Addressing Service Interruptions in Memory with Thread-to-Rank Assignment

Manjunath Shevgoor
University of Utah
shevgoor@cs.utah.edu

Rajeev Balasubramonian
University of Utah
rajeev@cs.utah.edu

Niladrish Chatterjee
NVIDIA
nchatterjee@nvidia.com

Jung-Sik Kim
Samsung Electronics
jungsik1.kim@samsung.com

***Abstract*—**

In future memory systems, some regions of memory will be periodically unavailable to the processor. In DRAM systems, this may happen because a rank is busy performing refresh. In non-volatile memory systems, this may happen because a rank is busy draining long-latency writes. Unfortunately, such service interruptions can introduce stalls in all running threads. This is because the operating system spreads the pages of a thread across all memory ranks. Therefore, the probability of a thread accessing data in an unavailable rank is high. This is a performance artifact that has previously not been carefully analyzed. To reduce these stalls, we propose a simple page coloring mechanism that tries to minimize the number of ranks over which a thread's pages are spread. This approach ensures that a service interruption in a single rank only stalls a subset of threads; non-stalled threads even have the potential to run faster at this time because of reduced bus contention. Our analysis shows that this approach is more effective than recent hardware-based mechanisms to deal with such service interruptions. For example, when dealing with service interruptions because of DRAM refresh, the proposed page coloring approach yields an execution time that is 15% lower than the best competing hardware approach.

I. Introduction

Future memory systems will frequently perform long-latency operations that will keep regions of memory busy for hundreds of nano-seconds. During these operations, these regions are unavailable to service read requests from the processor. Examples of such service interruptions include: (i) Refresh operations in high-capacity DRAM packages [30], (ii) A burst of writes in emerging non-volatile memories (NVMs) [2], [34] , (iii) Bulk page operations [39] such as memcopy, memset, etc.

In this paper, we first analyze this phenomenon and observe that the default OS page mapping policy is the root cause of refresh-induced stalls. We then devise a simple page mapping policy to mitigate this effect. Our analysis shows that such a software approach is more effective than other hardware-based approaches to combat the refresh problem. Our solution is also effective in mitigating stalls from long-latency NVM writes. The paper therefore makes a contribution that will be useful to the two segments that will compete for memory market share in the coming decade.

The paper primarily focuses on service interruptions from DRAM refresh. DRAM technology is expected to scale for another generation or two [14] and will be used widely in most systems for at least another decade. In addition, DRAM vendors will likely rely on 3D stacking (with or without TSVs) to boost per-package DRAM capacities for a few more generations. Many studies have shown that as DRAM capacities in a package grow, the package has to spend larger amounts of time performing refresh [43], [32], [30]. The refresh rate has to double if the operating temperature exceeds 85° C, a common scenario in many servers [25], [22], [38], [27]. At future technology nodes, the DRAM cell aspect ratio introduces a trade-off between hard error rates and data retention times [33]. DRAM cells may therefore offer lower retention times (and higher refresh rates) to keep hard error rates in check. For these reasons, refresh is viewed as an important challenge for future high capacity DRAMs.

In anticipation of this possible bottleneck, JEDEC is already taking steps to reduce the impact of the refresh process. The DDR4 standard includes a new fine granularity refresh (FGR) operation that can help reduce queuing delays for read requests from the CPU [15], [30]. However, we show later that the practical benefit from FGR is very small.

In addition to the steps taken by JEDEC, there has been a flurry of research papers attempting to alleviate refresh overheads. The approaches to this problem include the following: scheduling refresh during predicted idle times [43], avoiding refresh of recently read/written rows [9], reducing the number of rows refreshed based on retention characteristics of DRAM cells [23], providing error correction capabilities for leaky DRAM cells [20], pausing the refresh process to service processor read requests [32], and overlapping a lightweight refresh process with regular processor requests [49], [3]. Some recent solutions also involve the OS and application: ESKIMO [13] reduces refresh energy by not refreshing parts of the memory system that have been freed, RAPID [45] preferentially allocates areas of memory that have longer retention times, FLIKKER lowers the refresh rate for application pages that can tolerate lower fidelity [42].

Our analysis of state-of-the-art refresh techniques shows that there remains significant room for improvement. Modern systems implement a *Staggered Refresh* process where each memory rank is refreshed at a different time. Because refresh is a current-intensive process, this helps reduce the system's peak power requirement [30] and hence, system cost. This means that at most times, a portion of the memory system is unavailable to service processor requests. Modern memory controllers and operating sytems also scatter an application's pages across the entire memory system to boost memory system parallelism. This implies that if some memory rank is refreshing, all threads in the system could stall because they may all eventually need data that resides in that rank. In the example illustration in Figure 1, a refresh to Rank 1 stalls threads T1, T2, and T3 in the baseline. The combination of Staggered Refresh and page scattering results in a 36.9% increase in execution time over an idealized system with no refresh penalty (assuming 2 channels, 2 ranks per channel and 32 Gb chips at high temperature). State-of-the-art hardware-intensive refresh techniques are able to yield an execution time reduction of at most 4.1%, relative to this baseline.

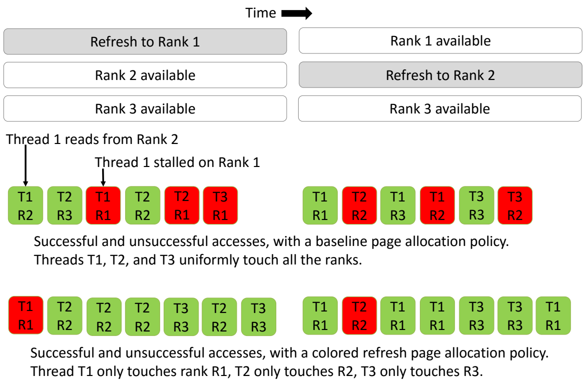

Time
Refresh to Rank 1
Rank 2 available
Rank 3 available
Rank 1 available
Refresh to Rank 2
Rank 3 available
Thread 1 reads from Rank 2
Thread 1 stalled on Rank 1
T1 R2
T2 R3
T1 R1
T2 R2
T2 R1
T3 R1
T1 R1
T2 R2
T1 R3
T1 R2
T3 R3
T3 R2
Successful and unsuccessful accesses, with a baseline page allocation policy.
Threads T1, T2, and T3 uniformly touch all the ranks.
T1 R1
T2 R2
T2 R2
T2 R2
T3 R3
T2 R2
T3 R3
T1 R1
T2 R2
T1 R1
T1 R1
T3 R3
T3 R3
T1 R1
Successful and unsuccessful accesses, with a colored refresh page allocation policy.
Thread T1 only touches rank R1, T2 only touches R2, T3 only touches R3.

Fig. 1. Stalls with baseline and Rank Assignment policies.

We also examine the write bottleneck in future NVMs. Multiple non-volatile cell technologies (e.g., PCM and Memristors) are expected to soon be mainstream. Both technologies suffer from long write latencies of several hundred nanoseconds [19], [36], [37], [17]. Typically, memory controllers defer writes and perform writes in bursts that can occupy the memory channel and the corresponding banks for many hundreds of cycles. Several recent papers have therefore attempted to optimize the write process in NVMs [36], [37], [17], [6], [48]. Similar to the refresh process, the regions of memory performing writes are unavailable for a long time, potentially causing stalls in all threads.

We design a simple solution, *Rank Assignment*, that can be implemented entirely in the OS page allocator and that is more effective than state-of-the-art hardware schemes that target service interruptions from refreshes and write drains. In the baseline described above, a single rank's refresh or write drain can stall every thread in the system. To prevent this, the OS allocates pages such that pages from one thread are assigned to the fewest number of ranks (ideally just one rank). This thread is only stalled when its rank is refreshing or draining writes.

