destroys the data in DRAM cells; restoring the data after reading requires a proper sequence of commands from CPU. Therefore, the data in DRAM can be corrupted by an erroneously transmitted command sequence due to operating DRAM at unsafely high frequency. As another example, prior work [47] reports that exploiting activation latency (i.e., *tRCD*) margin can corrupt the values in the accessed DRAM location and exploiting precharge latency (i.e., *tRP*) margin can corrupt an entire row in DRAM.

To keep the original blocks safe while a channel is in read mode, Hetero-DMR only accesses the Free Modules to satisfy LLC misses; it does not access the modules with original blocks. One challenge is that modules containing the original blocks must still be refreshed to retain their data; because the channel is operating unsafely fast, refresh commands can be misinterpreted (e.g., to a reset command) and thus corrupt the original blocks. Another challenge is that all modules in the same channel share the same external clock signals; as such, operating a channel's Free Module(s) unsafely fast may seem to require all modules to operate unsafely fast. To address both challenges, Hetero-DMR puts the modules containing original blocks in self-refresh mode during read mode; modules in self-refresh mode refresh on their own without receiving any refresh command from CPU. In self-refresh mode, DRAM chips also ignore all external clock signals [20] and use their internal clocks [20], which abide by specification. Figure 8b shows read mode under Hetero-DMR.

The heterogeneous operations between read mode and write mode is another reason we name our idea Hetero-DMR.

B. Challenge 2: How to Quickly and Reliably Detect Errors in Copies Caused by Unsafely Fast Read Mode?

Using the original blocks for recovery when copies become erroneous requires Hetero-DMR to first detect erroneous copies. We note that current server memory modules contain ECC chips dedicated to storing ECC bytes to enable CPU to quickly detect errors on-the-fly with accessing memory [37], [38]. As such, Hetero-DMR uses these existing module-level ECC bytes to quickly detect errors while accessing copies.

Operating copies unsafely fast worsens memory errors; how to reliably detect (i.e., maintain the same SDC - silent data corruption - rate as a conventional system) the worse errors under the existing physical form factors of server DIMMs and CPU-memory interfaces is challenging. The naive approach of adding more ECC bits to reliably detect the worse errors requires adding more ECC chip(s) per module and ECC bits per memory bus and thus violates memory standards.

To address this challenge, we observe CPUs currently use the module-level ECC bits to both detect *and* correct errors [37], [38]; intuitively, using the ECC budget just for detection, instead of also correction, strengthens detection. As such, instead of adding more ECC, Hetero-DMR reliably detects errors in each copy by using all of each copy's existing ECC just for detection. ECC decoding typically involves two steps - error detection, followed by correction if errors are detected. When using ECC to only detect errors, Hetero-DMR stops ECC decoding after detecting error(s); Hetero-DMR skips the ECC correction step because ECC can miscorrect in the presence of too many errors and cause SDC. Instead, Hetero-DMR reads the original block to correct detected error in a copy (see Section III-C). Note that this optimization of stopping ECC decoding after detecting error(s) does not modify memory modules as module-level ECC decode and encode logic both reside in CPU [37], [38].

Hetero-DMR further enhances error detection by adopting Bamboo-ECC [58], which uses all 64 data bytes in a memory block to compute eight Reed-Solomon ECC bytes together. Using all eight ECC bytes only for detection can detect all errors affecting up to eight bytes, even if some or all errors occur in the ECC bytes themselves. In addition to detecting

data errors, Hetero-DMR also detects all address bus errors by using the address of a block and all data in the block to compute the ECC for the block, similar to [72].

It is possible to cause more than eight bad bytes in an accessed block when operating memory faster than specification (e.g., due to command bus errors); we refer to an error spanning more than eight bytes in a memory block as an 8B+ error. The eight ECC bytes in a copy cannot guarantee to detect 8B+ errors. However, the probability that eight Reed Solomon ECC bytes fail to detect 8B+ errors is very small - only $\frac{1}{2^{(8*8)}}$; for an error to go undetected, all 64 code bits recomputed from the erroneous 64B of data read from memory must coincidentally and exactly match the 64 code bits read from memory. In other words, a system would encounter one SDC after encountering $2^{8*8} = 18446744073709600000$ detected 8B+ errors. As such, we propose slowing down how quickly a system detects 18446744073709600000 8B+ errors, and thus encounter one SDC, by ceasing to exploit memory margins after seeing a threshold number of 8B+ errors.

For each one hour epoch, Hetero-DMR counts how many errors it has encountered during the epoch; if Hetero-DMR counts more than the threshold number of errors, it slows down memory to specification for the rest of that epoch. At the beginning of the next epoch, Hetero-DMR replicates every data block and operates memory faster than its specifications again. Hetero-DMR uses a per-hour threshold of $\frac{18446744073709600000}{one\ billion\ years\ expressed\ in\ hours} \approx 2100000$ errors[2] to limit the meantime to SDC to one billion years assuming the unreal worst case where every access incurs 8B+ error when operating faster than specification. In comparison, servers target a meantime to SDC of 1000 years [44]; as such, the system-level SDC overhead due to operating faster than specification under Hetero-DMR translates to one over one million (i.e., 1000 years/1 billion years). For all practical purposes, a system-level overhead of one over one million is no overhead.

