# Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling

Jagadish B. Kotra[†] Narges Shahidi[†] Zeshan A. Chishti[‡] Mahmut T. Kandemir[†]

The Pennsylvania State University[†]
University Park, PA 16802
{jbk5155, nxs314, kandemir}@cse.psu.edu

Intel Labs[‡]
Hillsboro, OR 97124
zeshan.a.chishti@intel.com

## Abstract

DRAM cells need periodic refresh to maintain data integrity. With high capacity DRAMs, DRAM refresh poses a significant performance bottleneck as the number of rows to be refreshed (and hence the refresh cycle time, tRFC) for each refresh command increases. Modern day DRAMs perform refresh at a rank-level, while LPDDRs used in mobile environments support refresh at a per-bank level. Rank-level refresh degrades the performance significantly since none of the banks in a rank can serve the on-demand requests. Per-bank refresh alleviates some of the performance bottlenecks as the other banks in a rank are available for on-demand requests. Typical DRAM retention time is in the order of several milliseconds, viz, 64msec for environments operating in temperatures below 85 deg C and 32msec for environments operating above 85 deg C.

With systems moving towards increased consolidation (e.g., virtualized environments), DRAM refresh becomes a significant bottleneck as it reduces the available overall DRAM bandwidth per task. In this work, we propose a hardware-software co-design to mitigate DRAM refresh overheads by exposing the hardware address-mapping and DRAM refresh schedule to the operating system (OS). In our co-design, we propose a novel per-bank refresh schedule in the hardware which augments memory partitioning in the OS. Supported by the novel per-bank refresh schedule and memory-partitioning, we propose a refresh-aware process scheduling algorithm in the OS which schedules applications on cores such that none of the on-demand requests from the applications are stalled by refreshes. The evaluation of our proposed co-design using multi-programmed workloads from the SPEC CPU2006, STREAM and NAS suites show significant performance improvements compared to the previously proposed hardware-only approaches.

***CCS Concepts*** • **Computer systems organization** → **General-Hardware/software interfaces**; • **Hardware** → *DRAM Memory*; • **Software** → Operating Systems

***Keywords*** DRAM refresh, Operating Systems, Task Scheduling, Hardware-software co-design.

## 1. Introduction

Dynamic Random Access Memory (DRAM) is the predominant main memory technology used in computing systems today. DRAM cells use capacitors as data storage devices. Since capacitors leak charge over time, DRAM cells need to be periodically refreshed inorder to preserve data integrity. These periodic refresh operations block main memory accesses, therefore reducing main memory availability and increasing effective memory latency. This problem is even more accentuated in consolidated environments like virtualized systems.

With technology scaling enabling increase in number of homogeneous and heterogeneous cores on-chip [14, 20, 21, 23, 33], similar scaling is observed in the DRAM device densities as well over the last several decades. These scaling trends have enabled higher main memory capacities in all computing segments, paving the path for higher system performance and increasingly sophisticated software. However, as the total number of DRAM cells in a system continues to increase, the DRAM refresh overheads are on the rise and are threatening to dampen the performance benefits of DRAM capacity scaling. Recent studies have shown that for upcoming 32Gb DRAM devices, DRAM refreshes can cause a 30% reduction in overall system throughput [22].

Many recent papers have proposed hardware [15] [30] [12] and software [25] [35] solutions to mitigate the performance overheads caused by DRAM refreshes. These approaches can be broadly classified into two categories: (i) reducing the number of refreshes, and (ii) overlapping memory accesses with refreshes. Techniques belonging to the first category reduce refresh activity by refreshing each DRAM row at a different rate, dictated by the cell with the low-



*ASPLOS '17,* April 08-12, 2017, Xi'an, China



DOI: http://dx.doi.org/10.1145/3037697.3037724

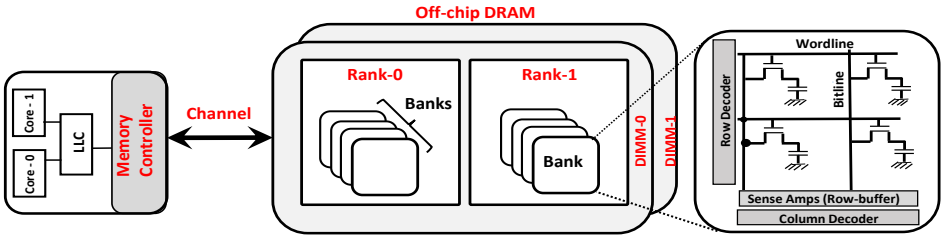



Figure 1: Basic DRAM organization.

est retention time in that row. While these techniques can reduce the number of refresh operations substantially, they rely on accurate retention time profiling, which is costly to implement and is highly prone to erratic changes in DRAM cell retention times [30]. The second category of techniques reduces the exposed refresh overhead by allowing regular DRAM accesses to proceed in parallel with refresh operations. The key idea behind these techniques is to confine the refresh activity to a portion of the DRAM (such as a bank or a subarray), so that refresh operations in one portion will not interfere with accesses to the other (non-refreshed) portions.

The most recent example of such finer-granularity refreshing adopted by the DRAM industry is the per-bank refresh scheme supported in LPDDR3 [10] and beyond. As opposed to the traditional all-bank refresh scheme in earlier LPDDRx generations (and current DDRx generations), a refresh command in the per-bank refresh scheme targets only one DRAM bank. Therefore, while a per-bank refresh command is busy refreshing rows in one bank, all the other banks are available to service regular DRAM accesses. In an ideal scenario, if all the DRAM requests that arrive at the DRAM controller during a refresh operation are headed to the available (non-refreshed) banks, then the refresh overhead can be fully hidden. However, in realistic scenarios, since memory requests generated by typical programs are often uniformly distributed across DRAM banks, the probability of a DRAM request being blocked by a per-bank refresh is quite high. Therefore, as shown in prior studies, per-bank refresh is only marginally effective in avoiding the DRAM refresh overheads [15].

In this paper, we propose a hardware-software co-design technique to mitigate the DRAM refresh overheads. Our technique exposes per-bank refresh to the operating system (OS) with the goal to enable higher overlap between refresh operations and regular memory accesses. The key idea behind our technique is to incorporate DRAM bank awareness and refresh schedule in the memory allocation and task scheduling decisions made by the OS. Specifically, our technique proposes the following two main changes to the operating system: (i) the OS memory allocator confines the memory allocated by a task to a subset of the available DRAM banks, and (ii) the OS task scheduler chooses the tasks scheduled during a quantum in such a way that the memory accesses made by these tasks do not span all the DRAM banks in the system. Furthermore, our technique proposes the following change to the refresh scheduler in the memory controller: rather than doing a round-robin scheduling of refresh commands to individual banks, the memory controller refreshes only those banks during a task scheduling quantum which are not expected to receive any memory requests during that quantum. With this careful collaboration between OS and the memory controller, our technique reduces the probability of per-bank refreshes interfering with regular DRAM accesses.

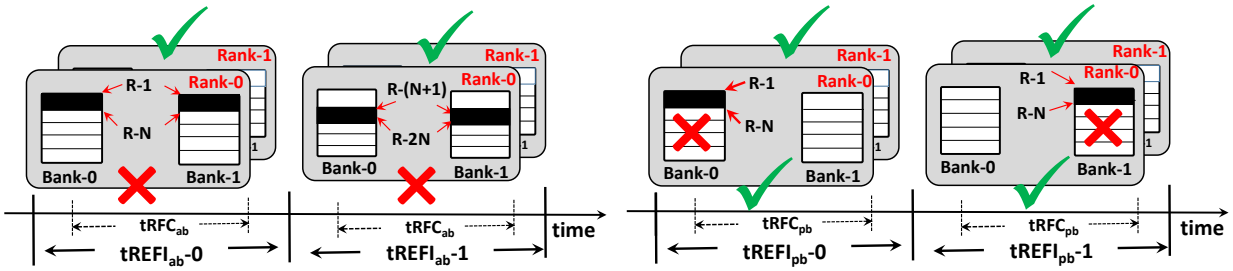
Extensive evaluations of our proposed technique on multi-programmed SPEC CPU2006 [6], STREAM [7] and NAS [5] workloads show that our technique achieves 16.2% and 6.3% performance improvement over all-bank and per-bank refresh for 32Gb DRAM chips, respectively. Our results also show that the co-design improves the performance by 14.6% and 6.1% on an average compared to previously proposed Adaptive Refresh (AR) [27] and per-bank Out-Of-Order refresh [15] respectively, without necessitating any modifications to the internal DRAM structures.

