ABSTRACT

PAN, XING. Providing DRAM Predictability for Real-Time Systems and Beyond. (Under the direction of Rainer Mueller.)

DRAM memory of modern multicores is partitioned into sets of nodes, each with its own memory controller governing multiple banks. Accesses can be served in parallel to controllers and banks, but sharing of either between threads results in contention that increases latency, as do accesses to remote controllers due to the non-uniform memory access (NUMA) design. Above DRAM, a last-level cache (LLC), typically at level 3 (L3), is shared by all cores while L1 and L2 caches tend to be core private. This NUMA design inflicts significant variations in execution time for applications due to variable latencies incurred by remote memory node accesses or contention in LLC and at memory banks/controllers.

DRAM cells must be refreshed periodically to maintain data validity. During a refresh, a successive memory space is blocked, and row buffer misses increase in the wake of refreshes. Such refresh delays not only reduce memory throughput but result in looser bounds on tasks' worst case execution time (WCET) since memory latencies vary significantly if interrupted by refreshes. What is more, with growing DRAM density and size, refresh latencies are rising fast as more memory needs to be refreshed.

Due to the above problems, this work contributes an approach to reduce memory access divergence and to eliminate refresh overhead. First, we propose two novel memory allocators called *CAMC* and *TintMalloc* for real-time and parallel systems, respectively. Controller/Node-Aware Memory Coloring (CAMC) is an allocator inside the Linux kernel for the entire address space to reduce access conflicts and latencies by isolating tasks from one another. CAMC improves timing predictability and performance over the Linux buddy allocator and prior coloring methods. It provides core isolation with respect to banks and memory controllers for real-time systems. To complement CAMC, we propose TintMalloc to color memory at the LLC, bank, and controller level to ensure accesses are directed only to local memory while reducing contention at the LLC/bank levels in software. After adding one line of code during initialization in each thread, existing applications automatically obtain colored heap space through regular malloc calls.

Second, we contribute *Colored Refresh* to hide DRAM refresh overhead completely for real-time cyclic executives by utilizing TintMalloc. Colored Refresh partitions DRAM memory at rank granularity such that refreshes rotate round-robin from rank to rank. The 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. Next, we extend *Colored Refresh* to the *Colored Refresh Server* (CRS), a scheduling paradigm with two servers, each with a unique color. CRS partitions DRAM into two groups such that each rank subject to refresh

is assigned one of the two colors. Each real-time task is also assigned one of these color and thus associated with a server. The servers are scheduled 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. CRS can remove all refresh overhead for severed real-time scheduling policies.

Experimental results with the Malardalen, SPEC, and Parsec benchmarks show that locality is increased, contention is decreased, and DRAM refresh delay is removed under our approach. With CAMC and TintMalloc, memory system performance increases for multicore NUMA platforms, and overall SPMD execution becomes more balanced at barriers compared to default memory allocation under Linux as well as prior coloring approaches. In addition, experimental results show that, in contrast to standard auto-refresh, both Colored Refresh and CRS enhance the memory throughput of real-time systems and make a task's WCET more predictable as DRAM density increases.

Providing DRAM Predictability for Real-Time Systems and Beyond

by
Xing Pan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Guoliang Jin	Hung-Wei Tseng
Huiyang Zhou	Rainer Mueller Chair of Advisory Committee

BIOGRAPHY

Xing Pan was born in Tianjin, China. He received his Bachelor's and Master's degree in Electronic Engineering from Nankai University. He then attended Computer Science Department in NC State University from 2013 to 2018 for Ph.D study.

ACKNOWLEDGEMENTS

First and foremost, I am very grateful to my advisor, Dr. Frank Mueller, for his continuous support to my study. When I began my Ph.D. study, my computer science background was not strong and my English is extremely bad. Without Dr. Mueller's consistent guidance and patience in the past five years, I definitely could not have reached this point.

I am very thankful to my committee members, Dr. Huiyang Zhou, Dr. Guoliang Jin, and Dr. Hung-Wei Tseng, for taking their valuable time to serve on my thesis committee. I would like to thank my family and all my friends in NC state.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Dynamic random-access memory (DRAM) is a type of random access semiconductor memory that stores each bit of data in a capacitor within an integrated circuit. Due to the simplistic structure of DRAM cell, DRAM can reach very high densities, with a very low price per bit. Today, DRAM is widely used in modern computer systems, including real-time systems and parallel systems. In this section, it is shown that DRAM accesses for NUMA architectures and DRAM refresh operations can result in variable and unnecessarily high memory latency. Next, we discuss the challenges for real-time and parallel systems due to the unpredictable memory access latency. At last, we propose our approach that provides DRAM predictability and increase memory performance for real-time systems and beyond.

1.1 NUMA Memory Architectures

Modern NUMA multicore CPUs partition sets of cores into a “node” with a local memory controller, where multiple nodes comprise a chip (socket). Memory accesses may be resolved locally (within the node) or via the network-on-chip (NoC) interconnect (from a remote node and its memory). Each core has a local and multiple remote memory nodes. The memory of a node consists of multi-level resources called channel, rank, and bank. The controller further provides access to different banks in parallel to increase memory throughput. Furthermore, in a NUMA architecture, L1 and L2 caches are often core private while the L3 cache, the last level cache (LLC), is shared among cores.

Fig. 1.1 depicts two sockets of such multicore chips. Even within each socket, core-local DRAM accesses (via the local memory controller), e.g., from core 0 via controller 0, have lower latency than remote accesses to other controllers on the socket, e.g., from core 0 to controller 1, as they traverse over the fast on-chip interconnect (Hypertransport/Quickpath for AMD/Intel). References to other sockets result in even longer latencies for both remote LLC (core 0 to the LLC of socket 1) and yet longer latencies for remote controllers (core 0 to controllers 3 or 4) as they transverse the off-chip interconnect (typically narrower, lower bandwidth Hypertransport/Quickpath lanes).

Memory references thus are non-uniform in access latency due to increasingly expensive access penalties for data obtained from LLC and DRAM.

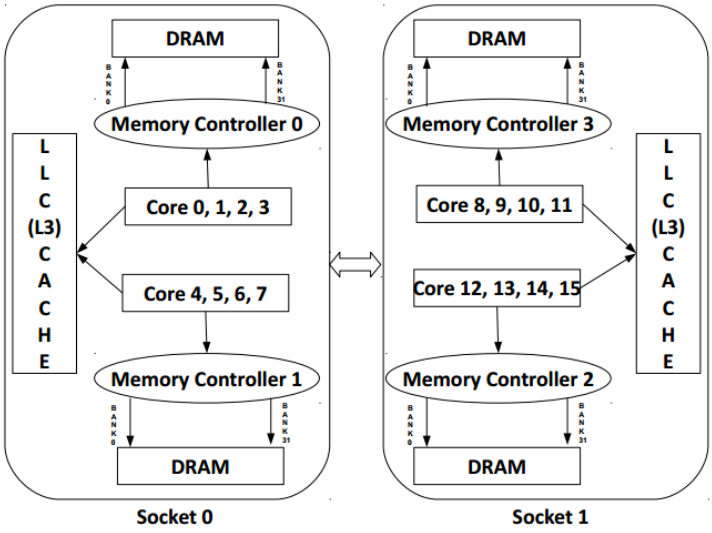

Figure 1.1 Architecture of memory and cache on AMD Opteron

1.1.1 On the Predictability Challenge for Real-time Systems

Real-Time systems are computing systems designed to adhere to not just the functional correctness but also temporal correctness [Liu]. This means that Real-time programs must guarantee a response within specified time constraints, referred to as a "deadline". A real-time system is temporally correct if all the timing requirements of real-time tasks can be met, i.e., no deadline is missed regardless of system load. Systems used for mission critical applications tend to be real-time control systems, such as automobiles, aerial vehicles, nuclear systems, etc. A delayed response in such systems could impact the quality of service or even result in catastrophic consequences. For example in avionics, flight control software must execute within a fixed time interval in order to accurately control the aircraft. Automotive electronics are subject to tight timing constraints on engine management

and transmission control systems derived from the mechanical systems that they control [DB11]. Depending upon the purpose of the systems and the implications of system correctness, real-time systems can be divided into two categories. Soft real-time systems can tolerate some misses of their temporal requirements, but a missed deadline can lead to a loss in quality of service. In contrast, hard real-time systems must guarantee every temporal requirements of an application to prevent catastrophic consequences. Such applications are considered safety critical. In order to guarantee the time requirements, predictability of the system behavior is the most important concern in real-time systems. Predictability is often achieved by bounding the worst-case execution time (WCET) of real-time tasks to meet their deadlines, as Fig. 1.2 shows.

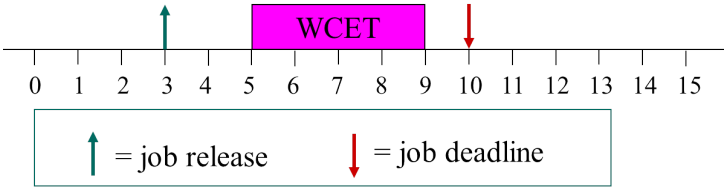

Figure 1.2 Worst-Case Execution Time for Real-Time Tasks

When tasks on different cores access NUMA memory concurrently, performance varies significantly depending on which node data resides and how banks are shared, such variation has two reasons. (1) The latency of accessing a remote memory node is significantly longer than accessing that of a local memory node. Although operating systems generally allocate from the local memory node by default, remote memory will be allocated when local memory space runs out. (2) Even with a single memory node, conflicts between shared-bank accesses result in unpredictable memory access latencies. As a result, system utilization may be low as the execution time of tasks has to be conservatively (over-)estimated in real-time systems. With the fluctuating memory access latency, the timing predictability of real-time systems is reduced, and the temporal requirements cannot be guaranteed anymore. As a result, for real-time systems running on NUMA platforms, system utilization may be low as the execution time of tasks has to be conservatively (over-)estimated.

1.1.2 On the Performance Challenge for Parallel Systems

Parallel computing is an area of compute-intense calculations where the execution of processes is carried out concurrently [AG89]. A parallel computational task typically consists of alternating parallel and serial sections. For a multi-core processor system, the parallel section is often expressed a common subtask runs on different cores operating independently on disjoint subsets of the data. By dividing large problems into smaller ones and combining results afterwards upon completion, the execution time of an application can be reduced while the system performance is enhanced.

For NUMA memory systems, different controllers and banks can be accessed in parallel, but sharing of either, even locally, may result in resource contention. Furthermore, non-local accesses can result in contention on the on-chip interconnect. Contention may also exist at the LLC level, typically due to large working set sizes that result in more data blocks being mapped to the same cache line than the LLC can hold given its associativity.

Application performance will degrade when data references result in frequent contention or suffer remote access penalties. It is thus imperative to try to keep as many references as possible local in order to improve memory performance while utilizing all cores of a processor. In addition, multi-threaded programs often utilize fork-join parallelism with data- or task-parallel execution in parallel sections using POSIX threads [But97] or OpenMP [Cha08]. At the end of such parallel sections, implicit or explicit barriers synchronize all threads. If execution is highly variable across threads in a parallel section, idle time is incurred for early arrivers at barriers in an unbalanced manner (see Fig. 1.3). Memory contention and non-uniform access penalties contribute to the aggregate cost of idle time, this resulting in underutilized processing resources.

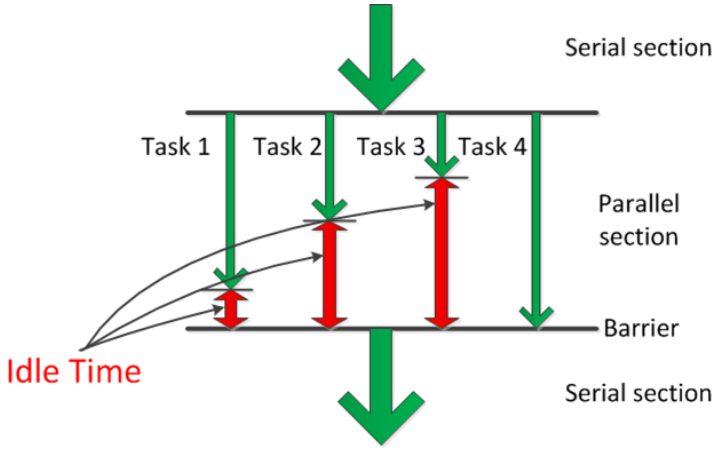

Figure 1.3 Imbalance caused by Idle Time

1.2 DRAM Refresh

Memory refresh is a background activity required during the operation of volatile dynamic random-access memory (DRAM), which is the most widely used memory in most computer systems and embedded systems. A DRAM cell is composed of an access transistor and a capacitor. Data is stored in a DRAM cell as a 1 or 0 (electrically charged/discharged). However, cells slowly leak their charge as time passes. This leakage volatility requires cells to be refreshed, or their data is lost. Memory refresh is the process of periodically reading information from an area of computer memory and

immediately rewriting the read information to the same area without modification. When memory is refreshed, the capacitor of DRAM cell is charged, and the data stored in DRAM is preserved.

To refresh memory, the DRAM controller periodically issues refresh commands, which are sent to DRAM devices via the command bus. This mode is called auto-refresh and recharges all the memory cells within the "retention time" (tRET), which is typically 64ms for commodity DRAMs [JED10]. In this mode, a refresh command is issued per interval, tREFI , for a duration/completion by tRFC. The DDR3 specification [JED10] generally 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(tREFI) is 7.8us (tRET/8192). The refresh completion time (tRFC) is the duration for each refresh command. Auto-refresh is triggered in the background by the DRAM controller while the CPU executes instructions.

1.2.1 Challenge

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. For commodity DRAM, the DRAM controller either schedules an automatic refresh to all ranks simultaneously (simultaneous refresh), or schedules 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 as follows:

A DRAM refresh command closes a previously open row and opens up a new row subject to refresh [2], even though data 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. In addition, as the density and size of DRAM grows, more ranks 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 refresh operations and thus reduces memory throughput.

As discussed above, the refresh problem significantly impacts real-time systems because predictable memory access latencies are imperative to assess task schedulability [WM01]. Furthermore, as DRAM density grows, more rows need to be refreshed within same refresh interval (tREFI), i.e., tRFC increases rapidly. Even with conservative estimates of growth in density for future DRAM technology, tRFC exceeds 1us at 32 Gb DRAM density [Liu12a]. 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.

1.3 Hypothesis

DRAM is widely used as main memory in digital electronics, such as modern computers and graphics cards, where low-cost and high-capacity memory is required. Non-uniform Memory Access (NUMA) architectures are common on server platforms with more than one system bus. They can harness large numbers of processors in a single system. But DRAM systems with NUMA architectures suffer from the following shortages:

(1) The latency of accessing a remote memory node is significantly longer than that of a local memory node.

(2) Even with a single memory node, conflicts between shared-bank accesses result in unpredictable memory access latencies.

(3) Contention may also exist at the last level cache (LLC), typically due to large working set sizes that result in more data blocks being mapped to the same cache line than the LLC can hold given its associativity.

(4) The response time of a DRAM access is impacted by refresh operations, which not only decreases memory throughput, but also makes memory access latency unpredictable.

We attempt to propose an approach to address these shortages in this dissertation. The hypothesis of this dissertation is as follows.

NUMA architecture with DRAM require a novel approach to avoid remote memory accesses, reduce contention among bank/LLC level, and remove the interference due to refresh, in order to increase the predictability of memory access latency and memory performance, which is critical for real-time systems and also benefits multi-core parallel system.

1.4 Contribution

To address these problems, this work contributes a novel approach to reduce memory access divergence and to hide DRAM refresh overhead. The first contribution is Controller-Aware Memory Coloring (CAMC), which colors the entire memory space (heap, stack, code and data segments) to avoid remote memory accesses, and to reduce memory bank contention. The second contribution is TintMalloc, which not only considers memory locality, but also supports last level cache (LLC) coloring. The third contribution is "Colored Refresh", which hides DRAM refresh overhead entirely in real-time cyclic executives. The forth contribution is the "Colored Refresh Server" to remove task preemptions due to refreshes and to hide DRAM refresh overhead entirely, which can be considered with several real-time scheduling policies.

Controller-Aware Memory Coloring (CAMC): CAMC is a memory allocator that “colors” pages of the memory space with locality affinity for controller- and bank-awareness suitable for mixed criticality, weakly hard, and soft real-time systems on NUMA architectures. In contrast to prior work

on non-NUMA, heap-constrained allocation [Yun14], CAMC colors the entire memory space of heap, globals, stack, and instruction segments via Linux kernel modifications transparent to the application. CAMC follows the philosophy of single-core equivalence [Man15]. It avoids (1) memory accesses to remote nodes and (2) conflicts among banks in an effort to make task execution more predictable via colored partitioning. Another novelty of CAMC is that it does not require *any* code modifications for applications. Tasks are *automatically* assigned one (or more) colors for memory regions disjoint from colors of other tasks in the system. Invocation of a command line utility prior to real-time task creation suffices to activate coloring in the kernel. The utility internally issues a single `mmap()` system call with custom parameters for color activation. We have modified the operating system (OS) kernel so that each task has its own memory policy. Heap, stack, static, and instruction (so-called text/code) segment allocations return pages adhering to this policy upon task creation as well as for expansions of stack or heap segments during execution due to heap allocations or deeply nested calls. In Chapter 2, we compare CAMC with standard buddy allocation and previous coloring techniques. We assess the performance of CAMC for codes from the NAS and Parsec benchmarks on a standard x86 platform, with and without real-time task sets. Results indicate (1) a lower memory latency for local controllers than remote ones; (2) monotonically increasing alternating stride patterns result in worse performance than prior access patterns used to trigger "bad" behavior; (3) CAMC increases the predictability of memory latencies; (4) it avoids inter-task conflicts; and (5) it is the only policy to provide single core equivalence when one core per memory controller is used. Thus, this work goes another step beyond prior work in improving real-time schedulability.

TintMalloc: TintMalloc is a heap allocator that "colors" memory pages with (1) locality affinity for controller- , (2) bank- and (3) LLC-awareness suitable for high performance computing on NUMA architectures. With TintMalloc, programmers can select one (or more) colors to choose memory controller, bank and LLC regions disjoint from those of other tasks. Our coloring allocator establishes memory and LLC isolation between tasks, so that each task only accesses its local memory controller, private memory banks and LLC. Due to this isolation, remote access penalties are avoided (except for shared data regions, which are typically smaller) and interference is reduced. The approach can keep the runtime of tasks in parallel sections more balanced, which reduces idle time and increases core utilization. TintMalloc only requires one line of code to be added to application initialization. An extension of the system call "mmap" is designed to indicate a thread's color. We have modified the OS kernel so that each task has its own dynamic allocation policy, which triggers either the legacy default allocation policy or TintMalloc's policy for mmap() system calls. Heap allocations by a task return pages adhering to the respective policy. This allows us to limit program modifications to just a single-line of code to select colors during initialization. As extensive experiments shown in Chapter 3, we observe that the latency of local memory controller accesses is much lower than that of remote memory controller accesses, and TintMalloc can (1) avoid memory accesses to remote

nodes, and (2) reduce conflicts among banks and thread interference in LLC. As a result, TintMalloc reduces the runtime of parallel programs by decreasing the idle time of parallel tasks, which makes them more balanced. Compared with CAMC, TintMalloc can also reduce LLC contention by cache coloring.

Colored Refresh: Colored Refresh is an approach that hides DRAM refresh overhead entirely. We exploit colored memory allocation (TintMalloc) to partition the entire memory space such that each real-time task receives different ranks. By strategically co-scheduling refreshes and competing memory accesses, memory reads/writes do not suffer from refresh interference. As a result, Colored Refresh reduces memory access latency and increases memory throughput, which tends to make real-time systems more predictable, particularly for large DRAM sizes. 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. In Chapter 4, (1) we analyze DRAM refresh of modern memory systems in detail, especially highlighting the impact of refresh delay under varying DRAM densities/sizes for real-time systems with stringent timing constraints. As observed, it is hard to predict an application's refresh overhead with auto-refresh. With growing DRAM density, the losses in DRAM throughput and performance caused by refreshes quickly become unacceptable for real-time systems. (2) A software solution, Colored Refresh, is contributed, which refreshes DRAM based on memory space coloring. Refresh overhead is entirely hidden since a memory rank is either being accessed or being refreshed, but not both. (3) We evaluate Colored Refresh performance with Malardalen benchmarks to confirm that regular memory accesses never suffer from refresh interference, i.e., the refreshes are completely hidden in a safe manner. Consequently, the refresh delays are hidden, DRAM access latencies are reduced, and application execution time becomes more predictable, even when DRAM density increases.

Colored Refresh Server: Colored Refresh Server (CRS) is an approach to remove task preemptions due to refreshes, hide the DRAM refresh overhead entirely, and to make real-time systems more predictable. Compared with Colored Refresh, CRS can be implemented for several real-time scheduling policies. CRS exploits colored memory allocation to partition the entire memory space into two colors corresponding to two server tasks (simply called servers from here on). Each real-time task is assigned one color and associated with the corresponding server, where the two servers have different static priorities. DRAM refresh operations are triggered by two tasks, each of which issues refresh commands to the memory of its corresponding server for a subset of a colors (DRAM ranks) using a burst refresh pattern. More significantly, by appropriately grouping real-time tasks into different servers, 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 CRS is small and remains constant irrespective of DRAM density/size. In contrast, auto-refreshed overhead keeps growing as DRAM density increases.

In Chapter 5, we develop the Colored Refresh Server (CRS), which refreshes DRAM based on memory space coloring and schedules tasks according to the server policy. Refresh overhead is almost entirely hidden since a memory space is either being accessed or refreshed, but never both at the same time. Experiments with the Malardalen benchmarks confirm that both refresh delays are hidden and application execution times become more predictable. Furthermore, compared to previous work [BM10], CRS not only removes refresh overhead, but also feasibly schedules short tasks (with periods less than the execution time of a burst refresh) by refactoring them as “copy tasks”.

1.5 Organization

This document is organized as follows. Chapter 2 presents Controller-Aware Memory Coloring (CAMC), which “colors” pages of the memory space with locality affinity for controller- and bank-awareness. Chapter 3 describes TintMalloc, which "colors" both memory and LLC. Chapter 4 describes Colored Refresh, which hides DRAM refresh overhead by memory coloring. Chapter 5 presents the Colored Refresh Server, which removes DRAM refresh overhead for several real-time scheduling policies. Chapter 6 presents the conclusion and discusses future work.

CHAPTER

2

CONTROLLER-AWARE MEMORY COLORING FOR MULTICORE REAL-TIME SYSTEMS

2.1 Introduction

Modern NUMA multicore CPUs partition sets of cores into a “node” with a local memory controller, where multiple nodes comprise a chip (socket). Memory accesses may be resolved locally (within the node) or via the network-on-chip (NoC) interconnect (from a remote node and its memory). Each core has a local and multiple remote *memory nodes*. A so-called *memory node* consists of multi-level resources called channel, rank, and bank. The banks are accessed in parallel to increase memory throughput. When tasks on different cores access memory concurrently, performance varies significantly depending on which node data resides and how banks are shared for two reasons. (1) The latency of accessing a remote memory node is significantly longer than that of a local memory node. Although operating systems generally allocate from the local memory node by default, remote memory will be allocated when local memory space runs out. (2) Even with a single memory node, conflicts between shared-bank accesses result in unpredictable memory access latencies. As a result, system utilization may be low as the execution time of tasks has to be conservatively (over-)estimated in real-time systems.

The idea of making main memory accesses more predictable is subject of recent research. Palloc [Yun14] exploits bank coloring on DRAM to allocate memory to specific DRAM banks. Kim et al. [Kim14] propose an approach for bounding memory interference and use software DRAM bank partitioning to reduce memory interference. Other approaches ensure that cores can exclusively access their private DRAM banks by hardware design [Wu13; She09]. Unfortunately, none of these approaches universally solve the problem of making memory accesses time predictable. Some of them require hardware modifications while others do not consider NUMA as a source of unpredictable behavior. Furthermore, programmers need carefully assign colors to each task and manually set coloring policy for real-time task set.

In operating systems, standard buddy allocation provides a "node local" memory policy, which requests allocations from the memory node local to the core the code executes on. Besides, The libnuma library offers a simple API to NUMA policies under Linux with several policies: page interleaving, preferred node allocation, local allocation, and allocation only on specific nodes. However, neither buddy allocation with local node policy nor libnuma library is bank aware. Furthermore, the libnuma library is restricted to heap memory placement at the controller level, and it requires explicit source code modifications to make libnuma calls.

This work contributes Controller-Aware Memory Coloring (CAMC), a memory allocator that automatically assigns appropriate memory colors to each task while combining controller- and bank-aware coloring for real-time systems on NUMA architectures. An implementation of CAMC on an AMD platform and its performance evaluation with real-time tasks provides novel insights on opportunities and limitations of NUMA architectures for time-critical systems. Memory access latencies are measured, the impact of NUMA on real-time execution is discussed, and the performance of DRAM partitioning is explored. To the best of our knowledge, this is the first work to comprehensively evaluate memory coloring performance for real-time NUMA systems.

Summary of Contributions: (1) CAMC colors the entire memory space transparent to the application by considering memory node and bank locality together — in contrast to prior work for non-NUMA allocations [Yun14], or "local node" policy in buddy allocation without bank awareness. Tasks are *automatically* assigned to one (or more) colors for memory regions disjoint from colors of other tasks in the system. CAMC follows the philosophy of single core equivalence [Man15]. It avoids (i) memory accesses to remote nodes and (ii) conflicts among banks in an effort to make task execution more predictable via colored partitioning. We modified the Linux kernel so that each task has its own memory policy. Heap, stack, static, and instruction (text/code) segment allocations return memory frames adhering to this policy upon task creation as well as for expansions of stack or heap segments dynamically for heap allocations or deeply nested calls.
(2) We compare CAMC with Linux' standard buddy allocator with "local node" policy and previous coloring techniques. We assess the performance of CAMC for Parsec codes on a standard x86 platform, with and without real-time task sets.

