The Colored Refresh Server for DRAM

Xing Pan, Frank Mueller
North Carolina State University, Raleigh, USA, *mueller@cs.ncsu.edu*

***Abstract*—Bounding each task's worst-case execution time (WCET) accurately is essential for real-time systems to determine if all deadlines can be met. Yet, access latencies to Dynamic Random Access Memory (DRAM) vary significantly due to DRAM refresh, which blocks access to memory cells. Variations further increase as DRAM density grows.**

This work contributes the "Colored Refresh Server" (CRS), a uniprocessor scheduling paradigm that partitions DRAM in two distinctly colored groups such that refreshes of one color occur in parallel to the execution of real-time tasks of the other color. By executing tasks in phase with periodic DRAM refreshes with opposing colors, memory requests no longer suffer from refresh interference. Experimental results confirm that refresh overhead is completely hidden and memory throughput enhanced.

I. Introduction

Dynamic Random Access Memory (DRAM) has been the memory of choice in embedded systems for many years due low cost combined with large capacity, albeit at the expense of volatility. As specified by the DRAM standards [1], [2], each DRAM cell must be refreshed periodically within a given refresh interval. The refresh commands are issued by the DRAM controller via the command bus. This mode, called auto-refresh, recharges all memory cells within the "retention time", which is typically 64ms for commodity DRAMs under 85°C [1], [2]. While DRAM is being refreshed, a memory space (i.e., a DRAM rank) becomes unavailable to memory requests so that any such memory reference blocks the CPU pipeline until the refresh completes. Furthermore, a DRAM refresh command closes a previously open row and opens a new row subject to refresh [3], even though data of the old row may be reused (referenced) before and after the refresh. Hence, the delay suffered by the processor due to DRAM refresh includes two aspects: (1) the cost (blocking) of the refresh operation itself, and (2) reloads of the row buffer for data displaced by refreshes. As a result, the response time of a DRAM access depends on its point in time during execution relative to DRAM refresh operations.

Prior work indicated that system performance is significantly degraded by refresh overhead [4], [5], [6], [7], a problem that is becoming more prevalent as DRAMs are increasing in density. With growing density, more DRAM cells are required per chip, which must be refreshed within the same retention time, i.e., more rows need to be refreshed within the same refresh interval. This increases the cost of a refresh operation and thus reduces memory throughput. 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 as a blocking term on response time. Atanassov and Puschner [8] 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 or the blocking term were augmented by the maximum possible refresh delay, many schedules would become theoretically infeasible, even though executions may meet deadlines in practice. Although Bhat et al. make refreshes predictable and reduce preemption due to refreshes by triggering them in software instead of hardware auto-refresh [3], the cost of refresh operations is only considered, but cannot be hidden. Also, a task cannot be scheduled under Bhat if its period is less than the execution time of a burst refresh.

This work was supported in part by NSF grants 1239246,1329780,1525609 and 1813004.

This work contributes the "Colored Refresh Server" (CRS) to remove task preemptions due to refreshes and to hide DRAM refresh overhead.

Contributions: (1) The impact of refresh delay under varying DRAM densities/sizes is assessed for real-time systems with stringent timing constraints.
(2) The Colored Refresh Server (CRS) for uniprocessors is developed to refresh DRAM via memory space coloring and shown to hide refresh overhead almost entirely .
(3) Experiments with real-time tasks confirm that both refresh delays are hidden and DRAM access latencies are reduced.

II. Background and Motivation

Today's computers predominantly utilize dynamic random access memory (DRAM), where each bit of data is stored in a separate capacitor within DRAM memory. To serve memory requests from the CPU, the memory controller acts as a mediator between the last-level cache (LLC) and DRAM devices. Once memory transactions are received by a DRAM controller from its memory controller, these read/write requests are translated into corresponding DRAM commands and scheduled while satisfying the timing constraints of DRAM banks and buses. A DRAM controller is also called a node that governs DRAM memory organized into channels, ranks and banks.

A. Memory Space Partitioning

We assume a DRAM hierarchy with node, channel, rank, and bank abstraction. To partition this memory space, we obtained a copy of TintMalloc [9], a heap allocator that "colors" memory pages with controller (node) and bank affinity.

TintMalloc allows programmers to 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. E.g., 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 it is allocated to. No other memory rank will ever be touched by it. By design, there is a penalty for the first heap allocation request with a color under TintMalloc. This penalty only impacts the initialization phase. After a "first touch" page initialization, the latency of any subsequent accesses to colored memory is always lower than that of uncolored memory subject to buddy allocation (Linux default). Also, once the colored free list has been populated with pages, the initialization cost becomes constant for a stable working set size, even for dynamic allocations/deallocation assuming they are balanced in size. Real-time tasks, after their initialization, experience highly predictable latencies for subsequent memory requests. Hence, a first coloring allocation suffices to amortize the overhead of initialization.