## 2. Background

In this section, we cover the background on basic DRAM organization introducing how the refreshes are scheduled to the DRAM banks by Memory Controller (MC). Also, we briefly touch upon how the OS (Linux) allocates memory by traversing through the free-lists, and finally give a brief primer on the current process scheduling algorithm used by the OS to schedule tasks on processor cores.

### 2.1 DRAM Organization

As shown in Figure 1, a typical DRAM hierarchy is made up of channels, ranks and banks [34] [21]. Each on-chip memory controller (MC) manages corresponding DIMMs by issuing various commands over the command bus, and the corresponding data is traversed over the data bus. Each DIMM, as shown in Figure 1, is made up of multiple DRAM ranks, while each rank further consists of multiple banks, as also shown in the figure. Each bank consists of DRAM cells laid out in rows (typically the size of a DRAM page, 4KB or 8KB) and columns connected by wordlines and bitlines, respectively. The data in each DRAM row is accessed by activating the row into DRAM sense-amplifiers (also referred to as row-buffers) through a RAS command, after which a




(a) (b)

Figure 2: (a) All-bank refresh. (b) Per-bank refresh with $tREFI_{pb}$ = $tREFI_{ab}$/(numBanks).

corresponding cache line from an activated row is accessed by a CAS command. Hence, a row-buffer caches the most recently opened row till it is precharged explicitly. Since accessing data from an already opened row (present in row-buffer) is faster, different row-buffer management policies have been proposed by researchers in the past [31] [19] [36], which aim at decreasing the overall memory latency.

### 2.2 DRAM Refresh Scheduling

DRAM cells are typically made up of an access transistor and a capacitor. Over time, DRAM cells leak charge and hence need to be refreshed periodically to maintain the data integrity. DRAM cell retention times (denoted by tREFW) are often a function of the operating temperatures and process variation [22] [15]. Typically, tREFW is 64msec for environments operating in temperatures ¡ 85 deg C, while it is halved to 32msec when temperature is beyond 85 deg C. Instead of refreshing all the DRAM rows at once, MC issues refresh command once in every refresh interval (denoted by tREFI). Typically, tREFI is in the order of $\mu$seconds and is generally 7.8 $\mu$secs for DDR3, while finer refresh granularities are supported for DDR4 in the 2x and 4x modes, where tREFI is 3.9 $\mu$secs and 1.95 $\mu$secs, respectively [27]. Each refresh operation issued by an MC lasts for a refresh cycle time (denoted by tRFC), which is typically in the order of several *nano seconds*. tRFC is a function of the employed tREFI and the number of rows to be refreshed. tRFC increases with the increase in the density of DRAM [15] [27] [12], causing significant performance bottlenecks for high capacity DRAMs. Commercial DDR cells are refreshed at a rank level, while the mobile LPDDRs can be refreshed at a per-bank granularity. Figures 2a and 2b illustrate the refresh operations when rows are refreshed at a rank-level and bank-level, respectively, in a system comprising of 2 ranks and 2 banks per rank.

#### 2.2.1 All-bank refresh

As shown in Figure 2a, a refresh operation issued at the rank-level refreshes a certain number of rows (say N) in all the banks in that rank. Figure 2a depicts that rows R-1 to R-N are refreshed in banks B-0 and B-1 during the first refresh interval (indicated by $tREFI_{ab}$-0), while rows R-(N+1) to R-2N are refreshed during the second refresh interval $tREFI_{ab}$-1. As can be observed in Figure 2a, in a given $tREFI_{ab}$, since all the banks in a rank are being refreshed, the entire rank-0 is not available as indicated by $\times$ in Figure 2a for rank-0 for $tRFC_{ab}$ duration, while rank-1 is available as indicated by ✓. Since the entire rank is not available to serve the on-demand memory requests during $tRFC_{ab}$, performance degradation is significant in all-bank refresh as opposed to per-bank refresh.

#### 2.2.2 Per-bank refresh

To increase the availability of the number of banks during refresh, LPDDRs allow refresh commands to be issued at a bank granularity. Figure 2b depicts the per-bank refresh employed by LPDDRs. Since refreshes are issued at a bank granularity, the refresh interval employed by per-bank (denoted by $tREFI_{pb}$) is smaller than that of $tREFI_{ab}$ and $tREFI_{pb}$ = $tREFI_{ab}$ / (numBanks). As can be observed in Figure 2b, rows R-1 to R-N in Bank-0 are refreshed in $tREFI_{pb}$-0; as a result, only Bank-0 is not available during $tRFC_{pb}$ (denoted by $\times$), while the other banks in Rank-0 are available to serve on-demand requests. In $tREFI_{pb}$-1, as can be observed from Figure 2b, the same rows R-1 to R-N in Bank-1 are refreshed, while Bank-0 is available for on-demand requests. Hence, banks in all the ranks are refreshed in a round-robin fashion in per-bank refresh [15]. Since not all the banks in a rank are refreshed in a given $tREFI_{pb}$ in per-bank refresh, performance degradation in per-bank refresh is not as catastrophic as in all-bank refresh.

### 2.3 Linux Memory Allocator

Linux uses a buddy memory allocator [4] to allocate physical addresses for applications. It maintains free-lists per zone to cater to the memory allocation requests. The traditional Linux memory allocator is oblivious to the DRAM bank organization, and consequently any given application can have memory allocated in all the DRAM banks depending on the memory footprint of the application. Such DRAM-oblivious memory allocation accommodates for better bank-level parallelism (BLP) for applications; however, in some multi-programmed environments, it can lead to memory interference as well [24] [37]. Such interference

in the multi-programmed environments result in not just the contention for memory bandwidth but also poor row-buffer locality in DRAMs, thereby degrading performance. To avert this memory interference, researchers have proposed DRAM bank-aware memory partitioning [24] [37], where the OS memory allocator is aware of the hardware address-mapping, viz, channel, rank and bank bits and can allocate memory such that certain applications will access certain DRAM banks, reducing the interference. However, since such a partitioning limits the bank-level parallelism (BLP), researchers have also proposed dynamic mechanisms to balance BLP vs row-buffer locality [18]. Hence the OS memory allocator plays a crucial role in managing various shared on-chip resources including the memory bandwidth.

### 2.4 Linux Process Scheduling

Linux kernel past 2.6.23 version uses Completely Fair Scheduler (CFS) to schedule tasks across processor cores [3] [26]. CFS uses a notion called the "virtual runtime", which indicates the next time-slice[1] when a task[2] will be scheduled. CFS employs time-ordered red-black tree data structure where the tasks are sorted by the vruntime. The left-most task in the red-black tree is chosen by the CFS scheduler as it is the oldest executed task among the runnable tasks. CFS manages red-black tree per CPU and in a multi-CPU system, CFS runs the load-balancer in the background to maintain an equal number of tasks in the per-CPU queues to maximize the overall throughput [26]. The time-slice of the CFS scheduler in Linux is typically in the order of 1-5msec [17][3]. Since the OS scheduler schedules the tasks on the CPU, it provides ample opportunities to schedule suitable tasks if some of the underlying hardware bottlenecks are exposed to the OS to improve the overall system throughput. In current-day systems, task scheduling in the OS is agnostic of the refresh scheduling in the DRAM. Such an independent schedule of tasks and DRAM refreshes causes significant performance problems. In this paper, we propose hardware-software co-design where the OS partitions the memory across tasks thereby enabling the OS task scheduler to schedule processes in a *refresh-aware fashion*.

## 3. Motivation

### 3.1 Performance Degradation due to DRAM Refresh

As explained in Section 2.2, since all banks in a rank are not available during refresh, all-bank refresh is more detrimental to performance compared to per-bank refresh since in the latter only one bank will be refreshed during a refresh interval. Figure 3 shows the performance degradation for different DRAM device densitities. As can be observed, for operating temperatures below 85 deg C where the DRAM

[1] We use time-slice and time quantum interchangeably in this paper.

[2] We use "application", "benchmark", and "task" interchangeably in this paper.

[3] We observed similar values for time-slice in our full-system experiments.