In Figure 1, a refresh to Rank 1 only stalls thread T1 when using the Rank Assignment page allocator. While other ranks are refreshing, this thread may even see a performance boost because of lower contention for the shared memory channel. The proposed page allocator therefore isolates the negative impact of refresh and write drains to a single thread while accelerating other threads assigned to that channel.

Traditionally, the OS and memory controller have adopted policies that spread requests from a thread across the entire memory system. We show that such policies must be questioned in the modern era and that it is better to co-locate an application's pages in a single rank. We further show that even an approximate mapping of pages to ranks is highly beneficial, thus lowering the burden on the OS.

The downside of this approach is that a thread cannot exploit rank-level parallelism; this has a very small performance penalty in the future because DDR4 provides high bank-level parallelism (16 banks per rank) [24].

Compared to the best competing hardware approach, Rank Assignment yields an execution time that is 15% lower, while requiring no changes to the memory controller, DDR standard, and memory chips.

II. Background

A. DRAM Basics

A modern high-performance processor typically has two to four DDR3 memory Controllers (MC). Each MC controls one 64-bit wide DDR3 channel and the 1-2 DIMMs connected to that channel. A DIMM has 1-4 ranks. Each rank consists of a collection of DRAM chips that together have an output width of 64 bits. A 64-byte cache line is fetched from a single rank with eight 64-bit transfers (a burst of eight). A rank is composed of eight independent banks in DDR3 and 16 banks in DDR4. Thus, a single channel can support multiple ranks and tens of banks, i.e., a high amount of memory system parallelism.

A single Activate command to a bank brings an entire row into a row buffer. Cache lines are read from or written to this row with Column-Reads Or Column-Writes. The bank must be Precharged before a new row can be fetched into the row buffer. The memory controller schedules these commands to all the banks while carefully satisfying all timing constraints.

B. Retention time and tREFI

The charge on a DRAM cell weakens over time. The DDR standard requires every cell to be refreshed within a 64 ms interval, referred to as the *retention time*. At temperatures higher than 85° C (referred to as Extended Temperature range), the retention time is halved to 32 ms to account for the higher leakage rate. The refresh of the entire memory system is partitioned into 8,192 smaller refresh operations. One such refresh operation has to be issued every 7.8 μs (64 ms/8192). This 7.8 μs interval is referred to as the *refresh interval, tREFI*. The DDR3 standard requires that 8 refresh operations be issued within a time window equal to 8×tREFI, giving the memory controller some flexibility when scheduling these refresh operations. Refresh operations are issued at rank granularity in DDR3 and DDR4.

We next describe the internals of the refresh operation.

C. tRFC and recovery time

Upon receiving a refresh command, the DRAM chips enter a refresh mode that has been carefully designed to perform the maximum amount of cell refresh in as little time as possible. During this time, the current carrying capabilities of the power delivery network and the charge pumps are stretched to the limit. The operation lasts for a time referred to as the *refresh cycle time, tRFC*. Towards the end of this period, the refresh process starts to wind down and some recovery time is provisioned so that the banks can be precharged and charge is restored to the charge pumps. Providing this recovery time at the end allows the memory controller to resume normal operation at the end of tRFC. Without this recovery time, the memory controller would require a new set of timing constraints that allow it to gradually ramp up its operations in parallel with charge pump restoration.

D. Performance penalty from refresh

On average, in every tREFI window, the rank is unavailable for a time equal to tRFC. So for a memory-bound application on a 1-rank memory system, the expected performance degradation from refresh is tRFC/tREFI. In reality, the performance degradation can be a little higher because directly prior to the refresh operation, the memory controller wastes some time precharging all the banks. Also, right after the refresh operation, since all rows are closed, the memory controller has to issue a few Activates to re-populate the row buffers. The performance degradation can also be lower than the tRFC/tREFI ratio if the processors can continue to execute independent instructions in their reorder buffer or if there are cache hits while the memory system is unavailable.

E. DRAM scaling

Refresh overheads are expected to increase dramatically in the future as DRAM chip capacities increase. Table I shows this scaling trend [30]. The number of cells that must be refreshed in every tRFC increases linearly with capacity. Therefore, we also see a roughly linear increase in tRFC. In future 32 Gb chips, the tRFC is as high as 640 ns, giving a tRFC/tREFI ratio of 8.2%. At high temperatures, this ratio doubles to 16.4%. tREFI will also reduce if DRAM cell capacitances reduce in the future [33]. In a 3D stacked package, the number of cells increases without a corresponding increase in pin count and power delivery [40] – this too results in a high tRFC.

Chip Capacity (Gb)	t_{RFC} (ns)	$t_{RFC_{1x}}$ (ns)	$t_{RFC_{2x}}$ (ns)	$t_{RFC_{4x}}$ (ns)
8	350	350	240	160
16	480	480	350	240
32	640	640	480	350
t_{REFI}	7800	7800	3900	1950

TABLE I
REFRESH LATENCIES FOR HIGH DRAM CHIP CAPACITIES [30].

F. Fine Granularity Refresh in DDR4

In response to these high tRFC refresh times, the DDR4 standard introduces Fine Granularity Refresh (FGR) operations [16]. FGR-1x is the same as the refresh process in DDR3. FGR-2x partitions each DDR3 refresh operation into 2 smaller operations. In essence, the tREFI is halved (refresh operations must be issued twice as often), and the tRFC also reduces (since each refresh operation does half the work). FGR-4x partitions each DDR3 refresh operation into 4 smaller operations. The tRFC and tREFI for these modes are also summarized in Table I.

These modes were introduced to reduce wait times for read requests that queue up during a refresh operation. In general, average queuing delays are reduced when a large refresh operation is broken into multiple smaller refresh operations. While the early projections of FGR were optimistic [15], the latest DDR4 parameters [30] reveal that FGR can introduce high overheads. A single FGR-2x operation has to refresh half the cells refreshed in an FGR-1x operation, thus potentially requiring half the time. But an FGR-2x operation and an FGR-1x operation must both incur the same recovery cost at the end to handle depleted charge pumps. The data in Table I shows that for 32 Gb chips, tRFC for FGR-2x mode is 480 ns, while tRFC for FGR-1x is 640 ns. The overheads of the recovery time are so significant that two FGR-2x operations take 50% longer than a single FGR-1x operation. Similarly, going to FGR-4x mode results in a tRFC of 350 ns. Therefore, four FGR-4x refresh operations would keep the rank unavailable for 1400 ns, while a single FGR-1x refresh operation would refresh the same number of cells, but keep the rank unavailable for only 640 ns. The high refresh recovery overheads in FGR-2x and FGR-4x limit their effectiveness in reducing queuing delays (see results in Section VI-E). Therefore, refresh remains an important problem.

G. NVM Writes

We next describe how writes are handled in memory systems. The memory controller typically prioritizes reads, and arriving writes are buffered in a write queue. When the write queue reaches a high water mark, the memory controller starts draining writes until the write queue reaches a low water mark [5]. The memory controller has to introduce a bus turnaround delay (tWTR) when switching from writes to reads in a given rank. To amortize this overhead, a number of writes are typically serviced in a rank before switching back to reads. Typically, memory controllers will drain 16-32 writes at a time to balance queuing delays for subsequent reads and the bus turnaround overhead. A write queue drain can therefore take hundreds of nano-seconds. During this write drain to a rank, other ranks on the channel are available for reads. A short tRTR timing delay (2 cycles) is introduced when switching between accesses to different ranks. Similar to refresh, the write queue drain can be viewed as a per-rank operation.