B. DRAM Refresh

Refresh commands are periodically issued by the DRAM controller to recharge all DRAM cells, which ensures data validity in the presence of electric leakage. A refresh command forces a read to each memory cell followed by a write-back without modification, which recharges the cell to its original level. The reference refresh interval of commodity DRAMs is 64ms under 85°C (185°F) or 32ms above 85°C, the so-called retention time, $tRET$, of leaky cells, sometimes also called refresh window, $tREFW$ [1], [2], [10], [11]. All rows in a DRAM chip need to be refreshed within $tRET$, otherwise data will be lost. In order to reduce refresh overhead, refresh commands are processed at rank granularity for commodity DRAM [12]. The DRAM controller can either schedule an automatic refresh for all ranks simultaneously (simultaneous refresh), or schedule automatic refresh commands for each rank independently (independent refresh). Whether simultaneous or independent, a successive area of multiple cells in consecutive cycles is affected by a memory refresh cycle. This area is called a "refresh bin" and contains multiple rows. The DDR3 specification [1] generally requires that 8192 automatic refresh commands are sent by the DRAM controller to refresh the entire memory (one command per bin at a time). Here, the refresh interval, $tREFI$, denotes the gap between two refresh commands, e.g., $tREFI = 7.8us$, i.e., $tREFW/8192$. 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.

Memory ranks remain unavailable during a refresh cycle, $tRFC$, i.e., memory accesses (read and write operations) to this region will stall the CPU during a refresh cycle. DRAM ranks can be refreshed in parallel under auto-refresh. However, the amount of unavailable memory increases when refreshing ranks in parallel. A fully parallel refresh blocks the entire memory space for $tRFC$. This blocking time not only decreases system performance, but can also result in deadline misses unless it is considered in a blocking term by all tasks.

Furthermore, a side effect of DRAM refresh is that 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 again, and the deferred memory access can continue. As a result, an additional overhead of $tRP + tRAS$ is incurred to close and re-open rows since the refresh purges all buffers. By considering both the cost of a refresh operation itself and the extra row close/re-open delay, DRAM refresh not only decreases memory performance, but also causes the response time of memory accesses to fluctuate. Due to the asynchronous nature of refreshes and task preemptions, it is hard to accurately predict and bound DRAM refresh delay. Depending on when a refresh command is sent to a bin (successive rows), two scheduling strategies exist: distributed and burst refresh (see [13]).

III. Design

The core 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 is issued until the refresh completes. As a result, memory latency increases and becomes highly unpredictable since refreshes are asynchronous. The central idea of our approach is to remove DRAM refresh interference by memory partitioning (coloring). Given a real-time task set, we design a hierarchical resource model [14], [15], [16] to schedule it with two servers. To this end, we partition the DRAM space into two colors, and each server is assigned a colored memory partition. (We show in [13] that two colors suffice, i.e., adding additional colors does not extend the applicability of the method, it would only make schedulability tests more restrictive.) By cooperatively grouping applications into two resource servers and appropriately configuring those servers (period and budget), we ensure that memory accesses can no longer be subject to interference by DRAM refreshes. Our approach can be adapted to any real-time scheduling policy supported inside the CRS servers. In this section, we describe the resource model, bound the timing requirements of each server, and analyze system schedulability.

A. Assumptions

We assume that a given real-time task set is schedulable with auto-refresh under a given scheduling policy (e.g., EDF or fixed priority), i.e., that the worst-case blocking time of refresh is taken into account. As specified by the DRAM standards [1], [2], the entire DRAM has to be refreshed within its retention time, $tRET$, either serially or in parallel for all K ranks. . We also assume hardware support for timer interrupts and memory controller interrupts (MC interrupts).

B. Task Model

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 job, task utilization $u_i = e_i/D_i$, and a hyperperiod H of $\mathcal{T}$. Furthermore, let
$tRET$ be the DRAM retention time,
L be the least common multiple of H and $tRET$, and
K be the number of DRAM ranks, and let k_i denote rank i.

C. DRAM Refresh Server Model

The Colored Refresh Server (CRS) partitions the entire DRAM space into two "colors", such that each color contains one or more DRAM ranks, e.g., $c_1(k_0, k_1...k_i)$, and $c_2(k_{i+1}, k_{i+2}...k_{K-1})$.

We build a hierarchical resource model (task server) [16], $S(W, A, c, p_s, e_s)$, with CPU time as the resource, where
W is the workload model (applications),
A is the scheduling algorithm, e.g., EDF or RM,
c denotes the memory color(s) assigned to this server, i.e., a set of memory ranks available for allocation,
p_s is the server period, and
e_s is the server execution time (budget). Notice that the base model [16] is compositional (assuming an anomaly-free processor design) and it has been shown that a schedulability test within the hyperperiod suffices for uniprocessors.

The refresh server can execute when
(i) its budget is not zero,
(ii) its available task queue is not empty, and
(iii) its memory color is not locked by a "refresh task" (introduced below). Otherwise, it remains suspended.

D. Refresh Lock and Unlock Tasks

We employ "software burst parallel refresh" [3] to refresh multiple DRAM ranks in parallel via the burst pattern (i.e., another refresh command is issued for the next row immediately after the previous one finishes [13]. In our approach, there are two "refresh lock tasks" (T_{rl1} and T_{rl2}) and two "refresh unlock tasks" (T_{ru1} and T_{ru2}), T_{rl1} and T_{ru1} surround the refresh for color c_1 and are allocated to server S_1 while T_{rl2} and T_{ru2} surround the refresh for color c_2 and are allocated by server S_2. The top-level task set $\mathcal{T}_\top$ of our hierarchical model thus consists of the two server tasks S_1 and S_2 plus another two tasks per color, with the highest priority, for refresh lock/unlock, T_{rl1} and T_{ru1} as well as T_{ru2} and T_{ru2}: $\mathcal{T}_\top = \{S_1, S_2, T_{rl1}, T_{ru1}, T_{rl2}, T_{ru2}\}$.