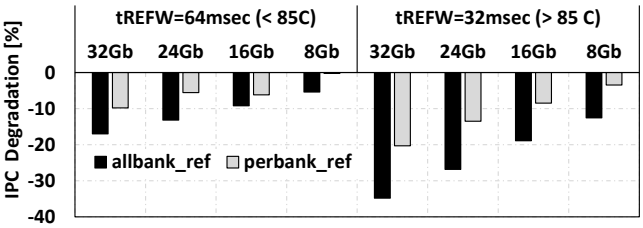



Figure 3: Performance degradation due to refresh.

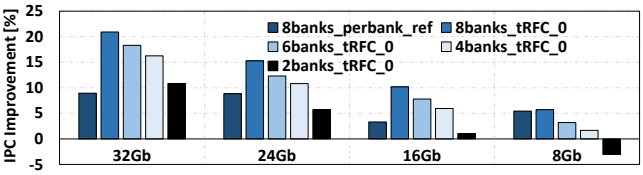



Figure 4: IPC Improvements for various DRAM densities, normalized to a scenario where each application uses all the 8 banks in a rank.

retention time (tREFW) is 64 msecs, as the chip density increases from 8Gb to 32Gb, performance degrades from 5.4% to 17.2% for all-bank refresh on an average. However, for per-bank refresh, the degradation on an average varies from 0.24% to 9.8%. This shows that refresh becomes much of a problem with growing DRAM densities since tRFC, the refresh cycle time increases from 350nsec for 8Gb to 890nsec for 32Gb device densities. Also, as device density increases from 8Gb to 32Gb, per-bank refresh also degrades performance significantly, by as much as 9.8% as can be observed from Figure 3.

DRAM refresh is much more detrimental to performance when the operating temperature is beyond 85 deg C, where the retention is 32 msecs, meaning the DRAM rows need to be refreshed twice as frequently. As can be observed from Figure 3, all-bank refresh degrades the performance by up to 34.8% for 32Gb chips on an average, while per-bank refresh degrades performance by up to 20.3%. This shows that DRAM refresh is an important problem that needs to be addressed for the future DRAMs with growing chip densities. The performance degradation due to refresh is expected to be even more pronounced in multi-programmed workloads where multiple high memory-intensive applications are often executed concurrently.

### 3.2 Refresh Cycle Time (tRFC) vs Bank Level Parallelism (BLP)

As explained in Section 2.3, the traditional Linux OS is agnostic of DRAM bank organization and allocates data for applications which span across all the DRAM banks. A positive side-effect of such an allocation scheme is increased bank level parallelism (BLP). In our scheme, since the OS partitions memory across tasks, it is important to understand how partitioning an application to access a subset of banks effects performance. Memory-partitioning by OS can increase the DRAM row-buffer locality for certain applications as there will not be any interference from other applications. Since our hardware-software co-design requires

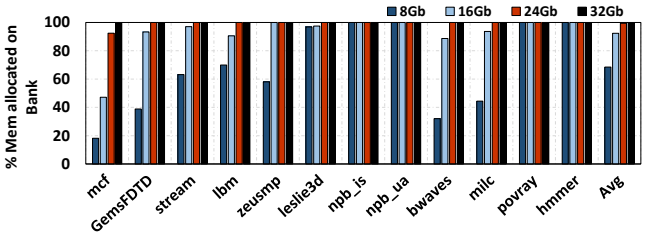



Figure 5: Percentage of memory that can be allocated on a single bank with increasing DRAM chip densities (normalized to total memory footprint).

partitioning applications' data across DRAM banks and our scheme will ultimately remove the entire tRFC overheads, we present results of various such scenarios in Figure 4. As can be observed from this figure, confining applications to a subset of available banks still yields better performance compared to the all-bank refresh if the entire tRFC overheads can be eliminated. Furthermore, confining applications to a maximum of 4 banks per rank (total of 8 banks per channel) still yields improvement in performance in future (16Gb, 24Gb, 32Gb) high-density DRAM chips. However, currently-available density of 8Gb with a lower tRFC, confining an application to few banks degrades the performance as expected, since the BLP is reduced.[4] This result shows that confining applications to a subset of DRAM banks can still yield significant improvements in performance if the entire DRAM refresh related overheads are eliminated.

### 3.3 Feasibility of Bank-Partitioning from a Capacity Stand-point

Having looked at the performance impact of memory-partitioning, we now evaluate the feasibility of memory-partitioning from a capacity stand-point. Since confining an application to a subset of banks limits the overall memory capacity available for an application, it is important to understand the capacity demands imposed by applications. If an application has high memory footprint, confining its data to a subset of banks will increase the number of page-faults in the system even though there is free memory available in the other DRAM banks. Such page-fault scenarios can cause significant degradation in performance. In this subsection, we evaluate the memory footprints of the SPEC CPU 2006 workloads using reference (large) input datasets and the feasibility of bank-partitioning for these applications from the capacity stand-point. Figure 5 shows the percentage of memory that can be allocated on each bank with different chip densities, normalized to the total footprint of each application. These results are collected by modifying the default Linux kernel buddy memory allocator, such that the kernel tries to allocate the maximum amount of memory on bank-0. If this cannot be done after a while, the fall-back mechanism would allocate data on other banks using the buddy memory allocator.

Figure 5 indicates that for the current-day DRAM chip density of 8Gb, on an average, 68% of applications' total footprint can fit into a single bank. And, this percentage of footprint that can be fit in a single bank increases with the increase in chip density, as can also be noted from Figure 5, making bank partitioning-based memory allocator more and more feasible[5] from a capacity stand-point.

## 4. Overview of Our Problem

### 4.1 Problem

Figure 6a depicts the modern day dual-core system, two cores C-0 and C-1 executing four tasks T0 - T3 (each denoted by a different pattern). As explained in Section 2.3, Linux allocates data for these tasks in a DRAM-oblivious fashion and hence the data for each task are allocated across all the DRAM banks. In Figure 6a, the memory allocated for each task by the OS is depicted with the same pattern as the task itself. Consequently, all the tasks T0-T3 can access data from any of the DRAM banks B0 - B3. Figure 6b shows the implications of all-bank refresh on a conventional system. Since none of the banks in a rank are available to serve the on-demand requests in all-bank refresh, the probability of the tasks T-0 and T-2 waiting on the data from the banks B0 - B3 is high. Figure 6b depicts such a scenario where cores C-0 and C-1 are stalled on the outstanding loads (depicted in the MC queue) to be served by the banks being refreshed. However, for per-bank refresh, since only one bank will be busy refreshing, the probability that both cores stalling due to a bank is low. As depicted in Figure 6c, there could be not-so-worse scenarios where only one core could be stalled due to per-bank refresh. However, since data of all the tasks are spread across all the DRAM banks, the worst-case scenario of both cores stalling due to a refreshed bank is still possible as depicted in Figure 6d. Hence, as observed in Section 3.1, allbank-refresh is more detrimental to performance compared to perbank-refresh.

### 4.2 Our Solution

Building on the per-bank refresh support[6], we propose a hardware-software co-design to mitigate DRAM refresh overheads by making changes in both the hardware and the OS. Our proposals are based on the observation that the DRAM retention time (tREFW) and the OS time quanta are in the same order of milliseconds, and include both hardware and software modifications with the goal of eliminating entire DRAM refresh overheads. To this end, we propose a novel and simple per-bank refresh schedule in the hard-

[4] Since per-bank refresh yields maximum benefit in 8Gb chips, we do not consider 8Gb in our experiments in the sub-sequent sections.

[5] Please refer to section 5.4.1 on how the applications with higher overall memory footprint are dealt in our co-design.

[6] Note that, per-bank refresh already performs significantly better compared to the other prior proposals which are built upon the all-bank refresh strategy as demonstrated in [15].