(3) Experiments quantify the non uniform latency between nodes and indicate that (i) monotonically increasing alternating stride patterns result in worse performance than prior access patterns believed to trigger the "worst" behavior; (ii) CAMC increases the predictability of memory latencies; and (iii) CAMC avoids inter-task conflicts. By comparison, CAMC is the only policy to provide single core equivalence when the number of concurrent real-time tasks is less than the number of memory controllers. By coloring real-time tasks and non-realtime tasks disjointly (with mappings to different memory controllers), real-time tasks increase their level of isolation from each other following the single core equivalence paradigm, which is is essential for improving the schedulability of real-time task sets and facilitate compositional analysis based on single-task analyses.
(4) An algorithm for the mapping of physical address bits is described for AMD processors. Its principles can be applied universally to any documented mapping.
(5) Instead of manual configuration by programmer, CAMC automatically assigns memory colors to tasks based on global utilization of memory colors. CAMC does not require *any* code modifications for applications. Invocation of a command line utility prior to real-time task creation suffices to activate coloring in the kernel. The utility issues a single `mmap()` system call with custom parameters for coloring.

2.2 Background

DRAM Organization: DRAM is organized as a group of memory controllers/nodes (Fig. 3.1), each associated with a set of cores (e.g., four cores per controller). Each controller governs multilevel resources, namely channel, rank, and bank. Each rank consists of multiple banks, where different banks can be accessed in parallel. Multiple channels further provide interleaving of memory accesses to improve average throughput. Each bank has a storage array of rows and columns plus a row buffer. When the first memory request to a row element is issued, a row of the array with the respective data is loaded into the row buffer before it is relayed to the processor/caches. Next, to serve this memory request, the requested bytes of the data are returned using the column ID. Repeated/adjacent references to this data in this row result in "memory bank hits" — until the data is evicted from the row buffer by other references, after which a "memory bank miss" would be incurred again. The access latency for a bank hit is much lower than for a bank miss. If multiple tasks access the same bank, they contend for the row buffer. Data loaded by one task may be evicted by other tasks, i.e., the memory access time and bank miss ratios increase as access latencies fluctuate on bank contention.

Memory Controller: The memory controller is a mediator between the last-level cache (LLC) of a processor and the DRAM devices. It translates read/write memory requests into corresponding DRAM commands and schedules the commands while satisfying the timing constraints of DRAM banks and buses. When multiple memory controllers exist, references experience the shortest memory latency when the accessed memory is directly attached to the local controller (node). A memory

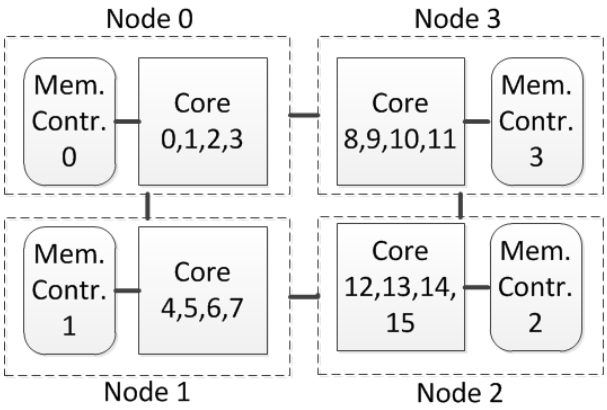

Figure 2.1 16 Cores, 4 Memory Controllers/Nodes

access from one node to memory of another incurs additional cycles of load penalty compared to local memory as it requires the traversal of the memory interconnect between cores. Overall, it is beneficial to avoid remote memory accesses not only for performance but also predictability (uniform latencies), and proper placement of data can increase the overall memory bandwidth which decreases its latency.

2.3 Controller-Aware Memory Coloring (CAMC)

In a NUMA system, a running task is subject to varying memory controller (node) access latencies and contention on memory banks. As described in Sec. 5.2, DRAM memory access latency is largely affected by: (1) where data is located, i.e., local vs. remote memory node; (2) how memory banks interleave; and (3) how much of the accesses contend.

In order to completely avoid remote memory node accesses and reduce bank contention, we design Controller-Aware Memory Coloring (CAMC), which is realized inside the Linux kernel (V2.6). It comprehensively considers memory node and bank locality to color the entire main memory space (heap, stack, static, and instruction segments) without requiring hardware or application software modifications. The entire memory space is partitioned into different sets, which we call "colors". Each memory bank receives a different color. CAMC forces an exact mapping for each active virtual page to a physical frame of the CAMC-indicated color. Such a color indicates a unique `bank color (bc)`, which translates a physical address to memory module locations: node, channel, rank, bank, columns, and rows. Based on this partition, CAMC optimizes the physical memory frame selection process to provide a private memory space for each task on their local memory node in order to make memory access latency stable and predictable.

In practice, it is hard to completely avoid remote accesses as tasks run concurrently and may incur complex memory reference patterns, e.g., due to data sharing. But if one were to conservatively assume remote references for *all* memory accesses, bounds on the WCET would be very loose, so that system utilization would be low. In contrast, we assume that only shared reference latencies are

bounded conservatively (to be remote) as CAMC guarantees locality and absence of controller/bank conflicts.

2.3.1 Address Mapping for Page Coloring

CAMC translates the physical address to a DRAM address and maps it onto the physical structure of main memory as described before (node, channel, rank, bank, columns, and rows). Some vendors only release bit-level mapping information under non-disclosure agreements (e.g., Intel — even though some prior work has published mappings for certain Intel processors) while others disclose this information in their architecture manuals (e.g., AMD, ARM). This work is based on the AMD Opteron hardware platform, but its principles apply universally to any documented mapping. On the AMD platform, we query PCI registers (documented in the architecture manual) and determine the bits that translate physical addresses to DRAM locations.

The memory controller/node of a frame is identified by the range of its physical address. Channel and rank ID bits are indicated by the "DRAM Controller Select Low Register" and 'DRAM CS Base Address Registers", respectively. After determining the frame's memory controller, channel, and rank information, we translate the physical address to the DRAM bank address by removing masked bits and normalizing. Next, we identify the bank, row, and column bits based on the "DRAM Bank Address Mapping Register".

Upon boot-up, our coloring mechanism is triggered within the OS. It scans all frames and calculates the color information for memory controller, channel, rank and bank per frame (and corresponding frame). Consider an AMD Opteron 6128 with four memory controllers, two channels per controller, two ranks per channel, and eight banks per rank (128 banks in total). After boot-up and page color initialization, the system groups the entire memory space into 128 colors and records which color a page belongs to in the page table.

2.3.2 CAMC User Interface

After boot-up, the system is ready for per-task CAMC allocation. Instead of manual configuration by the programmer in prior works, the user only needs to trigger memory coloring in CAMC. Subsequently, the coloring policy is applied automatically, i.e., all tasks are assigned appropriate memory colors without a programmer's manual selection. To turn on/off memory coloring in CAMC, we designed a coloring toggle capability, which is triggered via a single `mmap()` system call exploiting a backwards-compatible mmap extension to turn on/off and configure kernel coloring of memory pages per task. The parameters of this coloring toggle call indicate what kind of coloring action and how many colors should be assigned to real-time tasks during initialization (and can be changed by the programmer based on per-task memory requirement, default: 1 color/task).

Our enhanced `mmap()` retains the calling convention of standard `mmap` calls, which allocates

pages by creating new mappings in the virtual address space of the calling task. The (third) "protection" parameter allows the distinction of standard `mmap` vs. coloring `mmap` calls with full backwards compatibility for the former while triggering our kernel extensions for the latter. Specifically, a set bit 30 of the `mmap` third parameter (unused in Linux) triggers coloring; otherwise, calls experience standard (legacy) behavior. For colored `mmap()`, the first parameter indicates the color action (turn it on/off) and the number of colors to assign per real-time task. On the AMD Opteron platform, the color_num has a value range of 0-127. A sample call for coloring is as follows:

```
char * A = (char*) mmap(color_action+color_num,
    length, prot | (1<<30), flag, fd, offset);
```

2.3.3 Memory Policy Configuration

After CAMC is activated, an enhanced `mmap()` call registers (adds) the current user_id to the coloring_user list in the kernel. As there may be many other tasks running in the system, one may quickly run out of colored memory resource if the kernel assigns colored resources to every task. To avoid coloring for non real-time tasks and OS background processes, we further check the execution path of new tasks to determine whether this task should be colored. After CAMC activation, the user_id and execution_path of tasks is checked as they are spawned. If the user_id has been registered and the execution_path matches a user-specified configuration pattern, the OS kernel will configure the memory policy for this task to adhere to the supplied coloring constraints. In addition, a coloring flag, using_color, is set in the task_struct by kernel. Any subsequent memory allocation calls (including heap, stack, static, and instruction segments) will return pages based on memory policy and coloring requirements. Once a coloring memory policy has been established, this task is guaranteed to receive isolated (colored) memory pages in terms of controller locality and bank arbitration. In CAMC, no software/application source code or hardware architecture modifications are needed. The coloring memory policy is configured as depicted in Fig. 2.2.

CAMC maintains a table to record the utilization of global memory colors and each task's coloring allocation. Once a new coloring task is created, CAMC automatically selects one color (default 1, configurable to > 1) from memory regions disjoint from colors of other tasks in the system. If a task needs more memory space, CAMC assigns a new color after this task's pre-allocated colors have been exhausted.

Following the copy-on-write (COW) paradigm of Linux, when a fork system call is issued, the parent process' pages are shared (with read-only permission) between the child and parent processes. The memory space will not be copied for the child process until the child begins to execute. Whenever the child process calls the `do_exec` function, a separate copy of that particular page is made (actually, on the first write to such a page). The child process will then use the newly copied page and no longer shares the original one, which has now become exclusively owned by the parent. Under

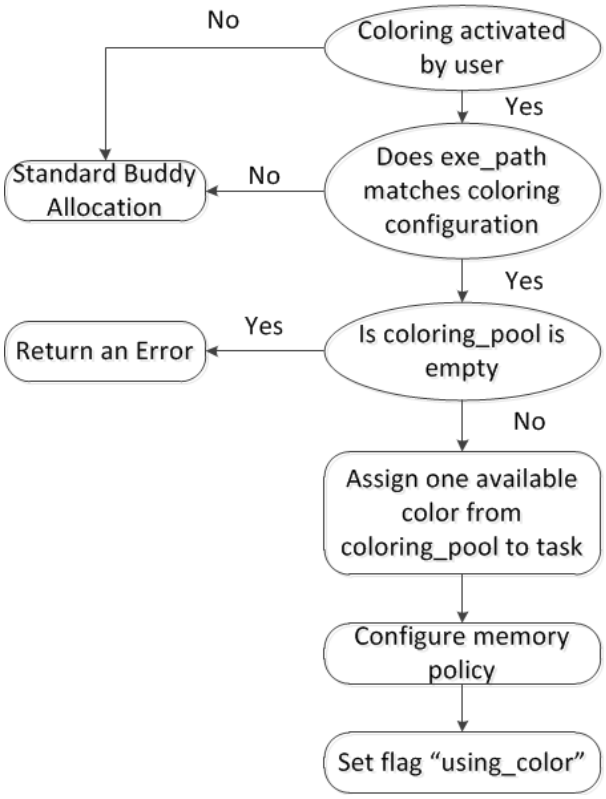

Figure 2.2 Program Flow to configure memory coloring

CAMC, the coloring memory policy is configured in the `do_exec` function so that the entire memory space is colored.

2.3.4 Page Allocation Design

CAMC is implemented by augmenting the Linux buddy allocator. We only handle the $order = 0$ case while higher orders are handled by the original buddy allocator, since user-level memory allocations are eventually performed in the page fault handler at page granularity (4KB, i.e., $order = 0$). CAMC thus handles the common kernel internal allocation requests (getting a page frame).

Furthermore, CAMC supports channel interleaving for multi-channel memory architectures. With channel interleaving, one page is spread evenly across channels at cache line granularity to increase the memory throughputs. The interleaving boundary is related with the size of cache line and determined by memory physical address, (6th bit of physical address on our platform, where a cache line is 64B). When channel interleaving is enabled, the color assigned to each memory bank does not only represent its memory location, but also indicates channel interleaving information, i.e., one color contains multiple memory banks (but a subset of the total number of banks). By assigning this color in CAMC, one task can access those banks at same time though multiple channels, while

isolation and predictability are still guaranteed by memory coloring.

Algorithm 1 Select colored page: find page of given size,color

```
1: INPUT: order
2: OUTPUT: page
3: if order==0 and (current->using_color) then
4:     for i = order ... MAX_ORDER do
5:         Get a memory list ID, MEM_ID, that matches requirements
6:         if Get Successful then
7:             return page from color_list[MEM_ID]
8:         else
9:             if free_list[i] is empty then
10:                continue //try next order
11:            else
12:                create_color_list (i, head page of the buddy set)
13:            end if
14:        end if
15:    end for
16:    return NULL /* no more pages of this color */
17: else
18:    return page from normal_buddy_alloc
19: end if
```

Algorithm 2 Create color list: move page from buddy to colored free_lists

```
1: INPUT: order, page
2: for i = 0 ... 2^(order-1) do
3:     append page to color_list[page_color]
4: end for
```

After configuring the memory policy, we need to determine which page to select at a page fault. This process is shown in Algorithms 1+2. Our approach instructs the kernel to maintain a free list and m color lists, where m denotes the total number of banks in DRAM system. At first, all color lists are empty and all free pages are in the non-colored free list of the buddy allocator. Upon a page fault, the returned page has to match memory coloring requirements if flag `using_color` is set. Orders greater than zero default to the standard buddy allocator while order zero requests traverse the corresponding colored free list to find an available page. E.g., when a task requests a color 0 page, the kernel traverses the `color_list[0]`. If free pages exist here, the kernel removes one such page and hands it to the user. Otherwise, the kernel traverses the general buddy free list and returns the first page with a matching color for this task. Any pages with non-matching colors encountered during the traversal are added to the corresponding color lists by calling the `create_color_list`

function. The call to `create_color_list` causes a buddy (of size = $2^{12+order}$) to be separated into 2^{order} single 4KB pages, which will be added to the respective color lists. When the task frees a memory space, the kernel adds each page to free lists corresponding to their color. In addition, the colors assigned to a task will be returned to the "coloring_pool" when this task calls `do_exit` to terminate upon which memory coloring resources are recycled. Thus, memory space can be configured for a specific memory controller and bank per task.

2.4 Evaluation Framework and Results

2.4.1 Hardware Platform

The experimental platform is a two-socket SMP with AMD Opteron 6128 (Magny Cours) processors with eight cores per socket (16 cores altogether). The 6128 Opteron processor has private 128KB L1 (I+D) caches per core, a private unified 512KB L2 cache, and a 12MB L3 cache shared across eight cores. There are two nodes per socket (4 nodes and eight memory controllers total), and nodes are connected via HyperTransport. The core frequency is between 800MHz-2GHz with a governor that selects 2GHz once a CPU-bound task starts running. There are two channels per memory controller, two ranks per channel, and eight banks per rank, i.e., 128 banks altogether. All banks can be accessed in parallel.

2.4.2 Memory Performance

We first investigate the memory performance impact of CAMC with a synthetic benchmark. The synthetic benchmark represents a performance stress test close to the worst possible case. In the experiment, a large memory space is allocated for varying numbers of threads (tasks) with CAMC. Each thread then performs many writes in this space. We record the execution time of every 524,288 (512*1024) memory writes. Since the only work for each thread is to access main memory, the execution time reflects the memory access latency, i.e., total execution time divided by the 524,288 accesses. We report the average memory access latency over multiple repeated experiments.

To assess the performance of memory controller coloring, we use large strides to defeat hardware prefetching and allocate a large address space to inflict capacity misses in all caches. Accesses follow a pattern where a thread writes to addresses with alternating (positive/negative) offsets increased by a fixed step size of at least cache line size. Consider split (64KB+64KB) I+D L1 caches with 64-byte caches lines. For an integer array, we select a step size of 64 bytes to touch each cache line exactly once. If a thread initially accesses the 256th array element, its next accesses are to the 272th (+16), 240th (-16), 288th (+32), 224th (-32) element etc.

2.4.2.1 Local vs. Remote Memory Controller Latency

We bound one thread to a specific CPU core (to defeat the Linux load balancer that may migrate tasks to different cores) and performed allocations for different memory controllers. Table 2.1 indicates the average memory access latency and standard deviation over a sequence of accesses per memory controller, when the task is running on core 1 (to reduce interference since core 0 often serves interrupts). As a result, memory controller 0 is accessed locally while other controllers are accessed remotely. We observe 60% lower memory access latency and 33% reduced standard deviation for local over remote references, simply due to resolving all references locally with our allocator.

Table 2.1 Access Latency for Core 0 to different Controllers

	average	standard deviation
controller 0	14.41 ns	0.56
controller 1	22.5 ns	0.65
controller 2 & 3	36.37 ns	0.84

Observation 1: Memory latency for local controllers is lower than for remote ones.

Table 2.1 also shows the variations in memory access latencies for a task (on core 1) accessing different remote memory controllers. The latency is the lowest (14.4ns) when accessing local controller 0. It increases to 22.5ns for controller 1 and is the highest (36ns) for controllers 2 and 3. This reflects the required number of hops over the on-chip interconnect discussed earlier.

Observation 2: Memory latency increases across remote controllers with the number of hops over the on-chip interconnect.

2.4.2.2 CAMC vs. Buddy Allocation with Local Node Policy

We compared the cost of CAMC and buddy allocation with "local node" policy. The synthetic benchmark executes 4 threads in parallel with 4 threads bound to cores 0-3, each allocating colored/buddy memory and accessing it as before. Table 2.2 depicts the average latency per access of all 4 threads and the standard deviation for a sequence of 100 experiments. The execution time (38ns) is shorter under CAMC due to a reduction in worst-case latency compared to about 53ns (buddy) on average, a 28.3% reduction. More significantly, the standard deviation of access times under CAMC is much lower than buddy allocation, which indicates that the memory access time becomes more predictable with coloring. Buddy shares memory controller and banks among the threads while CAMC accesses disjoint private banks per thread on the same controller.

Observation 3: Memory access time is reduced and becomes more predictable with CAMC coloring.

Table 2.2 Cost of CMAC Normalized to Buddy

	access latency		norm. allocation cost during:	
	latency	std.dev.	computation	initialization
buddy	53.21 ns	9.33	1	1
CAMC	38.22 ns	1.42	1	1.17

2.4.3 CAMC Overhead

Table 2.2 depicts *allocation overheads* normalized to standard buddy allocation. CAMC imposes no overhead over buddy allocation during regular program execution. But during initialization, CMAC has a 17% overhead during allocations over standard buddy allocation, which is explained as follows. The color lists are empty at program start, and any coloring request results in a traversal of the free_list until a page of the requested color is found. Any pages encountered during the free list traversal are further promoted to their respective index in the color lists. Currently, this initial overhead can be avoided by pre-allocating colored pages during initialization (and optionally freeing them). Alternatively, this overhead could be removed by reversing the design such that all pages initially reside in color lists and are demoted into the free list on demand.

To avoid the initial overhead, one can preallocate (and then free) the maximum number of pages per color that will ever be requested. Subsequent requests then become highly predictable. Typically, a first coloring allocation suffices to amortize the overhead of initialization.

Observation 4: CAMC imposes no overhead over buddy allocation during periodic real-time task execution. Its initialization overhead can be avoided by pre-allocating space for real-time system.

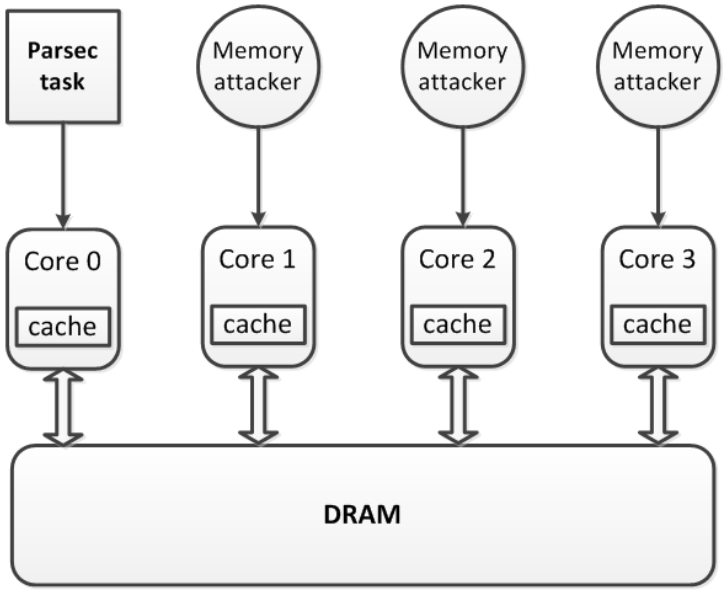

Figure 2.3 Mixed: 1 Parsec code + up to 3 memory attackers

2.4.4 System Performance

We next investigate performance and predictability for the PARSEC benchmark suite featuring multithreaded programs [**bienia08**]. In the experiment, we create a multi-task workload where several "memory attackers" run in the background to assess their interference on memory latency for a foreground task similar to prior work [Kim14; Yun14]. We call these background tasks the "memory attackers", represented by instances of the stream benchmark. E.g., consider 4 tasks in the experiment, one (foreground task) is a Parsec benchmark , and the others (background) are memory attackers (see Fig. 2.3).

2.4.4.1 Performance

We compare the execution time of shared vs. private (isolated) bank allocation (different controllers and different banks). Since CAMC coloring occurs automatically after activation, none of the benchmarks (neither any foreground benchmark nor the memory attackers) need to be modified, and each receives a disjoint colored space accessing only local node memory in private banks without inter-thread sharing. We deploy 3 memory attackers (Stream benchmark) and measure the wall-clock execution time of the foreground task to assess the impact of isolation via coloring. All tasks (memory attackers and the Parsec benchmark) are bound to different CPU cores. We also report results without background attackers for comparison.

We used 3 configurations: (1) In *same_bank*, the Parsec benchmark and all 3 memory attackers are colored so that they access the same bank. This configuration represents the worst case for buddy allocation even with "local node" policy. (2) In *diff_bank*, CAMC forces the foreground benchmark to share one memory controller/node (their local node) with attackers. However, they each are assigned a private bank/color. This is also called bank-level coloring. (3) In *diff_controller*, CAMC ensures that foreground task and attackers allocate pages from their private bank and private local controller for full task isolation.

Fig. 2.4 depicts the experimentally determined WCET for all Parsec benchmarks with background attackers (bars 1-3) and without (bar 4). We observe that the WCET is reduced under controller-aware coloring (private bank) in all experiments. Both *diff_bank* and *diff_controller* obtain better performance than *same_bank*. For bank-level coloring (*diff_bank*), the ferret benchmark gets the largest performance enhancement (28.9%) and the fluidanimate benchmark the smallest one (6.2%). For controller-level coloring (*diff_controller*), the canneal benchmark gets the largest performance enhancement (41.7%) and swaptions the smallest one (11.2%).

All 3 cases are relatively predictable in execution time (small variance), yet *diff_controller* has the tightest range of execution times of these methods, i.e., it is more predictable and the only one that provides single-core equivalence as it matches the last bar, *single run* (no attacker). Differences

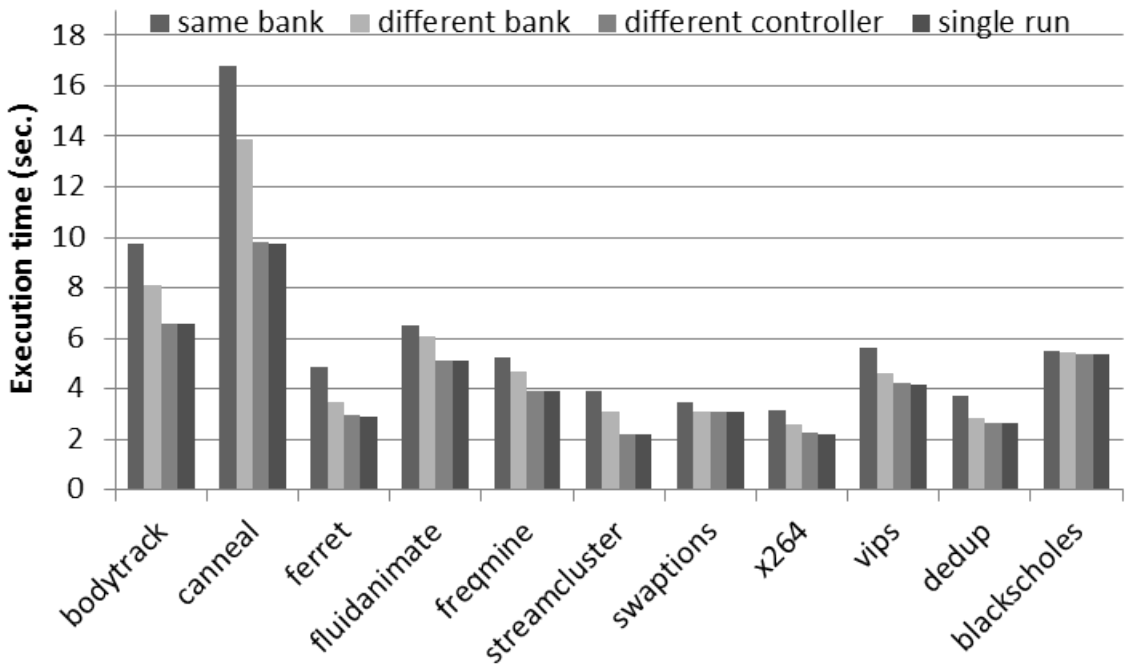

Figure 2.4 Parsec: diff. controller/diff. bank/shared bank/single

between the last two bars of 0.1% for most, 2.73% for X264, and 2.54% for ferret, are due to increased LLC contention for 4 tasks, which would be removed by LCC coloring.

Observation 5: CAMC increases the predictability of memory latencies by avoiding remote accesses and reducing inter-task conflicts. It is the only policy to provide single-core equivalence when one core per memory controller is used.

Not only foreground tasks (from the Parsec suite), background memory attackers (the Stream benchmark) also improve in performance under CAMC. The results indicate that *diff_controller* gets a 40% and *diff_bank* a 14.8% performance enhancement over *same_bank*.