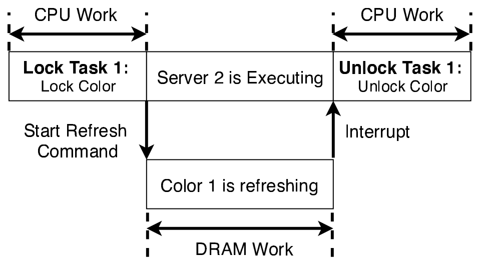

Fig. 1. Refresh Task with CPU Work plus DRAM Controller Work

When a refresh lock task is released (Fig. 1), the CPU sends a command to the DRAM controller to initiate parallel refreshes in a burst. Furthermore, a "virtual lock" is obtained for the colors subject to refresh. Due to their higher priority, refresh lock/unlock tasks preempt any server (if one was running) until they complete. Subsequently, the refresh lock task terminates so that a server task (of opposite color) can be resumed. In parallel, the "DRAM refresh work" is performed, i.e., burst refreshes are triggered by the controller. We use e_{r1} and e_{r2} to represent the duration of DRAM refresh per color $r1$ and $r2$, respectively. A CPU server resumes execution only if its budget is not exhausted, its allocated color is not locked, and some task in its server queue is ready to execute.

Once all burst refreshes have completed, an interrupt is triggered, which causes the CPU to call the refresh unlock task that unlocks the newly refreshed colors so that they become available again. This interrupt can be raised in two ways: (1) If the DRAM controller supports interrupt completion notification in hardware, it can be raised by the DRAM controller. (2) Otherwise, the length of a burst refresh, δ, can be measured and the interrupt can be triggered by imposing a phase of δ on the unlock task relative to the phase of the lock task of the same color. Interrupts are triggered at absolute times to reduce jitter (see Sect. IV). The overhead of this interrupt handler is folded into the refresh unlock task for schedulability analysis in the following. In practice, the cost of a refresh lock/unlock task is extremely small since it only programs the DRAM controller or handles the interrupt.

The periods of both the refresh lock and unlock task are $tRET$. The refresh lock tasks are released at $k * tRET$, while the refresh unlock tasks are released at $k * tRET + \delta$. The phases ϕ of T_{rl1} and T_{rl2} are $\frac{tRET}{2}$ and 0, respectively, i.e., memory ranks allocated to S_2 are refreshed first followed by those of S_1. Let us summarize:
$\mathcal{T}_\top = \{S_1, S_2, T_{rl1}, T_{ru1}, T_{rl2}, T_{ru2}\}$, where
$S_1 = (0, p_1, e_1, p_1), S_2 = (0, p_1, e_2, p_1)$,
$T_{rl1} = (tRET/2, tRET, e_{rl}, \delta), T_{rl2} = (0, tRET, e_{rl}, \delta)$,
$T_{ru1} = (tRET/2+\delta, tRET, e_{ru}, \delta), T_{ru2} = (\delta, tRET, e_{ru}, \delta)$.

The execution times e_{rl} and e_{ru} of the lock and unlock tasks are upper bounds on the respective interrupts plus programming the memory controllers for refresh and obtaining the lock for the former and just unlocking the the latter task, respectively. (They are also upper bounded by δ.) The execution times e_1 and e_2 depend on the task sets of the servers covered later, while their deadlines are equal to their periods (p_1 and p_2). The task set $\mathcal{T}_\top$ can be scheduled statically as long as the lock and unlock tasks have a higher priority than the server tasks. A refresh unlock task is triggered by interrupt with a period of $tRET$. Since we refresh multiple ranks in parallel, the cost of refreshing one entire rank is the same as the cost of refreshing multiple ones. Furthermore, the cost of the DRAM burst refresh, δ, is small (e.g., less than $0.2ms$ for a 2Gb DRAM chip with 8 ranks).

E. CRS Implementation

Consumption and Replenishment: The execution budget is consumed one time unit per unit of execution. The execution budget is set to e_s at time instants $k*p_s$, where $k \geq 0$. Unused execution budget cannot be carried over to the next period.

Scheduling: As described in Sec. III-D, the two refresh servers, S_1 and S_2, are treated as periodic tasks with their periods and execution times. We assign static priorities to servers and refresh tasks (lock and unlock). Instead of rate-monotonic priority assignment (shorter period, higher priority), static scheduling requires assignment of a strict fixed priority to each task (each server and each refresh task). The four refresh tasks receive the highest priority in the system. S_1 has the next highest priority and S_2 has a lower one than S_1. However, a server may only execute while its colors are unlocked. Tasks can be scheduled with any real-time scheduling policy supported inside the CRS servers, such as EDF, RM, or cyclic executive. During system initialization, we utilize the default hardware auto-refresh and switch to CRS once servers and refresh tasks have been released.

Example: Let there be four real-time tasks with periods and execution times of $T_1(16,4), T_2(16,2), T_3(32,8), T_4(64,8)$. DRAM is partitioned into 2 colors, c_1 and c_2, which in total contains 8 memory ranks ($k_0 - k_7$).