Emerging NVM cells will likely find a place in server memory systems in the near future. All of the emerging NVM cells suffer from long write latencies. While the tWR delay in DRAM is 15 ns [29], the same delay in PCM is 125 ns [19], and in Memristors is expected to range from 200-700 ns. Therefore, a write drain in these NVMs will keep a rank occupied for hundreds of cycles.

III. Motivation

Having described the basics, we now focus on creating a good baseline and analyzing it.

A. Peak power

DRAM refresh is performed at rank granularity. The memory controller can co-ordinate and issue refresh commands to every rank at the same time (*Simultaneous Refresh*) or stagger them so only one rank is refreshing at a time (*Staggered Refresh*). Commercial systems prefer the latter [30] because it limits the peak power of the memory system (since refresh is the most power-intensive operation performed within DRAM). Even though the average power consumption remains the same, limiting the peak power reduces system cost. With the Micron power calculator [28], we estimated that in a single channel with two ranks, with x4 4Gb chips [29], a single rank performing refresh consumes 6.34 W, while a rank that utilizes 30% of the channel bandwidth consumes 3.83 W. Thus, in a system with four channels and eight ranks, the difference in peak power for the staggered and simultaneous refresh mechanisms is 17.6 W. If we chose to go with simultaneous refresh, we would have to construct a board that is provisioned for the worst-case CPU power and the worst-case memory power (since it is possible that the CPU is executing many CPU-bound threads, and the memory system is refreshing at the same time). This implies a more expensive cooling system and power supply [35] (to the board and to the DIMMs). Having a power supply that is rated higher than the average power consumption also reduces the efficiency of the power supply, thereby increasing the energy costs [26]. The 17.6 W gap is for a 64 GB memory system. The gap would be much higher for large memory servers, such as the tera-byte servers that can be purchased today [10], [11].

B. Page mapping and stalls from refresh

While staggered refresh is favorable for peak power, compared to simultaneous refresh, it also has worse performance. Consider the following example. Simultaneous refresh ties up the entire memory system for 100 units of time; meanwhile, in a 4-rank system, staggered refresh ties up 1/4th of the memory system for 100 units, then the next 1/4th for the next 100 units, and so on. In theory, in both cases, a given memory cell is unavailable for time tRFC in that tREFI interval, so the effective memory availability appears to be similar. But in practice, staggered refresh introduces more stalls for the workload because of how application pages are scattered across ranks.

In order to balance the load between ranks and boost memory system parallelism for each thread, the OS spreads the pages from each thread over the entire available physical memory space. Our experiments show that there is an even distribution of read requests from a thread to different ranks with the default Linux kernel (2.6.13-1.1603sp13smp) in our simulations. In our 4-ranked baseline, we observe that the percentage of pages of a thread that are serviced by a rank can vary from 7-44%, but most workloads have a very uniform distribution of accesses. In every case, every rank has pages from every thread that is running.

In staggered refresh, when one of the ranks is performing refresh, a thread can continue to make progress as long as it is only reading data from the other three ranks (see the illustration in Figure 1). But this thread will eventually make an access to the rank being refreshed (since its pages are scattered equally across all four ranks). The access is stalled for hundreds of cycles; it becomes the oldest instruction in the thread's reorder buffer (ROB) and prevents the thread from making further progress. This is also true for all other threads in the system. By the end of the t_{RFC} refresh operation, it is possible that all threads are stalled and waiting on the rank being refreshed. The fact that three other ranks are available does not help throughput. Therefore, staggered refresh, while desireable from a peak power and cost perspective, is vulnerable to large slowdowns during refresh.

C. Performance with staggered refresh

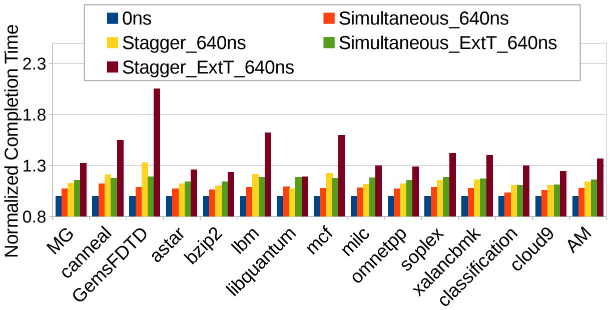


Fig. 2. Comparison between simultaneous, and staggered refresh.

Figure 2 shows a comparison of execution time with staggered and simultaneous refresh. We show results for a tRFC of 0 ns (an idealized memory system with no refresh penalties) and tRFC of 640 ns (representing a 32 Gb chip). We also show results for a 32 Gb chip at extended temperature (above 85° C), represented by "ExtT 640ns", that has a tRFC of 640 ns and a tREFI of 3.9 μs. On average, for a 32 Gb chip, simultaneous refresh has an execution time that is 12.6% lower than that of staggered refresh when operating in the extended temperature range. The idealized model has an execution time that is 27% lower than that of staggered refresh. Some benchmarks like GemsFDTD, lbm and mcf have large increases in execution time because these benchmarks have the highest percentage of refreshes that end up stalling a thread (see Figure 3). These benchmarks also have the highest number of cores stalled per refresh (see Figure 3).

Figure 3 quantifies the percentage of refresh operations that end up stalling a given thread in the workload. As discussed earlier, because a thread's pages are scattered across all ranks, every refresh operation has the potential to stall every thread. This is observed in a workload such as GemsFDTD. Across the entire benchmark suite, on average, a thread is stalled by half the refresh operations in the memory system.

Figure 3 also quantifies the stalls resulting from a typical refresh operation. The secondary Y-axis shows the number of cores that were stalled by a single refresh operation, where

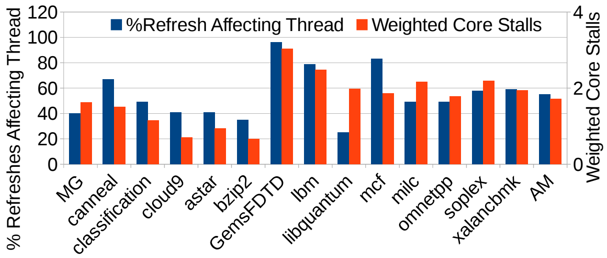


Fig. 3. Percentage of refreshes that stall a given thread, and Numbers of Cores that are stalled per Refresh, weighted by the fraction of t_{RFC} (for the ExtT-640ns case).

every core is weighted by the duration that it was stalled for. For example, if three cores were stalled, each for half the duration of a refresh operation, the weighted core stall time would be 1.5. While GemsFDTD experiences a weighted core stall time of over 3 (8 is the maximum), on average the weighted core stall time is 1.63.

The above discussion helps establish our baseline system – the staggered refresh mechanism that has low peak power and that attributes 27% of its execution time to refresh-related stalls for 32 Gb at extended temperature. Because of the staggered process, the penalty from refresh is much higher than the tRFC/tREFI ratio (16%). In theory, for our 4-rank system, the penalty can be as high as 4×tRFC/tREFI, but as shown by Figures 3, in practice not all threads are stalled all the time during every refresh, so the penalty is lower. However, the baseline provides a significant room for improvement with smarter refresh policies. We focus most of our results on the most aggressive DDR4 specification known to date (32 Gb chip at extended temperature).

D. Impact of NVM Writes

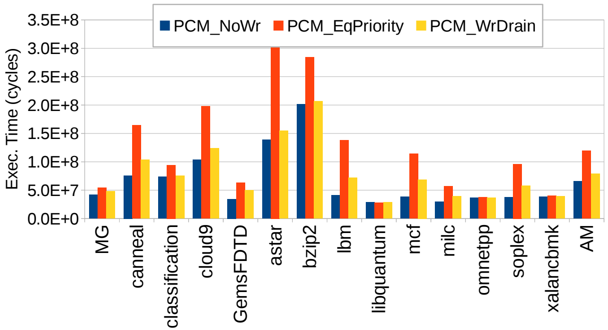


Fig. 4. Impact of high latency PCM writes on performance.