Since *same_bank* represents the worst case for standard buddy allocation, real-time tasks should be scheduled considering the *same_bank* WCET for safety. Under CAMC, the WCET of real-time tasks is much reduced compared to buddy.

2.4.4.2 Multiple Cores

We next executed a Parsec/X264 benchmark with multiple memory attackers (Stream benchmark) in the background on multiple cores. In 4 experiments, we ran Parsec/X264 with 0/3/7/15 memory attackers pinned to different cores. Fig. 2.5 depicts the runtime (left y-axis) of X264 (foreground) and Stream (background) (right y-axis, avg. and min/max as error bars) over the 3 allocation policies (x-axis).

We observe that the performance enhancement by CAMC becomes smaller as the number of background tasks increases. For 16 tasks, node-level coloring finally degrades to bank-level coloring.

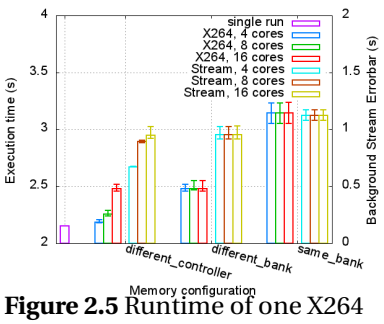

Figure 2.5 Runtime of one X264 and 3/7/15 Stream Tasks

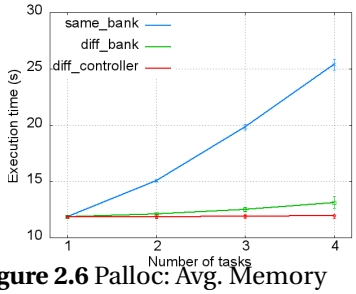

Figure 2.6 Palloc: Avg. Memory Latency for Controller/Bank/no Coloring

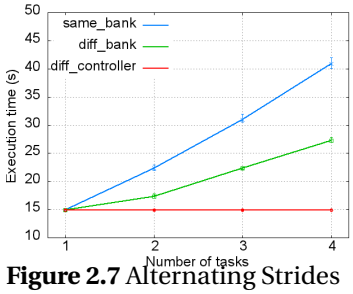

Figure 2.7 Alternating Strides for Controller/Bank/no Coloring

Notice that the predictability of background tasks (stream) also degrades for 16 tasks (cores) with CAMC matching that of the other allocators irrespective of the number of active task. This is due to contention within the shared queue of a memory controller before requests enter bank-specific queues. Even for 16 tasks, our approach still results in superior performance to normal buddy allocation (*same_bank*) where both controller and bank queues are shared by all tasks. However, compared to just one core, only the 4-core case under our policy provides single-core equivalence as this is the only configuration to avoid memory controller queue sharing. Furthermore, the 3 background Stream benchmarks result in better performance under CAMC with increasing variance under contention, which is uniformly higher for the other schemes and also our 16-core case.

Observation 6: CAMC results in superior performance for multi-core executions per controller, where both controller and bank queues are shared across tasks, but can no longer provide single core equivalence.

2.4.5 Real-Time Performance

We evaluated CAMC under rate-monotone scheduling for a task set composed of 2 periodic hard real-time tasks, (1) synthetic (alternating strides) and (2) IS_SER (NAS PB), sharing core 0 (task parameters depicted in Table 2.3) plus three non-real-time tasks (Stream) on cores 1,2,3 (omitted in the table). These cores share the same memory controller. Real-time tasks periodically execute jobs at a rate of 150 and 200ms under an execution time C of 90/60ms for a task utilization U of 0.6/0.3 for tasks 1 and 2, respectively.

Table 2.3 MC Tasks for Buddy Allocator

task$_i$	period	C$_i$	U$_i$
1: Synthetic	150 ms	90 ms	0.6
2: IS_SER	200 ms	60 ms	0.3

When tasks 1 and 2 execute together, CAMC isolates execution from background tasks (Stream) in `diff-controller` mode so that no deadlines are missed. For the CAMC diff-controller mode,

all non-real-time tasks are mapped to different memory controllers via coloring than real-time tasks. Although non-real-time task suffer more remote memory accesses, CAMC guarantees strict memory isolation for real-time tasks. Fig. 2.8 shows the corresponding Gantt chart from one execution of this scenario: Tasks 1 and 2 are released (arrays up) at time 0, synthetic has a shorter period and executes first followed by IS. Here, execution always results in a feasible schedule and all deadlines (arrows down) are met, which is what one would expect using response-time analysis to verify schedulability.

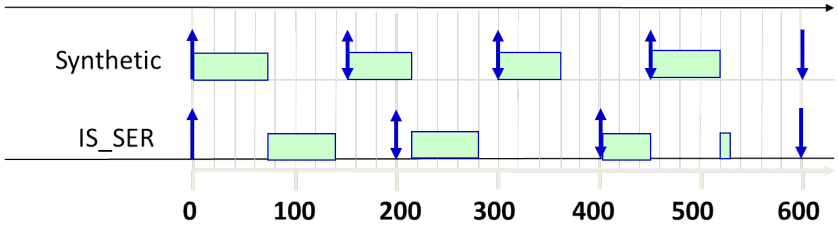

Figure 2.8 Feasible Schedule: diff-controller

In contrast, the `same-bank` configuration does not provide isolation between Tasks 1+2 and the background tasks (Stream), which causes deadlines to be missed. Fig. 2.9 depicts the same task set, but the executions of both synthetic and IS are longer due to Stream's interference. Task 1 executes first for 107ms, then task 2 (IS) runs but is preempted by the 2nd job of higher priority task 1 at 150, which was not enough time to finish, so the deadline of IS is missed at 200. The red box indicates this deadline miss. At the 2nd release of IS at 200, task 1 is still running, and when IS starts at 257, it only runs for 43ms before being preempted by the 3rd job of task 1 (running for just 90ms here due to variations in interference), but then continues at time 390 for another 10ms, which is again not enough to finish by its deadline of 400 (red box). The 3rd job of task 2 finally has enough time (50+37=87ms) to just finish by 594 since it is only preempted by one job of task 1 (running for 107ms). Overall, the interference of background tasks was sufficient to cause deadline misses, which one would not have expected based on calculated response times derived from isolated executions of tasks 1+2, i.e., interference causes schedulability analysis to not be compositional anymore with respect to single task executions. This also holds for buddy allocation (not depicted due to space limitations) or any other policy that causes interference.

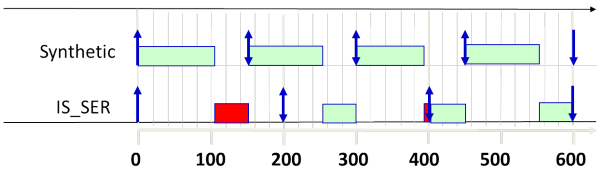

Figure 2.9 Deadline Misses (red) for same-bank

Observation 7: Schedulability analysis for real-time tasks remains compositional under CAMC, yet for other policies with interference, compostionality cannot be guaranteed: Deadlines of hard real-time tasks at higher priority can be missed if any other tasks run on other cores (even if just in the background).

Tables 2.4+ 2.5 depict the observed execution times (avg. over 100 runs, min./max. and standard deviation) for tasks 1 and 2, respectively, for the same 4 configurations as in previous experiments. Notice that a single task run (without background tasks) results in the smallest standard deviation, followed by `diff-controller` (adding minimal overhead due to LLC contention), and then others with higher interference at the bank/NUMA node level. These execution times also reflect the runtime behavior previously depicted in the Gantt chars. Table 2.6 quantifies the deadline miss rates for `same-bank` and `diff-bank` while the other policies always meet deadlines.

Table 2.4 Task 1: Synthetic Exec. Time

	SameBank	DiffBank	DiffContr.	SingleRun
avg.	90.6 ms	78.5 ms	62.5 ms	60.7 ms
max	107.1 ms	89.3 ms	75.8 ms	61.2 ms
min	80.4 ms	68.3 ms	61.5 ms	60 ms
std.dev.	4.88	4.33	2.46	0.44

Table 2.5 Task 2: IS_SER Exec. Time

	SameBank	DiffBank	DiffContr.	SingleRun
avg.	74.6 ms	67 ms	56.7 ms	54.3 ms
max	87.8 ms	74.4 ms	59.8 ms	56 ms
min	64.3 ms	58.8 ms	55.4 ms	53.7 ms
std.dev.	5.28	4.2	0.83	0.41

Table 2.6 Deadline Miss Rates

	SameBank	DiffBank	DiffContr.	SingleRun
deadline misses	82%	23%	0	0

2.4.6 Latency Comparison with Prior Work

We compared the performance of our approach with Palloc [Yun14], a DRAM bank-aware memory allocator that provides memory bank isolation on multicore platforms, but not memory controller locality as it does not support NUMA platforms. We utilize Palloc's latency benchmark [Yun14; Yun13],

which iterates through a randomly shuffled linked list whose size is twice that of the last-level cache (LLC) size.

We run one instance of the latency benchmark on core 0 (the "foreground" load) and co-run up to 3 latency benchmark instances in the background (cores 1-3). The actual number of background tasks varies (0-3), just as in prior work [Yun14]. We run experiments for the 3 memory settings of *same_bank*, *diff_bank*, and *diff_controller* for allocations of pages from different memory banks of disjoint memory nodes, where the latter utilizes a different controller per task (banks 0, 32, 64, 96 on the Opteron platform).

Fig. 2.6 shows the execution time (y-axis) of the latency benchmark over all memory accesses of the foreground task (on core 0) for varying numbers of tasks (x-axis), i.e., the aggregate number of background tasks plus one foreground task. The (very small) error bars show the range of execution times of background latency tasks. We observe that the execution time more than doubles for *same_bank* from 0 to 3 background tasks. This is due to significant bank-level conflicts as all tasks compete for accesses on the same memory bank. The execution time for *diff_bank* slightly increases by ≈4% from 0 to 3 background tasks. References from each task are isolated from one another as each task accesses a disjoint memory bank, i.e., no inter-task bank conflicts occur. The runtime for *diff_controller* is almost constant (slightly smaller than *diff_bank*) from 2-3 background tasks. *diff_controller* not only reduces bank conflicts but also avoids conflicts in the shared controller queue. Also, error bars are the smallest for *diff_controller*, i.e., CAMC provides higher predictability.

We next compare CAMC with Palloc [Yun14] utilizing our synthetic benchmark (striding back and forth with increasing offsets) under the same setup as for the Palloc latency benchmark. Fig. 2.7 uses the same x/y-axes as before. We observe that the execution time is still constant under *diff_controller* but increases steadily for *same_bank* and at a slope roughly twice as steep as *diff_bank*. This shows that the synthetic benchmark triggers a memory reference pattern that is *worse* than that of the latency benchmark. More significantly, it underlines the importance of controller-aware (and not just bank-aware) coloring. Bank sharing is still subject to conflicts between references that enter the shared controller queue before they are relayed to their bank queues. Only controller-aware coloring provides uniform access latencies in this observed worst case.

In comparison to the Palloc [Yun14] results, CAMC obtains similar performance for bank coloring (*diff_bank*), albeit on a different platform (AMD) than their work (Intel). CAMC goes beyond the capabilities of Palloc by further improving performance (diff controller) and making coloring applicable to NUMA multicores, where address bit selection for coloring is derived in a portable manner from PCI registers.

Observation 8: For single controller (UMA) platforms, CAMC is comparable to Palloc in performance. For multi-controller (NUMA), CAMC outperforms Palloc as the latter lacks NUMA awareness, i.e.. only CAMC provides single-core equivalence.

2.5 Related Work

The performance of multithreaded programs on NUMA multicores system has been studied extensively [Bla10; MV10; Mar10; Lac12; MG13; Yun15; PY16]. Scheduling or page placement has been proposed to solve the data sharing problem in NUMA system [Li93; MG12; Oga09; War15; Hua15]. However, compared with CAMC, these approaches introduce overhead and cannot eliminate the data sharing problem completely.

The basic idea of using DRAM organization information in allocating memory at the OS level is explored in recent work [Pan16; Kim14; Yun14; Liu12b; Awa10; Chi15]. Awasthi et al. [Awa10] examine the benefits of data placement across multiple memory controllers in NUMA systems. They introduce an adaptive first-touch page placement policy and dynamic page-migration mechanisms to reduce DRAM access delays in multiple memory controllers system but do not consider bank effects, nor do they provide task isolation. Pan et al. [Pan16] contribute an allocator that colors heap memory at LLC, bank, and controller level to ensure locality per level and requires modifications to applications. In contrast, CAMC colors the whole memory space (heap, stack, static, and instruction segments) without requiring application changes. Liu et al. [Liu12b] modify the OS memory management subsystem to adopt a page-coloring based bank-level partition mechanism (BPM), which allocates specific DRAM banks to specific cores (threads). Palloc [Yun14] is a DRAM bank-aware memory allocator that provides performance isolation on multicore platforms by reducing conflicts between interleaved banks. Our work differs from Palloc and BPM in that we not only focus on bank isolation but also consider memory controller locality, i.e., we avoid timing unpredictability originating from remote memory node accesses. Our approach extends to multi-memory-controller platforms commonly found in NUMA systems. It colors all memory segments, not just the heap, and requires no code changes in applications. Suzuki et al. [Suz13] combine cache and bank coloring to obtain tight timing predictions. Mancuso et al. [Man15] promote single core equivalence and combine several techniques to address contention at different levels of the memory, such as memory bandwidth (MemGuard), cache and memory bank. Yet, sharing within the memory controller results in varying of execution time depending on the number of cores. In contrast to these, our approach addresses both memory banks and memory controllers and ensures single core equivalence up to as many cores as there are memory controllers.

2.6 Conclusion

This work contributes the design and implementation of CAMC, a controller-aware memory coloring allocator for real-time systems. CAMC comprehensively considers memory node and bank locality to color the *entire* memory space and eliminates accesses to remote memory nodes while reducing bank conflicts. CAMC provides more predictable performance than the standard buddy allocator and outperforms previous work for the studied NUMA x86 platform. Experimental results indicate

that CAMC reduces memory latency, avoids inter-task conflicts, and improves timing predictability of real-time tasks even when attackers are present. Overall, this work is the first to assess the real-time predictability of DRAM partitioning on NUMA architectures.

CHAPTER

3

TINTMALLOC: REDUCING MEMORY ACCESS DIVERGENCE VIA CONTROLLER-AWARE COLORING

3.1 Introduction

Contemporary multicores provide a NUMA memory architecture, where L1 and L2 caches are often core private while the L3 cache, the last level cache (LLC), is shared among cores. Sets of cores further comprise a memory "node", where each node features a local memory (DRAM) controller. The controller further provides access to a different banks. A memory reference then is non-uniform in access latency due to increasingly expensive access penalties for data obtained from L1, L2, LLC, and DRAM.

Fig. 3.1 depicts two sockets of such multicore chips. Even within each socket, core-local DRAM accesses (via the local memory controller), e.g., from core 0 via controller 0, have lower latency than other controllers on the socket, e.g., from core 0 to controller 1 as they traverse over the fast on-chip interconnect (Hypertransport/Quickpath for AMD/Intel). References to other sockets result in even longer latencies for both remote LLC (core 0 to the LLC of socket 1) and yet longer for remote controllers (core 0 to controllers 3 or 4) as they transverse the off-chip interconnect (typically narrower, lower bandwidth Hypertransport/Quickpath lanes).

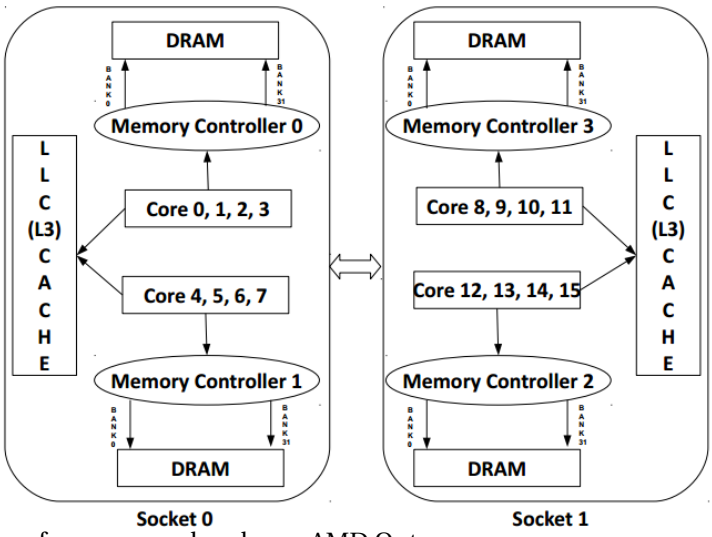

Figure 3.1 Architecture of memory and cache on AMD Opteron

In general different controllers and banks can be accessed in parallel, but sharing of either, even locally, may result in resource contention. Furthermore, non-local access can result in contention on the on-chip interconnect. Contention may also exist of the LLC level, typically due to large working set sizes that result in more data blocks being mapped to the same cache line than the LLC can hold given its associativity.

Application performance will degrade when data references result in frequent contention or suffer remote access penalties. It is thus imperative to try to keep as many references as possible local in order to improve memory performance while utilizing all cores of a processor.

Furthermore, multi-threaded programs often utilize fork-join parallelism with data- or task-parallel execution in parallel sections using POSIX threads or OpenMP. At the end of such parallel sections, implicit barriers synchronize all threads. If execution is highly variable across threads in a parallel section, idle time is incurred for early arrivers at barriers in an unbalanced manner. Memory contention and non-uniform access penalties contribute to the aggregate cost of idle time, i.e., unutilized processing resources.

In this work, we propose TintMalloc, a heap allocator that "colors" memory pages with (1) locality affinity for controller- , (2) bank- and (3) LLC-awareness suitable for high performance computing on NUMA architectures. With TintMalloc, programmers can select one (or more) colors to choose memory controller, bank and LLC regions disjoint from those of other tasks. Our coloring allocator establishes memory and LLC isolation between tasks, so that each task only accesses its local memory controller, private memory banks and LLC. Due to this isolation, remote access penalties are avoided (except for shared data regions which is typically smaller) and interference is reduced. The approach can keep the runtime of tasks in parallel section more balanced, which reduces idle time and increases core utilization.

For example, a task running on core 1 is assigned LLC color 0 and memory bank 0 from its local node (controller) 0. Another task on core 4 is assigned LLC color 1 and memory bank 1 from its local node 1. As a result, every task accesses a local memory controller instead of requiring remote node accesses. A task also has its private memory bank space and private LLC lines. Interference between tasks will be removed. This effectively shortens the execution time and makes execution more balanced for these sample tasks.

TintMalloc only requires one line of code to be added to application initialization. An initial system call indicates a thread's color, which is stored with the task control block inside the operating system (OS). We have modified the OS kernel so that each task has its own dynamic allocation policy, which triggers either the legacy default allocation policy or TintMalloc's policy for mmap() system calls. Heap allocations by a task return pages adhering to the respective policy. This allows us to limit program modifications to just a single-line of code to select colors during initialization.

We performed extensive experiments to assess the performance of TintMalloc for a set of benchmarks from the SPEC2006 and Parsec on a standard AMD Opteron hardware platform. Experimental results with TintMalloc and other heap allocators show the following: (1) The latency of local memory controller accesses is much lower than that of remote memory controller accesses. (2) TintMalloc avoids memory accesses to remote nodes, reduces conflicts among banks and thread interference in LLC. (3) It reduces the runtime of parallel programs. (4) TintMalloc decreases the idle time of parallel tasks and makes them more balanced.

Several approaches have been proposed to address contention between shared resources, e.g., scheduling algorithms based on data reference characteristics [Kim10b; Kim10a; Mur11], cache locality [MG11; CJ06; Per11; Suh04], and page coloring for DRAM partitioning [Yun14; Awa10; Liu12b]. Compared to them, our approach not only partitions memory banks and the LLC but also consider the locality of memory controllers. To our knowledge, it is the first paper to (a) color memory controllers and (b) combine memory controller, bank and LLC coloring together. Overall, TintMalloc effectively lowers contention for shared resources and reduces imbalance at boundaries of parallel sections in programs resulting in improved overall performance and core utilization, much in contrast to other allocators.

3.2 NUMA Memory Architecture

This section provides a brief primer of NUMA DRAM memories for just the aspects relevant this work.

3.2.1 Caches

Most modern CPU architectures have multiple levels of caches organized hierarchically within a single chip. For example, there are two sockets in our AMD Opteron 6128 system and the cache hierarchy in each socket is shown in Fig. 3.2. Each core has its private, local L1 and L2 caches and all cores share the LLC. A miss in L1 initiates an access to L2, and a miss in L2 initiates an access to L3. A miss in LLC initiates an access to memory.

The more cache hits, the faster the system performs. However, cache misses increase when multiple tasks access caches simultaneously since one task's reference may replace data in LLC of another task's prior references.

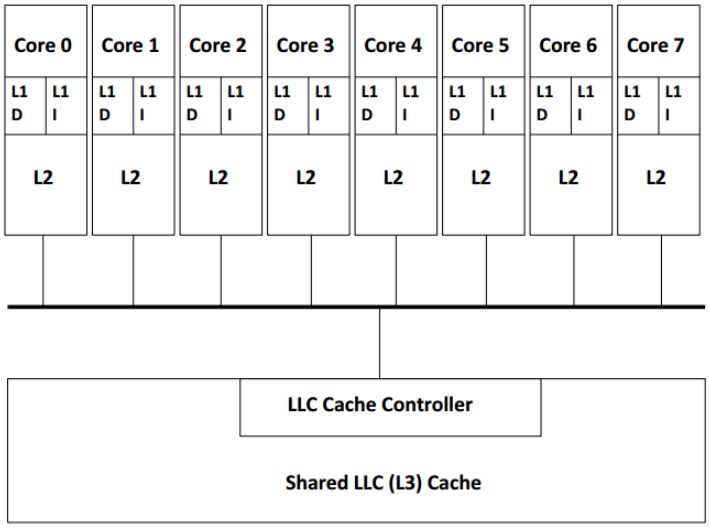

Figure 3.2 Cache Organization (AMD Opteron)

3.2.2 DRAM Memory

Sets of cores comprise a memory node in NUMA systems. Each such node has one local memory controller as depicted in Fig. 3.1. For example, the AMD Opteron system used in our experiments has four memory controllers over two sockets with four cores per controller. The DRAM memory behind a controller is organized into channels, ranks, and banks depicted in Fig. 3.3. On a same controller, accesses to different banks and channels may proceed in parallel, which provides the capability of interleaving memory accesses, thereby improving memory bandwidth/throughput.

A DRAM bank array is organized into rows and columns of individual data cells. Upon access to data, the corresponding row is selected and pulled from the array into the row buffer (incurring a row access strobe penalty). Once in the row buffer adjacent data may be accessed with just a column access strobe penalty, which is smaller than the row activation cost. This, spatial locality in the buffer

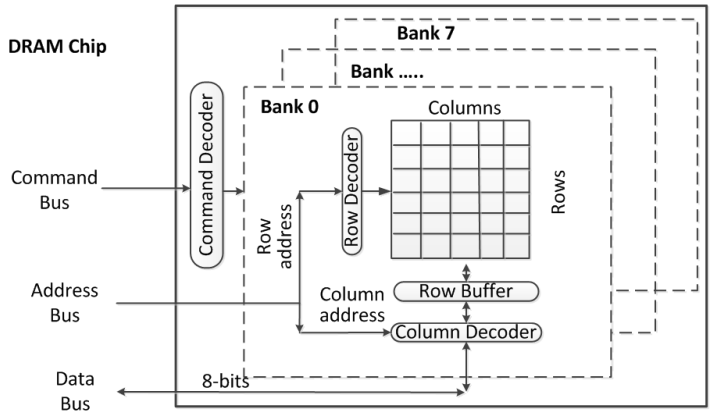

Figure 3.3 DRAM memory controller

can be exploited (while temporal locality is typically taken core of by the upper-level caches). When a row buffer is replaced, an additional precharge penalty is incurred to update the row in the array with any modified data from the row buffer due to memory writes. DRAM cells are also periodically refreshed by the controller so that they do not lose their data, which also flushes the row buffer.

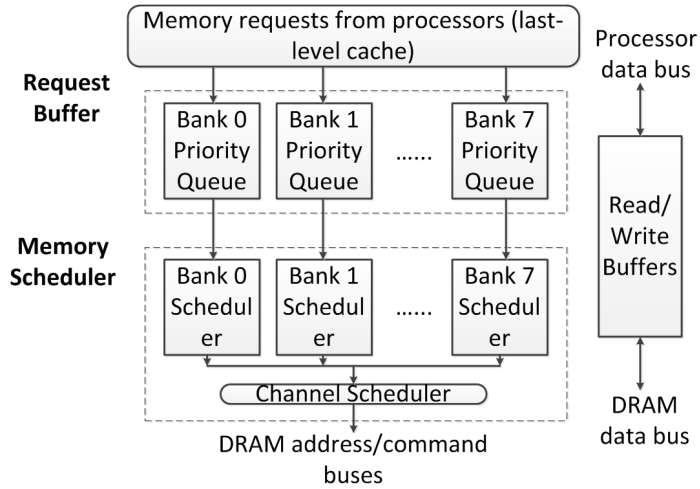

Figure 3.4 Logical Structure of DRAM Controller

When multiple tasks access (write to or read from) the same bank in multi-threaded programs, they contend for the row buffer. Data loaded by one task may be evicted by other tasks, i.e., the latency of memory accesses will increase if multiple threads access the same bank concurrently. Thus, the runtime of two tasks may differ even if their workloads are partitioned equally. In addition, any barrier in a parallel section may cause tasks to wait for the last arriving one, thereby incurring idle time, which becomes more common in NUMA systems due to memory access divergence.

The memory controller governs the activities across banks/channels of local memory arrays. An initial controller queue de-multiplexes requests to the respective parallel sub-components and then

issues DRAM commands for row/column accesses. Its operations are subject to timing constraints of banks and buses, which are typically configured at boot time and limited by manufacturing parameters (see Fig. 3.4). Multi-threaded programs can profit from avoiding resource contention by utilizing memory of the local controller, yet of different banks per thread. Access to the same bank increase latency due to contention, access to remote controllers increase costs due to propagation latencies over the on-chip (or cross-socket) interconnects and potential contention on interconnects and remote controllers/banks. Hence, data placement play a decisive role in ensuring that threads issue local accesses with lower contention and latency penalties instead of remote accesses of higher contention and latencies.