The four real-time tasks are grouped into two Colored Refresh Servers:
$S_1((T_1,T_2), RM, c_1(k_0,k_1,k_2,k_3), 16ms, 6ms)$ and
$S_2((T_3,T_4), RM, c_2(k_4,k_5,k_6,k_7), 16ms, 6ms)$.
In addition, refresh lock tasks T_{rl1} and T_{rl2} have a period of $tRET$ (64ms) and trigger refreshes for c_1 and c_2, respectively, i.e., T_{rl2} triggers refreshes for (k_4,k_5,k_6,k_7) with ϕ=0 while T_{rl1} triggers refreshes (k_0,k_1,k_2,k_3) with ϕ=32ms. Once refreshes have finished, the refresh unlock tasks T_{ru1} and T_{ru2} update corresponding memory colors to be available again.

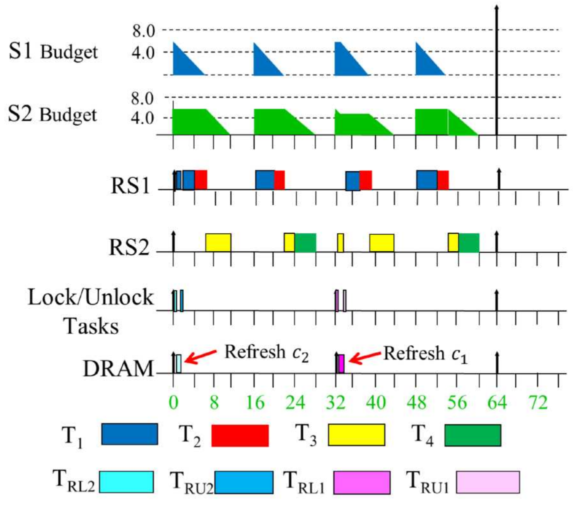

Fig. 2. Server scheduling example

Fig. 2 depicts the task execution for our CRS. Here, regular memory accesses from a processor of one color are overlaid with DRAM refresh commands of the opposite color, just by scheduling servers and refresh tasks according to their policies. We further observe that S_2 executes at time 32ms, even though S_1 has a higher priority than S_2. This is because color c_1 is locked by refresh task T_{rl1}. S_1 can preempt S_2 once c_1 is unlocked by T_{ru1}, i.e., after its DRAM refresh finishes.

F. Schedulability Analysis

In this section, we combine the analysis of the periodic capacity bound and the utilization bound [13] to bound the response time, quantify the cost of CRS, and analyze the schedulability of entire system, including the servers and refresh lock/unlock tasks, i.e., T_{rl1} (0, $tRET$, e_{rl}, $tRET$), T_{rl2} ($tRET/2$, $tRET$, e_{rl}, $tRET$), T_{ru1} (δ, $tRET$, e_{ru}, $tRET$), T_{ru2} ($tRET/2+\delta$, $tRET$, e_{ru}, $tRET$), S_1 (p_1, e_1), and S_2 (p_2, e_2), where we assume that the two refresh lock tasks have the same execution time (e_{rl}), as do the two refresh unlock tasks (e_{ru}). Compared to auto-refresh, we build a hierarchical resource model (by selecting period, budget, and workload for both servers), which not only guarantees schedulability but also has a lower cost than the overhead of auto-refresh. As a result of removing DRAM refresh interference, our Colored Refresh Server outperforms auto-refresh.

As described in Sec. III-D, the refresh tasks, T_{rl1}, T_{rl2}, T_{ru1}, and T_{ru2}, have the highest priority, S_1 has the next highest priority, followed by S_2 with the lowest priority. To guarantee the schedulability of a real-time system with static priority scheduling, we require that
(1) each task satisfies the TDA (time demand analysis) requirement, and
(2) the total utilization does not exceed 1, i.e.,
$\frac{e_1}{p_1} + \frac{e_2}{p_2} + 2 * \frac{e_{rl}}{tRET} + 2 * \frac{e_{ru}}{tRET} \leq 1$.

For hierarchical resource models [16], S_1 and S_2 are treated as periodic tasks.

With auto-refresh, the maximum response time of S_1 is $r_{s1}^{(k)} = e_{s1} + b$, where $b = \lfloor \frac{r_{s1}^{(k-1)}}{tREFI} \rfloor * (tRFC + tRP + tRAS)$ represents the refresh overhead.

The maximum response time of S_2 is:
$r_{s2}^{(k)} = e_{s2} + \lceil \frac{r_{s2}^{(k-1)}}{p_{s1}} \rceil * e_{s1} + b$, where $b = \lfloor \frac{r_{s2}^{(k-1)}}{tREFI} \rfloor * (tRFC + tRP + tRAS)$ represents the refresh overhead.

