Hiding DRAM Refresh Overhead in Real-Time Cyclic Executives

Xing Pan, Frank Mueller
North Carolina State University, Raleigh, NC 27695-8206, USA, mueller@cs.ncsu.edu

***Abstract*—Real-time systems with hard timing constrains require known upper bounds on each task's worst-case execution time (WCET) to determine if all deadlines can be met. One challenge in predictable execution is that Dynamic Random Access Memory (DRAM) cells must be refreshed periodically to maintain data validity, yet memory remains blocked during refresh, which results in overly pessimistic WCET bounds. This work contributes "Colored Refresh" to hide DRAM refresh overhead while preserving real-time schedulability for cyclic executives, which are widely used in highly critical systems. Colored Refresh partitions DRAM memory at rank granularity such that refreshes rotate round-robin from rank to rank. Real-time tasks are assigned different ranks via colored memory allocation. By cooperatively scheduling real-time tasks and refresh operations, memory requests no longer suffer from refresh interference. This reduces memory access latencies for tasks irrespective of DRAM density and size. Hence, Colored Refresh reduces a task's WCET and makes its execution more predictable.**

I. Introduction

DRAM is the de-facto standard memory technology of contemporary computers. Data is stored in DRAM cells that slowly leak their charge, i.e., they need to be refreshed to avoid loss of data. The DRAM controller periodically issues refresh commands, which are sent to DRAM devices. This mode is called auto-refresh and recharges all memory cells within the "retention time", typically 64ms for commodity DRAMs [1]. In this mode, a refresh command is issued per interval, $tREFI$, for a duration/completion by $tRFC$. During a refresh, a memory space (i.e., DRAM rank) becomes unavailable to memory requests (read or write) so that any such memory reference blocks until the refresh completes. Furthermore, a refresh closes all bank row buffers of this rank, even though spatial and temporal locality make future row buffer hits likely. As a result, memory accesses suffer from unpredictable bank row buffer misses around refreshes. These factors contribute not only to an increase in memory latency but also to significant latency fluctuations. In addition, as the density and size of DRAM grow, particularly due to the application of machine learning in embedded environments that require many Gigabytes of memory, more DRAM cells are required per DRAM chip, which must be refreshed within the same DRAM retention time, i.e., more rows need to be refreshed in one refresh cycle. This increases the length of a refresh operation and thus reduces memory throughput [2], [3], [4], [5].

Although the DRAM refresh impact can be reduced by proposed hardware solutions [6], [7], [8], [9], [10], such solutions take a long time before they become widely adopted. Hence, other work seeks to assess the viability of software solutions. RAIDR [2] lowers refresh overhead by exploiting inter-cell variation in retention time. RAPID [11] sorts pages by retention time and allocates long retention pages first. Smart Refresh [12] reduces unnecessary refreshes by maintaining a refresh-needed counter. Fine Granularity Refresh (FGR), proposed by JEDEC's DDR4 specification, reduces refresh delays by trading off refresh latency against frequency [13]. These approaches either heavily rely on specific data access patterns of workloads, or they have high implementation overhead. None hide refresh overhead to the extend that our work does. The refresh problem is even more significant for real-time systems because predictable memory access latencies are imperative to assess task schedulability [14]. Today's variable access latencies due to refreshes are counter-productive to tight bounds on a task's WCET, a problem that is only increasing with higher DRAM density/sizes. Due to the asynchronous nature of refreshes relative to task schedules and preemptions, none of the current analysis techniques tightly bound the effect of DRAM refreshes on WCET. Atanassov and Puschner [15] discuss the impact of DRAM refresh on the execution time of real-time tasks and calculate the maximum possible increase of execution time due to refreshes. However, this bound is too pessimistic (loose): If the WCET were augmented by the maximum possible refresh delay, many schedules would become theoretically infeasible, even though executions may make deadlines in practice. Also, as refresh overhead increases approximately linearly with growing DRAM density, it quickly becomes untenable to augment the WCET by ever increasing refresh delays for future high density DRAM. Bhat et al. make refreshes predictable and reduce the number of preemption due to refreshes by triggering them in software instead of hardware auto-refresh [16]. While they consider the cost of refresh operations, it cannot be hidden.

This work contributes "Colored Refresh" to hide DRAM refresh overhead entirely. Colored Refresh makes real-time systems more predictable, particularly for large DRAM sizes that are required for machine learning. It exploits colored memory allocation to partition the entire memory space such that each real-time task receives different ranks. More significantly, refreshes and competing memory accesses can be strategically co-scheduled so that memory reads/writes do not suffer from refresh interference. As a result, access latencies are reduced and memory throughput increases, which tends to result in schedulability of more real-time tasks. What is more, the overhead of Colored Refresh is small and remains stable irrespective of DRAM density/size. In contrast, auto-refreshed overhead keeps growing as DRAM density increases.

Contributions: (1) The impact of refresh delay *for real-time systems under varying DRAM densities/sizes* is shown to be hard to predict for auto-refresh and to increase with DRAM density to the point where deadlines will be missed.

(2) Colored Refresh is contributed, which refreshes DRAM using memory coloring. Refresh overhead is hidden since a memory rank is either being accessed or being refreshed, but not both. Thus, memory accesses no longer suffer from refresh interference, i.e., refreshes are hidden in a safe manner. (3) Experiments with Malardalen benchmarks confirm that both refresh delays are hidden and DRAM access latencies are reduced. Consequently, application execution time becomes more predictable and stable, even when DRAM density increases. (4) An experimental comparison with DDR4's FGR shows that Colored Refresh exhibits better performance and higher task predictability. (5) Our mechanism uses existing DRAM controller-initiated refresh capabilities and is otherwise implemented in software; no hardware change is required.

II. Background

DRAM Architecture: DRAM requests from the CPU are relayed by the memory controller, which acts as a mediator between the last-level cache (LLC) and DRAM devices. As a DRAM controller receives memory transactions from its memory controller, it translates memory requests into corresponding DRAM commands and schedules them while satisfying the timing constraints of DRAM banks and buses.

A DRAM bank array is organized into rows and columns of individual data cells (see Fig. 1). When a memory access request is resolved, the corresponding row that contains the data is selected and pulled from the bank array into the row buffer incurring a Row Precharge (close the old row in buffer) delay, tRP, and a Row Access Strobe (activate the new row) delay, $tRAS$. This is called a row buffer miss. Once loaded into the row buffer and opened, accesses of adjacent data in a row (spatial locality) incur just a Column Access Strobe penalty, $tCAS$ (row buffer hit) of much lower cost than $tRP + tRAS$ assuming an open page policy.

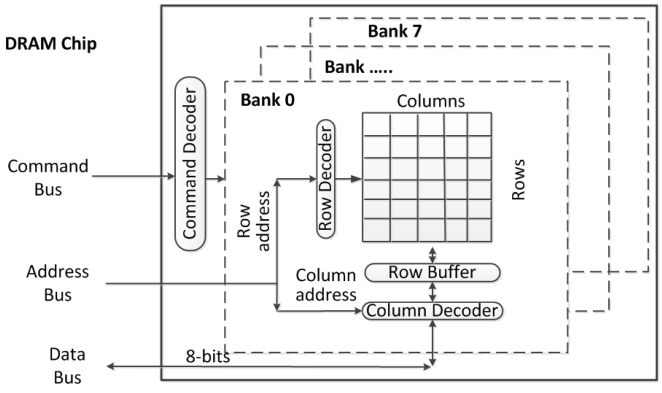

Fig. 1. DRAM Bank Architecture

DRAM Refresh: DRAM cells need to be recharged periodically to counter electric leakage and maintain data validity. The reference refresh interval of commodity DRAMs is 64ms under 85°C (185°F), the so-called retention time (R) of leaky cells, sometimes also called refresh window (tREFW) [1], [13], [17], [18]. To prevent data loss, all rows in a DRAM chip need to be refreshed within R (or tREFW). In order to reduce the refresh overhead, memory in each DRAM chip is divided into ranks, which are refreshed in parallel [19]. The DRAM controller can either schedule an automatic refresh to all ranks simultaneously (simultaneous refresh), or schedule automatic refresh commands to each rank independently (independent refresh). Whether simultaneous or independent, each memory refresh cycle affects a successive area of multiple cells in consecutive cycles. This area is called a "refresh bin" and contains multiple rows. The DDR3 specification [1] requires a DRAM controller to send 8192 automatic refresh commands to refresh the entire memory (one command per bin at a time). Here, the gap between two refresh commands, the so-called refresh interval (tREFI), is 7.8us (tREFW/8192). The so-called refresh completion time (tRFC) is the refresh duration per bin. Auto-refresh is triggered in the background by the DRAM controller while the CPU executes instructions.