C. Challenge 3: How to Correct Errors?

When detecting an error while accessing a copy, Hetero-DMR slows down the memory channel (i.e., the channel's CK_c/CK_t clock bits) to manufacturer specification to reliably read the copy's original block and use the latter's value to overwrite and thus correct the corrupted copy in DRAM (see Figure 8c). As such, Hetero-DMR only pays reliability tax for the $<$0.001% of accesses that are erroneous, similar to the ideal system (see Section I). Afterwards, Hetero-DMR speeds up the memory channel safely fast again to boost performance.

To correct normal errors (i.e., errors not due to operating faster than specification) in original blocks, Hetero-DMR uses ECC just like conventional systems. Note that both an original block and its copy can have the same ECC byte values because the optimization for detecting errors in copies (see Section III-B) only affects ECC decoding but not encoding; this allows Hetero-DMR to preserve the ability to broadcast writes to both the original block and its copy in one bus transaction to eliminate write bandwidth overhead (Section III-A).

[2]Under this threshold, Hetero-DMR can be active $\sim$100% of the time when considering the average error rate in an ambient temperature of 23 °C.

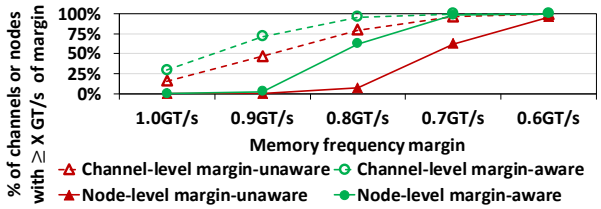


Fig. 11: Distribution of channel-level and node-level memory frequency margins.

D. Challenge 4: How to Mitigate Variability in Memory Frequency Margins?

1) Variability in a Channel: Different modules can have different margins (see Figure 2). Randomly choosing a module in a channel to operate unsafely fast can result in choosing the module with lowest frequency margin; this can limit the channel's maximum achievable frequency. To maximize a channel's achievable frequency, Hetero-DMR chooses the module with highest frequency margin in the channel to operate unsafely fast; we refer to the frequency margin of the chosen module as channel-level frequency margin.

We use Monte Carlo simulations to estimate the distribution of channel-level frequency margins under margin-aware selection; we also simulate a margin-unaware selection that chooses the first module in each channel to operate unsafely fast. We use the mean and standard deviation of the frequency margins in Figure 2a to create a normal distribution, similar to the prior work [71], which uses normal distribution when studying latency margins; for the reason described in Section II-B, the distribution only considers modules with 9 chips/rank. We use this normal distribution to randomly assign frequency margins to each module in the Monte Carlo simulations. Each experiment models a channel with two modules.

Figure 11 shows the distribution of channel-level frequency margins. 96% and 80% of channels have at least 0.8GT/s frequency margin under margin-aware selection and under margin-unaware selection, respectively.

2) Variability in a Node: Different channels in a node can have different frequency margins. Today's systems typically interleave data across many channels to balance bandwidth utilization; this can cause the slowest channel to become a bandwidth bottleneck that determines overall performance. For example, our Gem5 simulations show operating different channels in a node at different frequencies provides similar performance as operating all channels at the slowest channel's frequency. We refer to the lowest channel-level frequency margin in a node as node-level memory frequency margin.

The margin-aware selection (see Section III-D1) for mitigating variability in channel-level memory frequency margin also mitigates variability in node-level memory frequency margin. We use similar simulations (see Section III-D1) to estimate the distribution of node-level frequency margins; in these new simulations, each experiment models a node with

12 channels and two modules/channel. Figure 11 also shows the distribution of node-level frequency margins. Under the margin-aware selection, 62% and 98% of nodes have at least 0.8GT/s and 0.6GT/s frequency margins, respectively; they are 7% and 96% under the margin-unaware selection, respectively.

3) Variability in a HPC System: Different nodes in a HPC system can have different memory margins. A challenge is that HPC systems' job schedulers are currently unaware of different nodes' memory margins. When an MPI job runs on nodes with different memory margins, its total execution time can be determined by the slowest node with the lowest memory margin; as such, allocating both nodes with high frequency margins and nodes with low frequency margins to the same job essentially wastes the former's higher margins.

One solution to this issue is to enhance the system-wide job scheduler in HPC systems to group nodes with similar memory margins and schedule a job on nodes of the same group; we refer to it as margin-aware job scheduler. For example, the margin-aware job scheduler will group nodes in Figure 11 into three groups: 0.8GT/s, 0.6GT/s and 0GT/s; they occupy 62%, $98\% - 62\% = 36\%$ and $100\% - 98\% = 2\%$ of total nodes, respectively. To schedule a job that requests *X* nodes, the margin-aware job scheduler first determines the fastest group with at least *X* free nodes to fulfill the request; if every group have less than *X* free nodes, schedule the job on the fastest *X* free nodes among all groups. We implement the margin-aware job scheduler in Slurm [30], which is a popular HPC management software, by adding $\sim$30 lines of code; Section IV-C evaluate its the effectiveness.

E. Implementation Details

Activating and deactivating memory replication: Existing OS and hardware can work together to free up memory components (e.g., some banks, ranks, and modules) when memory utilization is low [27] to turn them off to save power. When Hetero-DMR sees half of the modules in a channel are free, it replicates every block in in-use modules to the free modules within the channel and then operates the latter as unsafely fast Free Modules to boost performance. When Hetero-DMR sees that less than half of the modules in the channel are free, it stops replication and operates all modules in the channel by manufacturer specification. While opportunistic memory replication can change the free modules' values, it does not require any new system-level handling to maintain correctness compared to the existing practice of turning off freed up modules, which also changes memory values.