With our CRS, S_1 and S_2 are co-scheduled with the refresh lock and unlock tasks. The maximum response time of S_1 is
$r_{s1}^{(m)} = e_{s1} + 2 * \lceil \frac{r_{s1}^{(m-1)}}{tRET} \rceil * (e_{rl} + e_{ru}) + \epsilon_1$,
where ϵ_1 is the refresh overhead that cannot be hidden by our CRS, which is
$\epsilon_1 = \sum_{n,k} \gamma$ for $n \in [0, L/p_2]$ and $k \in [0, L/tRET]$; also
$\gamma = e_{r1}$ if
(1) $(m+1)*p_1 > p_{rl1}*k > m*p_1$ and $p_{rl1}*k - p_1*m \leq r_{s1}^m$
(2) $(n+1)*p_2 > p_{rl1}*k > n*p_2$ and $p_{rl1}*k - p_2*n \geq r_{s2}^n$;
otherwise, $\gamma = 0$.

The maximum response time of S_2 is:
$r_{s2}^{(n)} = e_{s2} + \lceil \frac{r_{s2}^{(n-1)}}{p_{s1}} \rceil * e_{s1} + 2 * \lceil \frac{r_{s2}^{(k-1)}}{tRET} \rceil * (e_{rl} + e_{ru}) + \epsilon_2$,
where ϵ_2 is the refresh overhead that cannot be hidden by our CRS, which is
$\epsilon_2 = \sum_{m,k} \gamma$ for $m \in [0, L/p_1]$ and $k \in [0, L/tRET]$; also
$\gamma = e_{r2}$ if
(1) $(m+1)*p_1 > p_{rl2}*k > m*p_1$ and $p_{rl2}*k - p_1*m \geq r_{s1}^m$
(2) $(n+1)*p_2 > p_{rl2}*k > n*p_2$ and $p_{rl2}*k - p_2*n \leq r_{s2}^n$;
otherwise, $\gamma = 0$.

As defined in Sec. III-D, e_{r1} and e_{r2} represent the execution time of burst refreshes for the corresponding colors, respectively. r_{s1}^m and r_{s2}^n can be calculated by response time analysis under fixed-priority assignment. As we showed above, the periods of both T_{rl1} and T_{rl2} are the DRAM retention time, i.e., $p_{rl1} = p_{rl2} = tRET$.

This shows that overhead is only incurred when a refresh task is released but its corresponding server (accessing the opposite color) is not ready to execute. Here, the overhead of refresh operations cannot be hidden. But this overhead is a small fraction of the entire DRAM refresh cost. Besides, it is predictable and quantifiable. The refresh overheads, ϵ_1 and ϵ_2, under CRS can be optimized as discussed next.

Let us assume a task set is partitioned into two groups, each associated with its own server. The servers with periods p_1 and p_2 each have a periodic capacity and utilization bound that can be calculated [13]. For server S_1 and S_2, let PCB_1 and PCB_2 denote their periodic capacity bounds, while UB_1 and UB_2 denote their utilization bounds.

Compared to the response time under auto-refresh, CRS obtains a lower response time due to reduced refresh overhead, and requirement (1) is satisfied. We further assume that the execution times of refresh lock/unlock tasks (T_{rl1}, T_{rl2}, T_{ru1} and T_{ru2}) are identical (and known to be very small in practice). Since refresh tasks issue refresh commands in burst mode, CRS does not result in additional row buffer misses, i.e., e_{r1} and e_{r2} do not need to consider extra tRP or $tRAS$ overheads, which makes them smaller than their corresponding overheads under auto-refresh [3], i.e., requirement (2) is satisfied. Finally, our CRS not only bounds the response time of each server, but also guarantees system schedulability.

For a "short task", there is extra overhead under CRS due to the task copy cost [13]. The cost ($datasize * bandwidth$) can be folded into the response time of one sever if it has a copy task. However, the cost of task copying is much less than the delay incurred on real-time tasks by a refresh, i.e., a "short task" can be scheduled under our CRS.

IV. Implementation

CRS has been implemented in an environment of three components, a CPU simulator, a scheduler combined with a coloring tool, and a DRAM simulator. SimpleScalar 3.0 [17] simulates the execution of an application and generates its memory traces. Memory traces are recorded to capture last-level cache (LLC) misses, i.e., from the L2 cache in our case. This information 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. 3). The red/solid blocks and lines represent the LLC misses during application execution. The memory transactions of different applications are combined by a hierarchical scheduler according to scheduling policies (e.g., the priority of refresh tasks and servers at the upper level and task priorities within servers at the lower level). Furthermore, each memory transaction's physical address is colored based on the coloring policy (see "coloring tool" in [13]).

After scheduling and coloring, the memory traces are exposed to the DRAM simulator, RTMemController [18], to analyze the DRAM performance. All memory transactions

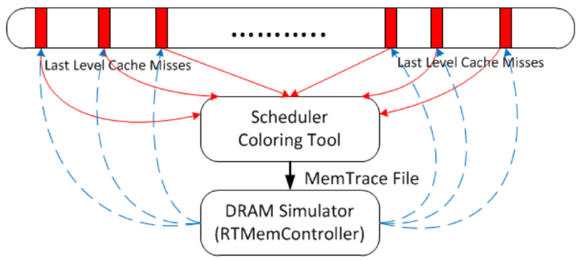

Fig. 3. System Architecture

of the trace are scheduled by RTMemController, and their execution times are calculated. Instead of using fixed memory latencies for every task, which is the default, we enhanced SimpleScalar to consider the average execution time of each task's memory transactions analyzed by RTMemController over all LLC misses, which includes the DRAM refresh overhead. At last, the result of RTMemController (execution time of each memory transaction) is fed back to SimpleScalar to determine the application's overall execution time. This models the execution time of each real-time application, including its DRAM performance per memory access.