Depending on the memory technology generation, the refresh granularity may vary. Nonetheless, memory ranks are unavailable during a refresh cycle (for tRFC units of time), i.e., memory accesses (read and write operations) to this region will stall the CPU during a refresh cycle. The resulting refresh overhead is tRFC/tREFI. As DRAM chip densities and sizes grow, each refresh bin contains more rows and the overall size becomes larger. But the more rows in a refresh bin, the longer the refresh delay and rank blocking times become. Refresh latency (equivalent to tRFC) is delimited by power constraints. Table I shows that the size of a refresh bin expands linearly with memory density, i.e., $tRFC$ increases rapidly as DRAM density grows and exceeds 1us at 32 Gb DRAM, even with conservative estimates of growth in density for scaling DRAM technology [2]. Ranks can be refreshed in parallel under auto-refresh, but this increases the amount of unavailable memory during a refresh. A fully parallel refresh blocks the entire memory space for tRFC. Such blocking not only decreases system performance, but may inflict deadline misses unless considered in a blocking term for schedulability analysis.

Tab. I. $tRFC$ per DRAM densities (data from [1], [13], [2])

Chip Density	total rows	number of rows per bin	tRFC
1Gb	128K	16	$110ns$
2Gb	256K	32	$160ns$
4Gb	512K	64	$260ns$
8Gb	1M	128	$350ns$
16Gb	2M	256	$550ns$
32Gb	4M	512	$\geq 1us$
64Gb	8M	1K	$\geq 2us$

As a side effect of DRAM refresh, a row buffer is first closed, i.e., its data is written back to the data array and any memory access is preempted. After the refresh completes, the original data is loaded back into the row buffer, and the deferred memory access can continue. In another words, the row which contains data needs to be closed and re-opened due to interference between refresh and an in-flight memory access. To close and re-open rows incurs an additional overhead of $tRP + tRAS$ since the refresh purges all buffers and often results in additional row buffer conflict misses, i.e., decreased memory throughput. Liu et al. [2] observe that the loss in DRAM throughput due to refreshes becomes untenable for large memories, reaching nearly 50% for 64 Gb DRAM.

Considering the cost of a refresh operation itself and the extra row close/re-open delay, DRAM refresh both decreases

memory performance and results in fluctuating response times of memory accesses. The asynchrony of refreshes plus task preemptions make it hard to accurately predict and bound DRAM refresh delay.

Refresh Mode and Scheduling Strategy: For commodity DDRx (e.g., DDR3 and DDR4), refresh operations are issued at rank granularity. A single refresh command for a given rank precharges all banks under this rank, which is called "All-Bank" refresh [19]. In contrast, recent LPDDRx DRAM [20] supports an enhanced "Per-Bank" mode to refresh cells at bank level while other banks in the same rank may be serviced. "Per-Bank" consumes more refresh time overall than "All-Bank" but achieves higher bank parallelism [21]. A refresh counter per rank maintains the address of the row to be refreshed and applies charges to the chip's row address lines. A timer then increments the refresh counter to step through the rows. Depending on when a refresh command to a bin (successive rows) is sent, two strategies exist: distributed and burst refresh.

Distributed Refresh: A single refresh operation is performed periodically (see upper Fig. 2). Once all rows are refreshed, the refresh cycle is repeated from the first row. With distributed refresh, the DRAM response time for memory accesses varies over a wide time range due to the spread of refreshes, and due to closing the row buffer time and again.

Burst refresh: A series of refresh cycles are performed, one after another ($tREFI = tRFC$), until all rows have been refreshed (see lower Fig. 2). After that, the memory is available for memory accesses until the next refresh. A burst refresh results in long periods of memory unavailability, which also affects task execution and results in longer memory latencies, yet such bursts occur less frequently.

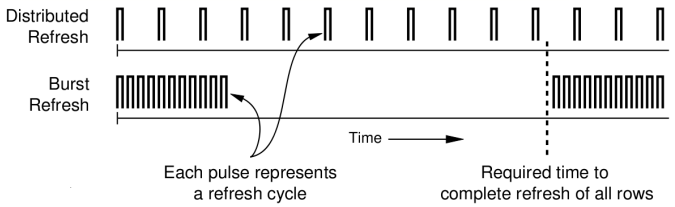

Fig. 2. DRAM Refresh Strategy

III. Design

The problem with the standard hardware-controlled auto-refresh is the interference between periodic refresh commands generated by the DRAM controller and memory access requests generated by the processor. The latter ones are blocked once one of the former are issued until the refresh completes. Since refreshes are asynchronous, memory latency becomes highly variable and unpredictable. The central idea of our approach is to hide DRAM refresh interference by memory partitioning (coloring) while making refresh synchronous to task execution. We partition memory so that each application is assigned a colored DRAM region. A *real-time schedule* can be adapted such that memory accesses will not be subject to interference by DRAM refreshes due to refresh-triggered task switching as described next.

Assumptions: Let a given real-time task set be schedulable under auto-refresh, i.e., the worst-case blocking of refresh is taken into account during schedulability analysis, including the row buffer misses due to closed rows. To this end, each worst-case execution time, e_i, is padded, by the refresh burst $tBST = m \times tRFC$, where $m = \lceil e_i/tREFI \rceil$. For a cyclic executive [22], [23], widely used in highly critical systems [24], e_i is iteratively calculated by determining the fixed point for

$$e_i(n) = e_i(0) + tRFC * \lceil e_i(n-1)/tREFI \rceil, \qquad (1)$$

where $e_i(0)$ is the original execution time before padding. We assume that tasks are independent, except for tasks that are sliced or copied, as discussed later. Henceforth, we will use e_i from Eq. 1 unless mentioned otherwise. In addition, for an application with input, we assume its maximum (worst-case) memory requirement can be triggered by a known input (in our experiments assumed to be the largest input). Furthermore, we also assume the timers in both task scheduling and DRAM refresh to be synchronized by the on-chip hardware clock, which is the case in practice [16].

A. Memory Space Partitioning

A memory node consists of one memory controller and multi-level resources, namely channel, rank, and bank. Banks are accessed in parallel to increase memory throughput. We obtained a copy of TintMalloc [25], an allocator that "colors" memory pages with controller and bank affinity suitable for NUMA architectures. With TintMalloc, programmers can select one (or more) colors to choose a memory controller and bank regions disjoint from those of other tasks. DRAM is further partitioned into channels and ranks above banks. The memory space of an application can be chosen such that it conforms to a specific color. For example, a real-time task can be assigned a private memory space based on rank granularity. When this task runs, it can only access the memory rank allocated to it. No other memory rank will ever be touched by it. With proper application design, this overhead of colored allocations impacts only the initialization phase and remains constant for a stable working set size, i.e., after their initialization, real-time tasks experience highly predictable latencies for subsequent memory requests.

B. Notation

Given that the memory space of a DRAM chip can be partitioned into multiple "colors" based on the DRAM architecture, such as node, channel, rank, and bank, our objective is to design a novel "Colored Refresh" policy, which systematically schedules DRAM refreshes based on DRAM rank coloring, i.e., at the refresh granularity level. By cooperatively scheduling task execution and DRAM refresh, Colored Refresh can hide the refresh overhead for real-time tasks and guarantee system schedulability. Most DRAM controllers allow the retention interval, R, to be configured [16] and can control which rank should be sent a refresh command. With Colored Refresh, the memory of a DRAM chip is refreshed by iterating over the ranks such that all tasks are colored by different DRAM ranks via TintMalloc. While we consider the most common DDR technology with rank refreshes in the following, our techniques are equally applicable to LPDDR and RLDRAM technology [26], where refreshes can be controlled at bank granularity. This simply requires coloring at bank level, which TintMalloc also supports.

Let us denote the set of periodic real-time tasks as $\mathcal{T} = T_1...T_n$, where each task, T_i, is characterized by (ϕ_i, p_i, e_i, D_i), or (p_i, e_i, D_i) if $\phi_i = 0$, or (p_i, e_i) if $p_i = D_i$ for a phase ϕ_i, a period p_i, (worst-case) execution time e_i, relative deadline D_i per task, and a hyperperiod H of $\mathcal{T}$ [27]. Also, let
R be the DRAM retention time,
L be the least common multiple of H and R, and
K be the number of DRAM ranks, and let k_i denote rank i. Let us further consider a cyclic executive schedule for $\mathcal{T}$, where
f is the frame size and f_i denotes the i^{th} frame (all of same size),
F is the number of frames in H. We next describe different refresh options.

C. Per-Rank Distributed, Harmonic Refresh

Let us first consider a schedule where the hyperperiod, H, is harmonic to the DRAM retention time, R. With Colored Refresh, the entire DRAM space is equally partitioned into "colors", such that each color contains one or more DRAM ranks. Refreshes are also equally divided over the number of frames following a one-to-one correspondence with colors, where c_i denotes color i of C total colors.

Example 1: Consider three real-time tasks, A=(16,4), B=(32,12), C=(64,16). For this task set, H=64ms (harmonic to R). Hence, L=64ms. Let $f = 8$ (explained later). This requires that tasks be arranged into frames of the cyclic executive's table according their periods and deadlines, and memory is colored according to Colored Refresh.