In Figure 4, we show the impact of different write scheduling policies in a system with PCM main memory (see Table II for detailed simulation parameters), relative to an idealized memory system with no writes (PCM_NoWr). The third bar (PCM_WrDrain) shows the execution time when write drains are performed with Hi/Lo water mark of 40/20 writes. The second bar (PCM_EqPriority) shows the execution time when reads and writes are given equal priorities and are both serviced in FCFS fashion. The best baseline is 21% worse than the idealized memory system, showing a significant room for improvement. On average, a write drain process keeps 8.6 banks (out of 16 banks in a channel) busy, stalling 5.7 of eight threads on average on every write drain. We also see minor performance differences (<1%) by co-ordinating the write drains in the ranks sharing a channel.

IV. Proposal: Rank Assignment

A. Current Approaches to Memory Mapping

In modern systems, contiguous virtual addresses of an application are scattered across the entire physical memory system. The hypervisor and OS map a new virtual page to a free physical page. These mappings, while influenced by well-defined page replacement policies, appear random for the most part. Further, memory controllers adopt address mapping policies that can scatter a single page across the memory channels, ranks, and banks. These policies were reasonable in the past decade – they were simple and yielded performance improvements. In Figure 5, we show the normalized execution times for our benchmarks for different address mapping policies. The first set of bars (No Refresh) represents an ideal system with a tRFC of 0ns. The second represents a system which performs staggerred refresh with a tRFC of 640ns. We consider the following address mapping policies. We also consider each address mapping policy with open and close page policies and pick the better of the two.

- ***Non-interleaved***, that places an entire physical page in one DRAM row to exploit row buffer locality.
- ***Channel-interleaved***, that exploits memory-level parallelism by scattering a page across all the channels.
- ***Bank-interleaved***, that places four consecutive cache lines in one bank, the next four in the next bank in the same rank, and so on, similar to the address mapping suggested by Kaseridis et al. [18].
- ***Bank-XOR***, that does bank interleaving, but calculates the bank ID by XOR-ing the row and bank pointed to by the address [50] .

The results in Figure 5 highlight the following two points:

First, in the absence of any refresh, scattering a page (channel-interleaving) is more effective than keeping a page in a single row (non-interleaved). The gap between the various policies is about 25%. This is because memory intensive applications that enjoy high row buffer hit rates, also enjoy higher bandwidth utilization when cachelines from a page are scattered between channels.

Second, for a future DDR4 DRAM system that has a tRFC of 640 ns and that implements staggered refresh, scattering a page between different channels or ranks can actually be harmful to performance. The channel interleaving scheme, that was a clear winner without refresh, now emerges as the least desireable address mapping policy with refresh by about 8%. Scattering a page across different channels increases the chances of a thread being stalled by a refresh operation.

For the rest of the paper, we use the optimal address mapping scheme from Figure 5: bank-XOR address mapping with an Open Page policy.

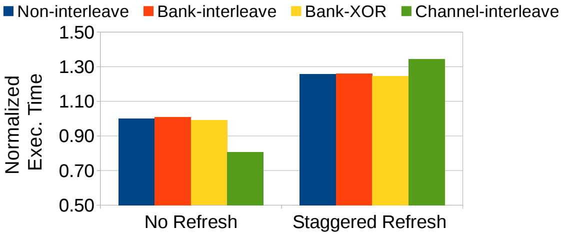


Fig. 5. Exec. time with different address mapping policies (average of all benchmarks).

These observations drive us to the following argument: while past systems (OS and memory controllers) were configured to scatter data across the entire physical memory system, such policies are worth re-considering in future systems. Traditional guidelines for page placement (using sophisticated address mapping and an unconstrained OS to scatter pages) will have to be replaced by new guidelines that recognize emerging phenomena (long refresh and write latencies, NUMA on a board, etc.).

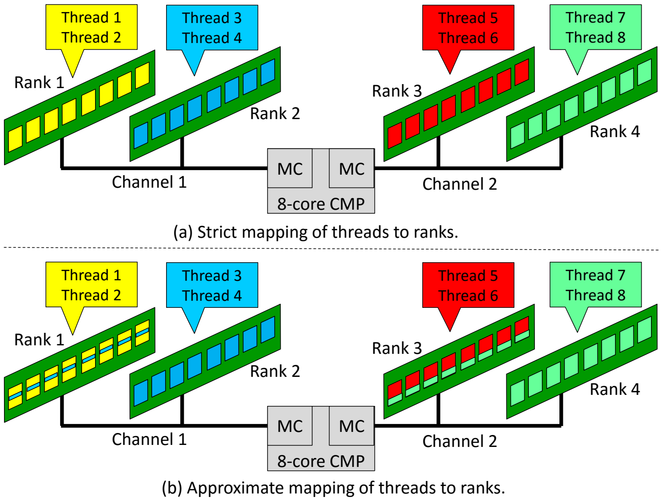
B. Overview

Our analysis in the previous section shows that staggered refresh and write drains incur a high performance penalty because even though only a fraction of the ranks are unavailable at a time, every thread has the potential to stall. The rest of this section focuses on the refresh problem, but the exact same solution also applies to the problem of long write drains in NVMs.

Our proposal is based on the simple notion that when (say) a quarter of the memory system has a service interruption, no more than a quarter of the threads should be stalled. This can be enforced if each thread places its data in a subset of the ranks. Figure 6a shows an example mapping of threads to ranks in an 8-core system running 8 threads, with 2 channels and 2 ranks per channel. Each thread places its pages entirely in one rank, i.e., two threads would essentially share a rank. When one rank is being refreshed, only up to two threads would be stalled. The remaining 2 threads mapped to that channel would run at higher than usual throughput because only 2 threads are competing for that channel's bandwidth. Figure 6b shows that the mapping of threads to ranks need not be strict. A few deviations can reduce the burden on the OS and achieve most of the benefit.

When a thread is created, the OS can assign it a *preferred* rank (the one with maximum unused capacity). The OS maps all pages from that thread to free pages in the preferred rank. This thread-to-rank affinity is maintained even if the thread is migrated to a different core. For this technique to be effective, we require an address mapping policy that places an entire page in a rank, e.g., the bank-XOR mapping.

There are many precedents of commercial page allocation policies that can place a page in specific locations. For example, Solaris implements locality groups [44] and Linux uses the *libnuma* library [12] for shared-memory multiprocessors, to designate a collection of CPUs that can access a collection of memory devices within a given latency. Lin et al. [21] modify the Linux Kernel to implement an OS based cache partitioning scheme. Similar page coloring approaches were also implemented in the SGI Origin to exploit NUMA latencies [7]. We build on this prior work to make the novel observation that smart page coloring has a significant impact on refresh and write management, more so than competing hardware-based schemes.



(a) Strict mapping of threads to ranks.

(b) Approximate mapping of threads to ranks.

Fig. 6. An example mapping with Rank Assignment (best viewed in color). The top figure shows a strict mapping, the figure below allows a few deviations.

C. Page Placement with Modified CLOCK

Ideally, we want a thread to be mapped only to its preferred ranks. But this places a higher burden on the OS because application requirements change over time and are not known beforehand. If the preferred rank for a thread is oversubscribed, it may be better to map a new page to a different rank than increase the page fault rate for the preferred rank. Such deviations will typically happen if some thread has a much larger working set than others and must spill into other ranks. There is no impact on correctness.