RTMemController is a back-end architecture for real-time memory controllers and was originally designed for DDR3 SDRAMs using dynamic command scheduling. We extended RTMemController to support burst refresh and DDR4 Fine Granularity Refresh (FGR). The performance of DRAM is analyzed by the enhanced RTMemController, which schedules the DRAM refresh commands at rank granularity.

The simulation environment also supports generation of an interrupt triggered by the DRAM controller when the bursts of a refresh task complete. Should a DRAM controller not support such an interrupt signal upon refresh completion, one can utilize a second timer. The refresh tasks are already triggered by a first periodic timer, say at time t. Once all DRAM refreshes have been issued by a refresh task, an absolute timer is installed for $t + tRFC$ (adding the refresh blocking time) to trigger the handler that unlocks the colors subject to refresh.

Color locks are implemented as attributes for scheduling at the top level, i.e., a flag per color suffices. Such a flag is set for colors of a refresh task before this refresh task is dispatched, and the flag is cleared inside the handler invoked upon refresh completion. We referred to a "virtual" lock earlier since the mechanism resembles a lock in terms of resource allocation for schedulability analysis. However, it cannot be implemented via a lock since a server task, if it obtained a lock, could not release it when interrupted by a refresh task. Instead, the refresh task would have to steal this lock, which is typically not supported by any API. Since we are implementing low-level scheduling directly, our flag solution is not only much easier to realize, it also has lower overhead as neither atomic instructions nor additional queues are required.

A discussion of how to extend this work to multi-processors and realize it completely in software is given in [13].

V. Evaluation Framework

We assess the Malardalen WCET benchmark programs [19] atop SimpleScalar 3.0 [17] combined with RTMemController [20]. The processor is configured 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 adjustable memory density (1Gb, 2Gb, 4Gb, 8Gb, 16Gb, 32Gb and 64Gb). The DRAM retention time, $tRET$, is 64 ms. Furthermore, there are 8 ranks, i.e., $K = 8$, and one memory controller per DRAM chip. (The approach requires a minimum of two ranks, which even small embedded systems tend to provide.) Refresh commands are issued by memory controllers at rank granularity.

TABLE I
Real-Time Task Set

Application	Period	Execution Time
cnt	20ms	3ms
compress	10ms	1.2ms
lms	10ms	1.6ms
matmult	40ms	10ms
st	8ms	2ms

Multiple Malardalen applications are scheduled as real-time tasks under both CRS (hierarchical scheduling of refresh tasks plus servers and then real-time tasks within servers) and auto-refresh (single-level priority scheduling). Execution times and periods (deadlines) per Malardalen task are shown in Table I. Here, the base for execution time is an ideal one without refreshes. This ideal method is infeasible in a practice, but it provides a lower bound and allows us to assess how close a scheme is to this bound. The real-time task set shown in Table I can be scheduled under either a dynamic priority policy (e.g., EDF) or a static priority policy (e.g., RM and DM). We assess EDF due to space limitations, but CRS also works and obtains better performance than auto-refresh under static priority scheduling.

The task set in Table I has a hyperperiod of 40ms, and is schedulable under EDF without considering refresh overhead. CRS segregates each Malardalen application into one of the two servers. As shown in Sec.III-F, Algorithms **??** and **??** assist in finding a partition with minimal refresh overhead. There may be multiple best configurations under CRS, but we only assess experiments with one of them due to symmetry.

We employ two servers (S_1 and S_2) and refresh tasks (T_{rl1}, T_{rl2}, T_{ru1} and T_{ru2}). Applications "cnt", "lms" and "st" are assigned to S_1 with 4ms periods and a 2.4ms budget, while application "compress" and "matmult" belong to S_2 with 4ms periods and a 1.6ms budget. The entire memory space is equally partitioned into 2 colors (c_1 and c_2), i.e., the 8 DRAM ranks comprise 2 groups with 4 ranks each. TintMalloc [9] ensures that tasks of one server only access memory of one color, i.e., tasks in S_1 only allocate memory from the 4 ranks belonging to c_1 while tasks in S_2 only allocate from c_2. Furthermore, memory within c_1 and c_2 is triggered by T_{rl1} and T_{rl2} to be refresh by the burst pattern. The memory space is locked, and the server allocated to this space/color is prevented to execute during refresh until it is unlocked by T_{ru1} and T_{ru2} when all refresh operations finish. The periods of all refresh tasks (T_{rl1}, T_{rl2}, T_{ru1} and T_{ru2}) are equal to the DRAM retention time $tRET$ (64ms), and their phases are 32ms and 0 for T_{rl1} and T_{rl2}, respectively.

VI. Experimental Results

Fig. 4 shows the memory access latency (y-axis) of auto-refresh normalized to that of CRS for all benchmarks at different DRAM densities (x-axis). The red/solid line inside the boxes indicates the median while the green/dashed line represents the average across the 5 tasks. The "whiskers" above/below the box indicate the maximum and minimum.