Fig. 3 depicts a feasible cyclic schedule for the above example. Above, *c(serial)* indicates the rank being refreshed. Since the execution time of B and C exceed f, their jobs need to be split into slices [22], [23] arranged by the network flow algorithm [28]. As shown in Fig. 4, a flow graph consists (left-to-right) of source, a layer of job nodes, a second layer of frame nodes, and a sink. Edges from the source are weighted by e_i/e_i, edges from jobs to frames are weighted by the fractional execution of a job placed in a frame, and edges from frames to sink are weighted by the aggregate utilized frame times over total frame length f. In the example, jobs 1 and 2 of task B are split over frames $\{1, 2\}$ and $\{5, 6\}$, respectively, while the job of C is split over $\{3, 4, 7\}$ by the network flow algorithm. Since the in-flow of 7 equals the out-flow, the schedule is valid. An invalid schedule would have an out-flow less than in-flow due to fractional e_i of a job that cannot be placed in a frame subject to release and deadline constraints relative to available frame times (which is further constrained by rules in Sec. III-G discussed later). A constructive algorithm may try to create edges from jobs to frames in a deadline-monotonic manner until is succeeds to place all execution or fails to do so. Task splitting can commonly be accomplished by refactoring a loop into consecutive subranges of loops per subtask that add up to the original loop length.

The schedule of Fig. 3 has a memory coloring solution for each task given in Table II. The entire DRAM space is refreshed by iterating over all 8 ranks, and the refresh duration of each rank is 8ms (R/K). Within one rank's refresh duration, all the refresh operations are scheduled as "burst refresh" (see

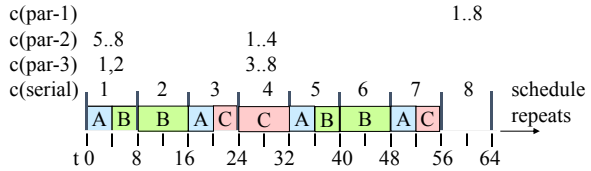

Fig. 3. Harmonic Schedule for H, R

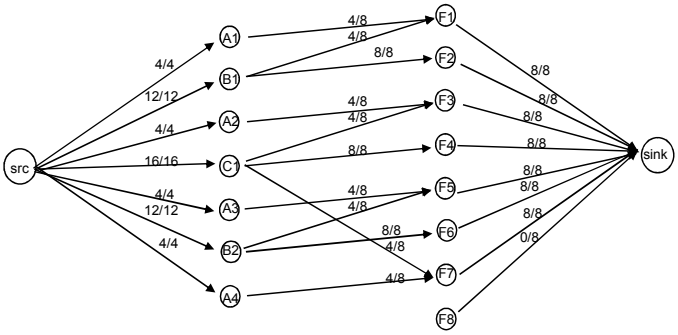

Fig. 4. Flow Graph for Task Splitting

Section II). Since f=8 and R=64 ms, we get F=8. This means all tasks are placed in 8 frames, and the entire memory space is grouped into 8 colors, where each color contains one rank. Throughout the schedule, one rank is under refresh during its corresponding frame while all other ranks are refresh-free during this frame.

Tab. II. Memory Assignment per Task

T	Occupied frames	available memory ranks (k_i)
A	f_1, f_3, f_5,f_7	c_2,c_4,c_6,c_8
B	f_1, f_2, f_5, f_6	c_3,c_4,c_7,c_8
C	f_3, f_4, f_7	c_1,c_2,c_5,c_6,c_8

D. Rank-Parallel, Harmonic Refresh

The DDR standard specifies that different ranks may be refreshed in parallel. Consider Fig. 3 again. Instead of refreshing one color (rank) per frame, we could refresh multiple colors at once (in a frame) in parallel. In the example, all ranks could simply be refreshed in frame 8 since the processor remains idle then, i.e., all colors (ranks) would become available to all tasks. As indicated by *c(par-1)* in the figure, this requires either a phase of $\phi = 56$ for the retention interval, R, relative to H or a software-triggered refresh burst at $t = 56$ synchronously triggered by the cyclic executive.

Let us consider schedules with tasks in every frame. We could then choose a subset of frames occupied by disjoint tasks so that refreshes are triggered only in these frame. In Fig. 3, we could choose frames 1 and 4 to refresh ranks $k_5...k_8$ and $k_1...k_4$, respectively. A feasible color assignment would be $((A, k_2), (B, k_3), (C, k_5))$ as no task execution coincides with a refresh in the same frame. As indicated by *c(par-2)* in the figure, two hardware refreshes with a period of H and phases $\phi_1 = 0, \phi_2 = 24$ would be required for frames 1 and 4, respectively.

Notice that there is no solution for *parallel*, zero-phased, single harmonic refreshes: If frame 1 was chosen, the colors for tasks A and B could not be refreshed, and ditto for frames 1 and 5. For frames 1, 3, 5, and 7, A's colors could not be refreshed, which only leaves the serial solution of one refresh per frame discussed before.

A phase of $\phi = 8$ works for frames 2, 4, 6, 8. And $\phi = 24$ works for frames 4, 8 but fails for other dual frame harmonic selections. However, if frame 8 was included, a refresh during frame 8 with $\phi = 56$ suffices to refresh all ranks in parallel, which also simplifies the method.

E. Copy Tasks

Should hardware capabilities or system design constraints not allow phasing, then a zero-phased rank-parallel refresh can be devised for the given example by introducing the novel concept of a "copy task".

Definition: Given a task T, a **copy task**, T', has identical control flow but, for both code and data, is referencing memory allocated from a different color. Local variables are segregated by executing on different stacks for T and T'. Heap and global variables of the original are combined in duplicate heap and global sections by transforming the source program accordingly. Code is duplicated as well for these colors. When the next job of a task has a different instance than the current one, *the local/global/heap variables of the current are copied to the next job's memory space* if jobs of this task are data dependent. This overhead is analyzed in Sec. III-L.

In Fig. 3, let A' and B' be the instances executed during the second half of H (t=32..64). As indicated by *c(par-3)* in Fig. 3, the colors (3..8) of A, B, C could be refreshed in frame 5 while colors (1,2) of A', B' can be refreshed in frame 1. This establishes a rank-parallel zero-phased refresh solution.

F. Rank-Parallel, Non-Harmonic Refresh

If H and R are non-harmonic, one task may occupy all frames in a refresh period L so that no solution for refreshing exists. This can be addressed in two ways, either by copy tasks or by adjusting the retention interval R.

Example 2: Consider a task set A=(20,8), B=(40,16). We still assume R=64 ms and K=8. In this case, H=40 ms and L=320 ms. Let f=8 (explained later). Fig 5 shows a feasible cyclic schedule. All frames are occupied by both A and B (Tab. III), i.e., no frames remain available.

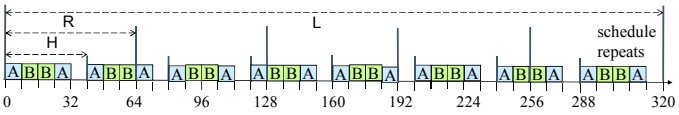

Fig. 5. Non-harmonic Schedule

Tab. III. Memory Assignment per Task

T	Occupied frames	available memory ranks (k_i)
A	f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8	none
B	f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8	none

We can hide refresh overhead via copy tasks, A' and B', for both A and B. When a job of this task is released, an instance is chosen for execution that differs from the color under refresh. With refresh phasing, let color 1 be refreshed during A/B in frame 1 while color 2 is refreshed during A'/B' in frame 2. For all other frames, either instance may execute (no refresh). Hence, all DRAM refresh overhead is hidden from task execution and thus need not be considered for schedulability analysis. Without refresh phasing, frames 1 and 5 may be chosen for refresh of color 1 and 2 plus A/B or A'/B' execution, if indicated by the schedule, complementary to the refresh color.

An alternative solution is to make the retention interval, R, harmonic to H. Consider $R = H = 40$ for example 2. One solution would be to refresh the colors of A and B in frames 1 and 2, respectively, with all other frames being refresh free. Another solution would be to perform all refreshes in frame 5, which is idle. In practice, system designers may only have the ability to alter the default retention interval of 64ms if they design the entire system. In many cases, ECUs of different manufacturers are integrated that have to adhere to common parameters, i.e., a fixed retention interval of 64ms, which is the factory default for any system today. To this end, we have developed a policy and algorithms to select frame and colors for refresh as well as assign colors to tasks in a systematic manner assuming a fixed retention interval without phasing.

G. Generalized Colored Refresh

We construct a cyclic executive schedule for $\mathcal{T}$, where
f is the frame size and f_i denotes the i^{th} frame (all of same size),
F is the number of frames in one R cycle, i.e., $F = R/f$. Thus, F is equal to the total number of colors, i.e., $F = C$.
After a refresh duration of tRFC, any ranks in the i^{th} set c_i are refreshed within the i^{th}frame f_i. By selecting an appropriate frame size (f), Colored Refresh can strategically co-schedule tasks and DRAM refresh to hide refresh overhead and guarantee schedulability. The basic idea is to alternate between a burst refresh for one set of ranks and another set of ranks, where ranks of each set are refreshed in parallel. We select an appropriate f using 5 rules:
(1) $f \leq min_{1 \leq i \leq n}(p_i/2)$, i.e., burst refreshes can alternate once all tasks are assigned disjoint memory colors under Colored Refresh.
(2) $\lfloor R/f \rfloor - R/f = 0$, i.e., f should divide R since DRAM refresh should be synchronous to the cyclic executive.
(3) $2f - gcd(p_i, f) \leq D_i$ for all i so that a task can execute in either of two adjacent frames.
(4) $\lfloor f/(R/K) \rfloor - f/(R/K) = 0$, i.e., R/K should divide f so that refresh becomes synchronous to other tasks and frames can be colored.
(5) Frames must be sufficiently long so that every job can execute non-preemptively within them, i.e., $f \geq max_{1 \leq i \leq n}(e_i)$.