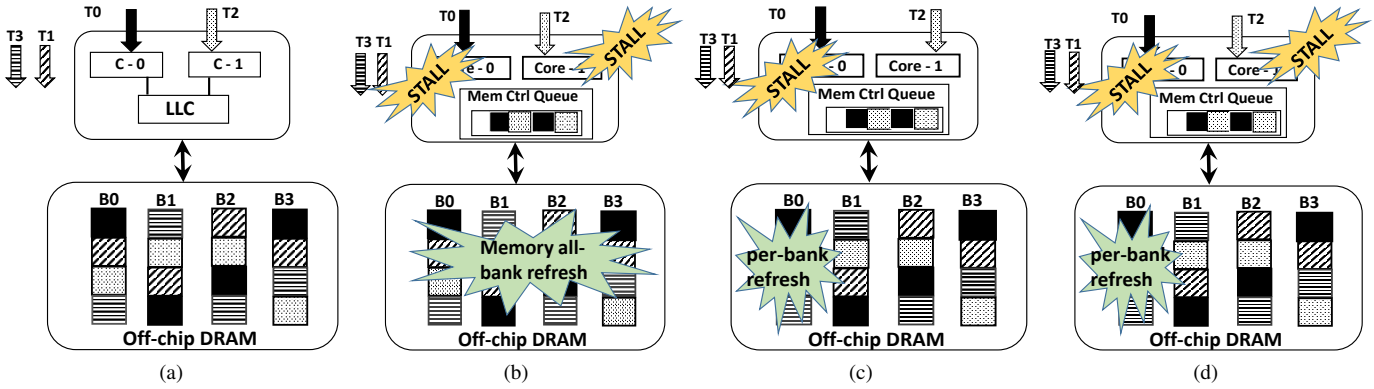



Figure 6: Implications of different refresh mechanisms on applications. Diagrams depict (a) Conventional dual-core system executing 4 tasks, (b) worse-case scenario where cores stalled due to all-bank refresh, (c) Not-so-worse scenario where only one core is stalled due to per-bank refresh, and (d) worse-case scenario where both cores can get stalled due to per-bank refresh.

ware which facilitates interesting solutions at the software-level. At the software-level, we use a simple soft-partitioning based memory allocator in the OS which augments the proposed per-bank refresh schedule in the hardware. Together, the proposed memory allocator and the proposed per-bank refresh scheduler enable the OS scheduler to schedule the applications in a *refresh-aware fashion*. That is, our proposed hardware per-bank scheduler and soft-partitioning based memory allocator present an opportunity for the OS to schedule an application which does not access the bank being refreshed in it's entire time-quantum[7]. This in turn increases the probability that an applications' on-demand requests are not stalled due to refresh.

## 5. Hardware-Software Co-design

### 5.1 Proposed Hardware Changes

Our proposed changes to the per-bank refresh schedule are depicted in Figure 7. Comparing Figures 2b and 7, it can be observed that, in our proposed schedule, in $tREFI_{pb}$-1, instead of refreshing rows R-1 to R-N of Bank-1, we refresh rows R-(N+1) to R-2N. That is, contrary to the default round-robin per-bank refresh scheduler, our per-bank refresh scheduler schedules refreshes to the same bank (to different rows) in successive refresh intervals until all the rows in a bank are refreshed. The pseudo-code for our new per-bank refresh scheduler is given in Algorithm 1.

**Implications of our per-bank refresh schedule:** Consider a typical system operating in environments below 85 deg C with a tREFW of 64 msec, containing 2 ranks and 8 banks per rank. In this system, with a total of 16 banks, using our proposed per-bank refresh schedule, all the rows in Bank-0 are done refreshing at the end of first 4msec. Since Bank-0 will be refreshed again only after the 64msec, Bank-0 will be available to serve the on-demand memory requests uninterruptingly after the first 4msecs in a 64msec refresh window. As covered in Section 2.4, this duration of 4msec coincides with the process scheduling time quantum used by the OS. Consequently, the new per-bank refresh scheduler enables interesting options for task scheduling in the OS if the applications' memory could be carefully partitioned such that *not* all the banks contain data from all the applications.

[7] Please refer to our best-effort process scheduler explained in Section 5.4.1 which picks the task with minimum amount of data allocated on bank being refreshed incase if there are high memory footprint tasks being executed on the system

**Algorithm 1** Proposed per-bank refresh schedule algorithm.

```
1: /* nextRefreshBank and the nextRefreshRank represents the bank and the corre-
   sponding rank that will be refreshed in the next tREFI_pb. */
2: refreshBankIdx = (nextRefreshRank * numBanksPerRank) + nextRefreshBank
3: /* numRowsRefreshed keeps track number of rows refreshed in a bank.*/
4: numRowsRefreshed[refreshBankIdx] += RowsPerRefresh;
5: if numRowsRefreshed[refreshBankIdx] < numRowsPerBank then
6:     nextRefreshBank = nextRefreshBank;
7: else
8: /* Done refreshing the entire bank. schedule the refresh to the next bank */
9:     numRowsRefreshed[refreshBankIdx] = 0;
10:    nextRefreshBank += 1
11: end if
12: if nextRefreshBank >= numBanksPerRank then
13:    nextRefreshBank = 0;
14:    nextRefreshRank = (nextRefreshRank + 1) % numRanks;
15: end if
```

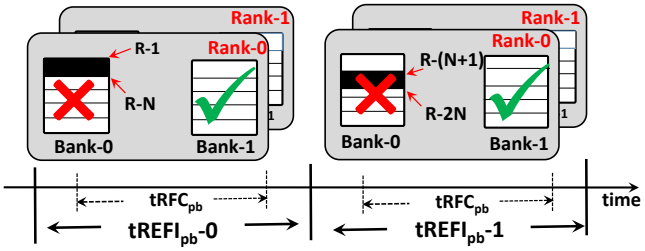



Figure 7: Proposed per-bank refresh schedule.

### 5.2 Proposed Software (OS) Changes

#### 5.2.1 Memory Partitioning Based Allocator

Various DRAM bank-aware memory partitioning algorithms have been proposed in [37] [24] to alleviate the interference across applications running on different cores. We envision two different ways of partitioning memory as depicted in Figures 8a and 8b.

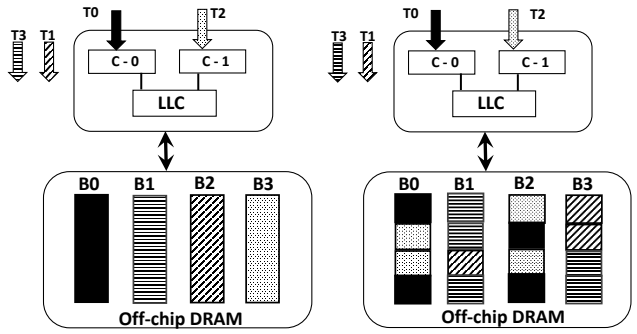



Figure 8: (a) Hard-partitioning, and (b) Soft-partitioning based memory allocators.

**Hard-partitioning based allocator:** Figure 8a shows the hard-partitioning based memory allocator. In such an allocator, each memory bank or a group of banks can host data only from a certain task. As depicted in Figure 8a, task T0's data is allocated in bank B-0, task T1's data is allocated in bank B-2, task T2's data is allocated in bank B-3, and T3's data is allocated in bank B-1.[8] Liu et al [24] proposed such a hard-partitioning based memory allocator. By allocating a task's data exclusively on a subset of banks, such an allocator alleviates the memory bank contention, thereby increasing row-buffer locality. However, there are certain drawbacks to such an allocator:

- Confining applications to certain set of banks results in poor bank-level parallelism (BLP) [18] for applications that do not have high row-buffer locality, e.g., irregular applications and pointer-based applications.
- Hard-partitioning can cause a task to page-fault when it is under-provisioned in terms of the number of banks, even if the other banks contain free memory. Such a scenario can be catastrophic to performance.
- With the increasing number of cores on-chip, hard-partitioning limits the overall memory bandwidth available for a task causing the performance to degrade compared to the baseline DRAM bank-agnostic memory allocation.

**Soft-partitioning based allocator:** An alternative to hard-partitioning is "soft-partitioning" where a group of tasks share a subset of DRAM banks, as depicted in Figure 8b. In the soft-partitioning scheme, instead of dedicating a DRAM bank to an application, a DRAM bank is loosely partitioned such that a group of tasks can share it. In Figure 8b, tasks T-0 and T-2 have data allocated in banks B-0 and B-2, while tasks T-1 and T-3 have data allocated in banks B-1 and B-3. Hence, such a soft-partitioning based allocator increases the overall memory utilization by sharing the capacity with other tasks and is more likely to reduce the number of the page-faults in a system. It also caters to the increased BLP, thereby increasing the overall memory bandwidth available for a task at the cost of row-buffer locality.

[8] Note that though in this example each task's data is allocated in only one bank, the OS can allocate multiple banks to a task. However, other tasks cannot have data allocated in these banks.