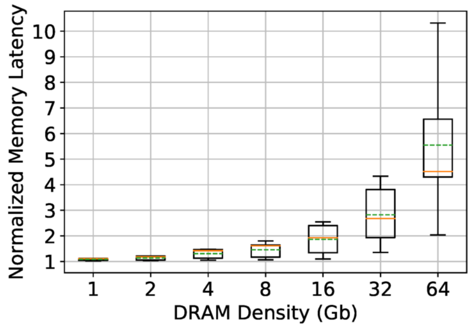

Fig. 4. Memory Latency of Auto-Refresh Normalized to CRS

We observe that CRS obtains better memory performance than auto-refresh, i.e., CRS reduces the memory latency due to refresh stalls for all DRAM densities. While auto-refresh suffers a small latency penalty at low DRAM density (8.34% on avg. at 1Gb density), this increases rapidly with density up to an unacceptable level (e.g., the average memory latency of auto-refresh increases by 455% relative to CRS at 64Gb). CRS avoids any latency penalty because memory requests of a real-time task do not (and cannot) interfere with any DRAM refresh since memory assigned to a server is not refreshed while the server executes. Hence, real-time tasks do not suffer from refresh overhead/blocking.

Observation 1: CRS avoids the memory latency penalty of auto-refresh, which increases with memory density under auto-refresh.

With growing density, the refresh delay increases not only due to longer execution time of refresh commands, but also because the probability of interference with refreshes increases. Fig. 5 illustrates this by plotting the number of memory references suffering from interference (y-axis) by task over the same x-axis as before. Memory requests of a task suffer from more refresh interference with growing density since longer refresh durations imply a higher probability of blocking specific memory accesses.

Observation 2: Auto-refresh results in high variability of memory access latency depending on memory access patterns and DRAM density while CRS hides this variability.

A. System Schedulability

Fig. 6 depicts the execution time of auto-refresh normalized to CRS (y-axis) over the same x-axis as before. We observe that execution times of tasks under auto-refresh exceed those

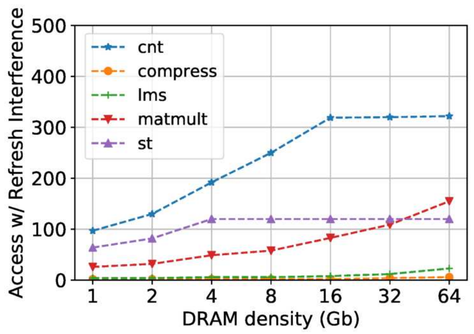

Fig. 5. Number of Memory Accesses with Refresh Interference

under CRS since the latter avoids refresh blocking. Execution times increase rapidly with DRAM density under auto-refresh. E.g., refreshes increase execution times by 3.16% for 8Gb and by 22% at 64Gb for auto-refresh. The execution time of each application under CRS remains constant irrespective of DRAM density. Since there is no refresh blocking anymore, changing density has no effect on performance.

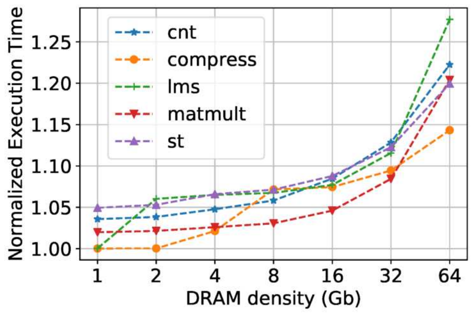

Fig. 6. Execution Time of Auto-Refresh Normalized to CRS

Fig. 7 depicts the overall system utilization factor (y-axis starting at 0.93) over DRAM densities (x-axis) of this real-time task set under different refresh methods. A lower utilization factor indicates better performance since the real-time system has more slack to guarantee schedulability. Auto-refresh experiences a higher utilization factor than CRS due to the longer execution times of tasks, which increases with density to the point where deadlines are missed (above factor 1.0) at 16, 32, and 64Gb.

In contrast, the utilization of CRS is lower and remains constant irrespective of densities. In fact, it is within 0.01% of the lower bound (non-refresh), i.e., scheduling overheads (e.g., due to preemption) are extremely low. Overall, CRS is superior because it co-schedules memory accesses and refreshes such that refresh interference is avoided.

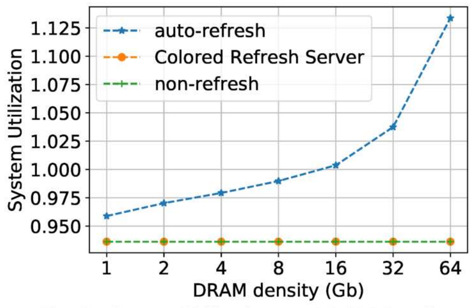

Fig. 7. System Utilization vs. DRAM Density

Observation 3: Compared to auto-refresh, CRS reduces the execution time of tasks and enhances system utilization by hiding refresh overheads, which increases predictability while preserving real-time schedulability. Furthermore, the performance of CRS remains stable and predictable irrespective of DRAM density while auto-refresh experiences increased overheads as density grows.

VII. Related Work

Contemporary DRAM specifications indicate increasing refresh latencies [1], [2], which prompted researcher to search for solutions. Recent works [12], [21], [22], [23], [6], [24] analyze DRAM refresh and quantify its penalty. The refresh overhead for DDR3+4 DRAM with high densities is discussed by Mukundan et al. [5]. While some focus on hardware to reduce the impact of DRAM refreshes [25], [26], [27], [28], others assess the viability of software solutions since hardware solutions take a long time before they become widely adopted.