Yet, shared data memory accesses of parallel programs can generally not be resolved with remote accesses for at least some of the threads and the associated contention and latencies. Fortunately, shared memory regions tend to be small in many data- and task-parallel programs. The focus of this work is on the larger portion of data, which is not shared among threads. The objective of the work is to avoid remote accesses by ensuring that memory allocated by a thread is assigned to the local controller in a disjoint bank from other threads and also disjoint LLC cache lines from other threads.

3.3 TintMalloc Design

TintMalloc is a novel memory allocation policy of the OS kernel. It has been implemented as part of the Linux kernel by modifying the mmap() system call code and task control block (TCB) data. TintMalloc colors the physical memory space by selecting memory frames for page allocation requests that comprehensively considers memory controller, bank and LLC locality. No hardware modifications are required, and the general techniques apply to any other architecture with virtual memory support and any other OS with a system call for memory allocation. TintMalloc responds to dynamical allocation requests of threads/tasks by selecting a physical memory frame local to the requesting core. Our assumption is that task-to-core allocations remain static, e.g., by explicitly pinning threads to cores once they have been created. The selection of the frame also ensures that the corresponding memory bank and LLC line are only used by the current thread to avoid conflicts/contention.

3.3.1 Frame Color Selection

Memory requests under the TintMalloc policy cause a lookup of the color(s) assigned to the current thread for this policy. A physical memory frame of 4KB size is subsequently chosen in accordance to the translation of addresses by the memory controller into node, channel, rank, bank, columns and rows, in this order.

The bank color, bc, of a physical page is determined as

$$bc = ((node * NN * NC + channel) * NR + rank) * NB + bank \quad (1)$$

where node (controller), channel, rank, and bank are specific to the physical frame; NN is number of nodes (controllers) available within a system; NC is number of channels within a controller; NR is number of ranks within a channel; and NB is number of banks within a rank.

This bit-level information is not released by some vendors (e.g., Intel, not even under non-disclosure agreements, even though prior work has reverse engineering/obtained data for specific Intel processors) citing that mappings could change and optimizations should not rely on such data. Other manufacturers, e.g., AMD and ARM, disclose this information in their architecture manuals, together with PCI-specific information in platform and BIOS configuration parameters. TintMalloc is highly portable, i.e., and can be easily be adapted to other platforms so long as hardware bit-level physical addressing information is available.

TintMalloc utilizes information on bit-level physical memory mapping. Its design is portable to any platform with known mappings. Our implementation is specific to the AMD Opteron platform, specifically the Opteron 6128 with bit mapping indicated in Fig. 3.5, where we combine fixed mappings with PCI register information obtained at runtime to determine address translation bits for node/controller (DRAM base/limit and limit system address registers indicate bits 16-20), channel (controller select low register), rank and bank (CS base address registers indicating bit 7 for the rank and bits 15, 16, and 18 for the bank) as well as row/column (bank address mapping register) — see AMD's architectural manuals — to select frame colors. The LLC color (set index) is given by bits 12-16.

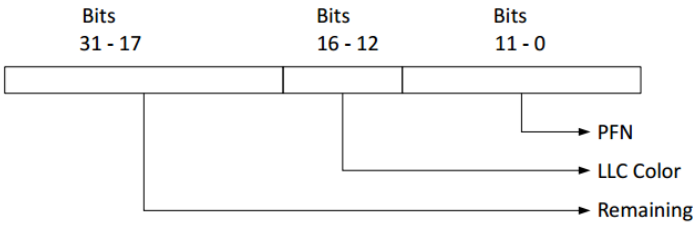

Figure 3.5 Cache Color Address Mapping bits (AMD Opteron)

TintMalloc is activated in the late phase of booting Linux at which time the bit-level information above is derived from PCI registers. For our Opteron 6128 platform, four memory controllers detected with two channels each, two ranks per channel and eight banks per rank. This amounts to $2^7 = 128$ banks altogether across all controllers on our platform suitable for coloring. The LLC also has $2^5 = 32$ colors (over 5 bits).

3.3.2 Coloring Policy Activation

TintMalloc groups frames of pages with valid page table mappings into separate lists for each of these colors to later serve allocation requests after the boot-up phase via the mmap() system call. We modified mmap() so that a zero-sized request is interpreted as the specification of color(s) by the calling thread for subsequent non-zero sized allocations. More specifically, a set bit 30 of the *protection* argument indicates that the first argument should be interpreted as the color and a mode, where the most significant bits specify the mode to indicate if the color should be set or cleared for memory (controller/bank) or LLC (see Fig. 3.6). A set color is recorded in the TCB of the corresponding calling thread/process (handled uniformly as a task in Linux). A thread may even call mmap() multiple times to establish a set of "owned" colors.

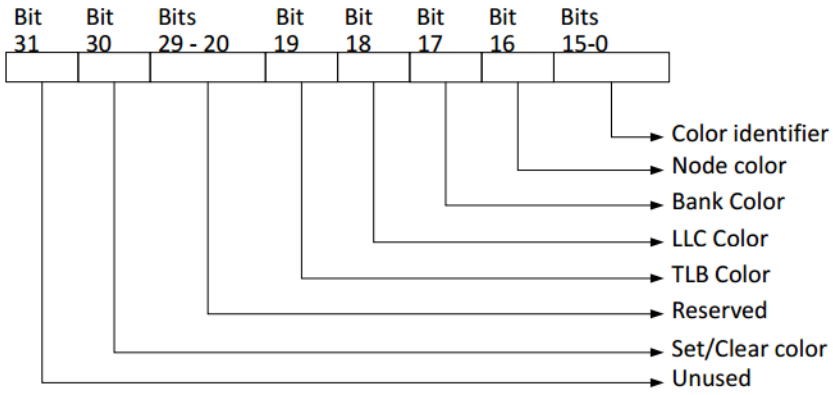

Figure 3.6 mmap() color parameter bits identifiers

For example, the following mmap() call adds the LLC color "c" to the LLC colors of the current thread:

```
int length = 0;
char * A = (char*) mmap(c | SET_LLC_COLOR, length, prot | COLOR_ALLOC, flag, fd, offset);
```

An analogous call with SET_MEM_COLOR would establish controller/bank colors so that any subsequent heap allocation (malloc/calloc call) results in colored page assignments.

TintMalloc divides the entire memory space and LLC into multiple partitions. Each task is guaranteed to only access its local memory node by receiving private (colored) memory and LLC spaces. This ensures controller locality, bank arbitration and cache isolation per task. Once a task activates coloring via mmap(), the OS kernel configures the task's memory policy to adhere to these color constraints. A single mmap() call during application initialization suffices to force any subsequent memory requests of this task to allocate only pages (frames) within the specified color set. If there is no memory left of a given color, mmap() will return an error code indicting that no more pages of this color are available.

Inside the OS kernel, zero-sized mmap() calls result in memory controller/bank and LLC colors to be saved in the `task_struct`, i.e., the TCB of Linux. In addition, two coloring flags `using_bank` and `using_llc`, are set in `task_struct` by kernel. Any subsequent dynamic allocation calls, e.g. malloc(), set aside pages within the coloring constraints by looking up the color set of a task in the TCB. Thus, malloc() calls remain unchanged, i.e., unlike prior work, they do not require source code changes to provide an additional color parameter. Again, *just 1-2 lines of code suffice* to subsequently color a task's entire heap space for controller-/bank- and LLC-aware locality/isolation so as to reduce memory access conflicts and reference latencies.

3.3.3 Heap Policies: Linux Buddy Allocations vs. TintMalloc

Linux currently uses a so-called "buddy allocator" by default, where memory is partitioned into "buddies" of exponentially increasing sizes (by powers of two, where the exponent is referred to as "order"). An allocation request is resolved by returning the matching order ($2^{12+order}$ bytes) or next larger memory region of the respective order-indexed buddy, where any remaining space is added to lower order free lists.

TintMalloc is currently restricted to serve only order-zero requests, i.e., $2^{12+0} = 4$KB, which suffices to handle all ordinary user heap requests in our test programs. Common applications allocate only small heap spaces (< 4KB) at a time, and none that we encountered use so-called huge pages (2MB) allocated from specially mounted memory devices.

TintMalloc maintains a free list and 128*32 color lists simultaneously inside the Linux kernel. Those color lists are defined as a matrix of `color_list[MEM_ID][cache_ID]`. At boot-up, these color lists are empty and all free pages are in the non-colored free list of the buddy allocator. A page fault by a program causes the kernel to invoke the `alloc_pages` routines to find a free page. In the function `alloc_pages`, we disable the "pcp list" and use the function `_rmqueue_smallest` to serve page allocation requests. The colored page selection process is shown in `Algorithm 3` and `Algorithm 4`.

In the algorithm, the kernel reads current task's coloring flag, `using_bank` and `using_llc`. `using_bank` or `using_llc` means the kernel should return a free page according to the memory or LLC coloring constraints. If both are set, the returned page has to match both the memory and the LLC requirements. Orders greater than zero default to the standard buddy allocator while order zero requests traverse the corresponding colored free list to find an available page. E.g., when a task requests a page for `MEM_ID 0` and `cache_ID 0`, the kernel traverses the `color_list[0][0]`. If this color list is not empty, the kernel removes one such page and returns it to the user. Otherwise, the kernel traverses the standard free_list to find an available free page of such a color and calls the function `create_color_list` (see `Algorithm 4`). The call to `create_color_list` causes a buddy (of size = $2^{12+order}$) to be separated into 2^{order} single 4KB pages, which will be added to

Algorithm 3 Colored Page Selection /* find page of certain size and color */

```
1: INPUT: order
2: OUTPUT: page
3: if order==0 and (current->using_bank or current->using_llc) then
4:     for i = order ... MAX_ORDER do
5:         if current->using_bank & current->using_llc then
6:             Get a memory list ID (MEM_ID) and last level cache list ID (LLC_ID) that match requirements
7:             set found_flag
8:         else if current->using_bank then
9:             Get a memory list ID (MEM_ID) that matches requirements
10:            set found_flag
11:        else if current->using_llc then
12:            Get a cache list ID (LLC_ID) that matches requirements
13:            set found_flag
14:        end if
15:        if found_flag then
16:            return page from color_list[MEM_ID][LLC_ID]
17:        else
18:            if free_list[i] is empty then
19:                continue //try next order
20:            else
21:                /* move page from buddy free_list to colored free_lists for next order */
22:                create_color_list (i,head page of the buddy set)
23:            end if
24:        end if
25:    end for
26:    return NULL /* no more page of this color */
27: else
28:    return page from normal_buddy_alloc
29: end if
```

the respective color lists. If available, the kernel will return a free page from the matching color_-list. Conversely, calls to free heap space by the application cause the kernel to add pages to the corresponding colored free lists.

In this manner, memory space can be configured for a specific controller, bank and LLC per application thread/process. Given our design, the overhead of colored allocations is higher for the first heap requests as the kernel traverses the general buddy free list. This higher cost typically impacts only the initialization phase of an application. Once the colored free list has been populated with pages, the overhead becomes constant for a stable working set size, even for dynamic allocations/deallocation assuming the are balanced in size (instead of always growing the utilized heap space).

Algorithm 4 create_color_list /*move page from buddy free_list to colored free_lists*/

```
1: INPUT: order, page
2: for i = 0 ... 2^(order-1) do
3:     page_bank = page[i].bank_color
4:     page_llc = page[i].llc_color
5:     append page to color_list[page_bank][page_llc]
6: end for
```

3.4 Experimental Platform

We perform experiments on a dual socket machine with two AMD Opteron 6128 processors. Each socket has 8 cores per for a total of 16 cores. Each core has private L1 caches for instructions and data (128KB each), a private unified L2 cache (512KB) and an L3 cache (12MB) shared among all 16 cores of both sockets. All caches have a line size of 128 bytes. Each socket has two memory controllers (so-called memory nodes) for a total of 8 controllers at machine level. Cores are connected via HyperTransport with a 1.8GHz link speed. Cores within a memory node are 1 hop apart, cores across nodes in the same socket are 2 hops apart, and cores of different sockets are 3 hops apart. The processing frequency of cores can be varied from 800MHz to 2GHz, but the CPU governor policy immediately elevates the frequency to 2GHz when a CPU-bound application is initiated. As mentioned before, there are 128 banks (colors) over 4 memory controllers and 32 LLC colors at the disposal of TintMalloc.

3.5 Experimental Result

We performed a set of experiments with synthetic benchmarks and standard benchmarks from the SPEC and Parsec suites to compare TintMalloc to the default buddy allocator of Linux and prior coloring approaches. All experiments were repeated ten times, and their averages are reported in the following.

3.5.1 Synthetic Benchmark Results

We designed a synthetic benchmark that allocates a large memory space. This space is subsequently accessed in a pattern with alternating strides such that each cache line is only accessed once. This ensures that references punch through the private L1/L2 and even the shared L3 caches and have to be resolved in DRAM. The access pattern starts with a write in the middle of our allocation, M, followed by a write to M+1C (where C=128 bytes is the cache line size), M-1C, M+2C, M-2C, etc. This access pattern defeats hardware pre-fetching and results in page faults for a large address space. The pattern is exercised for different numbers of threads, each of which obtains different heap space

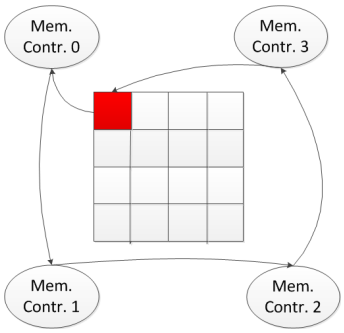

Figure 3.7 Access to remote controller (node)

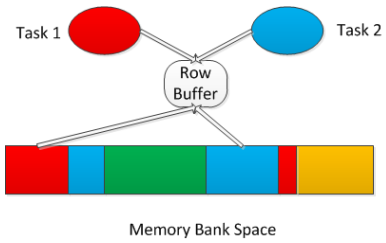

Figure 3.8 Access conflict in same bank

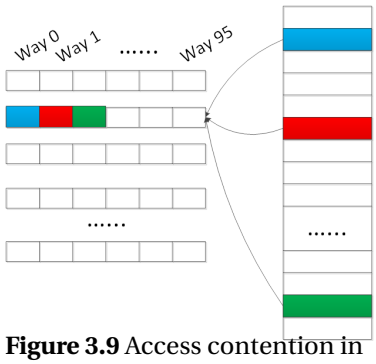

Figure 3.9 Access contention in the LLC

via buddy allocation and TintMalloc (with disjoint colors across threads for the latter).

In essence, this benchmark assesses the write latency of DRAM as it inflicts first cold and later capacity misses in L1/L2 caches for LLC coloring, or all caches for controller+bank coloring. Fig. 3.7 illustrates how one task may access a remote memory node and suffer the remote latency penalty under the buddy allocator. In addition, multiple tasks may share the same memory bank under buddy allocation. When two tasks access this bank at same time (Fig. 3.8), the second one will populate the row buffer and evict data from first one. This will inflate the memory access time of first task. The same problem also occurs in LLC accesses (Fig. 3.9). Here, the task's L3 cache miss rate will increase since other tasks evict its data from LLC.

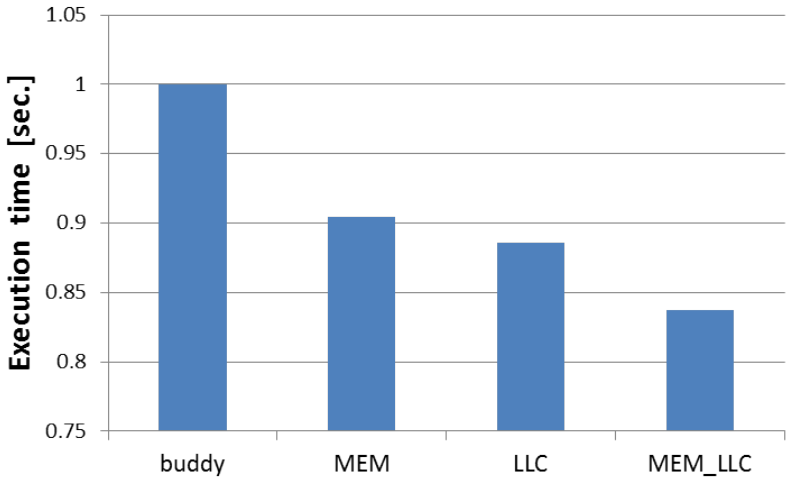

Figure 3.10 micro_result

Fig. 3.10 depicts the execution time of the synthetic benchmark on the y-axis for different coloring policies on x-axis. The shortest execution time is obtained with MEM/LLC, which indicates that both memory and LLC coloring are activated under TintMalloc. With MEM/LLC coloring, each

task obtains isolated resources as LLC and main memory are colored in concert. This reduces LLC access interference and bank level conflicts. In addition, the remote memory controller access penalty is never paid as accesses remain local. We observe that single memory coloring (MEM), single LLC coloring and MEM/LLC coloring all reduce the execution time. For MEM/LLC coloring, the execution time can be reduced by up to 17%.

3.5.2 Standard Benchmark Results

Next, we investigate TintMalloc's effectiveness for parallel programs using OpenMP of the PARSEC and SPEC benchmark suites. `Bodytrack, freqmine` and `blackscholes` in the Parsec benchmark suite and `lbm, art` and `equake` in the SPEC suite are the only OpenMP versions in those suites. We modified the OpenMP version of those benchmarks to include colored allocation in its initialization code (1 mmap() call per color) and timers at barriers to measure idle time.

The OpenMP programs contain data-intensive parallel tasks with implicit barriers at the end of each parallel section. Due to runtime differences, threads executing faster have to wait for the slower ones at a barrier. We measured the benchmark's runtime, total idle time, runtime per thread, and idle time per thread. The `idle time` indicates a thread's wait time at barriers (see `Algorithm 5`).

Algorithm 5 Measure idle time

```
1: nthreads = omp_get_num_threads()
2: tid = omp_get_thread_num()
3: Get the time "start"
4: pragma omp parallel /*begin pragma parallel section*/
5:       pragma omp for nowait
6:       For Loop /*computational section*/
7:       get the time "end[tid]"
8: end of pragma parallel section /*implicit barrier*/
9: Find maximum end time "max" from end[0] to end[nthreads]
10: Calculate idle time for each thread, idle[tid]=max-end[tid]
```

To show the impact of memory controllers, we vary the numbers of threads, their memory nodes, and the level of parallelism. There are a total of five configurations: `16_threads_4_nodes, 8_threads_4_nodes, 8_threads_2_nodes, 4_threads_4_nodes` and `4_threads_1_nodes`. For `16_threads_4_nodes`, we utilize all 16 cores and 4 memory controllers. 16 threads are pinned to different cores and share 4 controllers (nodes). For `8_threads_4_nodes`, 8 threads are pinned to 8 different cores and each pair of them accesses a common controller (their local memory controller)

disjoint from other pairs. For example, 8 threads are pinned to cores 0,1,4,5,8,9,12,13. In this case, tasks pinned to core 0 and core 1 access local controller 0. For `8_threads_2_nodes`, 8 threads are pinned to 8 cores from 2 nodes, e.g., cores 0-7. For `4_threads_4_nodes`, we select one core from each node and pin 4 threads to them, such as cores 0,4,8 and 12. For `4_threads_1_nodes`, 4 threads are pinned to 4 cores on the same node, e.g., cores 0-3.

For each configuration, we compared multiple coloring methods to standard buddy allocation and prior work. Our coloring methods are referred to as:

- `LLC coloring`: each thread has its private LLC colors but they share memory banks (uncolored).
- `Memory coloring (MEM)`: each thread has its private memory bank colors but they share the LLC (uncolored).
- `MEM+LLC coloring`: each thread has its private LLC colors and private memory banks colors. There is no sharing.
- `MEM+LLC(part)`: each thread has its private memory bank colors, but a group of threads shares private LLC colors.
- `LLC+MEM(part)`: each thread has its private LLC colors, but a group of threads shares private memory colors.

We compare TintMalloc to the standard buddy allocator of Linux (no coloring) and bank-level partitioning (BPM) [Liu12b]. BPM partitions memory banks and the LLC but it does not consider the memory controller in dynamic allocations.

`MEM+LLC(part)` and `LLC+MEM(part)` are different than `MEM+LLC` coloring. For example, there are a total of 32 LLC colors. For `MEM+LLC(part)` coloring with 16 threads, we create 4 thread groups. Each group has its private 8 LLC colors. Those 8 LLC colors are shared by the 4 threads in this group. For 8 threads in a parallel section, there are 2 threads per group sharing 8 LLC colors. In contrast, for `MEM+LLC` coloring, if 16 threads are in a parallel section, each thread has two private LLC colors. For 8 threads, each thread has four private LLC colors.

We compared `MEM+LLC` coloring, standard buddy allocation, previous work (BPM) and the best result from `MEM, LLC, MEM+LLC(part) and LLC+MEM(part)`. The results are shown in Figures 3.11, 3.12 , 3.13 and 3.14.

Fig. 3.11 shows normalized benchmark runtimes for these approaches relative to the standard buddy allocator. We observe that `MEM+LLC` coloring results in shorter runtimes than buddy and previous work (BPM) for all six benchmarks. For example, for 16 threads and 4 memory nodes, our approach reduces the average runtime by up to 29.84% over standard buddy allocation (for SPEC/lbm). We observe that some benchmarks' performance enhancements exceed than of our synthetic benchmark. This is caused by additional spatial locality resulting in cache hits for these codes. The synthetic benchmark, in contrast, cannot benefit from spatial locality at all since only one access occurs per cache line. The error bar shows the maximum and minimum runtime of each benchmark over 10 repeated experiments. We also observe that our approach reduces the deviation

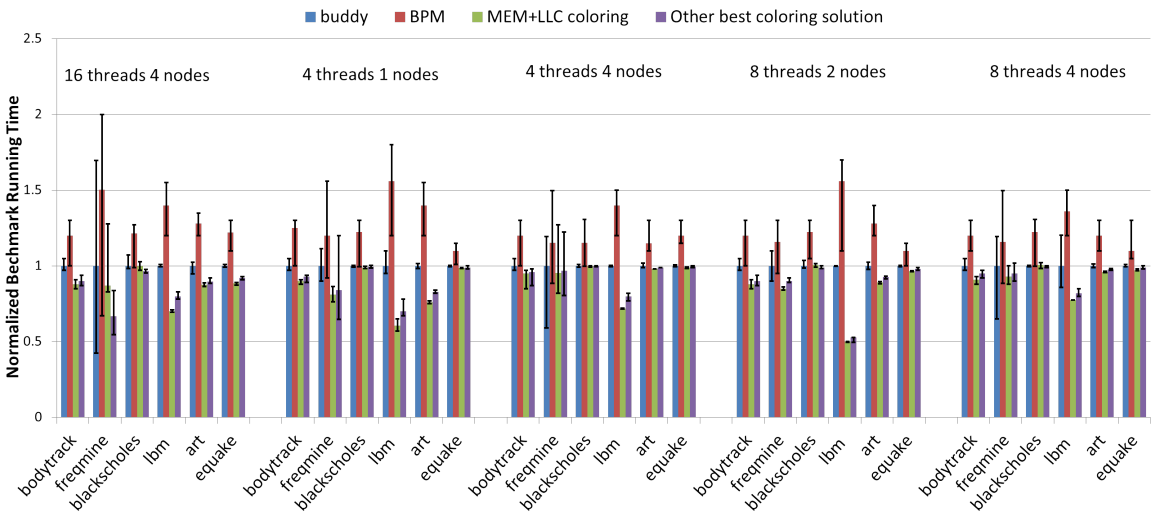

Figure 3.11 Benchmark running time

of runtime, i.e., it reduces the variance of execution time, which helps increase computational balance in parallel sections at barriers. In addition, the previous work (BPM) always results in longer runtimes than our coloring approach and the standard buddy allocator. This is because BPM only partitions memory banks and LLC but does not indicate a memory controller. In this case, tasks may access remote memory nodes and have to pay the remote access penalty. Of the different coloring configurations, `16_threads_4_nodes` experiences the largest performance boost over the 6 benchmarks. This is because more tasks increase the probability to access a remote memory, which results in more memory bank and LLC contention. Fig. 3.12 also indicates that a benchmark's idle time can be reduced by our coloring approach. For `16_threads_4_nodes`, our MEM+LLC coloring results in up to 74.3% lower idle time compared to standard buddy allocation due to more balanced computation (less runtime variation). In fact, we observe a correlation between idle reduction and benchmark runtimes across experiments.

Figures 3.13 and 3.14 indicate the runtime and idle time, respectively, spent by each thread in parallel sections. We observe that difference in runtime between the fastest and slowest thread under standard buddy allocation is larger than under our TintMalloc. For example, for SPEC/lbm benchmark in the `16_threads_4_nodes` configuration, the difference in maximum thread running time and minimum thread running time for buddy allocator is 4.38 times larger than that of the MEM+LLC coloring approach. In addition, the maximum thread runtime under MEM+LLC coloring is 30.77% smaller than for standard buddy. The maximum thread idle time of the lbm benchmark is also reduced by 75% under MEM+LLC coloring compared to buddy allocation. The results show that TintMalloc effectively results in more balanced parallel program execution and enhance performance at the same time.