Increasing Write Batch Size: Recall from Section III-A1, Hetero-DMR increases write batch size by 100X by proactively cleaning 12800 dirty blocks when a channel is in write mode; cleaning a dirty block means writing its value to memory and marking it as clean in LLC. One challenge is that cleaning a dirty block can cause an extra write if the block becomes dirty again after being cleaned; as such, Hetero-DMR first cleans least-recently used blocks as they are unlikely to be re-written prior to eviction.

Another challenge is that today's CPUs typically switch a channel to write mode when its write buffer becomes nearly full [48]; because it is small (e.g., 128 entries), a write buffer can become full before LLC has accumulated 12800 dirty blocks. To address this, we add a 128KB 64-way victim writeback cache per channel between LLC and the channel's write buffer, similar to [64]. Hetero-DMR caches each evicted dirty block in the writeback cache if the corresponding set has space and in the write buffer otherwise; this prevents the write buffer from quickly becoming full. While a channel is in write mode, Hetero-DMR drains the writeback cache's content to DRAM (by passing them through the write buffer) in addition to cleaning LLC. The writeback cache incurs small area overhead; for example, its area is 0.18% normalized to the total cache size (i.e., 68.25MB) of our test CPU. It also does not affect the memory command scheduler, which completely ignores the writeback cache; the scheduler only inspects the write buffer, just like conventional systems.

Handling permanent hardware faults: When a memory module develops a permanent yet ECC-correctable fault due to natural wear (e.g., a permanent column fault [74]), using the module to store copies can lead to costly repeated frequency transition to correct errors under Hetero-DMR. This issue can be addressed by dynamically remapping the copies to the good modules in the channel and remapping the original copies to the module with permanent fault.

Determining Margins: Hetero-DMR can borrow from [65] to profile a node's memory margins at boot time and to periodically re-profile when the node is idle. While [65] only talks about profiling tREFI, it can be extended to other parameters (e.g., frequency, tRCD, etc.). While Hetero-DMR requires profiling like [65], a primary difference is that [65] relies on profiling for both performance (i.e., to find faster parameters) and reliability, whereas Hetero-DMR relies on profiling only for performance, but not for reliability. For example, if errors become worse than profiled due to limited profiling duration or temperature spiking beyond the profiled temperature, Hetero-DMR can provide recovery by keeping the original blocks intact as Hetero-DMR only operates them according to manufacturer specification.

F. Discussion

Generality: Prior works [70], [76] report that the average memory utilization in Cloud systems is between 50% and 60%; as such, Hetero-DMR can also be a useful option in Cloud and data center, just like how CPU turbo-boost is useful in Cloud [4]. Hetero-DMR safely increases memory frequency when memory utilization is low, just like how CPU turbo-boost safely increases CPU frequency when CPU utilization (in terms of number of in-use cores) is low.

Our work only examines DDR4; DDR5 [15] is not available at the time of this writing. DDR5 may have similar frequency margins as DDR4. 3200MT/s DDR5 DRAM may have similar frequency margin as 3200MT/s DDR4 DRAM because the former operates at the same frequency as the latter. DDR5 DRAMs faster than 3200MT/s may have similar frequency

Cores	3.1GHz, 4-wide OoO, 2048 TLB entries, 224-entry ROB
L1$** **L1$ Prefetchers	Split Data-Inst, 64 kB, 8-way assoc, 3-cycle latency Stride (degree 2), Next-line (with auto turn-off)
L2$** **L2$ Prefetchers	1MB per core, 16-way assoc, 12-cycle latency Stride (degree 4), Next-line (with auto turn-off)
L3$	Size: Refer to Table III, 22ns latency
Memory Controller	DDR4, 4ranks/channel, 16-banks/rank, FR-FCFS scheduling policy with bank fairness, Hybrid page policy with 200 cycle timeout interval, XOR-based mapping function similar to Intel Skylake [67], Read queue: 256 entries/channel, Write queue: 128 entries/channel

TABLE IV: Simulated CPU and memory parameters.

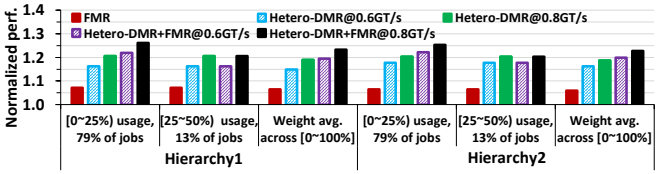


Fig. 12: Performance normalized to Commercial Baseline on average across six HPC benchmark suites. Figure 16 shows some per-benchmark normalized performance.

margins as 3200MT/s DDR5 DRAM because DDR5 JEDEC stipulates the same eye width for all DDR5 DRAMs, regardless of their data rate [15]; eye width is a measure of timing margin, which is the dual of frequency margin.

Impact on Aging: We do not think Hetero-DMR accelerates aging: First, Hetero-DMR does not increase operating voltage; increasing frequency without overvolting only causes temporary timing violation errors. Second, for all practical purposes, increasing frequency alone does not increase DIMM temperature (see Section II-A). Third, DRAM cells have practically infinite endurance [36].