Liu et al. [4] propose Retention-Aware Intelligent DRAM Refresh (RAIDR), which reduces refresh overhead by using knowledge of cell retention times. RAPID [29] is a similar approach, where pages are sorted by their retention time and then allocated in this order to select pages with longer retention time first. Compared to CRS, these techniques reduce but cannot hide refresh blocking.

Smart Refresh [30] identifies and skips unnecessary refreshes by maintaining a refresh-needed counter. Liu et al. proposed Flikker [31], a selective DRAM refresh that uses a reference bit per row to record and determine if this row needs to be refreshed. Our CRS is agnostic of data access patterns, and it does not require extra die space while its time overhead is very small. Bhati et al. [32] propose a new DRAM refresh architecture that combines refresh reduction techniques with the default auto-refresh. Unnecessary refreshes can be skipped, while ensuring that required refreshes are serviced. However, this approach does not hide refresh overhead completely, and it suffers from increased refresh latency for larger DRAM density/sizes.

Elastic Refresh [7] uses predictive mechanisms to decrease the probability of a memory access interfering with a refresh. Refresh commands are queued and scheduled when a DRAM rank is idle. In contrast, our CRS hides refresh delays for regular memory accesses under load. Chang et al. [33] make hardware changes to the refresh scheduler inside the memory controller/DRAM subarrays. Kotra et al. [34] 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 server paradigm, including the assessment of overheads (lock/unlock) and the composition of tasks. 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). Other DRAM technology, e.g., RLDRAM [35], makes memory references more predictable but is subject to the same refresh blocking, i.e., CRS is directly applicable to them as well. Bhat et al. [3] make DRAM refresh more predictable. Instead of hardware auto-refresh, a software-initiated burst refresh is issued at the beginning of every DRAM retention period. But the memory remains unavailable during the refresh, and any stalls due to memory references at this time increase execution time. Although memory latency is predictable, memory throughput is still lower than CRS due to refresh blocking, i.e., CRS overlays (hides) refresh with computation. Furthermore, a task cannot be scheduled if its period is less than the duration of the burst refresh.

VIII. Conclusion

A novel uniprocessor scheduling server, CRS, is developed that hides DRAM refresh overheads via a software solution for refresh scheduling in real-time systems. Experimental results confirm that CRS increases the predictability of memory latency in real-time systems by eliminating blocking due to DRAM refreshes.

References

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

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

[3] Balasubramanya Bhat and Frank Mueller, "Making DRAM refresh predictable," in *Euromicro Conference on Real-Time Systems*, 2010, pp. 145–154.

[4] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 1–12, 2012.

[5] 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 *International Symposium on Computer Architecture*. ACM, 2013.

[6] Prashant Nair, Chia-Chen Chou, and Moinuddin K Qureshi, "A case for refresh pausing in DRAM memory systems," in *High Performance Computer Architecture*. IEEE, 2013, pp. 627–638.

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

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

[9] Xing Pan, Yasaswini Jyothi Gownivaripalli, and Frank Mueller, "Tintmalloc: Reducing memory access divergence via controller-aware coloring," in *International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 363–372.

[10] 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.

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

[12] 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.

[13] Xing Pan and Frank Mueller, "The colored refresh server for DRAM," Tech. Rep. TR 2019-1, Dept. of Computer Science, North Carolina State University, 2019.

[14] Xiang Feng and Aloysius K Mok, "A model of hierarchical real-time virtual resources," in *Real-Time Systems Symposium*, 2002.

[15] Giuseppe Lipari and Enrico Bini, "Resource partitioning among real-time applications," in *Euromicro Conference on Real-Time Systems*, 2003.

[16] Insik Shin and Insup Lee, "Periodic resource model for compositional real-time guarantees," in *IEEE Real-Time Systems Symposium*, 2003.

[17] 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.

[18] Yonghui Li, Benny Akesson, and Kees Goossens, "Dynamic command scheduling for real-time memory controllers," in *Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 3–14.

[19] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *International Workshop on Worst-Case Execution Time Analysis (WCET),*, 2010.

[20] Y Li, B Akesson, and K Goossens, "Rtmemcontroller: Open-source wcet and acet analysis tool for real-time memory controllers," http://www.es.ele.tue.nl/rtmemcontroller/, 2014.

[21] 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.

[22] 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 *International Symposium on Computer Architecture*. ACM, 2013.

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

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

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

[26] 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 *High Performance Computer Architecture*. IEEE, 2014, pp. 356–367.

[27] Joohee Kim and Marios C Papaefthymiou, "Dynamic memory design for low data-retention power," *Lecture notes in computer science*, 2000.

[28] 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.

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

[30] 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 *International Symposium on Microarchitecture*. IEEE, 2007, pp. 134–145.

[31] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," *Architectural Support for Programming Languages and Operating Systems*, pp. 213–224, 2011.

[32] Ishwar Bhati, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob, "Flexible auto-refresh: enabling scalable and energy-efficient DRAM refresh reductions," in *International Symposium on Computer Architecture*, 2015.

[33] 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 *International Symposium on High Performance Computer Architecture*, Feb 2014, pp. 356–367.

[34] 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.

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