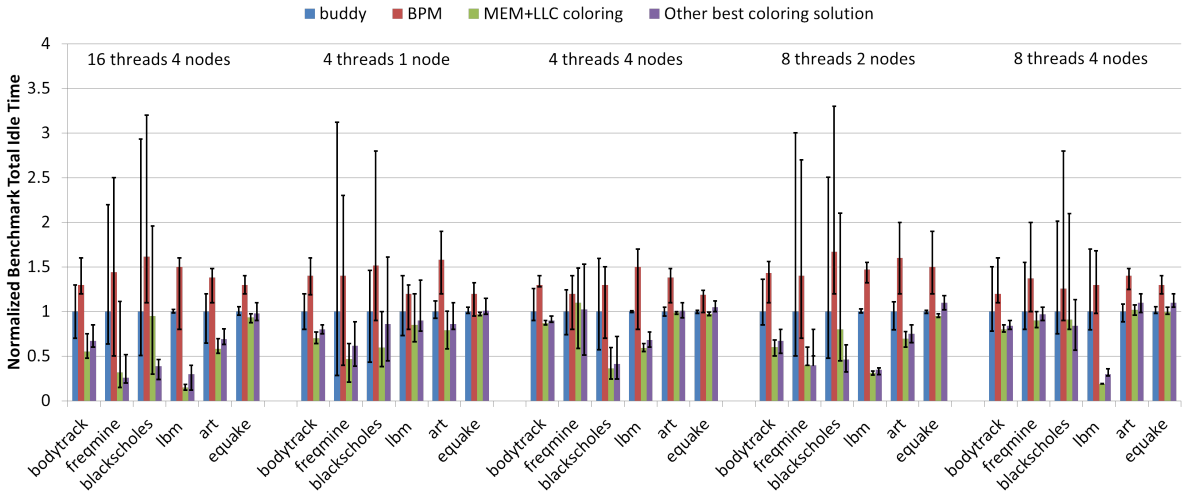

Figure 3.12 Benchmark idle time

Considering the four metrics comprehensively, we observe that the benchmark SPEC/lbm exhibits the largest performance enhancement under TintMalloc compared to buddy allocation. In addition, the Parsec/freqmine, Parsec/bodytrack and SPEC/art benchmarks are also sped up significantly. For those results, the averages and difference between maximum and minimum of each metric (benchmark runtime, total idle time, thread's runtime and idle time) are reduced by TintMalloc. This is because (1) these benchmarks allocate a large memory space on the heap, (2) they are memory intensive, i.e., their data space is accessed (data is reused) multiple times, and (3) the memory access patterns (and the data partition across threads) matches the per-thread first touch access allocation policy of the OS during initialization. In such cases, TintMalloc gets the most benefits in performance and load balance.

Furthermore, these benchmarks consist of alternating parallel and serial sections. The idle time reduction only affects performance enhancements for parallel sections while benchmark runtime reductions represent performance enhancements of the entire benchmark. Results indicate that the idle time reduction over all threads is larger than the runtime reduction due to more balanced barriers for most benchmarks under TintMalloc. For SPEC/equake, the benchmark's runtime and total idle time are also reduced by TintMalloc. However, the improvement in idle time is less than that of overall benchmark runtime. This is because the benchmark runtime is most affected by the proportionate reduction in runtime of the slowest thread, and the fastest thread's idle time will be reduced the most. After normalization, the ratio of runtime reduction may be larger than the benchmark's total idle time reduction (given that the idle time reduction of other threads is smaller than that of the fastest one).

In addition, we obverse that Parsec/blackscholes has the least performance improvement of the six benchmarks. Of all TintMalloc coloring solutions, MEM+LLC(part) is the best one and it reduces the runtime by 3.6% compared to buddy allocation for 16 threads on 4 nodes. This happens because

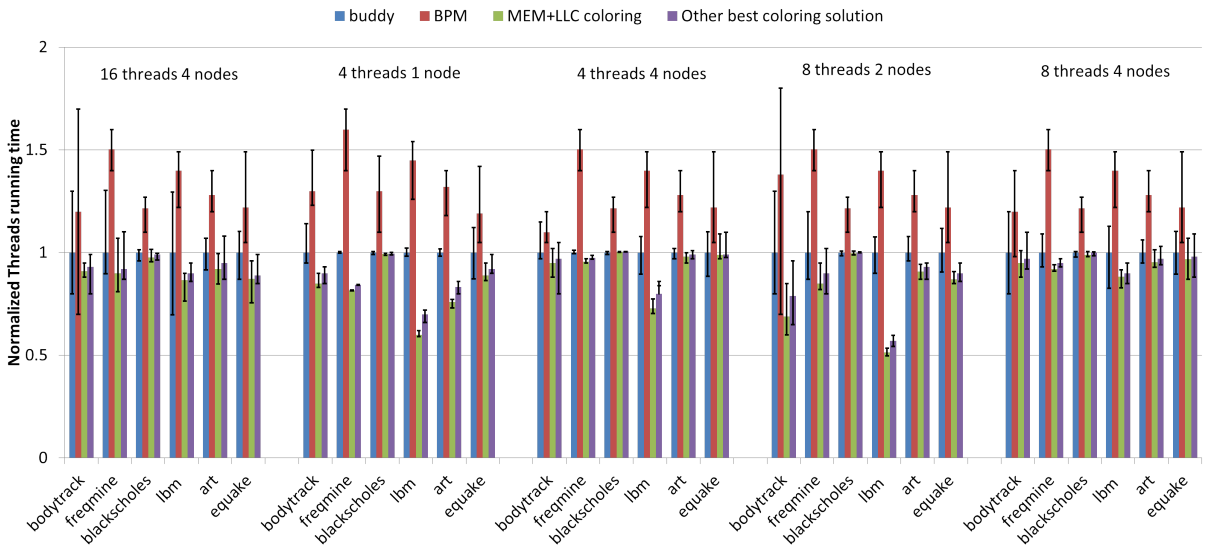

Figure 3.13 Threads running time

blackscholes reads a large mount of input data and is less memory intensive. Furthermore, the large fraction of the master thread's runtime prevents further performance enhancements since the master thread suffers from more restrictive memory allocation due to coloring. The larger the serial portion on the master thread is, the smaller will be the benchmark's performance enhancement.

The result also indicates that the MEM+LLC coloring approach is not always the best: For the Parsec/freqmine benchmark in the `16_threads_4_nodes` configuration, LLC+MEM(part) coloring outperforms MEM+LLC coloring in this case. This is because MEM+LLC coloring partitions the entire memory and LLC space, which restricts the overall memory space. This restriction increases the number of LLC cache conflict misses. Hence, the LLC+MEM(part) coloring method is an interesting trade-off for memory coloring without restrictions. Sometimes, "partly coloring", like LLC+MEM(part) or MEM+LLC(part), may result in better performance than "fully coloring", such as MEM+LLC.

3.6 Related Work

Blagodurov et al. [Bla10] and McCurdy et al. [MV10] describe performance problems that NUMA can present for multithreaded applications and investigate their causes. Marathe et al. [Mar10] propose a profiler to optimize data placement of multithreaded programs via hardware-generated memory traces. Lachaize et al. [Lac12] use profiling to help programmers understand why and which memory objects are accessed remotely and allow them to choose efficient application-level optimizations for NUMA systems. Majo et al. [MG13] study which factors limit the performance of multithreaded

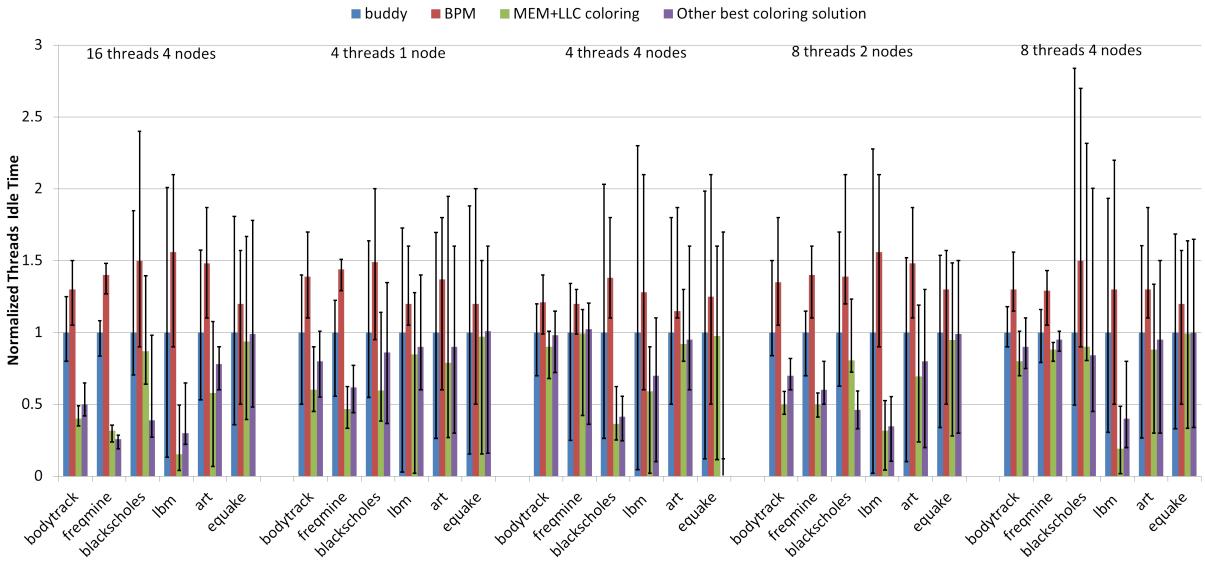

Figure 3.14 Threads idle time

programs on modern NUMA multicores and describe source-level techniques to address these problems. Yun et al. [Yun15] propose a new parallelism-aware memory interference delay analysis for parallel requests. TintMalloc address these problems by reducing memory access divergence on NUMA systems.

Several scheduling algorithms [Kim10b; Kim10a; Mur11] have been proposed to reduce memory contention. ATLAS [Kim10a] proposes a thread scheduling algorithm that optimizes the service order of threads periodically based on the amount of service they have attained from the memory controllers so far to improve system throughput. Kim et al. [Kim10a] dynamically divide threads with similar memory access behavior into two separate clusters (memory-non-intensive or memory-intensive) and employ different memory request scheduling policies in each cluster. Muralidhara et al. propose channel partitioning, which maps the data of applications that are likely to severely interfere with each other to different memory channels [Mur11]. In [Sud10], the frequently accessed data from different rows are dynamically migrated into the row buffer in memory, which can increase the memory row buffer usage and system performance. In contrast to TintMalloc, memory access contention cannot be removed entirely for all cases with their scheduling algorithms.

Several groups proposed scheduling or page placement to solve the data sharing problem. Li et al. [Li93] present a loop scheduling algorithm to exploit data locality and dynamically balance the load. Majo et al. [MG12] use program-level transformations to eliminate remote memory accesses. Ogasawara et al. [Oga09] propose an online method for identifying the preferred NUMA memory nodes of objects during garbage collection. Broquedis et al. [Bro10] propose a hierarchical approach named FORESTGOMP for the execution of OpenMP threads on multicore platforms. To maintain a good balance of threads, FORESTGOMP dynamically generates structured trees out of OpenMP pro-

grams and collects relationship information about threads and data. However, all these approaches introduce overhead and cannot completely eliminate data sharing.

Techniques to increase cache data locality that reduce shared memory contention are proposed in [MG11; CJ06; Per11; Suh04]. Majo et al. [MG11] describes two scheduling algorithms, maximum-local optimizes for maximum data locality and N-MASS reduces data locality to avoid the performance degradation caused by cache contention. Cho et al. [CJ06], Perarnau et al. [Per11] and Suh et al. [Suh04] employ software page coloring to partition shared caches for concurrently running threads, which eliminates the contention between threads and hence reduces conflicts at the cache level. DRAM partitioning is another scheme to reduce shared memory contention for parallel execution on multicore platform. The basic idea of using DRAM organization information in allocating memory at the OS level is explored in recent work [Yun14; Awa10]. Awasthi et al. [Awa10] examine the benefits of data placement across multiple memory controllers in NUMA systems. They introduce an adaptive first-touch page placement policy and dynamic page-migration mechanisms to reduce DRAM access delays in multiple memory controllers system but do not consider bank effects, nor do they reduce cache conflicts. Mi et al. [Mi10] develop a hardware/software co-design for bank coloring using address bit selection (XOR) but do not exploit virtual to physical address mapping purely in software as TintMalloc does. Palloc [Yun14] is a DRAM bank-aware memory allocator that provides performance isolation on multicore platforms by reducing conflicts between interleaved banks. Liu et al. [Liu12b] modify the OS memory management subsystem to adopt a page-coloring based LLC and bank level partition mechanism (BPM), which allocates specific LLC and DRAM banks to specific cores (threads). In contrast, our TintMalloc approach not only partitions memory banks and the LLC but also consider the locality of memory controllers, which is unprecedented in this combination.

3.7 Conclusion

This work contributes TintMalloc, a controller-aware memory and LLC coloring allocator for parallel systems. TintMalloc comprehensively considers memory node, bank and LLC locality to color the main memory and cache space without requiring hardware modifications. Only one additional line of an mmap() call in the initialization code suffices to trigger our controller-aware coloring heap allocation. This work describes the design and implementation of TintMalloc as an extension to the Linux kernel. Coloring address bits from PCI registers are utilized to determine locality and placement of a frame corresponding to a physical address, which makes the approach portable across x86 architectures with documented bit mappings (currently all AMD processors). With our approach, accesses to a remote memory node can be avoided for all tasks while bank and LLC access conflicts are reduced.

We assess TintMalloc in a number of experiments on a multicore platform with microbench-

marks as well as SPEC and Parsec OpenMP codes. Experimental results indicate that TintMalloc makes parallel tasks more balanced and enhances parallel system performance by reducing overall runtime and idle time at barriers.

CHAPTER

4

HIDING DRAM REFRESH OVERHEAD IN REAL-TIME CYCLIC EXECUTIVES

4.1 Introduction

DRAM is the de-facto standard technology for main memory 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 via the command bus. This mode is called auto-refresh and recharges all memory cells within the "retention time", typically 64ms for commodity DRAMs [JED10]. 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, 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 [Liu12a; Muk13; Nai13; Stu10].

Although the DRAM refresh impact can be reduced by proposed hardware solutions [Zhe08; Cha14; KP00; Rei], such solutions take a long time before they become widely adopted. Hence, other work seeks to assess the viability of software solutions. RAIDR [Liu12a] lowers refresh overhead by exploiting inter-cell variation in retention time. RAPID [Ven06] sorts pages by retention time and allocates long retention pages first. Smart Refresh [GL07] 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 [Sta12a]. These approaches either heavily rely on specific data access patterns of workloads, or they have high implementation overhead. None eliminate refresh overhead entirely. The refresh problem is even more significant for real-time systems because predictable memory access latencies are imperative to assess task schedulability [WM01]. 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 [AP01] 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 [BM10]. While they consider the cost of refresh operations, it cannot eliminated them.

This work contributes "Colored Refresh" to hide DRAM refresh overhead entirely. Colored Refresh makes real-time systems more predictable, particularly for large DRAM sizes. 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) DRAM refresh of modern memory systems is analyzed in detail. The impact of refresh delay under varying DRAM densities/sizes is highlighted for real-time systems with stringent timing constraints. We observe that it is hard to predict an application's refresh overhead with auto-refresh. Furthermore, the losses in DRAM throughput and performance caused by refreshes quickly become unacceptable for real-time systems with high DRAM density.

(2) Colored Refresh is contributed, which refreshes DRAM based on memory space coloring. Refresh overhead is entirely hidden since a memory rank is either being accessed or being refreshed, but not both. Thus, regular memory accesses never suffer from refresh interference, i.e., the refreshes are completely 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 is completely implemented in software, it does not require any hardware change.

4.2 Background

4.2.1 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 read/write 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. 5.2). 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 $tRP + tRAS$.

4.2.2 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 (tRET) of leaky cells, sometimes also called refresh window (tREFW) [JED10; Sta12a; Kim11; Mor05]. To prevent data loss, all rows in a DRAM chip need to be refreshed within tRET (or tREFW). In order to reduce the refresh overhead, memory in each DRAM chip is divided into ranks, which are refreshed in parallel [Bha16]. 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

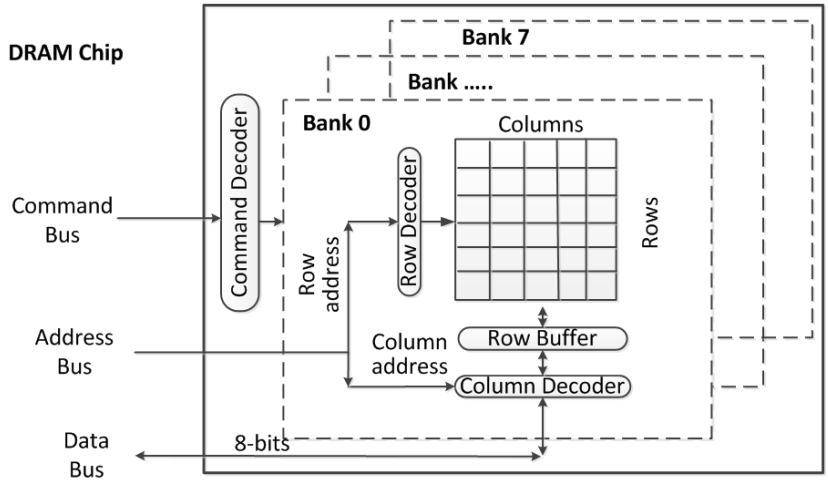

Figure 4.1 DRAM Bank Architecture

called a "refresh bin" and contains multiple rows. The DDR3 specification [JED10] generally 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 (tRFC), 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 (tRFC) is delimited by power constraints. Table 5.1 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 future DRAM technology [Liu12a]. Ranks can be refreshed in parallel under auto-refresh, but this increases the amount of unavailable memory increases 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 it is considered in a blocking term for schedulability analysis.

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 $tRP + tRAS$

Table 4.1 $tRFC$ per DRAM densities (data from [JED10; Sta12a; Liu12a])

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	$\geq 2us$

since the refresh purges all buffers and often results in additional row buffer misses, i.e., decreased memory throughput. Liu et al. [Liu12a] observe that the loss in DRAM throughput due to refreshes becomes untenable for large memories, reaching nearly 50% for 64 Gb DRAM.

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. The asynchrony of refreshes combined with task preemption makes it hard to accurately predict and bound DRAM refresh delay.

4.2.3 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 [Bha16]. In contrast, recent LPDDRx DRAM [Sta12b] 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 [Bha15]. For each rank, a refresh counter 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 scheduling strategies exist, namely distributed and burst refresh.

Distributed Refresh: A single refresh operation is performed periodically (see upper Fig. 5.3). Once all rows are refreshed, the refresh cycle is repeated by starting from the first row. With distributed refresh, the DRAM response time for regular 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 ($tREFI = 0$), until all rows have been refreshed (see lower Fig. 5.3). After that, the memory is available for memory accesses until the next refresh. A burst refresh results in long periods during which the memory is unavailable, which also affects task execution and results in longer memory latencies, yet such bursts occur less frequently.

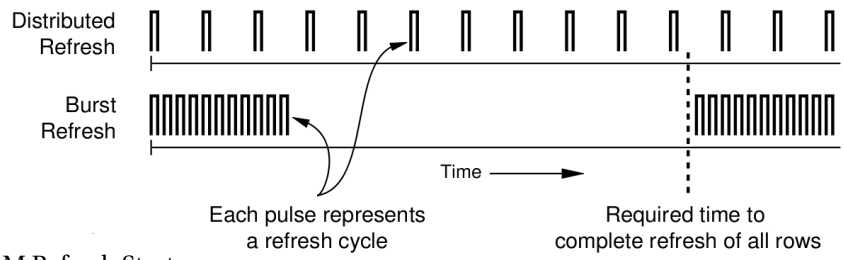

Figure 4.2 DRAM Refresh Strategy

4.3 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 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 remove DRAM refresh interference by memory partitioning (coloring). Under our approach, DRAM is partitioned, and each application is assigned a colored partition. 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. To this end, each worst-case execution time, e_i, is padded, by $tBST = m \times tRFC$, where $m = \lceil e_i / tREFI \rceil$. For a cyclic executive [BS88; Loc92], e_i can be calculated iteratively by determining the fixed point for

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

We assume that tasks are independent and tasks can be sliced. Our work accounts for any slicing or copying overhead in a term added to e_i. 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 reported to be the case in practive [BM10].

4.3.1 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. Pan et al. [Pan16] designed TintMalloc, 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. Real-time tasks, after their initialization, experience highly predictable latencies for subsequent memory requests.

4.3.2 Colored Refresh Design

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, we designed a novel "Colored Refresh" policy, which systematically schedules DRAM refreshes based on DRAM rank coloring, i.e., at the refresh granularity level. This section introduces the design of Colored Refresh. 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 refresh interval to be configured [BM10] and can control which rank should be sent a refresh command. With Colored Refresh, the memory space of a DRAM chip is refreshed by iterating over the ranks such that all tasks are colored by different DRAM ranks through TintMalloc.

Let us denote the set of periodic real-time tasks as $\mathscr{T} = T_1 ... T_n$, 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, and a hyperperiod H of $\mathscr{T}$ [Liu00]. 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.

With Colored Refresh, the entire DRAM space is equally partitioned into "colors", such that each color contains one or more DRAM ranks. The DRAM retention time, R, is also equally divided among the number of frames following a one-to-one correspondence with colors, where

c_i denotes color i of C total colors.

We construct a cyclic executive schedule for $\mathscr{T}$, where

f is the frame size and f_i denotes the i^{th} frame,

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, ranks contained in c_i are refreshed within 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 bust 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 based on the following 5 rules:

(1) $f \leq min_{1 \leq i \leq n}(p_i/2)$ such that 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.
(3) $2f - gcd(p_i, f) \leq D_i$ for all i.
(4) $\lfloor (R/K)/f \rfloor - (R-K)/f = 0$, i.e., f is an integer multiple of R/K.
(5) Frames must be sufficiently long so that every job can execute non-preemptively within them, i.e., $f \geq max_{1 \leq i \leq n}(e_i)$.

According to the 5 rules (2 of them derived from [BS88; Loc92]), 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 [Liu]. 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.

4.3.3 Modifications to the Task Set

Some conditions may require a task set $\mathscr{T}$ 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 [BS88; Loc92], 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 [Liu] — 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) and (5) [BS88; Loc92].
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 [BS88; Loc92]. 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 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 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) with memory colors derived from Colored Refresh based on the virtual frame size to satisfy rules (2)+(4).

As required by rule (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, i.e., $\frac{i}{j}$ is an integer. According to **Theorem 1**, we can select an appropriate f from $(0, D_{min}/2]$ to satisfy the above constrains. Finally, Colored Refresh guarantees schedulability (schedule tasks by frames whose size is f as a cyclic executive) and colors memory space (partition and assign memory to tasks at f'/R granularity) via frame fusing.

Algorithm 6 Task Fusion

```
1:  for f in (0, D_min/2] do
2:      f' = f, i = 1
3:      while f' ≤ R do
4:          f' += f, i++
5:          if (f' mod R/K) == 0 then
6:              j = f'/(R/K)
7:              if (i mod j) == 0 then
8:                  return f and f'
9:              else
10:                 continue
11:             end if
12:         end if
13:     end while
14: end for
```

• 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 $f = R/K$. We utilize a novel technique to address this case. Colored Refresh creates two instances of a task, the original one and a so-called "copy tasks". The two instances have identical control flow, but their data is referencing memory allocated from different colors. 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 global 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.

• 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 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 eliminated 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.

4.3.4 Schedulability of Colored Refresh

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 [BS88; Loc92]. 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: None of the $\hat{f}$ candidates that satisfy rules (3) and (5) also satisfy rules (1)+(2)+(4). Let $f_{min}=R/K$, which is the minimum frame size established by rules (2) and (4), i.e., both of these 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=0. 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, whats need 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. 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. □

4.3.5 Example

Example 1: Consider three real-time tasks, A=(16,4), B=(32,12), C=(64,16). Let R=64 ms, K=8, which means there are 8 ranks that should be refreshed every 64ms. For this task set, 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.

In this case, f is chosen to be 8, and tasks are arranged by the network flow algorithm [Liu] into frames of the cyclic executive's table according their periods and deadlines, and memory is colored according to Colored Refresh. Fig. 4.3 depicts a feasible cyclic schedule for the above example, where c (above) indicates the rank that is being refreshed.

The above schedule has a memory coloring solution for each task through Colored Refresh (see Table 4.2).

In this example, the entire DRAM space is refreshed by iterating over all 8 ranks, and the refresh

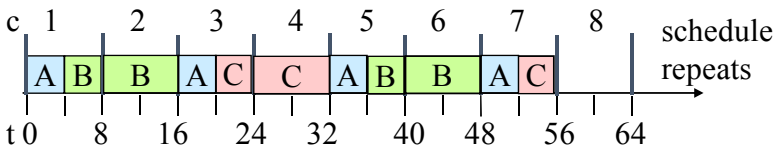

Figure 4.3 Harmonic Schedule for H, R

Table 4.2 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

duration of each rank is 8ms (R/K). Within one rank's refresh duration, all the refresh operations are scheduled as "burst refresh" (see Section 5.2.4). 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. One rank is under refresh during its corresponding frame while all other ranks are refresh-free during this frame. From the allocation rule for f_i and Fig. 4.3, we can identify the available colors per task (see Table 4.2), 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 $((A,k_2),(B,k_3),(C,k_5))$.

Example 2: In practice, the DRAM retention time (R) is not always harmonic with the hyperperiod of periodic real-time tasks (H). If H and R are non-harmonic, the phase of task execution time and refresh operation are different. As a result, one task may occupy all frames in a refresh period L. 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. (1)-(5) hold for f=8. Fig 4.4 shows a feasible cyclic schedule. Due to non-harmonic phasing, all frames are occupied by both A and B (Tab. 4.3), , i.e., no frames remain available.

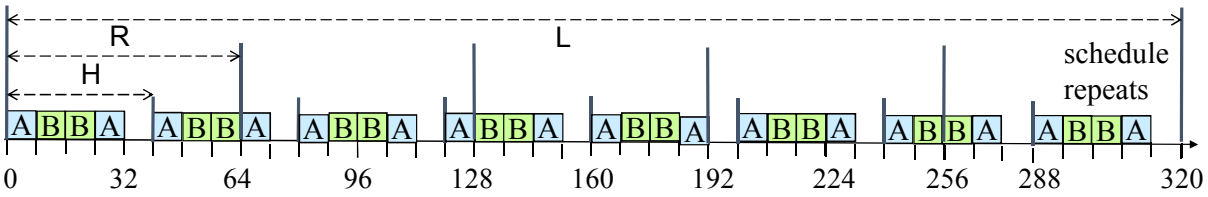

Figure 4.4 Non-harmonic Schedule

We can still remove refresh overhead via "copy tasks" for both A and B. A copy of, e.g., A', executes the same code (as A), but memory is allocated from different colors. When a job of this task is released, an instance is chosen for execution that differs from the color under refresh. E.g., let c_1 and c_2 be two instances of A. c_1 is refreshed during f_1 while c_2 is refreshed during f_2. If a job of A is scheduled

Table 4.3 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

in f_1, the instance colored to c_2 is executed and vice versa. For other frames (f_3-f_8), either instance may execute. Hence, all DRAM refresh overhead is hidden from task execution and thus need not be considered for schedulability analysis.