IV. Evaluation

We simulate a single-node system with Hetero-DMR (see Section IV-A) and an HPC system with Hetero-DMR (see Section IV-C) to demonstrate its performance benefit. We also emulate Hetero-DMR on a real system to check our simulated node-level performance benefit (see Section IV-B).

A. Single-node Simulation

We simulate a single-node system with Hetero-DMR in Gem5's [43] full system mode with Ramulator [59] as the memory subsystem. We evaluate the same memory hierarchies described in Section II-B. We simulate the CPU [13] in our test machine used in Section II; Section IV-B cross-validates our simulated performance against the real CPU's performance. Table IV summarizes the microarchitecture configurations. We evaluate the same benchmarks in Figure 5. We first use the KVM CPU in Gem5 to fast forward each benchmark to pass over its initialization stage. Next, we warm up caches via 15ms of atomic simulation and the branch predictors and prefetchers via 2ms of cycle-accurate simulation. Lastly, we perform 20ms of cycle-accurate simulation to evaluate performance.

We evaluate the following memory systems:

Commercial Baseline. This is a conventional memory system that provides the same performance regardless of memory usage. This system operates at manufacturer specification (see Manufacturer-specified Setting in Table II). Since Hetero-DMR has a 128KB writeback cache per channel (see Section III-E), we also add a 128KB writeback cache per channel to this baseline system for fair comparison; this improves the average performance of the baseline by 1%.

Hetero-DMR. Its performance depends on the node-level memory frequency margin. Hetero-DMR uses 0.8GT/s and 0.6GT/s node-level memory frequency margins (see in Section III-D). Hetero-DMR uses the memory latency margin setting described in Table II "Setting to Exploit Freq+Lat Margins". We increase read-write turnaround latency to 1us to model the latency of scaling memory frequency. After draining the writeback cache during write mode, Hetero-DMR notifies LLC to clean 12800 dirty cachelines. When memory usage is >50%, Hetero-DMR falls back to the same behavior and performance as Commercial Baseline.

Free-memory-aware Baseline: FMR [64] stores each block and its copy in two ranks and accesses the one that is currently in the faster state (e.g., in row buffer) to reduce latency; it efficiently keeps the original block and its copy up-to-date by broadcasting each write request to both. FMR uses the same memory setting as Commercial Baseline.

Hetero-DMR+FMR We also apply Hetero-DMR to FMR and call this Hetero-DMR+FMR. When memory utilization is <25%, Hetero-DMR+FMR stores two copies for every memory block, unlike Hetero-DMR alone, which stores only one copy. Hetero-DMR+FMR stores both copies in a Free Module and operates the Free Module at the same unsafely fast setting as Hetero-DMR. Hetero-DMR+FMR uses FMR's algorithm to dynamically select which of the two copies to read from to further reduce effective memory latency. When memory utilization is $> 25\%$, Hetero-DMR+FMR regresses to the behavior and performance of Hetero-DMR alone.

Results: Figure 12 shows performance normalized to Commercial Baseline under memory Hierarchy1 and Hierarchy2. Under each memory hierarchy, we present three memory usage scenarios (e.g., [0∼25%)). Under each memory usage scenario, each bar represents the normalized performance of each memory design on average across six HPC benchmark suites. The "[0∼100%]" memory usage bucket refers to the weighted average normalized performance across all memory usage scenarios (i.e., [0∼25%), [25∼50%), and [50∼100%]); the weights are the fraction of jobs having the corresponding memory usage (see Figure 1).

Across two memory frequency margins, Hetero-DMR provides 19% weighted average performance improvement over Commercial Baseline under memory Hierarchy1; the weights are 62% and 36% for 0.8GT/s and 0.6GT/s frequency margins respectively (see Section III-D).

Hetero-DMR improves performance by 18% over Commercial Baseline on average across two hierarchies; this average simultaneously considers the weights of different memory usage scenarios and different node-level memory frequency margins. Compared to FMR, the corresponding performance improvement of Hetero-DMR+FMR is 15%.

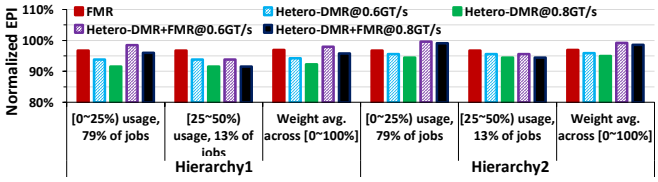


Fig. 13: EPI normalized to Commercial Baseline.

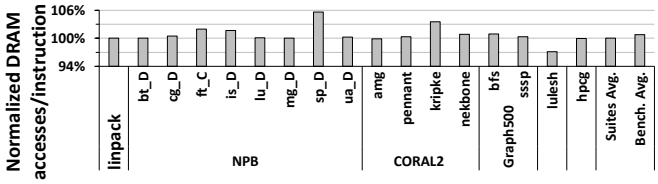


Fig. 14: Normalized DRAM accesses per instruction.