**Algorithm 2** Proposed memory-partitioning algorithm.

```
1: procedure GET_PAGE_FROM_FREELIST(..., unsigned int order, ..., struct zone
   *preferred_zone, int migratetype)
2: /* current –> pointer to the current task which requested the memory allocation
   */
3: /* free_list –> original free list maintained by the OS */
4: /* free_list_per_bank –> per bank free-list */
5: /* possible_banks_vector –> Bit mask representing bank bits.*/
6: /* lastAllocedBank represents the bank amongst the possible banks where the last
   memory request is allocated for the current task. */
7:     for each order in MAX_ORDER do
8:         count = 0
9:         for count < num_total_banks do
10:            allocBank = current–>lastAllocedBank;
11:            allocBank = (allocBank+1) % num_total_banks;
12:            if current–>possible_banks_vector[allocBank] is set then
13:                if free_list_per_bank[bank] is not empty then
14:                    /* Hit from a per bank free list */
15:                    page = free_list_per_bank[bank];
16:                    current–>lastAllocedBank = allocBank;
17:                    return page;
18:                else
19:                    /* Fetch a page from OS free-list */
20:                    page = list_entry(free_list, ....);
21:                    nr_free--; /* Decrementing the number of OS free pages */
22:
23:                    /* Since OS is exposed with hardware address-mapping infor-
   mation, we can get the bank id from the physical page address */
24:
25:                    bank = get_bank_id_from_page(page);
26:
27:                    if allocBank == bank then
28:                        /* Matches the round-robin bank */
29:                        current–>lastAllocedBank = allocBank;
30:                        return page;
31:                    else
32:                        /* Maintaining a cache of per bank free-lists*/
33:                        insert_in_to_free_list(free_list_per_bank, bank, page)
34:                    end if
35:                end if
36:            end if
37:            count++;
38:        end for
39:
40:    end for
41:    return NULL;
42: end procedure
```

Algorithm 2 shows the detailed pseudo-code for our general memory-partitioning allocator which can either hard-partition or soft-partition data across DRAM banks. We implemented and verified this algorithm in the actual Linux buddy allocator for our experiments. As can be observed from lines 15 and 33, we maintain a free-list of pages per bank so that a free page corresponding to a bank is known readily without traversing the OS free-list. Also, the possible_banks_vector used in line 12 is a bit-mask which represents the possible list of banks that contain data from this particular task. In our current implementation, this possible_banks_vectors bit-mask is an input taken from the user using debugfs [2] and cgroups [1] features in Linux. Hence, our partitioning based allocation presented in Algorithm 2 is generic for both the hard-and soft-partitioning schemes, and can be configured dynamically based on the possible_banks_vector bit-mask. One more important aspect to be noted in our implementation from lines 10-11 is that our

memory-partitioning allocator allocates pages such that the consecutive allocation requests[9] fall into different banks in a round-robin fashion by keeping track of lastAllocedBank per task, thereby improving BLP. In our experiments, we observed that soft-partitioning yields better performance as the number of applications running concurrently increases. This is because the memory bandwidth per task increases with soft-partitioning.

**Algorithm 3** Proposed refresh-aware process scheduling.

```
1:  procedure PICK_NEXT_TASK(struct rq *rq)
2:
3:      /* nextRefreshBank -> represents the next bank to be refreshed in DRAM
    based on the new per-bank refresh schedule */
4:
5:      struct task_struct *p;
6:      struct cfs_rq *cfs_rq = &rq->cfs;
7:      struct sched_entity *se;
8:      struct sched_entity *firstSchedEntity;
9:
10:     if !cfs_rq->nr_running then
11:         return NULL;
12:     end if
13:     found_task_flag = false;
14:     count = 0;
15:
16:     do
17:         count++;
18:         se = pick_next_entity(cfs_rq);
19:         set_next_entity(cfs_rq, se);
20:         cfs_rq = group_cfs_rq(se);
21:         p = task_of(se);
22:
23:         if count == 1 && cfs_rq then
24:             firstSchedEntity = se;
25:         end if
26:
27:         if cfs_rq && p->possible_banks_vector[nextRefreshBank] is not set then
28:             found_task_flag = true;
29:         else if cfs_rq && count >= η_thresh then
30:             found_task_flag = true;
31:             p = task_of(firstSchedEntity);
32:         end if
33:     while !flag_found_task;
34:     / ******* Some more Code ***/
35:     return p;
36: end procedure
```

### 5.2.2 DRAM Refresh-Aware Process Scheduling

The new per-bank refresh schedule and memory-partitioning proposed in the previous subsections provide an opportunity for the OS to schedule tasks in a *refresh-aware fashion*. The pseudo-code for the proposed refresh-aware process scheduler is presented in Algorithm 3. As can be noticed in Algorithm 3, nextRefreshBank represents the next bank to be refreshed based on our new per-bank refresh schedule. The code-snippet in the algorithm is the actual implementation in the Linux CFS scheduler, which returns the next task to be scheduled on a core. Our refresh-aware implementation is depicted in line 27 where the next runnable task chosen is the one that does not have any data allocated on the bank which will be refreshed in the next time quantum. This task to be scheduled is one among the tasks to the left in the red-black tree maintained by the CFS scheduler. Hence, by chosing the task which is left among the runnable tasks in the red-black tree, our scheduler tries to schedule a task which does not have any data allocated in the bank to be refreshed.

[9] We do not consider large-pages in our evaluation, hence each allocation granularity is 4KB.

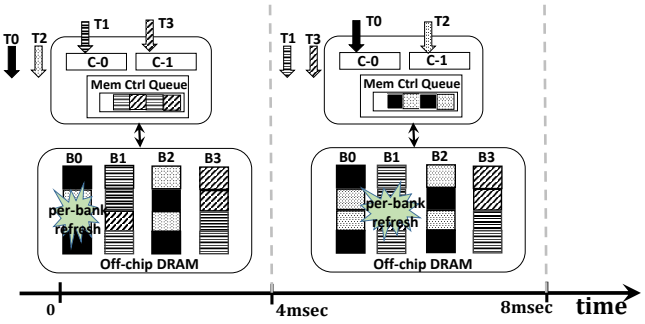



Figure 9: Our co-design depicted with soft-partitioning allocator.

### 5.3 Putting It All Together

Figure 9 depicts the bigger picture of how our co-design works. As discussed in Section 5.1, our proposed per-bank refresh schedule results in Bank B-0 being refreshed in the first 4msec, bank B-1 in 4-8msec, and so on. Figure 9 shows how data for tasks T0-T4 are allocated based on the soft-partitioning discussed in Section 5.2.1. The data of tasks T0 and T2 are allocated on banks B0 and B2, while T1 and T3 have their data allocated on banks B-1 and B-3. Since bank B-0 containing data allocated by tasks T-0 and T-2 will be refreshed in the first 4msec, our refresh-aware OS scheduler schedules tasks T1 and T3 on cores C-0 and C-1. After 4 msec, since bank B-1 will be refreshed from 4-8 msec, tasks T0 and T2 will be scheduled by our refresh-aware scheduler, thereby ensuring that none of the on-demand requests from the scheduled tasks are stalled by the refreshes.

### 5.4 Caveats

In a real-life system, there could be varying scenarios where the process scheduling is disrupted by the external factors. Such scenarios include:

- A high priority task enters the system warranting for it to be scheduled for more number of time quantums compared to the other tasks.
- It could be possible that the desired tasks to be scheduled (that do not have data allocated on the bank to be refreshed) are not in runnable queue as they are in other states e.g., sleep state.
- A non-maskable interrupt needs to be serviced immediately by the core.

In all the above scenarios, our refresh-aware scheduler might result in loss of fairness. To address these issues, "fairness_threshold", denoted by $\eta_{thresh}$ and depicted in line 29 of Algorithm 3, can be used to disable our refresh-aware co-schedule immediately by setting this parameter to 1 or gracefully by setting to some value like 2 or 3. This $\eta_{thresh}$

parameter can be used by the user to over-ride the refresh-aware scheduling decisions at-will using the sysctl_sched interface present in the Linux kernel.

#### 5.4.1 Large Memory footprint applications

The per-task (benchmark) footprints of some tasks used in our workloads are: mcf : 1.7GB; bwaves : 920MB; stream : 800MB; GemsFDTD : 850MB.