Consider example 1 in Fig. 3 again. With a given A=(16,4), B=(32,12), C=(64,16), and R=64 ms, let K=8, which means there are 8 ranks that should be refreshed every 64ms. H=64ms, which is harmonic with R. Hence, L=64ms.
By (1), $f \leq 8$.
By (2), f can be 2, 4, 8, 16 or 32.
(3) and (4) hold for f=8.

By (5), B and C are split into multiple job slices since their execution time is longer than f=8. Notice that besides the schedule in Fig. 3, other valid orders exist due to (3), e.g, the first two frames could be B and B, A, respectively.

From the allocation rule for f_i and Fig. 3, we can identify the available colors per task (see Table II), which are not being refreshed when the task runs. Colored Refresh assigns memory from these "available colors" though TintMalloc. Furthermore, memory requirements of all tasks should be satisfied after coloring. In this example, assuming a requirement of 1 rank per task, a feasible color assignment is $((A, k_2), (B, k_3), (C, k_5))$.

Consider the 5 rules (2 of them derived from [22], [23]) in general. We select an appropriate frame size (f) for a real-time task set. F frames are scheduled iteratively every R cycles, and all colors are refreshed within each R interval. Jobs are scheduled and placed in frames via the network flow algorithm [28]. Colored Refresh scans all frames within L and enumerates on a per-task basis the "available colors", which are the frames of a certain color not occupied by this task. Colored Refresh selects one of the available colors at a time to allocate memory from ranks conformant to the coloring policy. As a result, any real-time task only accesses refresh-free memory ranks within its execution. Thus, memory accesses are cooperatively scheduled with refresh operations, while the refresh overhead remains hidden from task execution.

H. Modifications to the Task Set

Some conditions may require a task set $\mathcal{T}$ to be modified:

• Condition (5) may not be satisfied by tasks T_i with long executions, where $e_i > f$. A well-known technique to address this is to split such tasks into job slices [22], [23], e.g., $k = 1..m$ with $e_{ik} = f$ for $k < m$ and $e_{im} = e_i - (m-1) \times f$, and then distribute them over the set of frames using the network flow algorithm [28] — in our case in conjunction with Colored Refresh to select colors for each frame.

Theorem 1: A task set schedulable as a cyclic executive can always be scheduled with a frame size in $(0, D_{min}/2]$, e.g., any positive number less than $D_{min}/2$ is an appropriate frame size for cyclic executive scheduling, where $D_{min} = min(D_1, D_2...D_n)$ is the minimum deadline among all tasks.

Proof: The selection of the frame size in a cyclic executive is determined by rules (3)+(5) [22], [23]. If $f \leq D_{min}/2$, then

$$2f - gcd(p_i, f) \leq D_{min} - gcd(p_i, f) < D_{min} \leq D_i,$$

i.e., for any frame size less than $D_{min}/2$, rule (3) holds.

Furthermore, rule (5) can be met by task splitting [22], [23]. Hence, any frame size within $(0, D_{min}/2]$ is appropriate for a task set under cyclic executive scheduling. □

• Conditions (1)+(4) may not be simultaneously satisfied as a task may occupy all frames and its available color set is empty, e.g., tasks with short periods $p_i < 2f$ or periods that do not divide the minimum frame size $f = R/K$. In this case, Colored Refresh creates two instances of a task, the original one and a so-called "copy tasks". When a job of this task is released, Colored Refresh selects the instance to execute with a color that is currently not being refreshed. The job instance can be statically determined for cyclic executives over L, which enumerates all job instances. When the next job is a different instance from current one, *the local/global/heap variables of the current are copied to the next job's memory space* if jobs of this task are data dependent. An arbitrary number of "copy tasks" can be created in order to satisfy different numbers of colors. The number of colors is determined by the amount of DRAM ranks in the system, and Colored Refresh usually selects an even number of colors. But odd numbers of colors are also supported by Colored Refresh. When the number of colors is odd, more than two instances are created by task copying. For example, three instances are needed (the original one and two "copy tasks") when there are 3 colors in system. Multiple "copy tasks" can be assigned the same memory color to save memory space.

Consider example 2 again, where H and R are non-harmonic with task set A=(20,8), B=(40,16), with R=64 ms and K=8. We determined H=40 ms and L=320 ms. Here, (1)-(5) hold for f=8. Fig 5 showed a feasible cyclic schedule for this frame size. Copy tasks, A' and B', were required as discussed before.

• Conditions (1)-(4) may not be simultaneously satisfied by a task set that requires a small f, e.g., an f derived from rule (3) that is less than the minimum frame size in rule (4). A novel technique to address this problem is to fuse multiple frames repeatedly into a "virtual frame", f', until this virtual frame fulfills the requirement of minimum frame size according to rule (4), i.e., until $f' \geq R/K$. By fusing adjacent frames, (see Algorithm 1), a task set with a small f is transformed into one with a virtual frame f' that is still scheduled by rules (1)+(3) but with memory colors derived from Colored Refresh based on the virtual frame size to satisfy rules (2)+(4).

Example 3: Consider R=64ms, K=8ms as before and a task set A=(3,2), B=(6,1) with f=3ms due to rules (3)+(5), the original rules for cyclic executives [22], [23]. Rule (2) is violated and can only be satisfied in conjunction with rules (1)+(3) for $f = 1ms$, which requires Task A to be split into A1 and A2 in order to satisfy rule (5). Now, rule (4) is violated, i.e., frames cannot be colored since the smallest coloring granularity is $f = R/K = 8ms$. We now *fuse* adjacent frames together until $f' = 8f = 8ms$ satisfies rule (4). While scheduling occurs at $f = 1ms$, refresh and coloring is coordinated at $f' = 8ms$. To this end, "copy tasks" A1', A2', and B' are created, which will be scheduled in a frame where the colors of A1, A2, and B are being refreshed. Colors are chosen by Colored Refresh from the available color not under fresh. Under f', L=192 and it suffices to execute A1' at t=96 and t=160 assuming that A1,A2 have one color while A1',B have another color. Thus, A2' and B' are not required in the example.

Algorithm 1 illustrates how frame fusion is accomplished, which is based on rules (2)+(4): $f' = i * f = \frac{R}{K} * j$ and $m * f' = R$, where i, j, m are integers and $m * j = K$. Since $\frac{R}{K}$ represents the memory coloring granularity, we require f to divide R/K (rule (4)), i.e., $\frac{j}{i}$ is an integer.

• A *huge task* is a task that allocates memory larger than the available refresh-free memory within DRAM, which is constrained by the number of memory ranks currently not under refresh while the task executes. For a system with K ranks, a huge task would have to exceed $K-1$ ranks in size, which means that a single task essentially monopolizes almost

Algorithm 1 Frame Fusion

```
for f in (0, D_min/2] do
    f' = f, i = 1
    while f' ≤ R do
        f' += f, i++
        if (f' mod R/K) == 0 then
            j = f'/(R/K)
            if (i mod j) == 0 then
                return f and f'
            else
                continue
            end if
        end if
    end while
end for
```

all memory for a large K, or that memory is relatively limited to begin with, say K=2, and more than half the memory is used by a single task (for K=2). If we account for such rare cases of skewed memory distribution, the refresh overhead of a huge task cannot be completely hidden but it can be constrained to a single burst of $tBST$. Only one such huge task may exist (due to memory capacity limits) per task set, i.e., this is a singular overhead term for this task's e_i. We do not consider huge tasks in the following as they are rare corner cases, which can be handled by assessing for their refresh overhead in the traditional manner as blocking.

I. Schedulability of Colored Refresh

For the following theorem, let us assume that task copy/split overhead is zero.

Theorem 2: Any task set (without a huge task) that is schedulable as a cyclic executive under auto-refresh by considering refresh delays in its execution time as a blocking term is also schedulable under Colored Refresh.

Proof: If a task set is schedulable as a cyclic executive, there exists an appropriate frame size to schedule it. This appropriate frame size should satisfy rules (3) and (5), see [22], [23]. With Colored Refresh, the frame size f is further constrained by (1), (2), and (4). What needs to be shown is that these rules do not further constrain the choice of f.

Let $\hat{f}$ be an appropriate frame size for the cyclic executive of the uncolored task set that guarantees schedulability without coloring by satisfying rules (3) and (5). We then need to find a new frame size, f, that satisfies rules (3) and (5) as well as the new rules (1), (2) and (4). We need to show that a task set is schedulable with f if it is schedulable with $\hat{f}$.