Figure 13 shows system-level (i.e., CPU+DRAM) Energy Per Instruction (EPI) normalized to Commercial Baseline. Although Hetero-DMR increase DRAM write power due to writing twice for each memory write request, it still improves EPI by 6% on average across two memory hierarchies. The reasons are: First, CPU idle power dominates dynamic power; Hetero-DMR improves CPU idle energy by improving performance, which outweighs the energy overheads of extra writes. Second, DRAM power relative to system power has been decreasing; for example, memory contributes 18% to system power in 2018 [42]. Third, because write bandwidth utilization is 15% of total bandwidth utilization on average (see Figure 15), write contributes to <15% of DRAM power. Although Hetero-DMR+FMR writes triple for each write request, it still shows similar average EPI as FMR.

The experiments in Figure 12 also simulate Hetero-DMR's write bandwidth overhead due to cleaning dirty cachelines (see Section III-A1); we measure this overhead as extra DRAM accesses per instruction normalized to Commercial Baseline. Figure 14 shows per-bench normalized DRAM accesses/instruction of Hetero-DMR+FMR@0.8GT/s over Commercial Baseline under Hierarchy1; they are similar across all Hetero-DMR and Hetero-DMR+FMR under both hierarchies. On average across six benchmark suites, the overhead is <1%.

B. Silicon Corroboration

To check our Gem5 setup, we simulate "Exploit Frequency+Latency Margins" (see Figure 5) to compare its simulated performance to its real-system performance, which is given in Figure 5. Figure 16 shows its simulated performance normalized to simulated Commercial Baseline and its real-system performance normalized to real-system Commercial Baseline; the average difference is 2%.

To check Hetero-DMR's simulated performance benefit (see Section IV-A), we use the test machine in Section II-B to emulate each benchmark's execution time under Hetero-DMR as: $exec\_time@unsafely\_fast - wr\_time@unsafely\_fast + wr\_time@safely\_slow$, where $exec\_time@unsafely\_fast$ is benchmark execution time measured on the real system under "Exploit

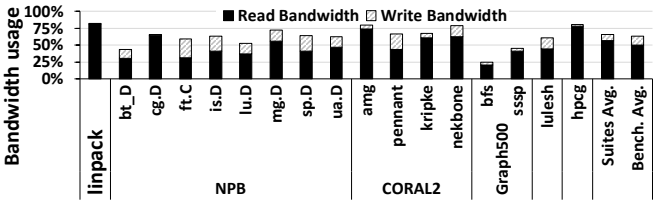


Fig. 15: Real-system memory bandwidth characterization.

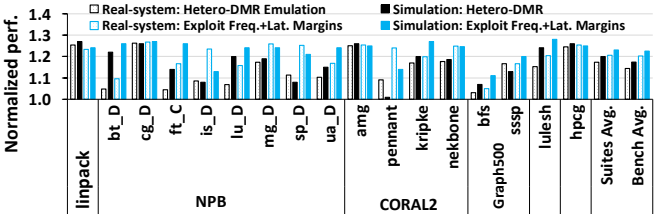


Fig. 16: Silicon corroboration under Memory Hierarchy1.

Frequency+Latency Margins"; wr_time is the time that the benchmark writes to DRAM when operating memory faster (i.e., $@unsafely\_fast$) or at spec (i.e., $@safely\_slow$). We model wr_time as $written\_data/bandwidth$. While this wr_time formula only accounts for bandwidth, we use it because the latency of individual writeback to DRAM has little impact on write time since all writebacks are independent, unlike read latency, which can stall subsequent reads from the same core. We use perf tool [28] to profile the total amount of data written to DRAM and the bandwidth during benchmark execution; Figure 15 shows the average bandwidth utilization of each benchmark when operating memory at manufacturer specification under Memory Hierarchy1.

We model the fact that Hetero-DMR reads only half of the modules of a channel in the common case (see Section III-A2). As such, our real-system experiments use two ranks/channel to emulate Hetero-DMR's performance in a system with four ranks/channel. Note that our simulations in Figure 12 model a system with four ranks/channel (see Table IV).

Figure 16 shows both Hetero-DMR's simulated performance, which is extracted from Figure 12, and emulated performance; both are normalized to Commercial Baseline. The average performance benefits are similar.

Compared to "Exploit Frequency+Latency Margins", Hetero-DMR provides 3% and 2% lower average performance benefit under emulation and under simulation, respectively; this is a small cost for rigorously maintaining reliability. Hetero-DMR is slightly faster than "Exploit Frequency+Latency Margins" for some benchmarks (e.g., linpack, amg, and hpcg). This is because these benchmarks perform better under two ranks/channel than under four ranks/channel in both real-system experiment and simulation. Recall that "Hetero-DMR" is emulated using two ranks/channel, whereas Exploit Frequency+Latency Margins uses four ranks/channel.

C. System-wide Simulation

Our single-node evaluations (see Sections IV-A and IV-B) have two limitations: First, they do not consider the performance impact of the system-level solution described in

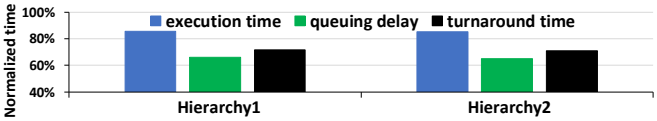


Fig. 17: System-wide evaluation of job execution time, queuing delay, and turnaround time.

Section III-D3. Second, they do not consider each job's queuing delay, which can significantly affect the job's overall performance. To address these limitations, we also simulate a HPC system with Hetero-DMR.