Since each workload comprises of multiple such tasks (up-to 8 for dual-core and 16 for quad-core), the total memory footprint of workloads used in our evaluation are in several GBs (maximum of up to 27.2GB for quad-core[1:4-ratio] for WL1).

However, the futuristic workloads may have much higher footprints where fitting the entire applications' memory in the soft-partitioned banks may not be entirely feasible. In such scenarios, our memory-partitioning and refresh-aware scheduling-algorithms can easily be generalized as follows:

- Soft-partitioning allocator initially exhausts the whole capacity from soft-partitioned banks after which the allocator falls-back to allocating data in other banks to cater to high-footprint tasks. For each task, OS can keep track of the percentage of memory allocated on each bank.
- Our refresh-aware scheduler can then perform best-effort-scheduling such that it schedules the task with a minimal percentage of allocated-data in the bank to be refreshed.

Such a generalized-mechanism will perform similar to default per-bank-refresh in the worst-case, since the refreshed-bank can still serve the on-demand requests for(tREFIpb-tRFC) duration out of $tREFI_{pb}$ duration.

## 6. Evaluation

### 6.1 Experimental Setup

We used a simulation based setup with modified Linux kernel to evaluate our co-design. For the simulation setup, we used the gem5 [13] simulator with the out-of-order CPU model integrated with NVMain [29] for the detailed memory model. The evaluated system configuration is given in Table 1, unless otherwise explicitly stated. Our default experiments are evaluated with 4 threads consolidated per core, with a default system executing 8 threads in total on 2 cores (a 1:4 consolidation ratio). As mentioned in Section 2.4, by registering a callback to switch_to( ) Linux system call in the gem5 simulator, we observed that each task in our workloads covered in Table 2 executes for a time-slice of 4 msec.

We used benchmarks from the SPEC CPU2006 [6] suite with the reference (large) input, STREAM [7] and UA from NAS [5] benchmark suite. We evaluated various multi-programmed workloads shown in Table 2, each using a mix of these benchmarks based on their memory intensities. We categorize an application with Misses Per Kilo Instruction (MPKI) higher than 10 as high memory intensive application, denoted by H in the table. Applications with MPKI values between 1 and 10 are categorized as medium, denoted by M, and those with MPKI values less than 1 are categorized as low. As shown in Table 2, our workloads are formed such that we cover a large spectrum of the memory intensity so that our performance results reported are representative and are not biased by high memory intensive workloads. For evaluation, we fast-forward applications to get to the region-of-interest after which the workload executes 100 million instructions to warm-up the LLC. We continue the simulation till each task in the workload executes a minimum of 200 million instructions. Once the last task finishes executing its 200 million instructions, we terminate the simulation and dump the statistics. Performance improvements reported in this section are the improvements in *harmonic mean* of the Instructions committed Per Cycle (IPC) of the workload relative to the baseline.[10]

| | | |
|---|---|---|
| **Hardware Config** | **Cores** | 2 cores @ 3.2GHz, Out-of-order, 8-wide issue, ROB: 128 Entries. |
| | **L1I/L1D $** | 32KB, 4-way associative, Hit latency: 2 cycles |
| | **L2 Cache** | 1MB per core, 2MB total, 16-way associative, Hit latency: 20 cycles, 64 Byte cache lines. |
| | **Memory** | DDR3-1600 [8], 1 channel, 1DIMM/channel, 2 ranks/DIMM, 8 banks/rank, FR-FCFS scheduler, open-row policy, Read/Write Queue Sizes: 64/64, Writes drained in batches [15] [16], Low/High watermarks: 32/54, 4KB DRAM row |
| | **Refresh Config** | $tREFI_{ab}$=7.8 $\mu$secs, tREFW=64msecs, $tRFC_{ab}$=530/710/890 nsecs for 16Gb/24Gb/32Gb, Rows/bank=256K/384K/512K for 16Gb/24Gb/32Gb, $tRFC_{ab}$-to-$tRFC_{pb}$ ratio = 2.3 [15], |
| **OS Config** | **Timeslice** | 4msec. |
| | **OS Scheduler** | CFS (round-robin). |
| | **Baseline Allocator** | Buddy Allocator without any memory partitioning. |
| | **Co-design Allocator** | Soft-partitioning based memory allocator. |

Table 1: Evaluated configuration.

| | Benchmarks | MPKI Category |
|---|---|---|
| WL-1 | mcf(8) | H |
| WL-2 | povray(8) | L |
| WL-3 | h264ref(8) | L |
| WL-4 | povray(4), h264ref(4) | L |
| WL-5 | GemsFDTD(8) | M |
| WL-6 | mcf(4), povray(4) | H + L |
| WL-7 | stream(4), h264ref(4) | M + L |
| WL-8 | bwaves(4), h264ref(4) | H + L |
| WL-9 | npb_ua(4), povray(4) | M + L |
| WL-10 | mcf(4), bwaves(2), povray(2) | H + L |

Table 2: Workloads used in evaluating our co-design in a dual-core system (1:4 consolidation ratio).

### 6.2 Co-design Results

Figure 10 shows the performance improvements of per-bank refresh and our refresh-aware co-design normalized to all-bank refresh for a dual core system with a 1:4 consolidation ratio. Note that, in our baseline, memory is not partitioned

[10] Note that in our baseline, each task is executed in a round-robin fashion with a time-slice of 4msec.

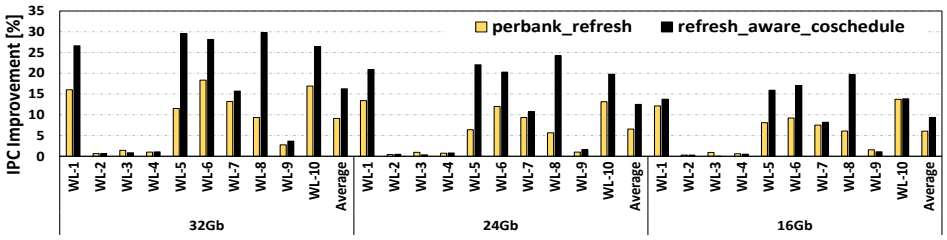



Figure 10: IPC improvement results (normalized to all-bank refresh).

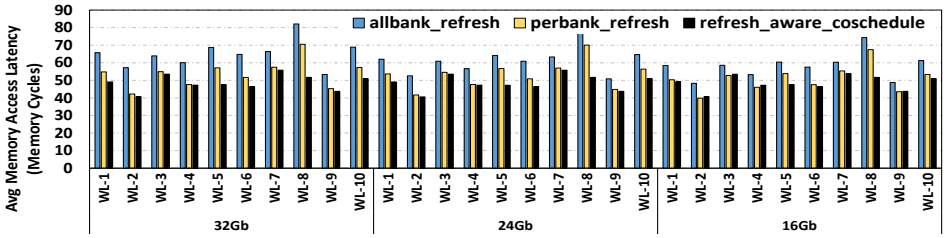



Figure 11: Average memory access latency results.

across tasks and a task can allocate data in all the 8 banks in a rank. For our co-design experiments, we confine each task to 6 banks within a rank so that not all tasks have data allocated on all 8 banks in a rank. Confining a task to 6 banks[11] in a dual-core system is the sweet-spot as it gives us good BLP (thereby reducing contention) as well as gives our co-design a flexibility to schedule tasks such that none of the on-demand requests are stalled by refreshes. As can be observed, our co-design scheme gives significant benefits over the all-bank refresh and per-bank refresh. Our co-design on an average improves the performance by 16.2%, compared to all-bank refresh, while it improves the performance by 6.3% over per-bank refresh for 32Gb chips. For 24Gb chip density, our co-design improves the performance by an average of 12.1% over all-bank refresh, and it improves the performance by an average of 5.4% over per-bank refresh. Furthermore, as the refresh overheads become less of a problem for 16Gb chips, our co-design improves the performance by 9.03% and 2.5% over the all-bank and per-bank refresh schemes, respectively. Figure 11 shows the corresponding average memory latencies in memory cycles per workload (lower the better in this graphs). As expected, the average memory access latencies are reduced significantly by our co-design as none of the tasks' on-demand requests are stalled by the refreshes.