4.3.6 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). Assuming a task set can be scheduled under auto-refresh, for any time span t,

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

where 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. The highest utilization under Colored Refresh is 1.

4.3.7 Discussion

The two examples in Sec. 4.3.5 featured a periodic real-time task set on a single processor. Nonetheless, 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. Furthermore, sporadic tasks and non-real-time tasks can also be placed into frames and scheduled with Colored Refresh. 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 [BM10].

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 the following cases:

1) A *long task* is a task 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 removed 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 removed 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.

2) A *Short task* is a task whose period is shorter than twice the minimum possible frame size (R/K). From (1), the period of a *short task* is

$$Period < (R/K) * 2.$$

For example, a task is defined as a *short task* if its period is less than 4 ms when the DRAM retention time is 64 ms and there are 32 ranks. A *short task* will occupy up to all frames of a color (up to its rank/color size), but once it exceeds this limit, it is split. A copy task may be required to hide all refresh overhead for the *short task* if its jobs are independent. For a *short task* whose jobs are dependent, Colored Refresh can also enhance utilization and make its refresh overhead more predictable by coloring it to a set of specific ranks.

4.3.8 Overhead of Colored Refresh

In order to hide DRAM refresh overhead, Colored Refresh suffers extra task coping 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 global 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 and smartphones [JED10; Sta12a]. 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.

4.4 Implementation

4.4.1 System Architecture

Our experimental framework consists of three components. (1) SimpleScalar 3.0 [Bur96] 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. 5.6). 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.

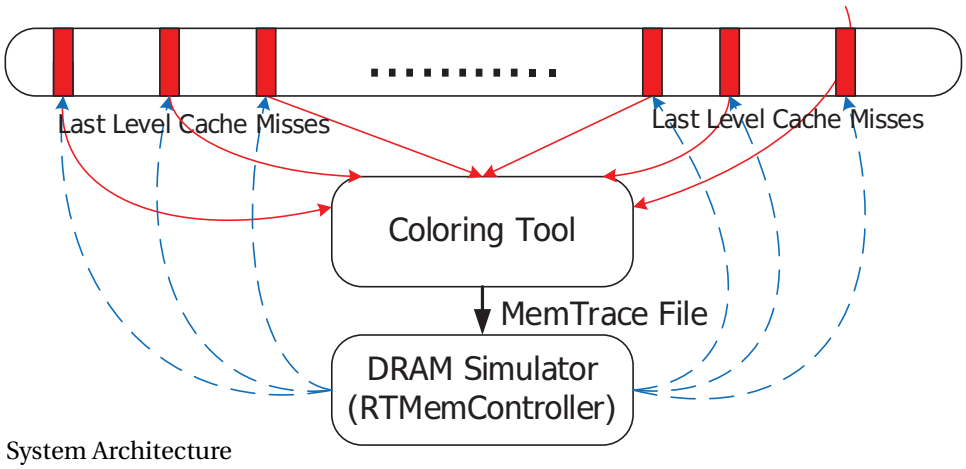

Figure 4.5 System Architecture

(3) Memory traces are relayed to RTMemController [Li14b], a back-end architecture for real-time memory controllers. It schedules each memory transaction and determines its execution time, which is fed back to SimpleScalar to determine the overall execution time. 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 Colored Refresh and DDR4 refreshes. 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.

4.4.2 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 [Pan16] to SimpleScalar so that it can select the color of physical addresses in memory. 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. We further retain page locality (and thus also 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), and the application only accesses this specific area as per coloring policy.

4.4.3 Discussion

While we evaluate our approach by simulator, it shall be noted that it can be ported onto a real architecture with engineering effort [BM10]. DRAM refreshes are synchronous with processor clock (if the clock is fixed) and can, in fact, optionally be disabled for a subset of ranks [Tov17]. Furthermore, per-rank refresh activation phasing can be reverse engineered by monitoring access latencies during the initialization of our approach.

Table 4.4 Task Set

Program	Period	Exec. Time
lms	20ms	4ms
compress	32ms	6ms
cnt	32ms	8ms
st	40ms	8ms
matmult	80ms	10ms

Table 4.5 Memory Assignment per Task

Program	Occupied frames	available colors (c_i)
lms	$f_1 - f_8$	coloring by task copying
compress	$f_1, f_3, f_4, f_5, f_7, f_8$	c_2, c_6
cnt	f_3, f_4, f_7, f_8	c_1, c_2, c_5, c_6
st	f_2, f_4, f_6, f_8	c_1, c_3, c_5, c_7
matmult	$f_2, f_3, f_4, f_6, f_7, f_8$	c_1, c_5

4.5 Evaluation Framework and Results

4.5.1 Experimental Setup

We assessed the Malardalen WCET benchmark programs [Gus10] 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 (1Gb, 2Gb, 4Gb, 8Gb, 16Gb, 32Gb 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.

Table 4.6 Colors per Task

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

4.5.2 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. 5.3).

Table 4.4 shows each Malardalen task's execution time 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 the real-time tasks of Table 4.4, 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. 4.6, where c indicates the color that is being refreshed (repeats at R=64ms). Table 4.5 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 4.6 indicates one feasible selection of colors after creating copy task $\hat{A}$, with a corresponding schedule shown in Fig. 4.6 for the intervals 176…184 and 240…248. Notice that at t=180 and t=240, $\hat{A}$ executes instead of A since $c = 7$ is being refreshed.

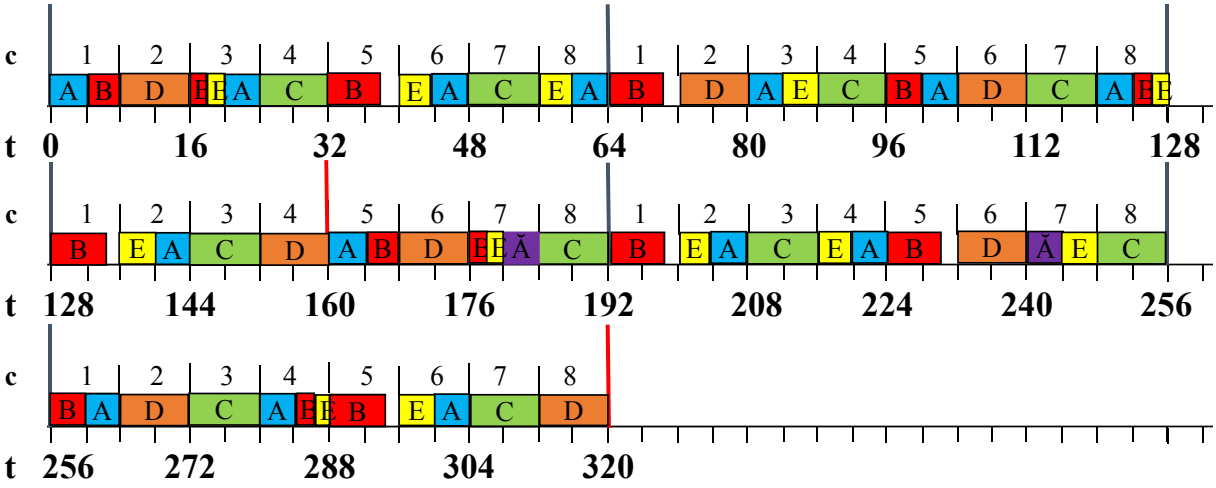

Figure 4.6 A Schedule with Copy Tasks

We compared the system utilization under Auto-Refresh, Colored Refresh, and Non-Refresh configurations. Fig. 4.7 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. 4.7 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 DRAM densities (<16Gb)

but becomes infeasible for higher densities (≥16Gb) so that deadlines are missed as additional refresh overhead pushes utilization to above 100%. Furthermore, since the maximum possible frame size for a standard cyclic executive could be 12ms (larger than all tasks' execution times), *matmult* requires task splitting just for our approach. Assuming a copy+split overhead (see Section 4.3.8) 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.

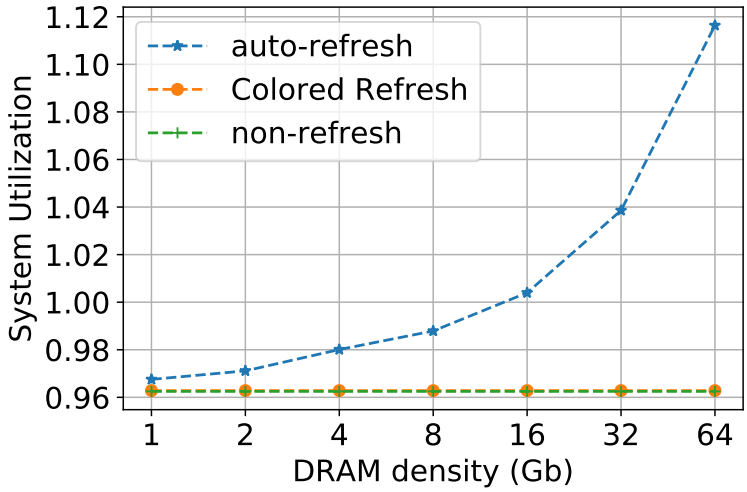

Figure 4.7 System Utilization per Refresh Method

Observation 1: Compared to Auto-Refresh, Colored Refresh reduces task execution time and enhances system utilization by eliminating 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.

4.5.3 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 4.3.2. Let us discuss the impact on DRAM performance under Colored Refresh.

Fig. 4.8 shows the normalized average memory access latency (y-axis) of Auto-Refresh over our approach for different tasks (x-axis). Since our approach removes 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. 4.8 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 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. 4.8 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 the overhead to issue more refresh commands, but also because the probability of interference with refreshes increases. Table 4.7 illustrates this by showing the number of memory references suffering from interference by all tasks per DRAM density.

Table 4.7 # Memory Accesses with Refresh Interference

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

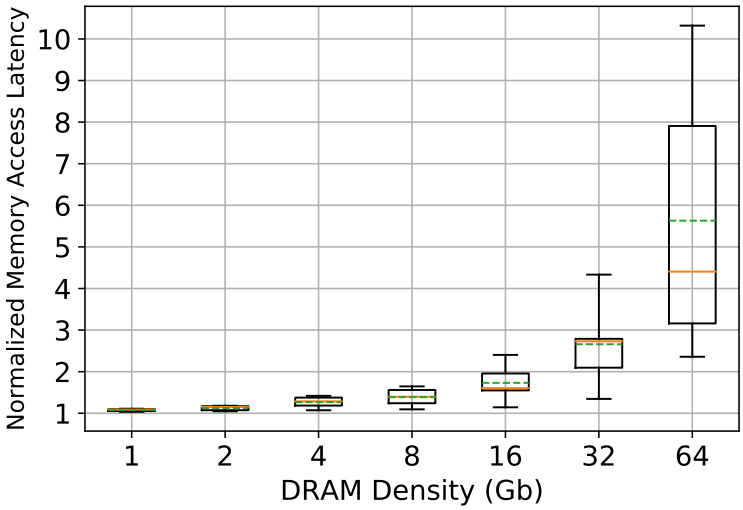

Figure 4.8 Normalized Memory Latency of Auto-Refresh relative to Colored Refresh

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

4.5.4 Fine Granularity Refresh

JEDEC's DDR4 DRAM specification [Sta12a] 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 [Sta12a]. 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% [Muk13].

Fig. 5.11 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 elimination of 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

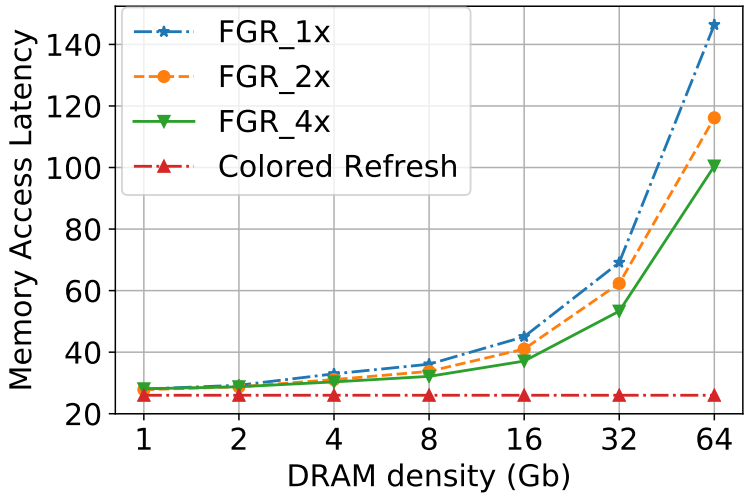

Figure 4.9 Colored Refresh vs. Fine Granularity Refresh (FGR)

access latency, respectively, relative to CRS. 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.

4.6 Related Work

We do not reiterate related work discussed in Sect. 4.1 due to space limitations here. Several DRAM refresh mechanisms are analyzed, and refresh penalties are quantified in recent work [Bha16; Emm08; Liu13; KL09; Nai13; Ham98]. Mukundan et al. [Muk13] discuss the refresh overhead for DDR4 DRAM with high densities.

Bhat et al. [BM10] 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. Selective DRAM Refresh [Liu11] 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 [GL07]. But the performance of Selective DRAM Refresh still heavily depends on the data access pattern. 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. [Stu10] 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.

4.7 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 eliminate DRAM refresh overheads at the software level for cyclic executive scheduling of real-time 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 eliminates refresh overhead completely 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.

CHAPTER

5

THE COLORED REFRESH SERVER FOR DRAM

5.1 Introduction

Dynamic Random Access Memory (DRAM) has been the memory of choice in most computer systems and embedded systems for many years. DRAMs owe their success to their low cost combined with large capacity, albeit at the expense of volatility. A DRAM cell is composed of an access transistor and a capacitor, where data is stored in the capacitor as 1 or 0 (electrically charged/discharged). However, capacitors slowly leak their charge over time. This leakage volatility requires cells to be refreshed, otherwise their data would be lost.

As specified by the DRAM standards [JED10; Sta12a], 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 [JED10; Sta12a]. 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 up a new row subject to refresh [BM10], even though data 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 [Liu12a; Muk13; Nai13; Stu10], a problem that is becoming more prevalent as DRAM 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. Even with conservative estimates of DRAM growth in density for future DRAM technology, the cost of one refresh operation, $tRFC$, exceeds 1 micro-second at 32 Gb DRAM size, and the loss in DRAM throughput caused by refreshes reaches nearly 50% at 64 Gb [Liu12a]. Some work focuses on reducing DRAM refresh latencies from both hardware and software angles. Although the DRAM refresh impact can be reduced by some proposed hardware solutions [Zhe08; Cha14; KP00; Rei], such solutions take a long time before they become widely adopted. Hence, other work seeks to assess the viability of software solutions by lowering refresh overhead via exploiting inter-cell variation in retention time [Liu12a; Ven06], reducing unnecessary refreshes [GL07; Liu11], and decreasing the probability of a memory access interfering with a refresh [Bha15; Stu10]. Fine Granularity Refresh (FGR), proposed by JeDEC's DDR4 specification, reduces refresh delays by trading off refresh latency against frequency [Sta12a]. Such software approaches either heavily rely on specific data access patterns of workloads or have high implementation overhead. More significantly, none of them can eliminate refresh overhead entirely.

For real-time systems, the refresh problem is even more significant. Bounding the worst-case execution time (WCET) of a task's code is key to assuring correctness under schedulability analysis, and only static timing analysis methods can provide safe bounds on the WCET [WM01]. 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 [AP01] 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. Furthermore, as the 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. Although Bhat et al. make refreshes predictable and reduce preemption from refreshes by triggering them in software instead of hardware auto-refresh [BM10], the cost of refresh operations is only considered, but cannot be eliminated. Also, a task cannot be scheduled under Bhat if its period is less than the execution time of a burst refresh.

This work contributes the "Colored Refresh Server" (CRS) to remove task preemptions due to

refreshes and to hide DRAM refresh overhead entirely. As a result, CRS makes real-time systems more predictable, particularly for high DRAM density. CRS exploits colored memory allocation to partition the entire memory space into two colors corresponding to two server tasks (simply called servers from here on). Each real-time task is assigned one color and associated with the corresponding server, where the two servers have different static priorities. DRAM refresh operations are triggered by two tasks, each of which issues refresh commands to the memory of its corresponding server for a subset of a colors (DRAM ranks) using a burst refresh pattern. More significantly, by appropriately grouping real-time tasks into different servers, 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 CRS is small and remains constant irrespective of DRAM density/size. In contrast, auto-refreshed overhead keeps growing as DRAM density increases.

Contributions:

(1) DRAM refresh of modern memory systems is analyzed in detail. The impact of refresh delay under varying DRAM densities/sizes is highlighted for real-time systems with stringent timing constraints. We observe that refresh overhead for an application is not easy to predict under standard auto-refresh. Furthermore, the losses in DRAM throughput and performance caused by refreshes quickly become unacceptable for real-time systems with high DRAM density.

(2) The Colored Refresh Server (CRS) is developed, which refreshes DRAM based on memory space coloring and schedules tasks according to the server policy. Refresh overhead is almost entirely hidden since a memory space is either being accessed or refreshed, but never both at the same time. Thus, regular memory accesses no longer suffer from refresh interference, i.e., the blocking effect of refreshes remains hidden in a safe manner.

(3) Experiments with the Malardalen benchmarks confirm that both refresh delays are hidden and DRAM access latencies are reduced. Consequently, application execution times become more predictable and stable, even when DRAM density increases. An experimental comparison with DDR4's FGR shows that CRS exhibits better performance and higher task predictability.

(4) CRS is realized in software and can easily be implemented on commercial off-the-shelf (COTS) systems.

(5) Compared to previous work[BM10], CRS not only eliminates refresh overhead, but also feasibly schedules short tasks (period less than execution time of burst refresh) by refactoring them as "copy tasks".

(6) Our approach can be implemented with any real-time scheduling policy supported inside the CRS servers.

5.2 Background and Motivation

5.2.1 DRAM Architecture

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 (see Fig. 5.1). 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 (see Fig. 5.1).

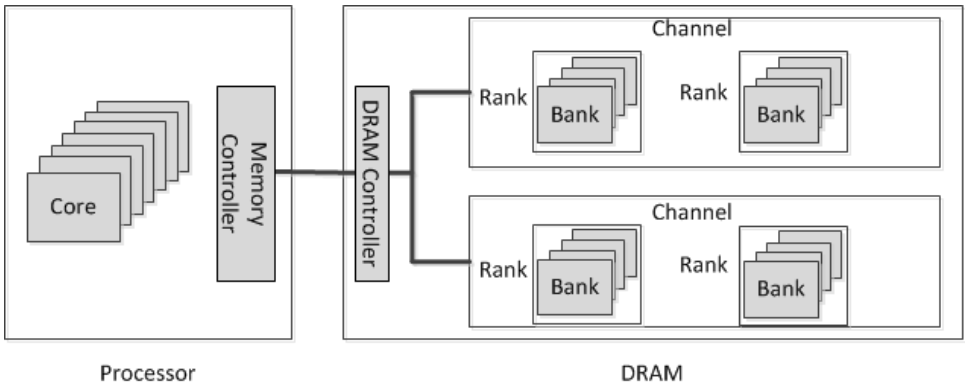

Figure 5.1 DRAM System Architecture

A DRAM bank array is organized into rows and columns of individual data cells (see Fig. 5.2). To resolve a memory access request, the row containing the requested data needs to first be copied from the bank array into the row buffer. As a side effect, the old row in the buffer is closed ("precharge") incurring a Row Precharge delay, tRP, and the new row is opened ("activate") incurring a Row Access Strobe delay, $tRAS$. This is called a row buffer miss. Once loaded into the row buffer and opened, accesses of adjacent data in a row due to spatial locality incur just a Column Access Strobe penalty, $tCAS$ (row buffer hit), which is much faster than $tRP + tRAS$.

5.2.2 Memory Space Partitioning

In the following, we assume a DRAM hierarchy with node, channel, rank, and bank abstraction. To partition this memory space, we utilize TintMalloc [Pan16], 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

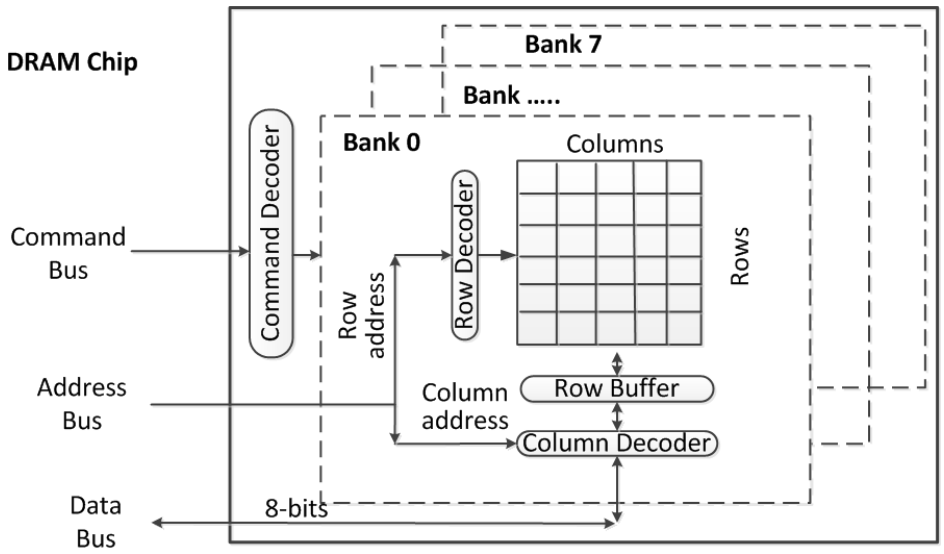

Figure 5.2 DRAM Bank Architecture

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). In addition, once our colored free list has been populated with pages, the initialization overhead 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.

5.2.3 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, $tREFW$ [JED10; Sta12a; Kim11; Mor05]. 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 [Bha16]. 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 [JED10] 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.

Depending on the memory technology generation, the refresh granularity may vary. Nonetheless, memory ranks are 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. The cost of a refresh operation is calculated as $tRFC/tREFI$. As density of DRAM chips grows, the size of each refresh bin becomes larger, i.e., it contains more rows. But the more rows in a refresh bin, the longer the refresh delay and memory block times become. The cost of a refresh operation, $tRFC$, is delimited by power constraints. Table 5.1 shows that the size of a refresh bin expands linearly with memory density so that $tRFC$ increases rapidly as DRAM density grows and exceeds 1us at 32 Gb DRAM, even with conservative estimates of growth in density for future DRAM technology [Liu12a]. 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.

Table 5.1 $tRFC$ for different DRAM densities (data from [JED10; Sta12a; Liu12a])

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	$\geq 2us$

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 row buffer again, 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. As a result, an additional overhead of $tRP + tRAS$ is incurred to close and re-open rows since the refresh purges all buffers. This tends to result in additional row buffer misses and thus decreased memory throughput. Liu et al. [Liu12a] observe

that the loss in DRAM throughput caused by refreshes quickly becomes untenable, reaching nearly 50% for 64 Gb DRAM.

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 combined with task preemption, it is hard to accurately predict and bound DRAM refresh delay.

5.2.4 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 within this rank, which is called "All-Bank" refresh [Bha16]. In contrast, recent LPDDRx DRAM [Sta12c] 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 [Bha15]. For each rank, a refresh counter 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 is sent to a bin (successive rows), two scheduling strategies exist, namely distributed and burst refresh (see Fig. 5.3).

Distributed Refresh: A single refresh operation is performed periodically. Once all rows are refreshed, the refresh cycle is repeated by starting from the first row. As Fig. 5.3 shows, distributed refresh only schedules one automatic refresh every $tREFI$. All refreshes are sent by the DRAM controller and performed in hardware. Distributed refresh is currently the most common method. However, the response time of regular memory accesses varies over a wide time range due to the spread of refreshes, and due to the overhead incurred by closing the row buffer.

Burst refresh: A series of refresh cycles are performed back-to-back ($tREFI = 0$) until all rows have been refreshed. After that, the memory is available for accesses until the next refresh, which is issued after $tRET$, the DRAM retention time. As shown in Fig. 5.3, sequential refreshes are performed successively at the beginning of each $tRET$ period. Although burst refresh can reduce extra row buffer misses, the cost of refresh operations still decreases DRAM system performance. More importantly, a burst refresh results in long periods during which the memory is unavailable, which also affects task execution and results in longer memory latencies, yet such bursts occur less frequently. For real-time system, a long memory blocking time may result in deadline misses, in particular if the period of a real-time task is short.

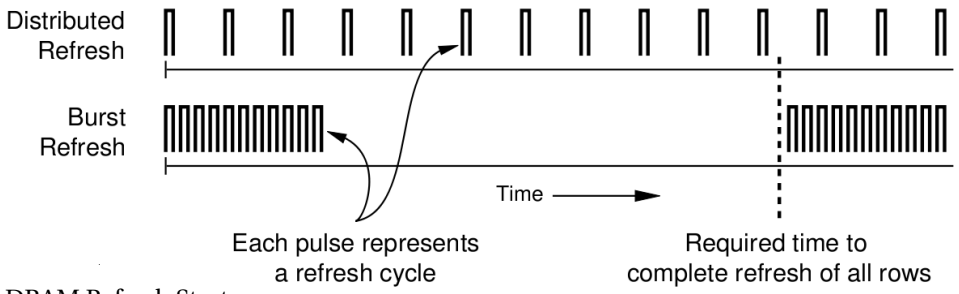

Figure 5.3 DRAM Refresh Strategy

5.3 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 [FM02; LB03; SL03] to schedule it with two servers. To this end, we partition the DRAM into two colors, and each server is assigned a colored memory partition. 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.