To reduce the burden on the OS, we consider a modified CLOCK replacement policy [41] that is simple and approximates the ideal assignment of threads to ranks. Muralidhara et al. [31] also use a similar page mapping algorithm to map pages to channels. The baseline CLOCK policy maintains a circular list of all pages, a bit for each page to indicate if it has been touched recently (reference bit), and a global pointer to the next eviction candidate. When a new page is requested, the pointer moves forward, till an unreferenced page is found. All pages that are encountered in this search, are marked unreferenced.

In our page allocator, when a thread is spawned, the OS looks for a rank that has the least utilization and that rank is assigned as the preferred rank for the new thread. Utilization is defined as the average queuing delay for that rank, where queuing delay includes wait time for the channel and wait time for page fault handling. Instead of one global pointer, the page allocator maintains a pointer for each rank. When a thread assigned to rank R1 requests a free page, the pointer for R1 is advanced. The pointer only looks for unreferenced pages belonging to R1. Unlike the baseline policy, encountered pages during this walk are not unreferenced. Ultimately, an unreferenced page may be found – this page is used by the thread and marked as referenced (so it is not used by trailing pointers). If the pointer reaches the last page in the list without finding an unreferenced page, it gives up and performs another walk, looking for a page from a second preferred rank. If that walk fails as well, we resort to the baseline CLOCK algorithm to find any unreferenced page, starting from the rank pointer that is trailing all other rank pointers. When all rank pointers reach the end of the list, the pointers roll to the start. Before rolling over, the reference bits for all pages are reset (with the exception of pages currently resident in the processor's TLBs). This reset process is common to the baseline and proposed allocators.

Consider an example with an aggressor thread T1 with a large and active working set co-scheduled with a submissive thread T2 with a small working set. T1 is assigned to rank R1 and T2 is assigned to rank R2. When T1 requests a page, it is preferentially allocated pages from R1. Its pointer quickly reaches the end of the list because many pages in R1 are marked as referenced. From that point on, subsequent pages are allocated from R2 until R2's pointer also reaches the end of the list. This algorithm does not lead to any additional page faults, but steers T1 to R1 as much as possible. Once the pointers roll over, at first, T1 evicts its own pages to make room for its pages. The baseline, on the other hand, would evict some of T2's pages to make room for T1's pages. We observed that the new policy leads to fewer page faults. This is because the aggressor thread has longer reuse distances, while the submissive thread has shorter reuse distances. Therefore, after the pointers have rolled over, it is better to evict the aggressor thread's pages than the submissive thread's pages.

D. Multi-Threaded Applications

A potential problem with the Rank Assignment approach is that while multi-programmed workloads can be controlled to only access a single rank, multi-threaded workloads may not be as easy to control. If a thread accesses a page that was created by another thread, the thread's accesses may no longer be localized to a single rank. The probability of this depends strongly on how much page sharing is exhibited by the application. In our benchmark suite, *canneal* had the least locality; up to 62.5% of accesses were steered to a rank different from the thread's assigned rank. As stated before, when such deviations happen, it only limits the performance improvement, but does not cause correctness problems. Regardless of what we observe on our benchmark suite, it is clear that the Rank Assignment approach will not be effective for some workload classes. For such a workload, a service interruption in a rank has the potential to stall all threads.

V. Methodology

A. Simulation

For the first part of our evaluation, we rely on detailed cycle-accurate simulations with Simics [46]. We interface Simics with the detailed USIMM memory model [4]. We update USIMM to model DDR4 and PCM memory systems (bank groups, DDR4/PCM timing parameters, etc.).

B. Simulator parameters

Salient Simics and USIMM parameters are summarized in Table II. While most of our experiments are run with 4MB of L2 cache, as part of our sensitivity analysis, we also provide performance results with 8MB of L2 cache (Section VI-K).

Processor	
Core Parameters:	UltraSPARC III ISA, 8-core, 3.2 GHz, 64-entry ROB, 4-wide ooo.
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache	4MB/8MB 64B,8-way, shared, 10-cycle
Coherence Protocol	Snooping MESI
DRAM/PCM Parameters	
Channels, ranks, frequency, banks	2 channels, 2 ranks/channel, 1600Mbps 16/8 banks/rank (DRAM/PCM)
Write queue water marks	10 (high) and 5 (low), for each Rank
Read Q Length	32 per channel
DRAM chips	32 Gb capacity at extended temperature (for most experiments)
DRAM/PCM Timing Parameters (DRAM cycles)	t_{RC} = 39/55 t_{RCD} = 11 t_{RRD_S} = 4 t_{RAS} = 28/45 t_{FAW} = 20 t_{RRD_L} = 5 t_{WR} = 12/100 t_{RP} = 11 t_{WTR_S} = 2 t_{RTRS} = 2 t_{CAS} = 11 t_{WTR_L} = 6 t_{RTP} = 6 t_{DTATA_TRANS} = 4 t_{CCD_L} = 5 t_{CCD_S} = 4
Refresh Interval	t_{REFI} = 7.8μ s t_{REFI_ExtT} = 3.9μ s
Refresh Period	t_{RFC} = 350ns (8Gb), 480ns (16Gb), 640ns (32Gb)

TABLE II
SIMULATOR AND DRAM TIMING [16], [29] PARAMETERS.

We adopt the bank-XOR address mapping policy. Our memory scheduler FR-FCFS, and Open Page policy. Reads are prioritized over writes until the write queue reaches a high water mark.

C. Peak power model

Our proposals do not impact the total activity within the memory system. By improving throughput, they have the potential to yield lower memory and system energy for a given task. The choice of staggered or simultaneous refresh does influence the peak memory power, already described in Section III. We rely on the Micron power calculator for x4 4 Gb DRAM chips for those estimates [28].

D. Workloads

We use multi-programmed workloads constructed out of SPEC2k6 benchmarks and multi-threaded workloads from PARSEC [1], NAS Parallel Benchmarks (NPB) and CloudSuite [8]. For SPEC2k6, we run 8 instances of each benchmark (rate mode). CloudSuite, PARSEC, and NPB are run with 8 threads. The SPEC workloads are fast-forwarded for

50 billion instructions before starting simulations and the NPB/CloudSuite/PARSEC programs are simulated at the start of their region of interest. Measurements are reported for 2 million DRAM accesses, which corresponds to 82M-1.5B total simulated instructions for each workload. By using DRAM access count to terminate an experiment, we ensure that each experiment measures roughly an equal amount of work. In the multi-programmed workloads, each program makes roughly equal progress in every experiment (we are running in rate mode and our policies do not prioritize any particular thread). Any slippage in multi-threaded workloads is compensated by wait times at barriers. Spinning at a barrier can increase instruction count (hence, IPC is a bad metric for multi-threaded workloads), but does not increase memory accesses.

E. *Longer simulations to evaluate page allocation*

Our cycle-accurate simulations are short enough that we do not encounter page faults or capacity pressures within a rank. Therefore, to test if our page mapping policy impacts page fault rates and thread-to-rank assignment, we run a collection of workloads for much longer durations (2.5B instructions per program) without cycle-accurate simulations. We run Simics in functional mode and simulate our page allocation algorithm. We run this experiment for multi-programmed workloads that mix large and small SPEC2k6 benchmarks. Since SPEC benchmarks are known to have small working sets and we want to stress the rank capacity, each rank is assumed to have a maximum capacity of only 0.125 GB. The results for these experiments are reported in Section VI-C.

VI. Results

A. *Impact of Rank Assignment on Refresh*

We focus most of our evaluation on the 32 Gb chip at extended temperature. Results for other systems are reported in Section VI-K. Section III has already established the baseline and idealized systems. The idealized system assumes a tRFC of 0 ns, i.e., no refresh. The baseline performs Staggered Refresh and assumes default page assignment, i.e., every application scatters its pages across all ranks. We also show the performance of Simultaneous Refresh in most figures.