Case 1: There exists an f that satisfies rules (1)-(5), i.e. $f = \hat{f}$. Then this f is a solution.

Case 2: Any of the $\hat{f}$ candidates that satisfy rules (3) and (5) do not satisfy one or more of the rules (1),(2),(4). Let $f_{min} = R/K$, which is the minimum frame size established by rules (2) and (4), i.e., both rules hold.
Let a task have a period $p_i < 2 * (R/K)$, which violates rule (1), and denote this task as T_c with a phase ϕ_c. For any such T_c, by creating "copy tasks", multiply their period by x and create x instances (the original one and $x-1$ "copy tasks"). $T_c^{'}$ with its enlarged period has an identical deadline and execution time but a phase of $\phi_c + p_c * i$, where $i \in [0, x-1]$. Furthermore, let Colored Refresh assign memory colors for these (x) tasks, and select an appropriate x by copying tasks until (1) holds. Here, rule (1) is satisfied by creating "copy tasks", while rule (5) is satisfied by "task splitting" (as discussed before). As a result, what needs to be shown is that f exists and satisfies rules (2)+(3)+(4).

Case 2a: If $\hat{f} \geq f_{min}$, we can always find an appropriate f which is in $[f_{min}, \hat{f}]$ to schedule tasks, i.e., since the frame size is equal or smaller than $\hat{f}$, there are at least as many frames between the release of a task and its deadline for f as there were for $\hat{f}$, so (2), (3) and (4) hold for all tasks i.

Case 2bi: if $\hat{f} < f_{min} \wedge f_{min} \leq D_{min}$, where $D_{min} = min(D_1, D_2 ... D_n)$, we can always find an appropriate f which is in $[f_{min}, D_{min}]$ to schedule tasks by creating "copy tasks". The period of the resulting set of "copy tasks" can be increased to be divisible by f, i.e., $f = gcd(p_i, f)$. As a result, there exists an f in $[f_{min}, D_{min}]$ to satisfy rules (2)+(3)+(4), and by choosing such an f, all rules are satisfied.

Case 2bii: if $\hat{f} < f_{min} \wedge f_{min} > D_{min}$, we can find an f such that $\lceil f_{min}/f \rceil - f_{min}/f = 0$, i.e., f_{min} is integer divisible by f. Furthermore, f is chosen such that it also satisfies rule (3), i.e., $f \leq D_i$. Then f is used for scheduling frames. We further construct a "virtual frame", f', by fusing frames of size f so that f' is a multiple of f_{min} and such that rules (2)+(4) hold for f', where f' is used for coloring (see Example 3). Furthermore, each f is associated with a memory color according to which virtual frame f' it belongs to. Hence, rules (2)+(3)+(4) hold by utilizing f' and f together.

Thus, for a task set in Case 2 with Colored Refresh, an f exists that satisfies all 5 rules and provides a solution to guarantee schedulability. Hence, the rules under Colored Refresh do not constrain the choice of f. □

J. Utilization

With Colored Refresh, system utilization is enhanced compared to auto-refresh. With auto-refresh, the blocking time of refresh operations has to be considered when deriving the worst case execution time (WCET) using the *original* e_i (i.e., not the modified one from Eq. 1). Assuming a task set can be scheduled under auto-refresh within a time span t, where

$$t \geq \sum_{i=1}^{n} \lceil \frac{t}{p_i} \rceil * e_i + b_t,$$

and b_t represents the blocking term due to DRAM refresh within t calculated as

$$b_t = \frac{t}{tREFI} * tRFC.$$

As a result, the upper bound on system utilization under auto-refresh is

$$U = \frac{t - b_t}{t} = \frac{t - \frac{t}{tREFI} * tRFC}{t} = 1 - \frac{tRFC}{tREFI}.$$

With Colored Refresh, the blocking term can be removed since the DRAM refresh overhead is hidden for real-time tasks, i.e., $t_{RFC} = 0$ and the 2nd term in Eq. 1 becomes zero. The highest utilization under Colored Refresh is 1.

K. Discussion

So far, we have considered a periodic real-time task set on a single processor. Colored Refresh can be implemented on multicore platforms where multiple task partitions are running simultaneously. Colored Refresh simply schedules the execution of multiple task partitions at a time when their

allocated colors are *not* being refreshed. Variables shared between tasks would require additional constraints to ensure they are accessed when not being refreshed. Furthermore, sporadic tasks and non-real-time tasks can also be placed into frames and scheduled with Colored Refresh with an admission test based on e_i from Eq. 1 similar to past work [22], [27]. Actually, within each frame, both distributed refresh and burst refresh could be implemented. Colored Refresh implements burst refresh at the start of each frame. After the burst refresh finishes, the time left in this frame becomes refresh-free for memory accesses, which provides more flexibility for other tasks to be scheduled [16].

The refresh task under Colored Refresh is still modeled as the highest priority task, but it can be co-scheduled with another real-time task of a different color on the same processor. This allows us to model resource constraints such that a real-time task of the same color as an active refresh task cannot preempt the refresh task due to their priority assignments.

Furthermore, we consider *long tasks* with (i) a period equal to or larger than DRAM retention time and (ii) slack between its period and execution time that is less than the minimum frame size.

$$p_i - e_i < \frac{R}{K}, \text{ and } p_i \geq R$$

The DRAM refresh overhead of a *long task* cannot be hidden completely through Colored Refresh, neither by splitting into multiple job slices nor by task copying. As a result, the utilization of a *long task* under Colored Refresh is

$$U \leq 1 - \frac{tRFC}{tREFI} * \frac{rankReq}{K},$$

where $rankReq = \lceil \frac{memAllocSize}{rankSize} \rceil$.
"rankReq" represents the lower bound of memory ranks that should be assigned to a *long task*, which is determined by how much memory it allocates and the size of one rank.
The system utilization of a *long task* is still enhanced by Colored Refresh compared to auto-refresh, although refresh overhead is only partly hidden for a *long task*. Furthermore, since we use a burst refresh in each frame, the remaining refresh overhead is predictable as a single task is colored to specific ranks. Notice that task splitting can be implemented by loop fission after the i^{th} iteration of an outer loop. Should a resource be held, it would be split across adjacent tasks executing back-to-back, where the former locks and the latter unlocks a mutex assuming POSIX-style locking.

L. Overhead of Colored Refresh

In order to hide DRAM refresh overhead, Colored Refresh imposes task copying and splitting costs. While not free, split overhead is predictable since split points are known statically. Furthermore, the cost of task copying is extremely small, as quantified by $\frac{globalMem}{bandwidth}$. Here, $globalMem$ denotes the cumulative size of variables that need to be copied from a current to the next job's memory space, and $bandwidth$ represents the memory bandwidth. We can determine if a task benefits from task copying by comparing the copy cost to the overhead it would suffer under refresh-incurred blocking instead:
$\frac{globalMem}{bandwidth} \leq \frac{tRFC}{tRFI} * e$, where e is the task's execution time and $\frac{tRFC}{tRFI} * e$ represents the overhead due to refresh (upper bound) that would have to be considered in a blocking term during schedulability analysis.

Example: The cost of one refresh operation is $tRFC$=350ns, and the length of a refresh interval is $tRFI$=7.8us for 8Gb DRAM density, which is common in commercial off-the-shelf embedded systems, GPU-enhanced embedded platforms for autonomous driving, and smartphones [1], [13]. If the execution time of a given task is 1ms and memory bandwidth is 10GB/s, $globalMem$=0.5M is the break-even point, i.e., the cost of task copying is lower for smaller copy sizes than suffering from refresh blocking. Notice that 0.5MB is larger than one I-frame of a typical MPEG stream, of which only one frame is needed roughly per 10ms at 30-60 frames/sec. Or consider two 250×250 double-precision matrices (which is less than 0.5MB) that are multiplied, with an execution time that far exceeds 1ms, i.e., no copy task would be required. After all, the execution time exceeds 1ms so that this task's period also has to be larger than the refresh duration. Thus, we conjecture that 0.5MB is quite sufficient to forward outputs of one job to the next for a real-time task with 1ms execution time and a short period in the same range. (Notice that conventional memory accesses with the task are already modeled by assessing the execution time of the task, which is actually reduced due to absence of refreshes compared to a scenario without coloring or copy tasks.)

The cost is strictly dominated by refreshes — overheads due to closing pages are very small as only one close would be inflicted per burst refresh when comparing open to closed page policies. Furthermore, memory can be partitioned even if the number of tasks exceeds the number of ranks since cyclic executives are non-preemptive. In fact, two colors/ranks suffice to hide a burst refresh of one half of the ranks followed later by a burst refresh of the other half. Additional colors may distribute many shorter burst refreshes, but since $f < tRFC$, this provides no benefit in costs or overhead. If one task has high memory demand, the two sets of memory may have more ranks in one set than another, and the task in question may be allocated more than one rank. We can further relax our assumptions and allow task dependencies, which are realized in cyclic executives by ensuring that job J_{ij} of Task T_i is released at or before J_{kl} of T_k, i.e., $r_{ij} \leq r_{kl}$, as are their deadlines, $D_{ij} \leq D_{kl}$ [27].