We simulate the benefit of adding Hetero-DMR to a Slurm-managed HPC system via Slurmsim [73]. We use Slurm configurations from an in-production HPC system – Grizzly [10], [29]; it has 1490 nodes with 36 physical cores and 128GB memory per node. We feed four months of 58K Grizzly job traces into Slurmsim. The overall node utilization (i.e., $\sum^{all\_jobs}(job\_execution\_time \times number\_of\_nodes\_of\_the\_job)/(total\_number\_of\_nodes \times 4months)$) is ~78%.

To simulate an HPC system with Hetero-DMR, we simulate each job's execution time by scaling it according to the performance of Hetero-DMR (as opposed to Hetero-DMR+FMR) normalized to Commercial Baseline. Because Hetero-DMR only benefits jobs with $< 50\%$ memory utilization, we use the job-level memory utilization distribution in Grizzly (see Figure 1) to probabilistically scale each job's execution time. The simulated HPC system uses the margin-aware job scheduler we implement in Slurm (see Section III-D). For example, the performance of Hetero-DMR@0.8GT/s normalized to Commercial Baseline under memory Hierarchy1 is 121%, on average; so for a job that occupies only nodes with 0.8GT/s memory frequency margin, we scale the job's execution time as $scaled\_execution\_time = original\_execution\_time \cdot \frac{1}{121\%}$. When a job occupies nodes with different data rates, we scale the job's execution time according to the performance of Hetero-DMR at the lowest node-level frequency margin among all of the job's occupied nodes.

Results: Figure 17 shows the normalized job execution time, queuing delay, and turnaround time over a conventional HPC system under two memory hierarchies; the turnaround time of a job is its queuing delay plus its execution time.

Hetero-DMR reduces job execution time by 15% on average across two memory hierarchies; this translates to 1.17x average speedup across the two hierarchies.

On average across two hierarchies, Hetero-DMR provides 1.4x turnaround-time-level speedup. It is higher than execution-time-level speedup because by making each node run 17% faster, Hetero-DMR also reduces average job queuing delay by 34% according to our simulation. We were initially surprised by this result; to check whether it is likely that making each node 17% faster can reduce job queuing delay by 34%, we re-simulate the conventional HPC system with 17% additional nodes. Our simulation shows that having 17% more nodes reduces average queuing delay by 33%, which is close to the 34% reduction due to making each node 17% faster.

To study the performance benefit of the margin-aware job scheduler (see Section III-D), we also evaluate Hetero-DMR with Slurm's default job scheduler, which allocates nodes to each job without considering their memory frequency margins. On average across two memory hierarchies, the margin-aware job scheduler provides 1.2x turnaround-time-level speedup over Slurm's default job scheduler.

V. Related Works and Existing/Prior Approaches

Memory Profiling. Memory companies and researchers have used memory profiling to improve memory performance in the context of consumer products (e.g., desktop, mobile systems). Memory profiling tests the error rates of memory at optimistic configurations (e.g., frequency, latency) to identify the memory's faster configurations that still have a low probability of triggering errors. Some memory module manufacturers (e.g., G.Skill, CORSAIR) use profiling to identify chips that can operate at a high data rate (e.g., 4000MT/s) to produce 'overclocked' unbuffered DIMMs for desktop systems. Compared to standard modules, these overclocked modules cost many times (e.g., ~2.65x [35]) as much due to the high cost of testing for memory manufacturers [40]. There are also no 'overclocked' server memory modules, the focus of this work, available on the market.

Some prior works [47], [60], [62] propose profiling regular (i.e., not 'overclocked') consumer memories (e.g., SODIMMs, LPDDR) directly in the end system to reduce cost compared to the commercial solutions above. However, profiling memory in the end-system lacks environmental stress test equipment, such as thermal and humidity-controlled test chambers; as such, operating memory at reduced latency based on end-system profiling results makes the memory system more vulnerable to occasional errors due to time-dependent changes caused by humidity, thermal cycling, corrosion, electrostatic discharge, power spike/voltage noise, dust etc. [32]. While it may be OK for consumer systems to experience occasional errors, they are unacceptable for server systems which have higher reliability requirement. Moreover, while these prior works exploit *latency* margins in the context of consumer products, this paper focuses on safely exploiting *frequency* margin in the context of server systems.

Existing Server Memory Protection. Server memories have ECC chips to protect against natural hardware failures during system lifetime. Simply reusing existing server memory ECC cannot achieve our goal - maintain the same level of reliability as before while exploiting memory frequency margin. Operating memory faster than specification increases fault rate; intuitively, when keeping ECC the same, increasing fault rate increases uncorrected error rate and thus reduces system reliability. Note that our test machine uses RDIMMs' ECC chips for ECC; it still encounters many uncorrected errors when operating at high frequency (see Figure 6).

A possible alternative is to add more ECC chips or bits per rank to strengthen existing ECC. However, adding more ECC chips or bits is costly as it requires custom DIMMs, modified CPU-memory interface standards, and additional

area. Moreover, it is difficult to decide how many chips or bits to add because it depends on the maximum number of bits per access that get corrupted when operating memory faster than its manufacturer specification. This number can vary across modules of the same model number, across different model numbers, brands, and generations.