Figure 7 shows the execution time for Rank Assignment. In these experiments, there are 8 threads and 4 ranks. For multi-programmed workloads, we assume that two threads are perfectly mapped to a single rank. For multi threaded workloads, pages are mapped to ranks depending on the first thread to touch the page. The results show that the Rank Assignment model has an execution time that is 18.6% lower than the Staggered Refresh baseline and only 11.4% higher than the idealized model.

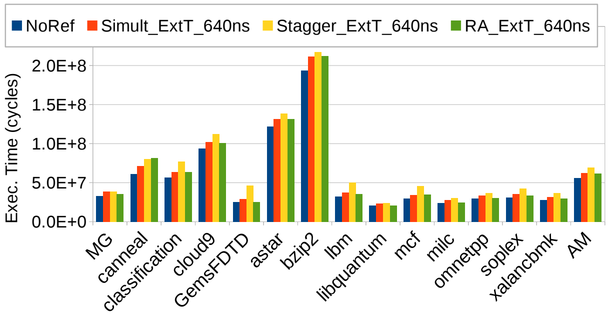


Fig. 7. Execution time for Rank Assignment and the baseline for a 32 Gb chip at extended temperature.

Similar to the analyses in Figures 3, we quantify how threads are impacted by refresh operations. In the baseline, a refresh operation can stall all the threads in the system. In Rank Assignment, a refresh operation stalls at most two threads. A refresh operation results in a weighted core stall time of 1.63 in the baseline (Figure 3) and 1.32 in Rank Assignment. The lower weighted-core-stall-time per refresh is the dominant factor in the performance improvement of Rank Assignment, relative to the Staggered Refresh baseline.

We also observe that multi-threaded workloads show a range of thread-to-rank affinities. Table III shows how some of the accesses from a thread are not serviced by the preferred rank.

Name	Max. non-preferred accesses	Execution time decrease with RA(%)
MG	15.86	9.0
canneal	62.59	-1
cloud9	21.23	10.1
classification	8.25	17.85

TABLE III
The percentage of page accesses that are not handled by the preferred rank for multi-threaded workloads .

The second factor is a possible improvement in row buffer hit rates because page coloring can reduce interference between threads in a bank. In our experiments, we noticed a very small change in row buffer hit rates.

The third factor is the boost in performance because some non-stalled threads experience lower bus contention during refresh. To get a sense for the contribution of this factor, we carried out the following experiment. We ran our simulator with six threads; threads T1 and T2 were mapped to rank R1, T3 and T4 were mapped to rank R2, T5 and T6 were mapped to rank R3, and rank R4 was left idle. Since rank R3 and R4 share a channel, we observed that T5 and T6 experienced lower bus contention delays and yielded 4.8% higher performance than threads T1-T4. Therefore, in our Rank Assignment experiments, 2 of the 8 threads see a similar performance boost during any refresh operation.

We examine the fourth and fifth factors in more detail in the next subsection.

B. *Write Queue Drains and Bank Contention*

In order to analyze the effect of Rank Assignment on write drains and bank contention, we assume tRFC=0.

When tRFC=0, RA causes a performance increase of 5.9%. This quantifies the combined benefit from lower bank contention and fewer stalls during write drains. Our results show that there is performance difference of 2.4% when Rank Assignment is performed on a system which performs neither refresh nor writes. This performance difference is entirely because of lower bank contention.

In summary, even though tRFC/tREFI is 16% for 32GB chips at extended temperatures, the observed penalty is only 10%. This is because different factors play roles in augmenting the benefits from page coloring: some overlap of computation with refresh, higher row buffer hit rates, higher throughput for non-stalled threads, lower bank contention, and more efficient write queue drains. The last four factors are not present in the Simultaneous Refresh baseline. Hence, Rank Assignment is able to out-perform Simultaneous Refresh by 4.1%, while having much lower memory peak power. The improvement relative to the Staggered Refresh baseline is much higher (18.6%).

C. Page Allocation Disruptions

Due to low simulation speeds, our cycle-accurate simulations last under 226 million processor cycles per core. In such experiments, we are able to perfectly map threads to ranks. Such simulations are not long enough to exhibit possible disruptions from the new page allocation algorithm. In particular, when different threads have varying working set sizes, it is possible that some threads may spread their pages across multiple ranks, or by forcing a large thread to steer most of its pages to one rank, it may incur a high page fault rate.

To test the algorithm described in Section IV-C on a heterogeneous workload, we run the programs in functional mode with Simics and track all page accesses. We assume that each rank has a capacity of 0.125 GB. Our workloads consist of four threads of varying working set sizes. We simulate these for 10B total instructions (2.5B instructions/core x 4 cores). We consider the following workloads composed of large and small working set benchmarks: lbm-libquantum-astar-milc, and astar-milc-soplex-xalancbmk.

Our results show that 30% of all accesses made by a thread are not to its assigned rank or the second-preferred rank. Figure 8 shows that the effectiveness of Rank Assignment decreases as pages from a thread are spread over more ranks. The Y-axis represents the execution time decrease and the X-axis represents percentage of pages that are mapped to a random rank (that is not the assigned rank). Even with a 50% page mapping error, Rank Assignment reduces execution time by 9.4%

Figure 8 also shows the effectiveness of the Rank Assignment scheme when the processor uses a larger cache, of 8MB. The memory system is a smaller bottleneck in this case, and the performance improvements are lower.

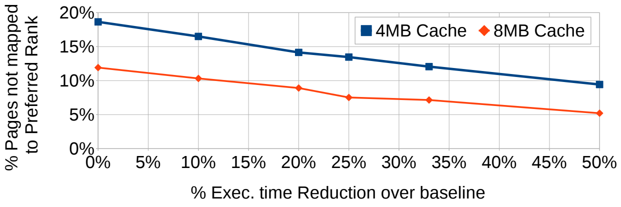


Fig. 8. Effect of mapping pages to non-preferred ranks.

D. Handling the Long Latency of NVM Writes

We model a PCM-based memory system with timing parameters from Lee et al. [19]. Figure 9 shows the performance of Rank Assignment. The first bar (Baseline) shows the execution time of the baseline system, the second bar (RA) shows the execution time of Rank Assignment. We find that RA reduces the execution time by 13.3%. In order to isolate the effects of reduced bank contention, we also show the performance of a system where no writes are performed. The third bar (NoWrites) in Figure 9 shows the execution time of the baseline system when no writes are serviced. The fourth bar (NoWrites_RA) shows the effect of RA when no writes are performed. There is a 6.6% reduction in execution time when RA mapping is used when no writes are performed. This reduction in execution time between NoWrites and NoWrites_RA is because of reduction in bank contention. Therefore we can conclude that out of the 13.3% execution time reduction, 6.7% is because of the effect of Rank Assignment on the long latency of NVM writes. Once again, some benchmarks like GemsFDTD, lbm, and mcf show the largest reduction in execution time. This is because in addition to reasons mentioned in Section III-C, they also have high write/read ratio.

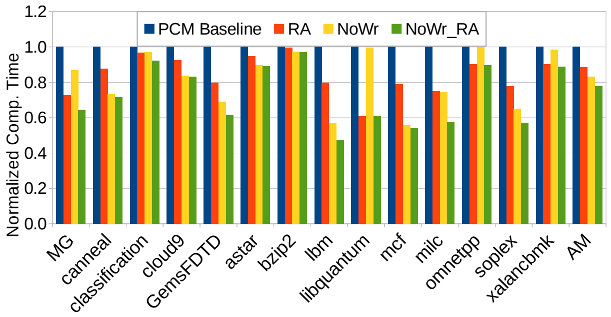


Fig. 9. Impact of Rank Assignment on PCM writes.

E. Comparisons to Prior Work

We now analyze competing hardware-based techniques that have been proposed in recent years.