IV. Implementation

System Architecture: Our experimental framework consists of three components. (1) SimpleScalar 3.0 [29] simulates application execution and generates memory traces for last-level cache (LLC) misses, which includes request intervals, physical address, command type, command size, etc. Each LLC miss results in a memory request (memory transaction) from processor to DRAM (see Fig. 6). Red/solid blocks and lines represent the LLC misses during application execution.

(2) Our coloring tool colors memory transactions based on a transaction's physical address and the coloring policy.

(3) Memory traces are relayed to RTMemController [30], a back-end architecture to assess commodity and real-time memory controllers, where we utilize only the commodity controller model. RTMemController schedules each memory transaction and determines its execution time, which is fed back to SimpleScalar to determine the overall execution time.

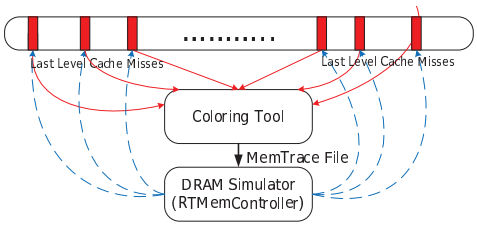

Fig. 6. System Architecture

Instead of using same memory latency for every tasks (the default), we enhanced SimpleScalar to use the average of RTMemController's memory latency for memory transactions per task, which includes the refresh overhead.

We extended RTMemController to support DDR4 refreshes and refresh for a subset of ranks (needed by Colored Refresh and supported by any DDR memory controller). The performance of DRAM is analyzed by the enhanced RTMemController, which schedules the DRAM refresh commands at rank granularity. Refresh is considered when the memory controller serves a memory transaction whose rank ID is derived from the physical address. If this rank is under refresh per schedule, a refresh command is issued in addition to the transaction. This incurs Refresh, Activate and Precharge commands plus purging of the bank's row buffer for a row buffer miss. Our enhanced RTMemController refreshes all ranks in round-robin order and completes all refreshes within DRAM retention time.

Coloring Tool: To hide the refresh overhead for real-time systems, our approach requires that each task be assigned a memory segment via colored memory allocation. We ported TintMalloc [25] to SimpleScalar so that it can select the color of physical addresses in memory for any memory request (stack, code, heap, data segments). A memory coloring policy is configured for each application assigning it as many colors as needed to meet the application's memory requirement. TintMalloc's port reads an application's memory trace and scans the physical addresses accessed. To color a memory space, the Rank_ID of each physical address is calculated and then checked if it maps to the colors assigned to this application. In our case, the rank ID is determined by bits 15-17 of the physical address. If the Rank_ID does not match, these bits are set to the task's respective color. Otherwise, the physical address remains unchanged. To avoid duplicate physical addresses, TintMalloc's port not only changes the rank ID of the physical address, but also assigns it to a free page of the corresponding color (see Algorithm 2). We further retain page locality (and thus cache locality) of physical addresses, i.e., if two physical addresses originally reside in the same page, they still share a page after coloring. Once colored, all physical addresses in a trace belong to a particular memory segment (color). The application only accesses this specific area as per coloring policy.

Algorithm 2 Memory Coloring

1: Input: memTran, task
2: page = ColoredFreeList[task→color]
3: memTran→addr = (memTran→addr && 0xfff) —— (page→addr && 0xfffff000)

Discussion: Copy tasks introduce the main change to memory behavior. A copy task is created separately from its original instance, i.e., we can treat the copy task as a different task than original one with precedence constraints enforced by the schedule (release times and deadlines). The memory state of the original task is transferred to that of the copy task before it runs and transferred back after it has run. These memory transfers warm up caches, but they have to be accounted for as overhead, which we analyze in our work (and Sec III-L shows that it is very small).

While we evaluate our approach by simulator, it shall be noted that it can be ported onto a real architecture with engineering effort [16]. The scheduling policies of Colored Refresh can be realized by modifying the scheduler inside a real-time operating system kernel. Our mechanism utilizes existing DRAM-triggered periodic refreshes combined software for coloring tasks in a cyclic executive without requiring any hardware change. DRAM refreshes are synchronous with processor clock (if the clock is fixed) and can, in fact, optionally be disabled for a subset of ranks [31]. Furthermore, per-rank refresh activation phasing can be reverse engineered by monitoring access latencies during the initialization of our approach.

V. Evaluation Framework and Results

Experimental Setup: We assessed Malardalen WCET benchmark programs [32], modified to operate on much larger data structures and with higher loop bounds and also to match multiples of 1ms to facilitate analysis, on the SimpleScalar 3.0 processor simulator with split data and instruction caches of 16KB size each, a unified L2 cache of 128KB size, and a cache line size of 64B. The memory system is a JEDEC-compliant DDR3 SDRAM (DDR3-1600G) with varied memory density (1, 2, 4, 8, 16, 32, and 64Gb). The DRAM retention time (R) is 64 ms. Furthermore, there are 8 ranks (K=8) and one memory controller on a DRAM chip. Refresh commands are issued by memory controllers at rank granularity.

The Malardalen benchmarks are used to evaluate different types of worst-case execution time (WCET) analysis tools and methods. In our experiment, we analyze the WCET of Malardalen benchmarks and evaluate the real-time performance of our Colored Refresh. Due to space limitations, we constrain results to the Malardalen benchmarks, but our approach also works well with other workloads, such as memory-intensive and CPU-intensive programs. As shown in Sec. III-L, our Colored Refresh obtains performance improvements even for memory-intensive (MPEG stream) or CPU-intensive (matrices multiplication) workloads.

We make a case for platforms with more ranks for real-time systems. Our platform has 1 channel with 4 ranks, where each memory controller has 2 channels. The more the better, and developers may deliberately choose platforms to benefit from more ranks early on during the design.

A. Real-time Tasks

Multiple Malardalen applications are scheduled as real-time tasks under Colored Refresh, each with the largest input to trigger maximum (worst case) memory requirements (see assumptions in Sec. III).

Tab. IV. Tasks and their Memory Assignment

Program	Period	Exec. time	Occupied frames	Available colors (c_i)
lms	20ms	4ms	$f_1 - f_8$	colored with copy task
compress	32ms	6ms	f_1,f_3,f_4,f_5,f_7,f_8	c_2,c_6
cnt	32ms	8ms	f_3,f_4,f_7,f_8	c_1,c_2,c_5,c_6
st	40ms	8ms	f_2,f_4,f_6,f_8	c_1,c_3,c_5,c_7
matmult	80ms	10ms	f_2,f_3,f_4,f_6,f_7,f_8	c_1,c_5

Tab. V. Colors/Task

T	program	color
A	lms	c_7
$\hat{A}$	lms copy task	c_8
B	compress	c_2
C	cnt	c_1
D	st	c_3
E	matmult	c_5

Table IV shows each Malardalen task's execution time (padded by adjusting loop bounds to the indicated values) and period (deadline). Here, the execution time is measured under an ideal method that performs no refreshes. Although this ideal method is infeasible in a reality, we model it in simulation as it provides a lower bound.

For these real-time tasks of Table IV, the hyperperiod H is 160ms, and L is 320ms (given H and R=64ms). By rules (1)-(5) in Colored Refresh, f=8ms is chosen for our system (K=8), and a feasible cyclic schedule is shown in Fig. 7, where c indicates the color that is being refreshed (repeats at R=64ms). Table IV shows the available memory colors per task. Colored Refresh assigns one of its available colors to *compress*, *cnt*, *st* and *matmult*. But for *lms*, "copy tasks" are created since it occupies all frames. Table V indicates one feasible selection of colors after creating copy task $\hat{A}$ (see in Fig. 7) for the intervals $176 \ldots 184$ and $240 \ldots 248$. Notice that at t=180 and t=240, $\hat{A}$ executes instead of A since $c = 7$ is being refreshed. Preceding each $\hat{A}$, the job of A (t=160 and t=220) copies shared state from color $c_7 \rightarrow c_8$, and the $\hat{A}$ jobs from $c_8 \rightarrow c_7$, which is included as 0.05% of the WCET of lms (task A) according to Sec. III-L.

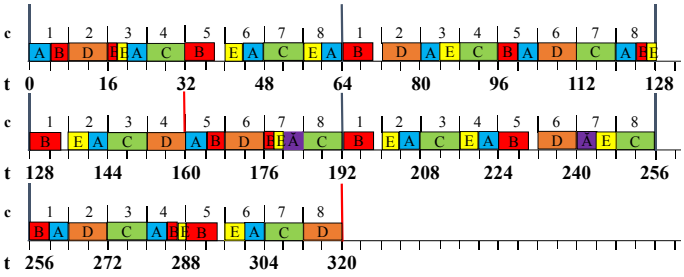

Fig. 7. A Schedule with Copy Tasks

We compared the system utilization under Auto-Refresh, Colored Refresh, and Non-Refresh configurations. Fig. 8 shows the utilization (y-axis starting at 0.96) for different DRAM densities (x-axis) of this real-time task set under different refresh methods. A lower utilization indicates better performance since the real-time system has more slack to guarantee schedulability.