Emerging memory technologies. High Bandwidth Memory (HBM) [3] and DDR5 [15] also improve bandwidth compared to mainstream server memory (e.g., DDR4). HBM is primarily a GPU memory. However, adopting HBMs in CPUs - the focus of this paper - is costly due to HBM's high price and high in-package area overhead; doing so is also an overkill as HBMs are designed to satisfy GPU bandwidth requirement, which is orders of magnitude higher than CPU. High-speed DDR5 is still several years from market; at that time, CPUs may also increase their memory bandwidth requirement (e.g., due to core count scaling) and, therefore, will still benefit from the additional bandwidth Hetero-DMR provides.

Exploiting CPU Frequency Margin. Prior works have exploited frequency margin in CPUs (e.g., [55]). Exploiting frequency margin in CPU requires different approaches from exploiting frequency margins in memory because CPUs have vastly different architectures from memory. For example, CPUs have no concept of read mode and write mode and thus do not operate heterogeneously for read mode and write mode as does Hetero-DMR.

VI. Conclusion

This paper performs the first public broad study to characterize frequency margin for server memory. We find that under standard voltage and cooling, commodity RDIMMs can operate, on average, 27% faster than manufacturer specification without errors for 99.999%+ of memory accesses. We propose Hetero-DMR to safely exploit memory margins in HPC systems to improve both bandwidth and latency. On average across six HPC benchmark suites and across two memory hierarchies, Hetero-DMR improves node-level performance by 18% and 15% over a Commercial Baseline and over FMR, respectively. System-wide simulations show Hetero-DMR provides 1.4x average speedup on job turnaround time over a conventional HPC system.

Acknowledgment

This work was supported by National Science Foundation (NSF) under grants 1942590 and 1919113 and by Los Alamos National Laboratory under the CSSE ASC Program. We thank Matthew Hicks and Michael Moukarzel from Virginia Tech for their help with thermal chamber experiments. We thank Advanced Research Computing at Virginia Tech for providing computational resources. We thank the reviewers for providing helpful feedback to improve the paper.

This manuscript has been approved for unlimited release and has been assigned LA-UR-21-23881. This work has been co-authored by two employees of Triad National Security, LLC which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes.

The authors would like to thank Jerry Ahrens from AMD for his technical feedback. The authors would like to thank AMD internal reviewers, Mike Ignatowski and Vilas Sridharan, for their review comments. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] "Amazon AWS compute optimized instances." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html.
[2] "Amazon AWS for HPC." https://aws.amazon.com/hpc/.
[3] "AMD HBM," https://www.amd.com/en/technologies/hbm.
[4] "AWS and Intel," https://aws.amazon.com/intel/.
[5] "Basics of Clock Jitter," https://www.mouser.com/pdfdocs/src-tutorials/Basics-of-Clock-Jitter.pdf.
[6] "CORAL2," https://asc.llnl.gov/coral-2-benchmarks/.
[7] "GIGABYTE C621 AORUS XTREME motherboard," https://www.gigabyte.com/us/Motherboard/C621-AORUS-XTREME-rev-10/sp#sp.
[8] "Google Cloud for HPC." https://cloud.google.com/solutions/hpc.
[9] "Graph500," https://graph500.org/.
[10] "Grizzly HPC system," https://www.top500.org/system/178972/.
[11] "HPC workloads for running on Microsoft Azure." https://azure.microsoft.com/en-us/solutions/high-performance-computing/#features.
[12] "Intel Cache Allocation Technology," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.
[13] "Intel Xeon W-3175X Processor," https://ark.intel.com/content/www/us/en/ark/products/189452/intel-xeon-w-3175x-processor-38-5m-cache-3-10-ghz.html.
[14] "JEDEC DDR4 SDRAM," https://www.jedec.org/standards-documents/docs/jesd79-4a.
[15] "JEDEC DDR5 SDRAM," https://www.jedec.org/standards-documents/docs/jesd79-5.
[16] "LANL DIMM temperature measurements," ftp://hpc-ftp.lanl.gov/data/operational/LA-UR-20-24989-sedc_data.h5.
[17] "LINPACK," http://www.netlib.org/linpack/.
[18] "Memory stress test," https://www.simmtester.com/News/PublicationArticle/164.
[19] "Message Passing Interface." https://computing.llnl.gov/tutorials/mpi/.
[20] "MICRON 8Gb DDR4 SDRAM." https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.
[21] "MICRON DDR4 SDRAM system-power calculator," https://www.micron.com/support/tools-and-utilities/power-calc/.
[22] "MICRON Technical Note," https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn0018.pdf.
[23] "MICRON Technical Note: Calculating Memory Power for DDR4 SDRAM," https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf.
[24] "Microsoft Azure for High-Performance Computing." https://azure.microsoft.com/en-us/solutions/high-performance-computing/.
[25] "Microsoft Azure high performance computing VM sizes." https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc?toc=/azure/virtual-machines/linux/toc.json&bc=/azure/virtual-machines/linux/breadcrumb/toc.json.
[26] "MIRA HPC system," https://www.top500.org/system/177718/.
[27] "PASR: Partial Array Self-Refresh Framework," https://lwn.net/Articles/478049/.
[28] "perf tool," https://perf.wiki.kernel.org/index.php/Main_Page.
[29] "Raw memory usage measurement for HPC systems in LANL." https://usrc.lanl.gov/data/LA-UR-19-28211.php.

[30] "SLURM," https://slurm.schedmd.com/documentation.html.

[31] "stressapptest." https://github.com/stressapptest/stressapptest.

[32] "Stringent Tests from ICs to Modules Ensure DRAM Reliability," https://www.atpinc.com/blog/dram-testing-module-chips-ic-burn-in-quality-characteristics.