F. Fine Granularity Refresh (FGR)

Section II describes the FGR mechanism being introduced in DDR4. A short refresh operation lowers queuing delays for reads, but increases the overall time that memory is unavailable (because of recovery overheads after every refresh).

Figure 10 shows results for a number of fine granularity refresh mechanisms. The first two bars show the idealized no-refresh model and the baseline Staggered Refresh model. The third and fourth bars represent Staggered Refresh with FGR-2x and FGR-4x, respectively. We assume optimistic tRFCs with no additional overhead for these cases, i.e., the tRFCs for the baseline, FGR-2x, and FGR-4x are 640 ns, 320 ns, and 160 ns, respectively. The fifth and sixth bars represent Staggered Refresh with FGR-2x and FGR-4x, respectively, but with realistic DDR4 timing parameters [30] (Table I). The results show that no-overhead FGR by itself is effective (10.1% improvement over the Staggered Refresh baseline), but this effectiveness disappears once realistic overheads are considered (24% execution time increase over the baseline).

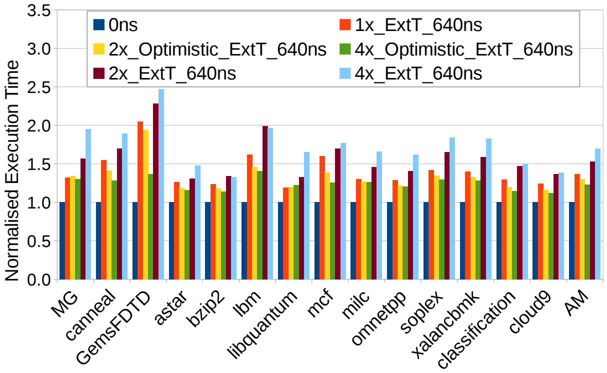


Fig. 10. FGR with and without overheads (32 Gb at ExtT).

G. Adaptive Refr. (AR), Preemptive Command Drain (PCD)

The above results indicate that FGR is not effective in our baseline model. However, it may be effective in other settings. For example, FGR may be effective when performed during memory idle times.

Recent work by Mukundan et al. [30] also shows that FGR can yield improvements, the optimal refresh granularity varies across applications, and an Adaptive Refresh (AR) scheme can dynamically select the optimal refresh granularity. The AR model does not help in our baseline because FGR is never effective.

There are two important reasons for this difference. Mukundan et al. [30] assume a processor model similar to the IBM Power7 where the command queue is shared by multiple ranks. This leads to command queue seizure in their baseline, where requests to a rank being refreshed take up most of the command queue entries and throttle memory throughput. In our simulation model, we assume per-rank command queues, so the entries waiting for a rank being refreshed do not impact throughput in other ranks. Even when we assume a single command queue for all ranks, we observe that command queue seizure is rare if we assume a 32-entry read queue. For an 8-core system to fill up a 32-entry read queue, each 64-entry ROB must average more than 4 misses, i.e., a per-thread MPKI higher than 62.5, which is relatively uncommon.

In our simulation model, we see that Preemptive Command Drain (PCD) [30] reduces execution time by 0.9%.

H. Refresh Pausing (RP)

Refresh Pausing (RP) [32] begins a refresh operation as soon as a rank is idle, but pauses it when the memory controller receives a request from the CPU. Refresh Pausing partitions the refresh operation into smaller fragments and squeezes them into idle periods. We model an optimistic version of RP that resembles the model in the original paper [32] and a realistic version of RP that is based on the recently released DDR4 timing parameters. The optimistic version of RP (formulated before DDR4 timing was revealed) assumes that a 640 ns refresh operation can be paused up to eight times, where each fragment takes 80 ns. Similar to DDR4's FGR, we assume that the realistic version of RP can be paused only at FGR intervals. Figure 11 shows a significant (14%) execution time reduction with optimistic RP, but an overall increase in execution time (7.2%), when FGR overheads are added.

I. Elastic Refresh (ER)

Elastic Refresh (ER) [43] tries to perform refresh when the memory is idle. The DDR3 standard requires that 8 refresh operations be performed within an 8×tREFI time window, but allows the memory controller some flexibility in when to issue those refresh operations. If ER detects that the rank has been idle for a specified time, it issues a refresh command. Any leftover refreshes have to be issued at the end of the 8×tREFI time window. We model the static ER scheme described in [43]. Figure 11 shows the potential benefit with this scheme (4.2% over the baseline).

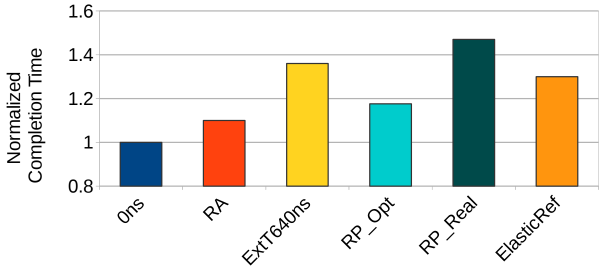


Fig. 11. Completion times normalized to 0ns.

J. Summary

We have modeled multiple recent hardware-based refresh solutions: two that partition a refresh operation into smaller fragments (FGR, RP), two that attempt to perform refresh during memory idle times (RP and ER), and one that prevents command queue seizure (PCD). Of these, we observed that the fine-grained schemes (FGR, RP) are not effective. Those schemes may be more effective in a processor model that is more vulnerable to command queue seizure. PCD yielded a 0.9% benefit while ER yielded a 4.2% benefit, relative to the Staggered baseline. Recall that the improvement with Rank Assignment was 18.6%. We therefore believe that the Rank Assignment scheme is simpler and more effective than these competing schemes that require changes to the hardware (DRAM chips, memory controller, and DDR standards). Similarly, in terms of NVM write optimizations, RA reduced runtime by 13.1%. Prior hardware-based proposals (e.g., Staged Reads [5]) yield lower improvements (12%).

Many of the proposals in prior work are orthogonal to the Rank Assignment approach. We therefore believe that they can be combined with Rank Assignment to yield higher improvements. We leave the evaluation of such combined schemes for future work.

K. Sensitivity Analysis

We have carried out a sensitivity analysis over a number of parameters. We experimented with a system that had eight cores and four ranks on a single channel. The execution time reduction with Rank Assignment was again 18.6%. For smaller chips at temperatures under 85° C, where the refresh bottleneck is smaller, we observed execution time reductions of 10.8% (32 Gb chips), 8.8% (16 Gb chips), and 7.9% (8 Gb chips) with Rank Assignment.

VII. Related Work

Many techniques have been proposed to mitigate the long latency of PCM writes. Lee et al. [19] use narrow rows to mitigate the high latency of PCM. Yue et al. [48] and Qureshi et al. [37] make use of the asymmetric nature of PCM writes to speed up the write process. Jiang et al. [17] limit the number of write iterations performed, and use ECC to correct any bits that are incorrectly written. Qureshi et al. [36] pause the iterative write process in PCM when a read request arrives, thereby avoiding stalls because of the long write latency.

All the techniques mentioned above are orthogonal to our proposal. Rank Assignment attempts to keep the memory channel busy even when certain parts of the channel are refreshing. Unlike many of the proposals listed above, Rank Assignment does not incur power or storage over heads.

Liu et al. [24] modify the Linux kernel to implement thread to bank mapping. They show that most workloads can achieve 90% of their max. performance with only 16 DRAM banks. Xie et al. [47] dynamically assign banks to threads based on the memory characteristics of the thread.

Muralidhara et al. [31] map pages to DRAM channels based on the MPKI and DRAM row buffer hit rate of the application. This helps reduce bank contention and also prioritize performance critical DRAM accesses. The implementation of our Rank Assignment mapping is partly inspired by the implementation presented in [31].