Fig. 8 shows that this real-time task set is always schedulable under our approach with any DRAM density. Under Auto-Refresh, the task set is schedulable for small densities ($<$16Gb) but becomes infeasible (above 1.0) for higher densities ($\geq$16Gb) so that deadlines are missed as additional refresh overhead pushes utilization to above 100%. Also, since the maximum possible frame size for a standard cyclic executive is 12ms (larger than all tasks' execution times), *matmult* requires task splitting for our approach. Assuming a copy+split overhead (see Sec. III-L) for *lms+matmult*, respectively, Colored Refresh suffers 0.05% overhead relative to Non-Refresh, which originates from one additional context switch plus cold cache misses for lms (2.789usecs) and two times two copy overheads (16KB/10GBps=1.6usecs) each for matmult. This overhead is extremely small and remains constant as DRAM density grows.

Observation 1: Compared to Auto-Refresh, Colored Refresh reduces task execution time and enhances system utilization by hiding refresh overhead completely, which increases predictability while preserving real-time schedulability.

As DRAM density grows, the performance of Auto-Refresh decreases (resulting in a larger utilization for the same task set) since higher DRAM density implies more refresh overhead, i.e., additional rows are refreshed per interval. In contrast, Colored Refresh remains stable with increasing DRAM density. For example, with 1Gb DRAM density, Auto-Refresh results in 0.47% larger utilization than Colored Refresh. However, when DRAM density grows to 32Gb and then 64Gb, the task set's utilization under Auto-Refresh is 7.57% and 15.35% higher, respectively, than Colored Refresh. This is because tasks only access colored memory not subject to DRAM refresh delays under our approach.

Observation 2: Colored Refresh obtains stable and predictable performance irrespective of DRAM size while Auto-Refresh's overhead increases significantly with DRAM size.

B. DRAM Performance

Each task only accesses the coloring memory space assigned to it in our approach, while the memory controller sends refresh command as described in Section III-G. Let us discuss the impact on DRAM performance under Colored Refresh.

Fig. 9 shows the normalized average memory access latency (y-axis) of Auto-Refresh over our approach for different tasks (x-axis). Since our approach hides the entire DRAM refresh overhead, the memory latency of our approach equals to non-refresh at each DRAM density. Furthermore, the memory latency of our approach remains stable/same with growing DRAM density.

Red/solid lines inside the boxes mark the median while the green/dotted lines denote averages across the 5 tasks, "whiskers" above/below the box indicate the maximum/minimum. Fig. 9 shows that the memory latency is increased for all benchmarks by auto-refresh compared to our approach. Our Colored Refresh obtains a small latency reduction at low DRAM density (6.5% on avg. at 1Gb density) that

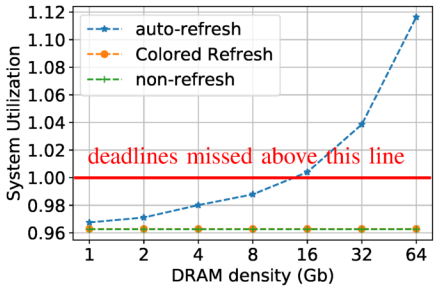

Fig. 8. System Utilization per Refresh Method

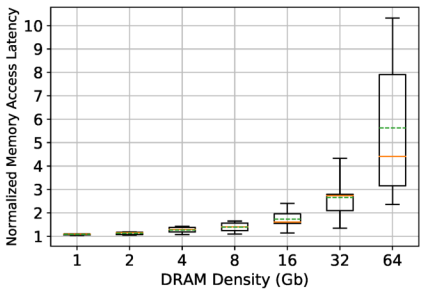

Fig. 9. Memory Latency of Auto-Refresh Normalized to Colored Refresh

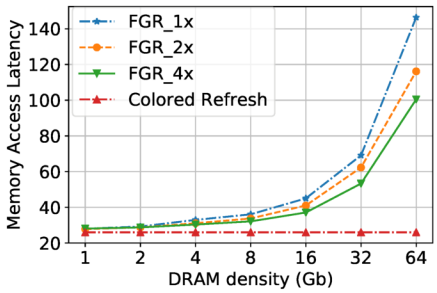

Fig. 10. Colored Refresh vs. Fine Granularity Refresh (FGR)

increases rapidly with density (e.g., 82.2% at 64Gb density). Memory access latency is reduced because memory requests of a real-time task cannot interfere with any DRAM refresh under our approach. When this memory space needs to be refreshed, the task accessing it will be suspended by switching to another task, which is not "colored" to this memory group. Our approach ensures that the memory colors accessed by real-time tasks cannot suffer from refresh overhead unless they are *long* tasks, i.e., Colored Refresh obtains the same memory performance as Non-Refresh.

Observation 3: Colored Refresh reduce memory access latency compared to Auto-Refresh, and this benefit increases with growing DRAM size and density.

Auto-refresh not only increases memory latency, it also causes memory performance to highly fluctuate across applications. Fig. 9 also shows that different tasks suffer different memory access latencies under Auto-refresh dependent on their memory access patterns. E.g., for a density of 16Gb, the latencies of "compress" and "matmult" with auto-refresh are increased by 14.3% and 140.3% compared with our approach (equals to Non-refresh), respectively. With growing density, the refresh delay increases not only due to the overhead to issue more refresh commands, but also because the probability of interference with refreshes increases. Table VI illustrates this by showing the number of memory references suffering from interference by all tasks per DRAM density.

Tab. VI. # Memory Accesses with Refresh Interference

Density	1Gb	2Gb	4Gb	8Gb	16Gb	32Gb	64Gb
Number	1243	1586	2128	2351	2524	2565	2644

Observation 4: Auto-refresh results in high variability of memory latencies depending on access patterns and DRAM density while Colored Refresh hides this variability.

C. Fine Granularity Refresh

JEDEC's DDR4 DRAM specification [13] introduces a Fine Granularity Refresh (FGR) to counter increases in DRAM refresh overhead by creating a range of refresh options that provide a trade-off between refresh latency and frequency.

We compared Colored Refresh with three FGR refresh options, namely 1x, 2, and 4x refresh modes. 1x refresh is a direct extension of DDR2 and DDR3 refreshes. A certain amount of refresh commands are issued, and each command takes tRFC time. The refresh interval (tREFI) of 1x refreshing is 7.8 us [13]. 2x and 4x refreshing require refresh commands to be sent twice and four times as frequently, respectively. The interval (tREFI) is correspondingly reduced to 3.9 us (1.95 us) for 2x (4x) modes. More refresh commands mean fewer DRAM rows are refreshed per command, and, as a result, the refresh latencies tRFC for 2x and 4x modes are shorter. However, when moving from 1x to 2x and then 4x, while tREFI scales linearly, tRFC does not but instead decreases at a rate of less than 50% [3].

The same task set as before (Table IV) and the configuration (Table IV and V) are used for this FGR experiment using the same simulator but with FGR configuration. Fig. 10 compares memory access latency (y-axis) for different DRAM densities (x-axis) under FGR 1x, 2x, 4x to Colored Refresh. We observe that although 4x outperforms 1x and 2x, Colored Refresh uniformly obtains the best performance and lowest memory access latency due to hiding any refresh blocking. Furthermore, the performance of FGR decreases with growing DRAM density. For example, at 1 Gb density, FGR 4x, 2x and 1x have 7.7%, 6.9%, and 8% higher memory access latency, respectively, relative to Colored Refresh. This loss increases to 463%, 347%, and 286% at 64Gb. In contrast, Colored Refresh hides all refresh costs and memory access latencies remain the same irrespective of DRAM densities, i.e., our approach obtain a same performance as Non-refresh.

Observation 5: Colored Refresh exhibits better performance and higher task predictability than DDR4's FGR.

VI. Related Work

We do not reiterate related work discussed in Sect. I due to space limitations here. Several DRAM refresh mechanisms are analyzed, and refresh penalties are quantified in recent work [19], [33], [34], [35], [4], [36]. Mukundan et al. [3] discuss the refresh overhead for DDR4 DRAM with high densities.

Chang et al. [37] make hardware changes to the refresh scheduler inside the memory controller/DRAM subarrays. Kotra et al. [38] use LPDDR-technology for bank-partitioned scheduling without deadlines. Our work focuses on how real-time deadlines can be supported while hiding refresh via hierarchical scheduling in a cyclic executive. Our work focuses on commodity DDR-technology, which is widely used in the embedded field and only supports rank partitions under refresh, but our methodology is equally applicable to LPDDR bank-partitioning (with its added flexibility). Not only LPDDR but

also other DRAM technology, e.g., RLDRAM [26], makes memory references more predictable but is subject to the same refresh blocking, i.e., our refresh hiding is directly applicable to them as well.