5.3.1 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 [JED10; Sta12a], the entire DRAM has to be refreshed within its retention time, $tRET$, in a manner where all K ranks are refreshed either serially or in parallel. We also assume hardware support for timer interrupts and memory controller interrupts (MC interrupts).

5.3.2 Task Model

Let us denote the set of periodic real-time tasks as $\mathscr{T} = \{T_1 ... T_n\}$, 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, and a hyperperiod H of $\mathscr{T}$. 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.

5.3.3 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 rank, 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) [SL03], $S(W, A, c, p_s, e_s)$, 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).

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.

5.3.4 Refresh Lock and Unlock Tasks

We employ "software burst parallel refresh" [BM10] 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). In our approach, there are two "refresh lock tasks" (T_{rl1} and T_{rl2}) and two "refresh unlock tasks" (T_{ru1} and T_{ru2}), where $S_1 = \{$ and S_2. T_{rl1} and T_{ru1} refresh color c_1 allocated to server S_1 while T_{rl2} and T_{ru2} refresh color c_2 allocated by server S_2. The top-level task set $\mathscr{T}_\top$ 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}, i.e.,

$\mathscr{T}_\top = \{S_1, S_2, T_{rl1}, T_{ru1}, T_{rl2}, T_{ru2}\}$.

As Fig. 5.4 shows, when a refresh lock task is released, 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 during time of DRAM refresh work for each color. Notice that a CPU server resumes execution only if its budget is

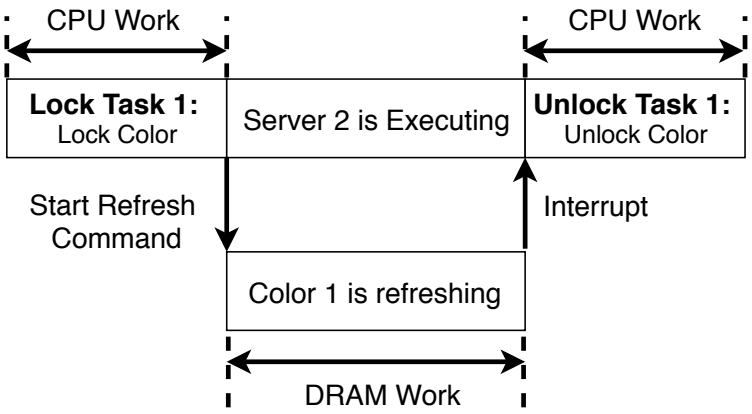

Figure 5.4 Refresh Task with CPU Work plus DRAM Controller Work

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 refresh unlock task that unlocks the newly refreshed colors so that they become available again. Notice that 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) In the absence of DRAM controller support, 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. The overhead of this interrupt handled 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 the both 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:

$\mathscr{T}_\top = \{S_1, S_2, T_{rl1}, T_{ru1}, T_{rl2}, T_{ru2}\}$, 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 $\mathscr{T}_\top$ can be scheduled statically as long as the lock and unlock tasks have a higher priority than the server tasks.

A refresh unlock refresh 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), and derived from the DRAM density according Table 5.1.

5.3.5 CRS Implementation

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

Scheduling: As we described in Sec. 5.3.4, 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-monotone priority assignment (shorter period, higher priority), static scheduling requires assignment of a strict fixed priority to each subject task (servers and refresh tasks). 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 refresh for c_1 and c_2, respectively, i.e., T_{rl2} triggers refresh for (k_4, k_5, k_6, k_7) with ϕ=0 while T_{rl1} triggers refresh (k_0, k_1, k_2, k_3) with ϕ=32ms. Once refresh finishes, the refresh unlock tasks T_{ru1} and T_{ru2} update corresponding memory color to be available again.

Fig. 5.5 depicts execution of this task set with our CRS. We observe that 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 DRAM refresh finishes.

5.3.6 Copy Task

To schedule short tasks, we next propose the concept of a "copy task". A short task is defined as one whose period is less than the burst refresh cost. Such tasks are not schedulable in [BM10].

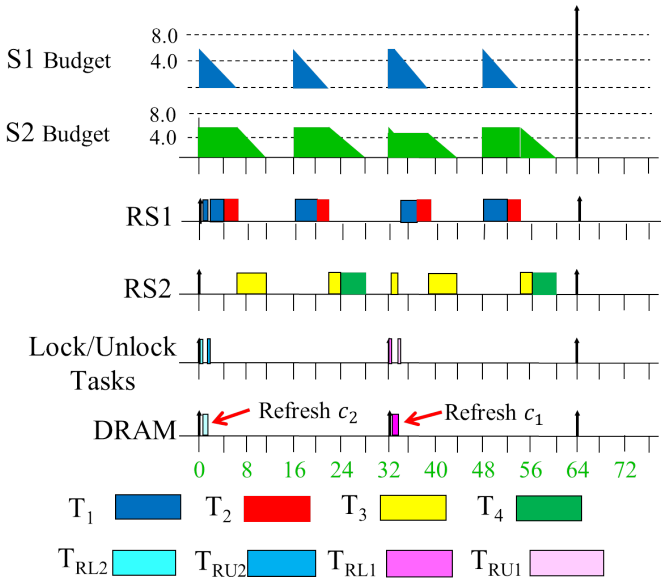

Figure 5.5 Server scheduling example

Our novel approach creates two instances of a task, the original one and a so-called "copy tasks". The two instances have identical control flow, but their data is referencing memory allocated from different colors so that the two instances belong to different servers. When a job of this task is released, our approach selects the instance to execute based on which server is running. Once one instance starts, its data is forwarded (copied) from another color if the last previous instance to run had been allocated to a different color than the current one. Notice that differences in colors of consecutive instances can be determined statically over the entire hyperperiod, i.e., it is possible to perform this check statically so that the copy subroutine is triggered for exactly for those instances prefixed by a different color instance. The trade-off between CRS's refresh hiding and the forwarding cost, calculated as $datasize * bandwidth$, is evaluated Section 5.5.

5.3.7 Schedulability Analysis within a Server

In this section, we analyze the schedulability of tasks within a server by modeling the "Periodic Capacity Bound" and the "Utilization Bound". For each server, $S(W, A, c, p_s, e_s)$, we can bound the periodic capacity for its period and budget that guarantees the schedulability of workload W and scheduling algorithm A. Similarly, when characterizing its period, budget and scheduling algorithm, we determine a utilization bound for its workload W that guarantees the schedulability of this server. Let us derive the periodic capacity bounds and the utilization bounds for the EDF algorithm and the RM algorithm, respectively.

EDF: For our CRS, DRAM refresh operations are performed by the refresh tasks (lock and unlock), which are outside of the servers. As a result, there is no refresh overhead within each server.

The resource demand bound function [SL03] under EDF is:

$dbf(t) = \sum_{T_i \in W} \lfloor \frac{t}{p_i} \rfloor * e_i$

(i) Periodic Capacity Bound (PCB): For a sever with period p_s and budget e_s, its lowest supply bound function during t time units $lsbf(t)$ is [SL03]:

$lsbf(t) = \frac{e_s}{p_s} * (t - 2 * (p_s - e_s))$

In order to guarantee schedulability of tasks within a server, $\forall\ 0 < t \leq H : dbf(t) \leq lsbf(t)$, where H is the hyperperiod of tasks within this server.

$dbf(t) \leq lsbf(t) = \frac{e_s}{p_s} * (t - 2 * (p_s - e_s))$.

We have $PCB = \frac{e_s}{p_s}$, where

$e_s \geq \frac{\sqrt{(t-2p_s)^2 + 8p_s * dbf(t)} - (t-2p_s)}{4}$.

(ii) Utilization Bound (UB): For a task set W, its utilization bound U_W can be calculated as $p' * U_W \leq lsbf(p')$ [SL03], where p' is the smallest period in task set W.

$U_W \leq \frac{lsbf(p')}{p'} = \frac{e_s}{p_s} * (\frac{p' - 2(p_s - e_s)}{p'}) = \frac{e_s}{p_s} * (1 - \frac{2(p_s - e_s)}{p'})$.

RM: The response time of a task is

$r_i^{(k)} = e_i + \sum_{T_k \in HP(W,T_k)} \lceil \frac{r_i^{(k-1)}}{p_k} \rceil * e_k$, as talked before, there is no refresh overhead within each server.

(i) Periodic Capacity Bound (PCB): The linear service time bound function $ltbf(t)$ represents the upper bound of service time to supply t time units of a resource [SL03]:

$ltbf(t) = \frac{p_s}{e_s} * t + 2(p_s - e_s)$.

To guarantee schedulability of tasks within a server, e.g., for T_i, the service time to supply $r_i^{(k)}$ should be less than its period, p_i, i.e.,

$ltbf(r_i^{(k)}) = \frac{p_s}{e_s} * r_i^{(k)} + 2 * (p_s - e_s) \leq p_i$.

As a result, the periodic capacity bound (PCB) is

$PCB = \frac{e_s}{p_s}$, where

$e_s \geq \frac{\sqrt{(p_i - 2p_s)^2 + 8 * p_s * r_i^{(k)}} - (p_i - 2p_s)}{4}$.

(ii) Utilization Bound (UB): For a task set W, its utilization bound, U_W, can be calculated as $U_W = \frac{e_s}{p_s} * (ln2 - \frac{p_s - e_s}{p'})$, where p' is the shortest period in task set.

5.3.8 Schedulability Analysis

In this section, we combine the analysis of the periodic capacity bound and the utilization bound" in Sec. 5.3.7 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 the two refresh lock tasks have same execution time (e_{rl}), and the two refresh unlock tasks have same execution time (e_{ru}). Compared with auto-refresh, we build a hierarchical resource model (configure the 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, our Colored Refresh Server has a better performance than auto-refresh by removing DRAM refresh overhead.

As described in Sec. 5.3.4, 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:
(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 [SL03], 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 can not 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 can not 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. 5.3.4, e_{r1} and e_{r2} represent the execution time of burst refresh for corresponding color, respectively. r_{s1}^m and r_{s2}^n can be calculated by response time analysis under fixed-priority assignment. As we showed above, 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. In this case, the overhead of refresh

operations cannot be hidden. But this overhead is a small proportion of the entire DRAM refresh cost. Besides, it is predictable and quantifiable. The refresh overhead, 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 (shown in Sec. 5.3.7). 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.

The following algorithms find the lowest refresh overhead for each server. Algorithm 7 searches the entire range of available budgets while and uses Algorithm 8 to quantify the refresh overhead. This search is performed off-line, i.e., it does not incur overhead during real-time task execution.

Algorithm 7 Optimize Refresh Overhead

```
1: Input: Two given workloads (W1 and W2) which related with two servers (S1 and S2)
2: for p1 in (0, Hyper-period of W1] do
3:     for p2 in (0, Hyper-period of W2] do
4:         for e1 in [PCB1 * p1, p1] do
5:             for e2 in [PCB2 * p2, p2] do
6:                 Calculate UB1 based on (p1,e1), UB2 based on (p2,e2), see Sec. 5.3.7
7:                 if sum_{Ti in W1} Utilization(Ti) <= UB1 and sum_{Tj in W2} Utilization(Tj) <= UB2 then
8:                     for m in [0, L/p1] do
9:                         e1 = Refresh_Overhead_Calculate(1, m, (p1, e1), (p2, e2))
10:                        Calculate r_s1^m
11:                        if r_s1^m >= p1 then
12:                            break
13:                        end if
14:                        TotalCost1 += e1
15:                    end for
16:                    for n in [0, L/p2] do
17:                        e2 = Refresh_Overhead_Calculate(2, n, (p1, e1), (p2, e2))
18:                        Calculate r_s2^n
19:                        if r_s2^n >= p2 then
20:                            break
21:                        end if
22:                        TotalCost2 += e2
23:                    end for
24:                    if TotalCost1 + TotalCost2 < min_overhead then
25:                        budget1 = e1
26:                        budget2 = e2
27:                        min_overhead = TotalCost1 + TotalCost2
28:                    end if
29:                end if
30:            end for
31:        end for
32:    end for
33: end for
34: return budget1 and budget2
```

Algorithm 8 Refresh_Overhead_Calculate

```
1: Input:index, i, (p_1, e_1), (p_2, e_2)
2: for k in [0, L/tRET] do
3:     if index==1 then
4:         for n in [0, L/p_2] do
5:             if (i+1)*p_1 > tRET*k > i*p_1 and (n+1)*p_2 > tRET*k > n*p_2 then
6:                 calculate r^i_s1 and r^n_s2
7:                 if tRET*k − p_1*i ≤ r^i_s1 and tRET*k − p_2*n ≥ r^n_s2 then
8:                     return e_r1
9:                 else
10:                    return 0
11:                end if
12:            end if
13:        end for
14:    end if
15:    if index==2 then
16:        for m in [0, L/p_1] do
17:            if (m+1)*p_1 > tRET*k > m*p_1 and (i+1)*p_2 > tRET*k > i*p_2 then
18:                calculate r^m_s1 and r^i_s2
19:                if tRET*k − p_1*m ≤ r^m_s1 and tRET*k − p_2*i ≥ r^i_s2 then
20:                    return e_r2
21:                else
22:                    return 0
23:                end if
24:            end if
25:        end for
26:    end if
27: end for
```

With Algorithm 7 and 8, our CRS reduces the refresh overhead by selecting appropriate periods and budgets for each server. Compared to the response time under auto-refresh, CRS can obtain 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 [BM10], 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 (see Sec. 5.3.6). The cost ($datasize * bandwidth$) can be modeled into response time of one sever if it has a copy task. However, as the discussion in Section 5.5 will show, 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.

5.4 Implementation

5.4.1 System Architecture

CRS has been implemented in a simulation environment of three components, a CPU simulator, a scheduler combined with a coloring tool, and a DRAM simulator. SimpleScalar 3.0 [Bur96] 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. 5.6). 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.

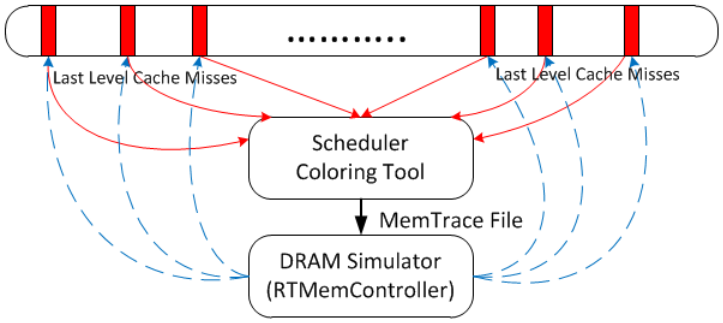

Figure 5.6 System Architecture

After scheduling and coloring, the memory traces are exposed to the DRAM simulator, RTMemController [Li14b], to analyze the DRAM performance. All memory transactions 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.

The 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. Once all DRAM refreshes have been issued by a refresh task, a relative timer with offset $tRFC$ (refresh blocking time) is installed 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. This flags 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 refereed 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 no atomic instructions or additional queues are required.

5.4.2 Coloring Tool

To hide the refresh overhead for real-time systems, our approach requires that each task be assigned a memory color via colored memory allocation. We ported TintMalloc [Pan16] to SimpleScalar so that it can select the color of physical addresses in memory. In the experiments, the entire DRAM is split into two colors corresponding to the two servers, and each application is assigned to one of them. We can adjust the number of ranks associated with one color, e.g., in order to meet an 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 it is checked if it belongs 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. Example: Consider a total of 8 ranks, let ranks 0-3 belong to color_1 while ranks 4-7 are in color_2. When an application is assigned to color_1, TintMalloc ensures that all its pages are in the 0-3 rank range by resetting bits 15-17 of the physical address (in the page range). To avoid duplicated physical addresses, TintMalloc's port not only changes the rank ID of the physical address, but also assigns this address to a new free page of the corresponding color. We further retain page locality of physical addresses, i.e., if two physical addresses originally reside in the same page, they still share the same page after coloring. Once applications are colored this way, all physical addresses of a trace belong

to a particular memory segment (color), and a task only accesses this specific area as per coloring policy.

5.4.3 Discussion

This paper shows that the refresh overhead of a periodic real-time task set on a single processor can be hidden by our CRS. CRS could be generalized to multicore platforms under partitioned parallel scheduling of tasks with respect to cores. CRS could simply schedule the subset of tasks associated with the partition of a given core using CRS' hierarchical server model on a per-core basis, where servers receive different memory colors to guarantee when their allocated colors are not being refreshed while a server executes.

We evaluate our approach in a hardware simulator. But unicore software refresh control has been demonstrated on different hardware platforms [BM10], and CRS could be implemented with similar software refresh controls on such platforms (with some engineering overhead). DRAM refreshes are synchronous with the processor clock (if the clock is fixed) and can, in fact, optionally be disabled for a subset of ranks on contemporary single- and multi-core systems [Tov17]. Furthermore, the phase when a per-rank hardware refresh start could be reverse engineered by monitoring access latencies during the initialization of a CRS-controlled system on these platforms.

5.5 Evaluation Framework and Results

The experimental evaluation assesses the performance of CRS relative to standard DRAM auto-refresh in four experiments. The first investigates the memory performance enhancements of CRS. The second illustrates how CRS eliminates the refresh delay from execution times and guarantees the schedulability of real-time system. The third and fourth compare CRS with DDR4 Fine Granularity Refresh (FGR) and previous work, respectively.

5.5.1 Experimental Setup

We assess the Malardalen WCET benchmark programs [Gus10] atop SimpleScalar 3.0 [Bur96] combined with RTMemController [Li14a]. 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. Refresh commands are issued by memory controllers at rank granularity.

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

Table 5.2 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

(single-level priority scheduling). Each Malardalen task's execution time and period (deadline) are shown in Table 5.2. 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 5.2 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 obtain better performance than auto-refresh under static priority scheduling. In this section, our CRS is evaluated by scheduling the above task set with our CRS and standard EDF policy (no constraint).

The hyper-period of the task set in Table 5.2 is 40ms, and the task set is schedulable under EDF without considering refresh overhead ("refresh-free"). CRS segregates each Malardalen application into one of the two servers. As Sec.5.3.8 shown, Algorithms 7 and 8 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 [Pan16] 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.

5.5.2 Memory Performance

Fig. 5.7 shows the normalized memory access latency (y-axis) of auto-refresh compared to CRS for all benchmarks at different DRAM densities (x-axis). The red/upper line inside the boxes indicates the median while the green/lower line represents the average across the 5 tasks. The "whiskers" above/below the box indicate the maximum and minimum.

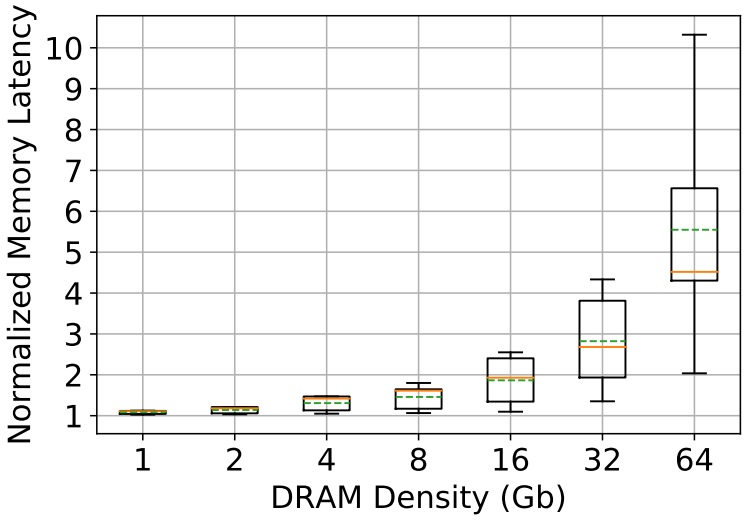

Figure 5.7 Normalized Memory Latency of Auto-Refresh relative to CRS

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. When this memory subspace needs to be refreshed, the respective server is suspended so that the other server may execute, which accesses memory of opposite color (not subject to refresh). In short, CRS co-schedules servers and refresh tasks such that any memory subspace can either be accessed by a processor or refreshed by the DRAM controller, but not by both at the same time. 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.

Auto-refresh not only increases memory access latency, it also causes memory performance to highly fluctuate across applications. Fig. 5.7 shows that different tasks suffer different latency penalties dependent on their memory access patterns. E.g., for a density of 16Gb, “compress” suffers a 9.7% increased latency while “cnt” suffers more than 154% increased latency. 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. 5.8 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 eliminates this variability.

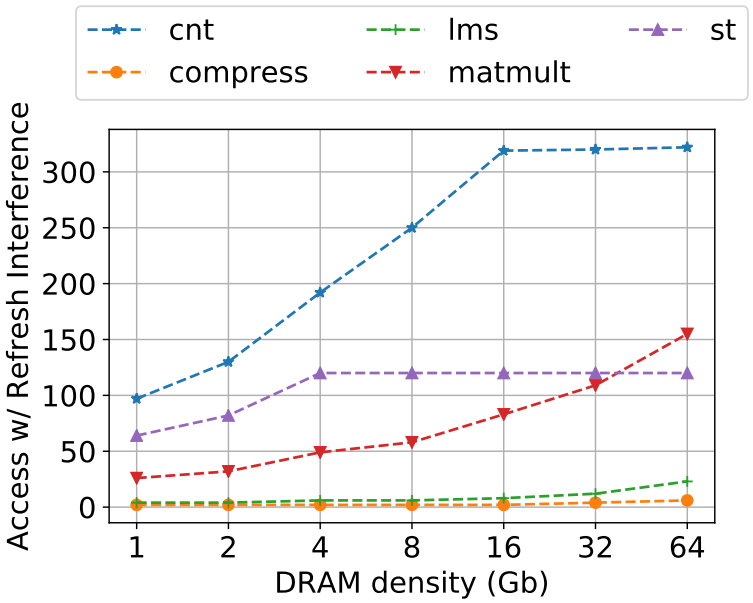

Figure 5.8 Number of Memory Accesses with Refresh Interference

5.5.3 System Schedulability

Let us compare the execution time of each task under auto-refresh and CRS. Fig. 5.9 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 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 8 Gb and by 22% at 64Gb for auto-refresh. Furthermore, the execution time of each application under CRS is the same irrespective of DRAM density, i.e., it remains constant. Since there is no refresh blocking anymore, changing density has no effect on performance.

Fig. 5.10 depicts the overall system utilization (y-axis starting at 0.93) over 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. Auto-refresh experiences from higher utilization than CRS due to the longer execution times of tasks, which increases with density to the point where it exceeds 1 such that deadlines are missed at 64 Gb.

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.

Observation 3: Compared to auto-refresh, CRS reduces the execution time of tasks and enhances system utilization by eliminating refresh overheads completely, which increases predictability while preserving real-time schedulability. Furthermore, the performance of CRS remains stable and

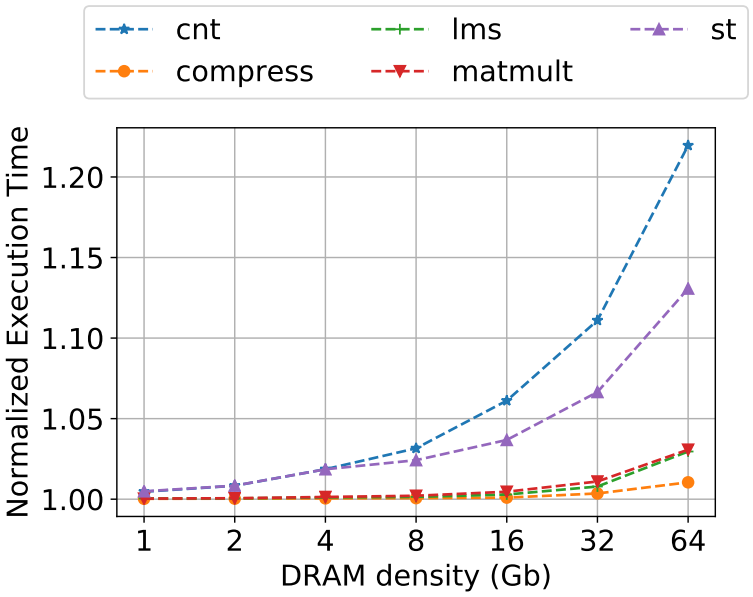

Figure 5.9 Execution Time of Auto-Refresh relative to CRS

predictable irrespective of DRAM density while auto-refresh experiences from increased overheads as density grows.

5.5.4 Fine Granularity Refresh

JEDECs DDR4 DRAM specification [Sta12a] introduces a Fine Granularity Refresh (FGR) that attempts to tackle increases in DRAM refresh overhead by creating a range of refresh options to provide a trade-off between refresh latency and frequency. We compared CRS with three FGR refresh options, namely the 1x, 2x, and 4x refresh modes. 1x 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 is 7.8us [Sta12a]. 2x and 4x require refresh commands to be sent twice and four times as frequently, respectively. The interval, $tREFI$ is correspondingly reduced to 3.9us and 1.95us for 2x and 4x, respectively. More refresh commands mean fewer DRAM rows are refreshed per command, and, as a result, the refresh latencies, $tRFC$, for 2x and 4x are shorter. However, when moving from 1x to 2x and then 4x, while $tREFI$ scales linearly, $tRFC$ does not. Instead, $tRFC$ decreases at a rate of less than 50% [Muk13].