The workloads WL-2, WL-3 and WL-4 are low memory-intensive, and hence not many of their on-demand requests are stalled by the refreshes in the baseline itself. Consequently, they do not get any improvement in performance from our co-design (as can be noted from Figure 10). WL1 as presented in Table 2 comprises of mcf applications which has a very high MPKI, compared to the other benchmarks categorized as high. Since our approach confines each task to 6 out of the 8 banks, the tasks executing at the same time contend for bandwidth from the confined banks. As a result, the improvement in performance is significant but still not as significant as other high MPKI workloads. As can be observed, WL5 comprising of the medium intensive applications like GemsFDTD and WL8 comprising of a mix of the high and low MPKI workloads give very good improvements as there is not much contention for bandwidth in the confined banks.

### 6.3 DDR4 Fine Granularity Refresh Results

Figure 12 shows how our co-design fares with DDR4-1600. As discussed in Section 2.2, DDR4 supports 1x, 2x and 4x refresh modes [9]. DDR4 1X mode employs a $tREFI_{ab}$ of 7.8 $\mu$secs, while 2x and 4x modes employ 3.9 $\mu$secs and 1.95 $\mu$secs respectively. While the $tREFI_{ab}$ is halved from 1x to 2x and 2x to 4x modes, $tRFC_{ab}$ for 2x/4x modes is scaled only by a factor of 1.35x/1.63x as observed in [15] [27]. Consequently, DDR4-2x and DDR4-4x modes fare worse than DDR-1x as more number of refresh commands are issued in a given refresh window, thereby causing more number of on-demand request stalls. To mitigate these refresh overheads, Adapative Refresh (AR) [27] dynamically switches between the DDR4-1x and DDR4-4x modes by monitoring the DRAM channel utilization at runtime. We present the comparison results of our co-design with AR in Section 6.5. As the entire refresh overheads are masked in our co-design, our co-design performs significantly better on an average compared to DDR4-1x, DDR4-2x and DDR4-4x modes, as can be noted in Figure 12 as we schedule tasks in a refresh-aware fashion.

### 6.4 Results with Lower DRAM Retention Time

As covered in Section 2.2, the DRAM retention time is halved to 32msecs in environments operating beyond 85 deg C. As a result, the DRAM refresh overheads become detri-

[11] We have experimented with 4 and 2 banks as well. While they improve performance,the improvements are not as high as 6 banks case.

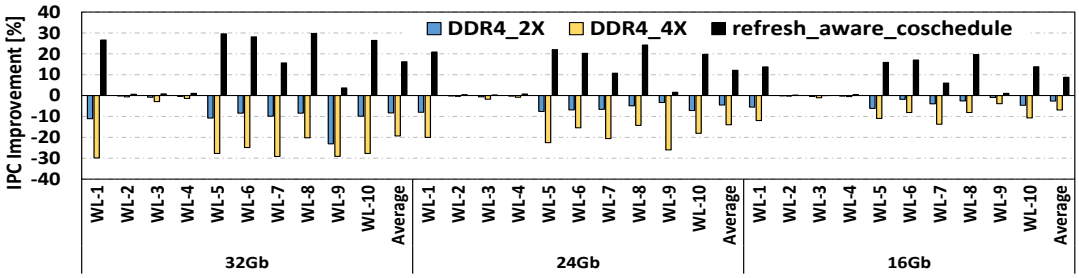



Figure 12: Comparison with FGR in DDR4 (normalized to allbank-refresh DDR4-1x mode baseline).

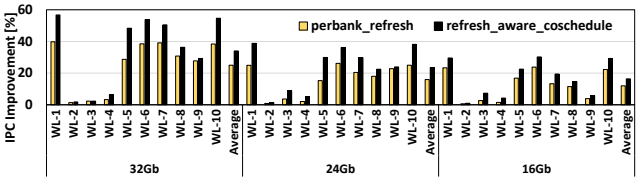



Figure 13: Results with the 32msec retention time.

mental to the overall system performance. Using a refresh window, tREFW of 32msec, Figure 13 shows the performance improvements acheived by our co-design.[12] As in other performance graphs, all the results are normalized to all-bank refresh baseline. Our co-design refresh improves the performance in such high temperature environments on an average by 34.1% over all-bank refresh and 6.7% over per-bank refresh for 32Gb chips. In 24Gb chips, our co-design improves the performance on an average by 23.4% and 6.3% over the all-bank and per-bank refresh, respectively, while in 16Gb chips, the average performance improvements are 16.4% and 3.9%, respectively, over all-bank and per-bank refresh.

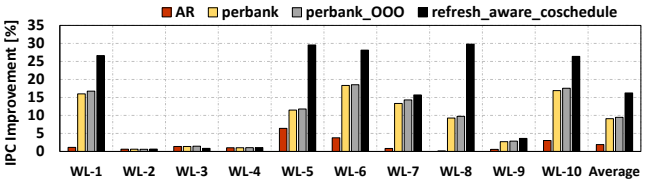



Figure 14: Comparison results for 32Gb chips (normalized to all-bank refresh).

### 6.5 Comparison with Previous Proposals

Figure 14 shows how our co-design fares over some of the previously proposed hardware-only solutions. Since our mechanism is based on the per-bank refresh, we compare it with the out-of-order (OOO) per-bank refresh proposed by Chang et al [15]. Apart from doing a OOO per-bank refresh, they further propose parallelizing accesses going to the refreshed bank by assuming sub-array support. Since our mechanism does not assume these additional support (modifications) to a DRAM bank, we compare our co-design only with OOO per-bank refresh. In OOO per-bank refresh, while deciding which bank to be refreshed, they look at the transaction queue and decide the target bank as the one with the lowest number of outstanding requests. As can be observed in Figure 14, just performing an OOO per-bank refresh does not improve the performance considerably. In our experiments, we observed that this is primarily a timing issue. Even though there are no requests queued to the target bank when deciding which bank to be refreshed, as the refresh operation lasts for several hundred $nanoseconds$ ($tRFC_{pb}$), we observed the outstanding requests to the bank being refreshed increased from the point the decision is taken. This is primarily because the data of each task are spread across all the banks. As a result, the average performance improvement brought in by the OOO per-bank refresh is marginal compared to the per-bank refresh but is significant around 9.5% compared to the all-bank refresh for 32Gb chips. Our co-design performs significantly better compared to the OOO per-bank refresh improving the performance on an average by 6.1%. In the interest of space, we could not present the results for other chip densities, but they seem to follow the same trend.

Figure 14 also presents results compared to another previously proposed hardware-only solution, Adaptive Refresh (AR) [27]. As discussed in Section 7, AR switches between the DDR4-1x and DDR4-4x modes dynamically by monitoring the channel bandwidth utilization. AR is an optimization proposed on top of DDR4 all-bank refresh. As can be noted from Figure 14, AR improves the performance by 1.9% on average compared to all-bank refresh but still does not perform as well as the per-bank refresh. Similar observations have also been noted by Chang et al [15]. Compared to AR, our co-design improves the performance on an average by 14.6%.

### 6.6 Sensitivity Results

Figure 15 shows the sensitivity results of our co-design with varying number of cores and varying number of tasks per core. In the interest of space, we present the average improvements over all the workloads (and not each workload result) across different chip densities. As can be observed, our co-design consistently fares better than both all-bank and per-bank refresh across various consolidation ratios. Confining to use just 1 DRAM channel, in a dual-core system as the consolidation ratio decreases from 1:4 to 1:2, a task's

[12] Note that we used a 2msec time-slice for 32msec retention time in our experiments, which still falls in typical OS time-slice duration of 1-5msec [17].

data can only be allocated on 4 banks per rank, as opposed to 6 banks per rank in the 1:4 consolidation ratio. This is because, assigning more than 6 banks per rank for each task allows only one task to be available to be scheduled on a dual-core system remaining the other core idle, thereby resulting in the under-utilization of cores. Hence, on a dual-core system with the 1:2 consolidation ratio, memory is partitioned such that each task allocates data on 4 banks in a rank, making a total of 8 banks. Consequently, the available BLP is reduced compared to the 1:4 consolidation ratio scenario. However, the 1:2 consolidation ratio still fares better compared to the all-bank and per-bank refresh. Our co-design improves the performance by 14.2%,11.2%,8.9% over all-bank refresh in 32Gb, 24Gb and 16Gb chips, respectively. However, by scaling up the number of DIMMs per channel from 1 to 2, it is possible for each task to allocate data on more number of banks, resulting in improved BLP and reduced contention and ultimately higher performance benefits. As can be observed, our co-design also gives good performance improvements as we increase the number of cores and the corresponding number of tasks per bank over the all-bank and per-bank refresh scenarios as well.