Bhat et al. [16] try to make DRAM refresh predictable for real-time embedded system. Instead of hardware auto-refresh, a software-initiated burst refresh is issued at the beginning of every DRAM retention time period. After this refresh burst completes, there is no refresh interference for regular memory accesses during the remainder of DRAM retention time. But the memory remains unavailable during the DRAM retention period, and any stall time due to references to memory under refresh increases rapidly when DRAM density/size grows. Although memory latency is predictable, memory throughput is still lower due to a refresh delay compared to our Colored Refresh.

Pan et al. [39] exploit hierarchical scheduling of refresh, lock/unlock, and server tasks at the top level and real-time tasks inside servers subject to memory partitioning (coloring). Their method supports priority scheduling inside a server and requires interrupts from the memory controller when a refresh completes, or from an equivalent timer. In contrast, we focus on cyclic executives and show how refresh cost can be hidden without hardware modifications.

Selective DRAM Refresh [40] uses a reference bit per row to record and determine if this row needs to be refreshed, which reduces die space relative to Smart Refresh [12]. But the performance of Selective DRAM Refresh still heavily depends on the data access pattern. VRL-DRAM [41] develops a detailed analytical model to estimate the minimum latency of a refresh operation in order to reduce the refresh performance overhead. But it still cannot hide all refresh overhead, and it is implemented inside the memory controller. More importantly, VRL-DRAM cannot provide predictable memory performance by reducing refresh overhead. It is still too pessimistic to predict tight WCETs for real-time systems. Our Colored Refresh is agnostic of data access patterns, and it does not need extra die space while its time overhead is small.

Stuecheli et al. [5] describe Elastic Refresh, which uses predictive mechanisms to decrease the probability of a memory access interfering with a refresh. Elastic Refresh queues refresh commands and schedules them when a DRAM rank is idle. This way, some interferences between memory accesses and refreshes can be avoided. However, as tRFC increases with growing DRAM density, the probability of avoiding interferences decreases. In contrast, our Colored Refresh can hide all refresh delays for regular memory accesses, and its performance is not affected by increasing DRAM density.

Hardware solutions have not been adopted by any vendor. Since our mechanism is completely implemented in software (no hardware change required), it is applicable on today's devices. Since our focus is on available hardware platforms, we did not compare to hardware solutions [6], [7], [8], [9], [10]. Other software solutions (e.g., RAIDR [2], Smart Refresh [12], RAPID [11], et. al.) either heavily rely on specific data access patterns of workloads, or they have high implementation overhead, which makes the problem much worse for real-time system. To the best of our knowledge, our Colored Refresh is the first solution that can hide all refresh overhead for real-time system, i.e., we do not compare to those schemes because worst-case execution times remain just as pessimistic for [2], [12], [11] etc. as they cannot hide refresh overhead.

VII. Conclusion

We analyzed the impact of DRAM refresh delay on the predictability and performance for real-time systems. We proposed Colored Refresh, a novel refresh method to hide DRAM refresh overheads at the software level for cyclic executive scheduling of real-time systems, e.g., in highly critical systems. With Colored Refresh, a memory rank is either accessed by a processor or refreshed by the DRAM controller at any time, but not both. Experimental results confirmed that our approach hides refresh overhead for real-time task execution, thus enhancing their predictability and memory throughput. Compared to previous work, Colored Refresh can be safely implemented without much overhead (less than 1% for task copying and splitting) irrespective of increasing DRAM density/sizes, yet with better memory performance than DDR4 Fine Granularity Refresh.

Acknowledgment

We would like to thank the anonymous reviewers for their thorough reading and constructive criticism that lead to an improved paper. This work was supported in part by NSF grants 1813004, 1239246, and 0958311.

References

[1] JEDEC STANDARD, "DDR3 SDRAM SPECIFICATION," 2010.

[2] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," in *ISCA*, 2012, pp. 1–12.

[3] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F Martínez, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *ISCA*, 2013.

[4] Prashant Nair, Chia-Chen Chou, and Moinuddin K Qureshi, "A case for refresh pausing in DRAM memory systems," in *HPCA*, 2013, pp. 627–638.

[5] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C Hunter, and Lizy K John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO*, 2010, pp. 375–384.

[6] Hongzhong Zheng, Jiang Lin, Zhao Zhang, Eugene Gorbatov, Howard David, and Zhichun Zhu, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *MICRO*, 2008, pp. 210–221.

[7] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014, pp. 356–367.

[8] Joohee Kim and Marios C Papaefthymiou, "Dynamic memory design for low data-retention power," in *PATMOS*, 2000.

[9] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Hardware/Software Codesign and System Synthesis*, 2011.

[10] Yinhe Han, Ying Wang, Huawei Li, and Xiaowei Li, "Data-aware dram refresh to squeeze the margin of retention time in hybrid memory cube," in *International Conference on Computer-Aided Design*, 2014.

[11] Ravi K Venkatesan, Stephen Herr, and Eric Rotenberg, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *ISCA*, 2006, pp. 155–165.

[12] Mrinmoy Ghosh and Hsien-Hsin S Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs," in *MICRO*, 2007, pp. 134–145.

[13] Standard, JEDEC, "DDR4 SDRAM," 2012.

[14] J. Wegener and F. Mueller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, 2001.

[15] Pavel Atanassov and Peter Puschner, "Impact of dram refresh on the execution time of real-time tasks," in *Workshop on Application of Reliable Computing and Communication*, 2001.

[16] Balasubramanya Bhat and Frank Mueller, "Making DRAM refresh predictable," in *ECRTS*, 2010, pp. 145–154.

[17] Heesang Kim, Byoungchan Oh, Younghwan Son, Kyungdo Kim, Seon-Yong Cha, Jae-Goan Jeong, Sung-Joo Hong, and Hyungcheol Shin, "Characterization of the variable retention time in dynamic random access memory," *IEEE Transactions on Electron Devices*, 2011.

[18] Yuki Mori, Kiyonori Ohyu, Kensuke Okonogi, and R-I Yamada, "The origin of variable retention time in dram," in *Electron Devices Meeting*, 2005.

[19] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob, "DRAM refresh mechanisms, penalties, and trade-offs," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 108–121, 2016.

[20] Standard, JEDEC, "LPDDR3 SDRAM Specification," 2012.

[21] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible auto-refresh: enabling scalable and energy-efficient DRAM refresh reductions," in *ISCA*, 2015.

[22] Theodore P Baker and Alan Shaw, "The cyclic executive model and ada," in *RTSS*, 1988.

[23] C Douglass Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Real-time systems*, vol. 4, no. 1, pp. 37–53, 1992.

[24] Tom Fleming, Sanjoy Baruah, and Alan Burns, "Improving the schedulability of mixed criticality cyclic executives via limited task splitting," in *International Conference on Real-Time Networks and Systems*, 2016, pp. 277–286.

[25] Xing Pan, Yasaswini Jyothi Gownivaripalli, and Frank Mueller, "Tint-malloc: Reducing memory access divergence via controller-aware coloring," in *IPDPS*, 2016, pp. 363–372.

[26] M. Hassan, "On the off-chip memory latency of real-time systems: Is ddr dram really the best option?," in *RTSS*, Dec 2018, pp. 495–505.

[27] J. Liu, *Real-Time Systems*, Prentice Hall, 2000.

[28] Jane Liu, "Real-time systems. 2000," .

[29] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar toolset," Tech. Rep. CS-TR-96-1308, University of Wisconsin - Madison, CS Dept., July 1996.

[30] Yonghui Li, Benny Akesson, and Kees Goossens, "Dynamic command scheduling for real-time memory controllers," in *ECRTS*, 2014, pp. 3–14.

[31] Konstantinos Tovletoglou, Dimitrios S Nikolopoulos, and Georgios Karakonstantis, "Relaxing dram refresh rate through access pattern scheduling: A case study on stencil-based algorithms," in *International Symposium on On-Line Testing and Robust System Design*, 2017.

[32] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *Workshop on Worst-Case Execution Time Analysis*, 2010, pp. 137–147.

[33] Philip G Emma, William R Reohr, and Mesut Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications," *IEEE Micro*, pp. 47–56, 2008.

[34] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *ISCA*, 2013.

[35] Kinam Kim and Jooyoung Lee, "A new investigation of data retention time in truly nanoscaled DRAMs," *IEEE Electron Device Letters*, pp. 846–848, 2009.

[36] Takeshi Hamamoto, Soichi Sugiura, and Shizuo Sawada, "On the retention time distribution of dynamic random access memory (dram)," *IEEE Transactions on Electron devices*, 1998.

[37] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving dram performance by parallelizing refreshes with accesses," in *HPCA*, Feb 2014, pp. 356–367.

[38] Jagadish B. Kotra, Narges Shahidi, Zeshan A. Chishti, and Mahmut T. Kandemir, "Hardware-software co-design to mitigate dram refresh overheads: A case for refresh-aware process scheduling," *SIGPLAN Not.*, vol. 52, no. 4, pp. 723–736, Apr. 2017.

[39] Xing Pan and Frank Mueller, "The colored refresh server for dram," in *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, May 2019.

[40] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011, pp. 213–224.

[41] Anup Das, Hasan Hassan, and Mutlu Onur, "Vrl-dram: Improving dram performance via variable refresh latency," in *DAC*, 2018.