Fig. 5.11 depicts memory access latency (y-axis) normalized to CRS over DRAM densities (x-axis) for FGR 1x, 2x, and 4x. We observe that although 4x outperforms 1x and 2x, our approach uniformly provides the best performance and lowest memory access latency due to elimination of refresh blocking. After all, CRS hides the entire refresh operation while FGR reduces the refresh blocking time. Furthermore, the performance of FGR decreases with growing DRAM density. E.g., at 64 Gb

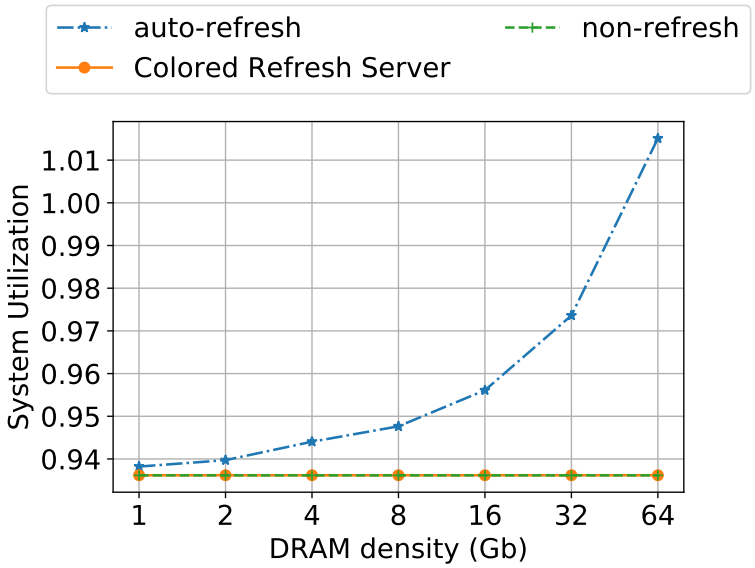

Figure 5.10 System Utilization

density, memory requests suffer an additional 17.6%, 20.7%, and 30.8% delay under FGR 4x, 2x and 1x, respectively relative to CRS. This cost increases to 343.7%, 376.4%, and 454.8% at 64Gb. CRS, in contrast, hides refresh costs so that memory access latencies remain the same irrespective of DRAM densities.

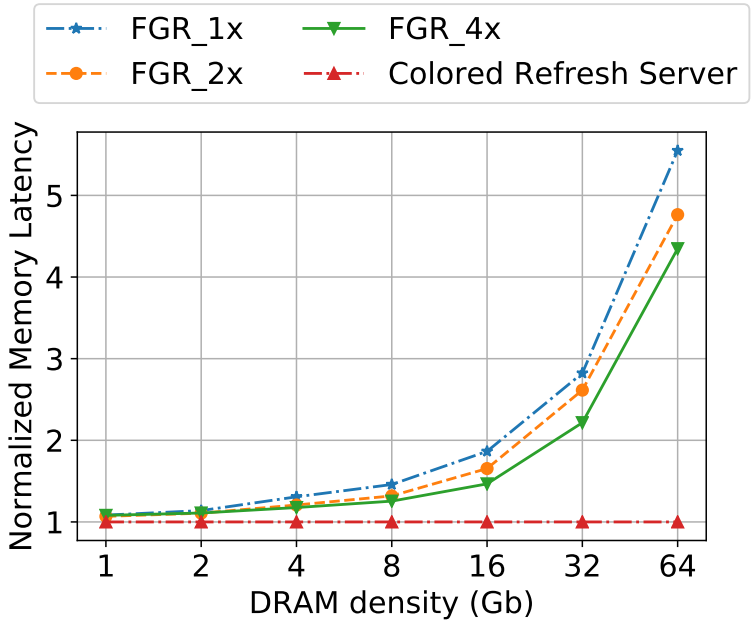

Figure 5.11 Memory Latency under FGR Schemes Normalized to CRS

Observation 4: CRS exhibits better performance and higher task predictability than DDR4's FGR.

5.5.5 Comparison with Prior Work

Bhat et al. [BM10] utilized burst patterns to reduce refresh delay and increase timing predictability. We compare the performance of CRS with the "burst-refresh" policy of [BM10]. Fig. 5.12 depicts the memory access latency (y-axis) normalized to CRS under for different DRAM densities (x-axis) for three refresh schemes. We observe that burst-refresh has a better performance than standard auto-refresh since it reduces blocking by preempting lower priority tasks while refreshing. But it cannot reduce the cost of refresh operations, which by far exceeds the interference delay. As a result, the performance of burst-refresh still suffers as it decreases rapidly with growing DRAM density. In contrast, CRS not only incurs a constant preemption cost to issue the DRAM burst, but then resumes tasks while some ranks are refreshed while the other ranks are accessed by the tasks, which effectively hides the cost of refresh. Hence, our approach outperforms burst-refresh. As mentioned before, memory access latencies remain constant under CRS irrespective of density.

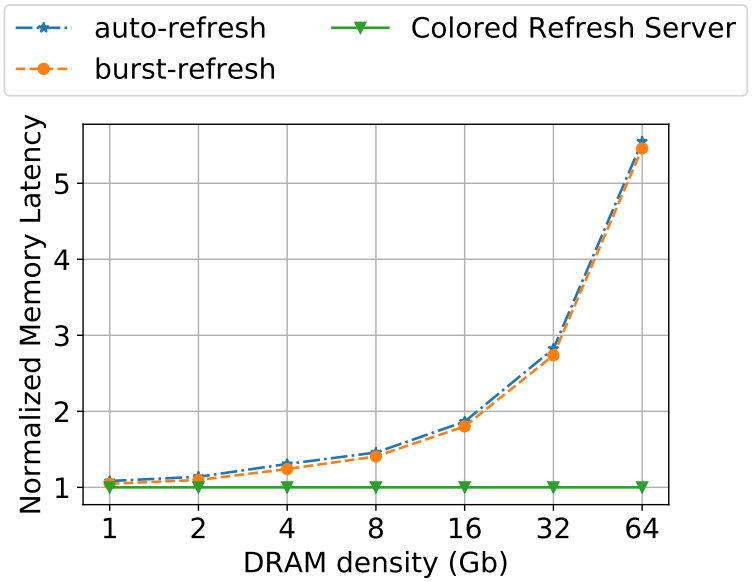

Figure 5.12 Memory Latency per Refresh Scheme Normalized to CRS

In addition, a task cannot be scheduled under Bhat's approach [BM10] if its period is less than the execution time of a burst refresh. However, such a task can be scheduled under CRS by "task copying" (see Sec. 5.3.6). The cost of task copying is extremely small, as quantified by $\frac{globalMem}{bandwidth}$. Here, $globalMem$ denotes the cumulative size of global variables that need to be copied from a current to the next job's memory space, while $bandwidth$ represents the memory bandwidth. We can determine if a short 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 and smartphones [JED10; Sta12a]. 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 250x250 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 since the execution time exceeds 1ms so that this task's period also has to be larger than the refresh duration. Thus, we conjecture that for a real-time task with 1ms execution time and a short period in the same range, 0.5MB is quite sufficient to forward outputs of one job to the next.

Observation 5: CRS obtains better performance and higher task predictability than burst refresh of the closest prior work, and CRS can schedule short tasks which prior work cannot [BM10].

5.6 Related Work

Contemporary DRAM specifications indicate increasing refresh latencies [JED10; Sta12a], which prompted researcher to search for solutions. Recent works [Bha16; Emm08; Liu13; KL09; Nai13; Ham98] analyze DRAM refresh mechanism and quantify its penalty. The refresh overhead for DDR3+4 DRAM with high densities is discussed by Mukundan et al. [Muk13]. While some focus on hardware to reduce the impact of DRAM refreshes [Zhe08; Cha14; KP00; Rei], others assess the viability of software solutions since hardware solutions take a long time before they become widely adopted.

Liu et al. [Liu12a] propose Retention-Aware Intelligent DRAM Refresh (RAIDR), which reduces refresh overhead by using knowledge of cell retention times. By exploiting the variation of DRAM cell retention time, RAIDR groups DRAM cells into several bins based on the measured minimum retention time cross all cells in a corresponding bin. Refresh overhead is reduced by RAIDR since rows are refreshed at different rates based on which bin they belong to. However, the retention time of a DRAM cell is sensitive to temperature, voltage, internal DRAM noise, manufacturer variability, and data access patterns. It may be risky to schedule refreshes at intervals beyond DRAM specifications as the retention time of cells is at least variable, if not unstable. Retention-Aware Placement in DRAM (RAPID) [Ven06] 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, RAPID not only suffers from similar risks as RAIDR, but also heavily relies on high memory locality to obtain better performance.

Smart Refresh [GL07] identifies and skips unnecessary refreshes by maintaining a refresh-needed counter. With this counter, a row that has been read or written since a refresh need not be refreshed again. Thus, memory performance is enhanced since the total number of refreshes is reduced. However, the performance of Smart Refresh heavily relies on knowledge about the data access pattern and has a high die space cost to maintain the refresh-needed counters. Liu et al. proposed Flikker [Liu11], a selective DRAM refresh that uses a reference bit per row to record and determine if this row needs to be refreshed. Rows that are labeled "non-critical" will not be refreshed in order to reduce unnecessary refreshes. But the performance of Selective DRAM Refresh still heavily depends on the data access pattern. 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. [Bha15] propose an new DRAM refresh architecture, which 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 eliminate refresh overhead completely, and it suffers from increased refresh latency for larger DRAM density/sizes.

Elastic Refresh [Stu10] 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. 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 CRS hides all refresh delays for regular memory accesses, and its performance is not affected by increasing DRAM density. Bhat et al. [BM10] 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. 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 refresh, and any stalls due to memory references at this time increase execution time, an overhead that rises rapidly with growing DRAM density/size. Although memory latency is predictable, memory throughput is still lower than CRS due to refresh blocking. Furthermore, a task cannot be scheduled if its period is less than the duration of the burst refresh.

5.7 Conclusion

We identify the problem of DRAM refresh interference with memory references and its adverse effect on predictability and performance for real-time systems. A novel scheduling server, CRS, is developed that eliminates DRAM refresh overheads via a software solution for refresh scheduling in real-time systems. By distributing tasks into servers under CRS and recharging DRAM cells via refresh tasks, memory accesses and DRAM refreshes are co-scheduled through colored memory allocation. Thus, a memory space can either be accessed by a processor or be subject to refresh at any given time, but not both. Thus, the WCET of real-time tasks is reduced and becomes more predictable under CRS compared to standard auto-refresh, since the refresh activity is hidden during task

execution. Experimental results confirm that our approach eliminates refresh overhead completely and enhances memory throughput. Compared to previous work, CRS can be implemented with low overhead, and this overhead remains constant irrespective of DRAM density. Our evaluation shows that CRS outperforms DDR4 Fine Granularity Refresh and other prior work. CRS could be integrated with any real-time scheduling policy supported inside the CRS servers with few (if any) modifications. It can be implemented on commercial off-the-shelf (COTS) systems, even in the absence of hardware support for interrupts upon refresh completion. Overall, CRS increases the predictability of memory latency in real-time systems by eliminating blocking due to DRAM refreshes, even for future DRAM sizes with higher density.

CHAPTER

6

CONCLUSION AND FURTHER WORK

In this chapter, we present the conclusions drawn from our work and the scope of future work.

6.1 Conclusion

This work contributes four approaches (CAMC, TintMalloc, Colored Refresh, and Colored Refresh Server) to avoid remote memory accesses, reduce memory bank/LLC contention, and remove DRAM refresh overhead. CAMC is designed as a controller-aware memory coloring allocator for multicore mixed criticality, weakly hard, and soft real-time systems. It comprehensively considers memory node and bank locality to color the *entire* memory space. Coloring at the kernel level is activated via a command line utility prior to task creation. Subsequently, applications are receiving colored pages for private heap, stack, static, and instruction segments automatically *without* any source code modifications. To complement CAMC, LLC coloring is considered in the second approach: TintMalloc is a controller-aware memory and LLC coloring allocator for multicore NUMA systems. TintMalloc comprehensively considers memory node, bank and LLC locality to color main memory and cache space without requiring hardware modifications.

Furthermore, the impact of DRAM refresh delay on the predictability and performance is discussed for real-time systems. We develop Colored Refresh, a novel refresh scheme to hide DRAM refresh overhead for real-time cyclic executives. The primary idea of Colored Refresh is to coordinate memory accesses and refreshes via memory coloring. With Colored Refresh, a memory rank is either

accessed by a processor or refreshed by the DRAM controller at any time. Thus, the WCET of real-time tasks is reduced and becomes more predictable under colored refresh compared to standard auto-refresh. To extend Colored Refresh, we develop Colored Refresh Server for all real-time scheme policies. By distributing tasks into servers under CRS and recharging DRAM cells via refresh tasks, memory accesses and DRAM refreshes are coscheduled through colored memory allocation. Thus, a memory space can either be accessed by a processor or be subject to refresh at any given time, but not both. Thus, the refresh activity is hidden during task execution, and the predictability of memory latency in real-time systems is increased even for future DRAM sizes with higher density.

We assess these approaches in a number of experiments with microbenchmarks as well as standard benchmarks (such as SPEC, Parsec, and Malardalen). Experimental results indicate the following: (1) Accesses to a remote memory node can be avoided for all tasks while bank and LLC access conflicts are reduced. (2) Parallel tasks become more balanced and system performance is enhanced by reducing overall runtime and idle time at barriers. (3) Refresh overhead is completely eliminated for task execution, thus enhancing the memory throughput. Overall, these approaches provide more predictability and better performance than the standard buddy allocator and outperform previous work for NUMA x86 platforms.

6.2 Further Work

This work can be extended in three aspects: (1) Coordination between our current coloring allocator and Intel Cache Allocation Technology (CAT) [Cor15], (2) optimization for interleaved memory, and (3) generalization of the Colored Refresh Server to multi-core platforms.

`Synchronization of coloring and CAT`: To help prevent last level cache (LLC) contention from occurring, Intel has developed CAT to enable more control over the LLC cache and how cores use it. Using CAT, one can reserve portions of the cache for individual cores so that only these cores can allocate into them. As a result, other applications may not evict cache lines from these reserved portions of the cache. In future, this technology work could be explored in detail and combined with our coloring allocator. The cache is divided into ways (vertical partition) and these ways can be divided or shared among cores with CAT. In contrast, our coloring allocator partitions LLC horizontally. By synchronizing both, the cache can be divided at a finer granularity, and we can explore its advantage for multicore systems.

`Interleaved memory optimization`: In computing, interleaved memory is a design made to compensate for the relatively slow speed of dynamic random-access memory (DRAM) or core memory, by spreading memory addresses evenly across memory banks. That way, contiguous memory reads and writes are using each memory bank in turn, resulting in higher memory throughput due to reduced waiting for memory banks to become ready for desired operations. However, on modern NUMA architectures, interleaved memory causes more *remote* memory accesses if banks

are on the remote memory nodes. Furthermore, even on the local memory node, memory bank contention is also increased in multicore systems due to spreading data crossed banks. Future work can solve these problems via novel coloring techniques for multicore systems on NUMA architectures.

`Multi-core Colored Refresh Server`: Multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU). Microprocessors currently used in almost all personal computers are multi-core. On multi-core systems, all cores share the main memory and suffer DRAM refresh interference. Our current approaches, both Colored Refresh and Colored Refresh Server, are implemented on a single core processor. Future work can extend the Colored Refresh Server to multi-core platforms, and consider more constraints within the hierarchical resource model, such as memory bandwidth and last level cache. Furthermore, both Colored Refresh and Colored Refresh Server are implemented in a simulator. Some previous works [Tov17; BM10] control the DRAM refresh on an real system. In future work, the our refresh schemes could be implemented on actual hardware platforms as well. The scheduling policies of Colored Refresh and Colored Refresh Server can be realized by modifying the scheduler inside of the operating system's kernel (for hard real-time systems) or that of user-level scheduling libraries (for soft or non real-time systems).

BIBLIOGRAPHY

[AG89] Almasi, G. S. & Gottlieb, A. *Highly Parallel Computing. Series in Computer Science and Engineering*. 1989.

[AP01] Atanassov, P. & Puschner, P. "Impact of DRAM refresh on the execution time of real-time tasks". *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*. 2001.

[Awa10] Awasthi, M. et al. "Handling the problems and opportunities posed by multiple on-chip memory controllers". *International Conference on Parallel Architectures and Compilation Techniques*. 2010, pp. 319–330.

[BS88] Baker, T. P. & Shaw, A. "The cyclic executive model and Ada". *Real-Time Systems Symposium*. 1988.

[BM10] Bhat, B. & Mueller, F. "Making DRAM refresh predictable". *Euromicro Conference on Real-Time Systems*. 2010, pp. 145–154.

[Bha15] Bhati, I. et al. "Flexible auto-refresh: enabling scalable and energy-efficient DRAM refresh reductions". *AInternational Symposium on Computer Architecture*. ACM. 2015.

[Bha16] Bhati, I. et al. "DRAM refresh mechanisms, penalties, and trade-offs". *IEEE Transactions on Computers* **65**.1 (2016), pp. 108–121.

[Bla10] Blagodurov, S. et al. "A case for NUMA-aware contention management on multicore systems". *International Conference on Parallel Architectures and Compilation Techniques*. 2010, pp. 557–558.

[Bro10] Broquedis, F. et al. "Structuring the execution of OpenMP applications for multicore architectures". *International Parallel & Distributed Processing Symposium*. IEEE. 2010, pp. 1–10.

[Bur96] Burger, D. et al. "Evaluating Future Microprocessors:The Simplescalar Toolset". CS-TR-96-1308. 1996.

[But97] Butenhof, D. R. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[Cha14] Chang, K. K.-W. et al. "Improving DRAM performance by parallelizing refreshes with accesses". *International Symposium on High Performance Computer Architecture*. IEEE. 2014, pp. 356–367.

[Cha08] Chapman, B. et al. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008.

[Chi15] Chisholm, M. et al. “Cache sharing and isolation tradeoffs in multicore mixed-criticality systems”. *IEEE Real-Time Systems Symposium*. 2015.

[CJ06] Cho, S. & Jin, L. “Managing distributed, shared L2 caches through OS-level page allocation”. *MICRO-39*. IEEE Computer Society. 2006, pp. 455–468.

[Cor15] Corporation, I. “Improving Real-Time Performance by Utilizing Cache Allocation Technology” (2015).

[DB11] Davis, R. I. & Burns, A. “A survey of hard real-time scheduling for multiprocessor systems”. *ACM computing surveys (CSUR)* (2011).

[Emm08] Emma, P. G. et al. “Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications”. *IEEE micro* (2008), pp. 47–56.

[FM02] Feng, X. & Mok, A. K. “A model of hierarchical real-time virtual resources”. *Real-Time Systems Symposium*. 2002.

[GL07] Ghosh, M. & Lee, H.-H. S. “Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs”. *IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2007, pp. 134–145.

[Gus10] Gustafsson, J. et al. “The Mälardalen WCET Benchmarks – Past, Present and Future”. *International Workshop on Worst-Case Execution Time Analysis (WCET)*. 2010, pp. 137–147.

[Ham98] Hamamoto, T. et al. “On the retention time distribution of dynamic random access memory (DRAM)”. *IEEE Transactions on Electron devices* (1998).

[Hua15] Huang, P. et al. “An Isolation Scheduling Model for Multicores”. *IEEE Real-Time Systems Symposium*. 2015.

[JED10] JEDEC STANDARD. “DDR3 SDRAM SPECIFICATION” (2010).

[Kim11] Kim, H. et al. “Characterization of the variable retention time in dynamic random access memory”. *IEEE Transactions on Electron Devices* (2011).

[Kim14] Kim, H. et al. “Bounding memory interference delay in COTS-based multi-core systems”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2014.

[KP00] Kim, J. & Papaefthymiou, M. C. “Dynamic memory design for low data-retention power”. *Lecture notes in computer science* (2000).

[KL09] Kim, K. & Lee, J. “A new investigation of data retention time in truly nanoscaled DRAMs”. *IEEE Electron Device Letters* (2009), pp. 846–848.

[Kim10a] Kim, Y. et al. “ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers”. *International Symposium on High Performance Computer Architecture*. IEEE. 2010, pp. 1–12.

[Kim10b] Kim, Y. et al. “Thread cluster memory scheduling: Exploiting differences in memory access behavior”. *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE. 2010, pp. 65–76.

[Lac12] Lachaize, R. et al. “MemProf: A Memory Profiler for NUMA Multicore Systems.” *USENIX Annual Technical Conference*. 2012, pp. 53–64.

[Li93] Li, H. et al. “Locality and loop scheduling on NUMA multiprocessors”. *International Conference on Parallel Processing*. Vol. 93. 1993, pp. 140–147.

[Li14a] Li, Y et al. *RTMemController: Open-source WCET and ACET analysis tool for real-time memory controllers*. `http://www.es.ele.tue.nl/rtmemcontroller/`. 2014.

[Li14b] Li, Y. et al. “Dynamic command scheduling for real-time memory controllers”. *Euromicro Conference on Real-Time Systems*. IEEE. 2014, pp. 3–14.

[LB03] Lipari, G. & Bini, E. “Resource partitioning among real-time applications”. *Euromicro Conference on Real-Time Systems*. 2003.

[Liu00] Liu, J. *Real-Time Systems*. Prentice Hall, 2000.

[Liu12a] Liu, J. et al. “RAIDR: Retention-aware intelligent DRAM refresh”. *ACM SIGARCH Computer Architecture News*. Vol. 40. 3. IEEE. 2012, pp. 1–12.

[Liu13] Liu, J. et al. “An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms”. *International Symposium on Computer Architecture*. ACM. 2013.

[Liu] Liu, J. W. *Real-time systems. 2000*. Prentice Hall.

[Liu12b] Liu, L. et al. “A software memory partition approach for eliminating bank-level interference in multicore systems”. *International Conference on Parallel Architectures and Compilation Techniques*. 2012, pp. 367–376.

[Liu11] Liu, S. et al. “Flikker: saving DRAM refresh-power through critical data partitioning”. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 213–224.

[Loc92] Locke, C. D. “Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives”. *Real-time systems* **4**.1 (1992), pp. 37–53.

[MG11] Majo, Z. & Gross, T. R. “Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead”. *International Symposium on Memory Management*. Vol. 46. 11. ACM. 2011, pp. 11–20.

[MG12] Majo, Z. & Gross, T. R. “Matching memory access patterns and data placement for NUMA systems”. *International Symposium on Code Generation and Optimization*. 2012, pp. 230–241.

[MG13] Majo, Z. & Gross, T. R. “(Mis) understanding the NUMA memory system performance of multithreaded workloads”. *International Symposium on Workload Characterization*. 2013, pp. 11–22.

[Man15] Mancuso, R. et al. “WCET(m) Estimation in Multi-Core Systems using Single Core Equivalence”. *Euromicro Conference on Real-Time Systems*. 2015.

[Mar10] Marathe, J. et al. “Feedback-directed page placement for ccNUMA via hardware-generated memory traces”. *Journal of Parallel and Distributed Computing* **70**.12 (2010), pp. 1204–1219.

[MV10] McCurdy, C. & Vetter, J. “Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms”. *International Symposium on Performance Analysis of Systems & Software*. 2010, pp. 87–96.

[Mi10] Mi, W. et al. “Software-hardware cooperative DRAM bank partitioning for chip multiprocessors”. *Network and Parallel Computing*. Springer, 2010, pp. 329–343.

[Mor05] Mori, Y. et al. “The origin of variable retention time in DRAM”. *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*. IEEE. 2005.

[Muk13] Mukundan, J. et al. “Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems”. *International Symposium on Computer Architecture*. ACM. 2013.

[Mur11] Muralidhara, S. P. et al. “Reducing memory interference in multicore systems via application-aware memory channel partitioning”. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 374–385.

[Nai13] Nair, P. et al. “A case for refresh pausing in DRAM memory systems”. *High Performance Computer Architecture*. IEEE. 2013, pp. 627–638.

[Oga09] Ogasawara, T. “NUMA-aware memory manager with dominant-thread-based copying GC”. *Conference on Object Oriented Programming Systems, Languages and Applications*. 2009, pp. 377–390.

[Pan16] Pan, X. et al. “TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring”. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pp. 363–372.

[PY16] Pellizzoni, R. & Yun, H. “Memory Servers for Multicore Systems”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2016.

[Per11] Perarnau, S. et al. “Controlling cache utilization of hpc applications”. *Proceedings of the international conference on Supercomputing*. ACM. 2011, pp. 295–304.

[Rei] Reineke, J. et al. “PRET DRAM controller: Bank privatization for predictability and temporal isolation”. *CODES+ISSS*.

[She09] Shen, X. Z. S. D. K. “Hardware execution throttling for multi-core resource management”. *USENIX Annual Technical Conference*. 2009.

[SL03] Shin, I. & Lee, I. “Periodic resource model for compositional real-time guarantees”. *RTSS*. 2003.

[Sta12a] Standard, JEDEC. “DDR4 SDRAM” (2012).

[Sta12b] Standard, JEDEC. “LPDDR3 SDRAM Specification” (2012).

[Sta12c] Standard, JEDEC. “LPDDR3 SDRAM Specification” (2012).

[Stu10] Stuecheli, J. et al. “Elastic refresh: Techniques to mitigate refresh penalties in high density memory”. *IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2010, pp. 375–384.

[Sud10] Sudan, K. et al. “Micro-pages: increasing DRAM efficiency with locality-aware data placement”. *ASPLOS* **45**.3 (2010), pp. 219–230.

[Suh04] Suh, G. E. et al. “Dynamic partitioning of shared cache memory”. *The Journal of Supercomputing* **28**.1 (2004), pp. 7–26.

[Suz13] Suzuki, N. et al. “Coordinated bank and cache coloring for temporal protection of memory accesses”. *International Conference on Computational Science and Engineering*. 2013.

[Tov17] Tovletoglou, K. et al. “Relaxing DRAM refresh rate through access pattern scheduling: A case study on stencil-based algorithms”. *23rd IEEE International Symposium on On-Line Testing and Robust System Design(IOLTS)*. 2017.

[Ven06] Venkatesan, R. K. et al. “Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM”. *International Symposium on High-Performance Computer Architecture*. IEEE. 2006, pp. 155–165.

[War15] Ward, B. C. “Relaxing Resource-Sharing Constraints for Improved Hardware Management and Schedulability”. *IEEE Real-Time Systems Symposium*. 2015.

[WM01] Wegener, J. & Mueller, F. “A comparison of static analysis and evolutionary testing for the verification of timing constraints”. *Real-Time Systems* (2001).

[Wu13] Wu, Z. P. et al. “Worst case analysis of DRAM latency in multi-requestor systems”. *IEEE Real-Time Systems Symposium*. 2013.

[Yun13] Yun, H. et al. “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2013.

[Yun14] Yun, H. et al. “PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2014, pp. 155–166.

[Yun15] Yun, H. et al. “Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems”. *Euromicro Conference on Real-Time Systems*. 2015.

[Zhe08] Zheng, H. et al. “Mini-rank: Adaptive DRAM architecture for improving memory power efficiency”. *IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2008, pp. 210–221.