[33] "TestEquity 123H Temperature/Humidity Chamber," https://www.testequity.com/product/TestEquity-123H-Temperature-Humidity-Chamber-North-America-Version-17267-1.

[34] "Tolerance level in Linux kernel," https://www.kernel.org/doc/Documentation/x86/x86_64/machinecheck.

[35] "VENGEANCE® LPX 32GB (2 x 16GB) DDR4 DRAM 4000MHz C19 Memory Kit - Black," https://www.corsair.com/us/en/Categories/Products/Memory/VENGEANCE-LPX/p/CMK32GX4M2F4000C19.

[36] "What Is DRAM's Future?" https://semiengineering.com/what-is-drams-future/.

[37] "Intel E7500 Chipset MCH Intel x4 Single Device Data Correction (x4 SDDC) Implementation and Validation," 2002, https://www.intel.com/content/dam/doc/application-note/e7500-chipset-mch-x4-single-device-data-correction-note.pdf.

[38] "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," 2013, http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf.

[39] "DDR5 vs DDR4 – All the Design Challenges & Advantages," January 2021, https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets/.

[40] Z. Al-Ars, "DRAM fault analysis and test generation," *Ph.D. dissertation*, 2005.

[41] D. Bailey, E. Barszcz, J. Barton, D. S. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. A. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks summary and preliminary results," *Proceedings of Supercomputing'91*, pp. 158–165, 1991.

[42] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition.*, October 2018.

[43] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[44] D. C. Bossen, "CMOS Soft Errors and Server Design," *IEEE Reliability Physics Tutorial Notes*, 2002.

[45] K. W. Cameron, "Energy Efficiency in the Wild: Why Datacenters Fear Power Management," *Computer*, vol. 47, no. 11, pp. 89–92, 2014.

[46] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM performance by parallelizing refreshes with accesses," in *Proceedings of HPCA'14*, 2014.

[47] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *Proceedings of SIGMETRICS'16*, 2016.

[48] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged Reads: Mitigating the impact of DRAM writes on DRAM reads," in *Proceedings of HPCA'12*, 2012.

[49] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *Proceedings of SC '18*, 2018.

[50] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *Proceedings of ICAC'11*, 2011.

[51] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *Proceedings of MICRO'12*, 2012.

[52] Q. Deng, L. Ramos, R. Bianchini, D. Meisner, and T. Wenisch, "Active Low-Power Modes for Main Memory with MemScale," *IEEE Micro*, vol. 32, no. 3, pp. 60–69, 2012.

[53] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance Conjugate-gradient Benchmark," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 3–10, Feb. 2016.

[54] L. Ecco and R. Ernst, "Tackling the Bus Turnaround Overhead in Real-Time SDRAM Controllers," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1961–1974, Nov 2017.

[55] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Proceedings of MICRO'03.*, 2003.

[56] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *Proceedings of MICRO'19*, 2019.

[57] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, August 2013.

[58] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," in *Proceedings of HPCA'15*, 2015.

[59] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016.

[60] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *Proceedings of SIGMETRICS'17*, 2017.

[61] H. Li and L. Wolters, "Towards A Better Understanding of Workload Dynamics on Data-Intensive Clusters and Grids," in *Proceedings of IPDPS'07*, 2007.

[62] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *Proceedings of ISCA'13*, 2013.

[63] H. Luo, T. Shahroodi, H. Hassan, M. Patel, A. G. Yaglikçi, L. Orosa, J. Park, and O. Mutlu, "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in *Proceedings of ISCA'20*, 2020.

[64] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying Memory Underutilization in HPC Systems and Using It to Improve Performance via Architecture Support," in *Proceedings of MICRO'19*, 2019.

[65] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *Proceedings of ISCA'17*, 2017.

[66] I. Peng, R. Pearce, and M. Gokhale, "On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems," in *Proceedings of SBAC-PAD'20*, 2020.

[67] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *Proceedings of USENIX Security'16)*, 2016.

[68] D. Redmayne, E. Trelewicz, and A. Smith, "Understanding the effect of clock jitter on high speed ADCs."

[69] D. A. Reed and J. Dongarra, "Exascale Computing and Big Data," *Commun. ACM*, vol. 58, no. 7, p. 56–68, Jun. 2015.

[70] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of SoCC'12*, 2012.

[71] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "VARIUS: A Model of Process Variation and Resulting Timing Errors for Microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3–13, 2008.

[72] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient Die-Stacked DRAM Caches," in *Proceedings of ISCA'13*, 2013.

[73] N. A. Simakov, R. L. DeLeon, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, A. K. Patra, and T. R. Furlani, "Slurm Simulator: Improving Slurm Scheduler Performance on Large HPC Systems by Utilization of Multiple Controllers and Node Sharing," in *Proceedings of PEARC'18*, 2018.

[74] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of ASPLOS'15*, 2015.

[75] M. Strevell, D. Lambiaso, A. Brendamour, and T. Squillo, "Designing an Energy-Efficient HPC Supercomputing Center," in *Proceedings of ICPP'19: Workshops*, 2019.

[76] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of EuroSys'20*, 2020.

[77] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé, "Main Memory in HPC: Do We Need More or Could We Live with Less?" *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 3:1–3:26, Mar. 2017.