VIII. Conclusions

This paper considers a simple technique to hide service interruptions in future DRAMs and NVMs. Service interruptions such as refresh happen at the granularity of ranks. We show that fine-grained refresh approaches, as supported by DDR4, are not effective in hiding refresh penalties. Our experiments with elastic forms of refresh also yielded small improvements. By mapping pages from a thread to few ranks, we show that it is possible to stall only a fraction of threads when a rank is interrupted. The Rank Assignment mechanism is effective in removing a large fraction of refresh and write drain overheads, while requiring no changes to the hardware. The only modifications are to the OS, that only has to perform a best-effort mapping of threads to ranks.

Acknowledgment

We thank the anonymous reviewers for their suggestions. This work was supported in part by Samsung, and by NSF grants 1302663 and 1423583.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” Princeton University, Tech. Rep., 2008.

[2] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, “Phase Change Memory Technology,” 2010, http://arxiv.org/abs/1001.1164v1.

[3] K. K.-W. Chang, D. Lee, Z. Chishti, C. Wilkerson, A. Alameldeen, Y. Kim, and O. Mutlu, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” in *Proceedings of HPCA*, 2014.

[4] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, “USIMM: the Utah SImulated Memory Module,” University of Utah, Tech. Rep., 2012, UUCS-12-002.

[5] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, “Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads,” in *Proceedings of HPCA*, 2012.

[6] S. Cho and H. Lee, “Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy, and Endurance,” in *Proceedings of MICRO*, 2009.

[7] J. Corbalan, X. Martorell, and J. Labarta, “Page Migration with Dynamic Space-Sharing Scheduling Policies: The case of SGI 02000,” *International Journal of Parallel Programming*, vol. 32, no. 4, 2004.

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in *Proceedings of ASPLOS*, 2012.

[9] M. Ghosh and H.-H. Lee, “Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs,” in *Proceedings of MICRO*, 2007.

[10] HP, “HP ProLiant DL980 Generation 7 (G7),” 2013, http://h18004.www1.hp.com/products/quickspecs/13708_na/13708_na.pdf.

[11] IBM Corporation, “IBM System x3950 X6,” 2014.

[12] Intel Corp., “Optimizing Applications for NUMA,” 2011, https://software.intel.com/en-us/articles/optimizing-applications-for-numa.

[13] C. Isen and L. John, “ESKIMO - Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy of DRAM subsystem,” in *Proceedings of MICRO*, 2009.

[14] ITRS, “International Technology Roadmap for Semiconductors, 2013 Edition,” 2013.

[15] JEDEC, “DDR4 Mini Workshop,” 2011, http://www.jedec.org/sites/default /files/JS_Choi_DDR4_miniWorkshop.pdf.

[16] JEDEC, *JESD79-4: JEDEC Standard DDR4 SDRAM*, 2012.

[17] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, “Improving write operations in MLC phase change memory,” in *Proceedings of HPCA*, 2012.

[18] D. Kaseridis, J. Stuecheli, , and L. K. John, “Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-Core Era,” in *In Proceedings of MICRO*, 2011.

[19] B. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting Phase Change Memory as a Scalable DRAM Alternative,” in *Proceedings of ISCA*, 2009.

[20] C.-H. Lin, D.-Y. Shen, Y.-J. Chen, C.-L. Yang, and M. Wang, “SECRET: Selective error correction for refresh energy reduction in DRAMs,” in *Proceedings of ICCD*, 2012.

[21] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems,” in *Proceedings of HPCA*, 2008.

[22] J. Lin, H. Zheng, Z. Zhu, E. Gorbatov, H. David, and Z. Zhang, “Software Thermal Management of DRAM Memory for Multi-core Systems,” in *Proceedings of SIGMETRICS*, 2008.

[23] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, “RAIDR: Retention-aware intelligent DRAM refresh,” in *Proceedings of ISCA*, 2012.

[24] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, “A software memory partition approach for eliminating bank-level interference in multicore systems,” in *Proceedings of PACT*, 2012.

[25] S. Liu, B. Leung, A. Neckar, S. Memik, G. Memik, and N. Hardavellas, “Hardware/Software Techniques for DRAM Thermal Management,” in *Proceedings of HPCA*, 2011.

[26] D. Meisner, B. Gold, and T. Wenisch, “PowerNap: Eliminating Server Idle Power,” in *Proceedings of ASPLOS*, 2009.

[27] Micron, “DDR3 Thermals: Thermal Limits, Operating Temperatures, Tools and System Development,” 2007.

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

[29] Micron Technology Inc., “Micron DDR3 SDRAM Part MT41J1G4,” 2009, http://download.micron.com/pdf/datasheets/dram/ddr3/4Gb_DDR3_SDRAM.pdf.

[30] J. Mukundan, H. Hunter, K.-H. Kim, J. Stuecheli, and J. F. Martinez, “Understanding and Mitigating Refresh Overheads in High-Density DDR-4 DRAM Systems,” in *Proceedings of ISCA*, 2013.

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

[32] P. Nair, C. Chou, and M. Qureshi, “A Case for Refresh Pausing in DRAM Memory Systems,” in *Proceedings of HPCA*, 2013.

[33] P. Nair, D.-H. Kim, and M. Qureshi, “ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates,” in *Proceedings of ISCA*, 2013.

[34] D. Niu, C. Xu, N. Muralimanohar, N. Jouppi, and Y. Xie, “Design Trade-offs for High Density Cross-point Resistive Memory,” in *Proceedings of ISLPED*, 2012.

[35] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, , and J. Underwood, “Power Routing: Dynamic Power Provisioning in the Data Center.” in *Proceedings of ASPLOS*, 2010.

[36] M. Qureshi, M. Franceschini, and L. Lastras, “Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing,” in *Proceedings of HPCA*, 2010.

[37] M. K. Qureshi, M. Franceschini, L. Lastras, and A. Jagmohan, “PreSET: Improving Read Write Performance of Phase Change Memories by Exploiting Asymmetry in Write Times,” in *Proceedings of ISCA*, 2012.

[38] S. Ankireddi and T. Chen, “Challenges in Thermal Management of Memory Modules,” 2008, http://www.electronics-cooling.com/2008/02/challenges-in-thermal-management-of-memory-modules.

[39] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization,” in *Proceedings of MICRO*, 2013.

[40] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. Udipi, “Quantifying the Relationship between the Power Delivery Network and Architectural Policies in a 3D-Stacked Memory Device,” in *Proceedings of MICRO*, 2013.

[41] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley and Sons, 2009.

[42] S.Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn, “Flikker: Saving DRAM Refresh-power through Critical Data Partitioning,” in *In Proceedings of ASPLOS*, 2011.

[43] J. Stuecheli, D. Kaseridis, H. Hunter, and L. John, “Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory,” in *In Proceedings of MICRO*, 2010.

[44] *Sun Studio 12 Update 1: OpenMP API User’s Guide*, Sun Microsystems, Inc., 2009.

[45] R. Venkatesan, S. Herr, and E. Rotenberg, “Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM,” in *In Proceedings of HPCA*, 2006.

[46] Wind River, “Wind River Simics Full System Simulator,” 2007, http://www.windriver.com/products/simics/.

[47] M. Xie, D. Tong, K. Huang, and X. Cheng, “Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning,” in *Proceedings of HPCA*, 2014.

[48] J. Yue and Y. Zhu, “Accelerating Write by Exploiting PCM Asymmetries,” in *Proceedings of HPCA*, 2013.

[49] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, “CREAM: A Concurrent-Refresh-Aware DRAM Memory System,” in *Proceedings of HPCA*, 2014.

[50] Z. Zhang, Z. Zhu, and X. Zhand, “A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality,” in *Proceedings of MICRO*, 2000.