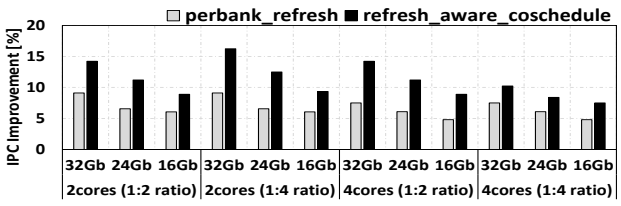



Figure 15: Sensitivity results (normalized to all-bank refresh).

## 7. Related Work

Many recent papers have proposed hardware and software solutions to mitigate the DRAM refresh overheads. These solutions reduce the DRAM refresh overheads by either skipping unnecessary refreshes or allowing DRAM accesses to proceed in parallel with refreshes.

Multiple previous papers have exploited the fact that most of the DRAM cells have high retention times and do not need to be refreshed as often as the small number of weak cells with low retention times. Liu et al. proposed RAIDR [22], a retention-aware refresh technique which enables 75% of the refresh activity to be eliminated. Bhati et al. proposed modifications to the existing auto-refresh functionality in order to enable such refresh skipping [12]. Other software techniques such as Flikker [25] and RAPID [35] take retention times into account while allocating the critical program data and the OS pages, respectively. All these techniques rely on building a retention time profile for the entire DRAM, which requires extensive testing and could incur a substantial runtime overhead. Furthermore, recent work has shown that DRAM cell retention times exhibit large variations with both time and temperature [22] [30], making "retention time profiling" unreliable and difficult to implement.

Other prior work attempts to reduce refresh overheads by scheduling refresh commands in periods of low DRAM activity. Elastic Refresh proposed by Stuecheli et al. [32] postpones up to 8 refresh commands in order to find idle periods when these refresh commands could be scheduled. Similarly, Co-ordinated Refresh [11] attempts to schedule refreshes when DRAM is in the self-refresh mode. While these techniques could save refresh power for low memory intensity workloads, they may not work well for memory-intensive workloads where periods of low memory activity are scarce.

Another approach adopted by prior techniques to reduce refresh overheads is to use finer granularity refresh modes. We already presented how our co-design fares quantitatively relative to Adaptive Refresh [27] and DDR4 2x and 4x modes. Adaptive Refresh (AR) chooses one of the three available refresh modes (1x, 2x and 4x) in DDR4, based on monitoring the runtime DRAM bandwidth utilization. Another technique, refresh pausing [28], aborts refresh commands upon receiving DRAM requests and then resumes them subsequently. However, supporting this functionality requires the memory controller to have intricate vendor specific knowledge of the refresh implementation within the DRAM device.

Finally, some recent papers have proposed techniques to overlap memory accesses with refreshes. The per-bank feature in LPDDR3 allows one DRAM bank to be refreshed while other banks can be accessed in parallel. Chang et. al [15] and Zhang et al. [38] have proposed techniques to enable bank-granularity and sub-array granularity refresh commands in order to allow more parallelism between refreshes and requests. These techniques require changes to the DRAM subarray architecture. In comparison, our technique enables parallelism of refreshes and requests by careful hardware-software co-design, without requiring any DRAM modifications. If such DRAM modifications are incorporated into the future DRAMs, we expect our co-design to yield even better performance improvements. This is because, exposing the sub-array structures to the OS can enable soft-partitioning at a sub-array granularity, resulting in reduced contention and increased BLP.

## 8. Conclusion

In this work, we proposed a hardware-software co-design to mitigate the DRAM refresh overheads for high density DRAMs. In particular, we proposed a novel per-bank refresh schedule for DRAM banks which is further combined with our proposed soft memory-partitioning scheme to present ample opportunites for the OS to schedule tasks in a refresh-aware fashion. By exposing the hardware address-mapping and the per-bank refresh schedule to the OS, our co-design framework enables the OS scheduler to schedule a task such that none of the on-demand requests of a scheduled task are stalled by refreshes in a given time quantum. Hence, by being refresh-aware, our proposed co-design masks the en-

tire refresh overheads at the small cost of bank-level parallelism. Our proposed co-design improved the overall performance on an average by 16.2%/12.1%/9.03% over all-bank refresh and by 6.3%/5.4%/2.5% over per-bank refresh for 32Gb/24Gb/16Gb DRAM chips, respectively, without warranting any changes to the internal DRAM structures unlike the prior proposals.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants 1439021, 1439057, 1213052, 1409095, and 1629129. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Linux cgroups. http://goo.gl/tTiwSl.

[2] Linux debugfs. https://goo.gl/sdBhIh.

[3] Linux CFS Scheduler. https://goo.gl/hjVjJl, .

[4] Understanding the Linux Kernel. http://goo.gl/8P7gJR, .

[5] NAS. https://www.nas.nasa.gov/publications/npb.html.

[6] SPEC 2006. https://www.spec.org/cpu2006/.

[7] STREAM. https://www.cs.virginia.edu/stream/.

[8] JEDEC. DDR3 SDRAM Standard, 2012.

[9] JEDEC. DDR4 SDRAM Standard, 2012.

[10] JEDEC. Low Power Double Data Rate 3 (LPDDR3), 2012.

[11] I. Bhati, Z. Chishti, and B. Jacob. Coordinated refresh: Energy efficient techniques for DRAM refresh scheduling. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED, 2013.

[12] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob. Flexible auto-refresh: Enabling scalable and energy-efficient DRAM refresh reductions. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, 2015.

[13] 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*, 2011.

[14] J. D. Booth, J. B. Kotra, H. Zhao, M. Kandemir, and P. Raghavan. Phase detection with hidden markov models for dvfs on many-core processors. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, ICDCS, 2015.

[15] K. K. W. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *the 20th International Symposium on High Performance Computer Architecture*, HPCA, 2014.

[16] 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 the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA, 2012.

[17] V. V. Fedorov, A. L. N. Reddy, and P. V. Gratz. Shared last-level caches and the case for longer timeslices. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS, 2015.

[18] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *IEEE International Symposium on High-Performance Comp Architecture*, HPCA, 2012.

[19] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2011.

[20] O. Kislal, M. T. Kandemir, and J. B. Kotra. Cache-aware approximate computing for decision tree learning. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW, 2016.

[21] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das. Re-NUCA: A practical nuca architecture for reram based last-level caches. In *2016 IEEE International Parallel and Distributed Processing Symposium*, IPDPS, 2016.

[22] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. Raidr: Retention-aware intelligent DRAM refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA, 2012.

[23] J. Liu, J. B. Kotra, W. Ding, and M. Kandemir. Network footprint reduction through data access and computation placement in noc-based manycores. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC, 2015.

[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 the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2012.

[25] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[26] J. Lozi, B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys, 2016.

[27] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, 2013.

[28] P. Nair, C. C. Chou, and M. K. Qureshi. A case for refresh pausing in DRAM memory systems. In *IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA, 2013.

[29] M. Poremba and Y. Xie. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *IEEE Computer Society Annual Symposium on VLSI*, ISVLSI, 2012.

[30] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for DRAM systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2015.

[31] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA, 2000.

[32] J. Stuecheli, D. Kaseridis, H. C. Hunter, and L. K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *the 43rd Annual International Symposium on Microarchitecture*, MICRO, 2010.

[33] K. Swaminathan, J. B. Kotra, H. Liu, J. Sampson, M. Kandemir, and V. Narayanan. Thermal-aware application scheduling on device-heterogeneous embedded architectures. *2015 28th International Conference on VLSI Design*, 2015.

[34] X. Tang, M. Kandemir, P. Yedlapalli, and J. B. Kotra. Improving bank-level parallelism for irregular applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2016.

[35] R. K. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in DRAM (rapid): software methods for quasi-non-volatile DRAM. In *The Twelfth International Symposium on High-Performance Computer Architecture*, HPCA, 2006.

[36] P. Yedlapalli, J. B. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam. Meeting midway: Improving CMP performance with memory-side prefetching. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2013.

[37] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2014.

[38] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie. Cream: A concurrent-refresh-aware DRAM memory architecture. In *The 20th International Symposium on High Performance Computer Architecture*, HPCA, 2014.