Controller-Aware Memory Coloring for Multicore Real-Time Systems

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

Abstract

Memory latencies vary in non-uniform memory access (NUMA) systems so that execution times may become unpredictable in a multicore real-time system. This results in overly conservative scheduling with low utilization due to loose bounds on the worst-case execution time (WCET) of tasks. This work contributes a controller/node-aware memory coloring (CAMC) 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 Linux' buddy allocator and prior coloring methods. It provides core isolation with respect to banks and memory controllers for real-time systems. To our knowledge, this work is first to consider multiple memory controllers in real-time systems, combine them with bank coloring, and assess its performance on a NUMA architecture.

CCS Concepts • **Computer systems organization** → *Real-time systems*;

Keywords memory access, NUMA, real-time predictability

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 [22] exploits bank coloring on DRAM to allocate memory to specific DRAM banks. Kim et al. [6] 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 [18, 21]. 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

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167196

prior work for non-NUMA allocations [22], 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 [12]. 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 Background

DRAM Organization: DRAM is organized as a group of memory controllers/nodes (Fig. 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.

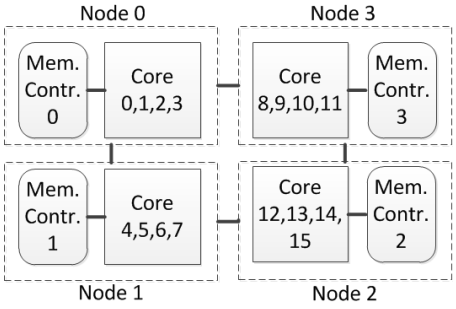

Fig. 1. 16 Cores, 4 Memory Controllers/Nodes

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 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.

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. 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.

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.

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);
```

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.

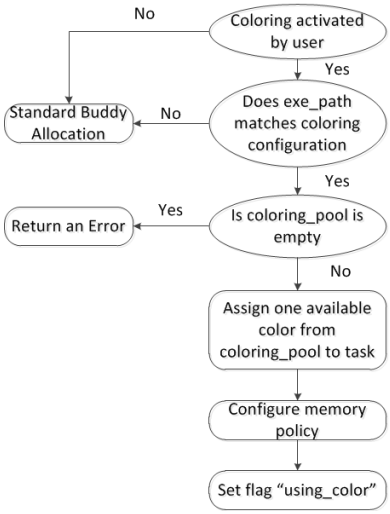

Fig. 2. Program Flow to configure memory coloring

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

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.

4 Evaluation Framework and Results

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.

4.2 CAMC vs. Buddy with Local Node Policy

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.

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 1 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 1: Memory access time is reduced and becomes more predictable with CAMC coloring.

Table 1. 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

4.3 CAMC Overhead

Table 1 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 2: 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.

4.4 System Performance

We next investigate performance and predictability for the PARSEC benchmark suite featuring multithreaded programs [2]. 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 [6, 22]. 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. 3).

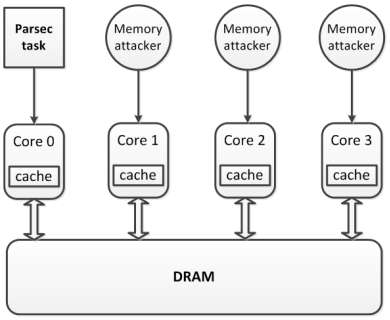

Fig. 3. Mixed: 1 Parsec code + up to 3 memory attackers

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. 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 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.

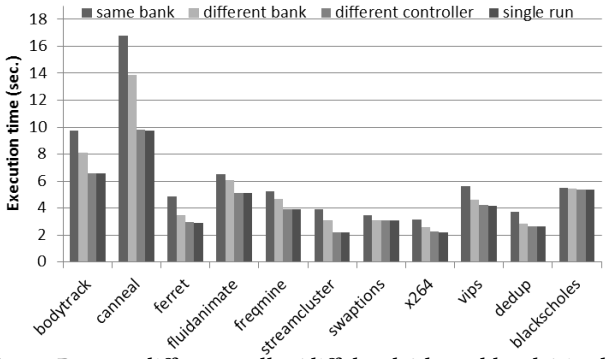

Fig. 4. Parsec: diff. controller/diff. bank/shared bank/single

Observation 3: 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.

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. 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. 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 4: 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.

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) 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. 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. 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.

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. 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.

Observation 5: 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 3+ 4 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 5 quantifies the deadline miss rates for `same-bank` and `diff-bank` while the other policies always meet deadlines.

Table 3. 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 4. 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 5. Deadline Miss Rates

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

4.6 Latency Comparison with Prior Work

We compared the performance of our approach with Palloc [22], 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 [22, 24], 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 [22]. 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).

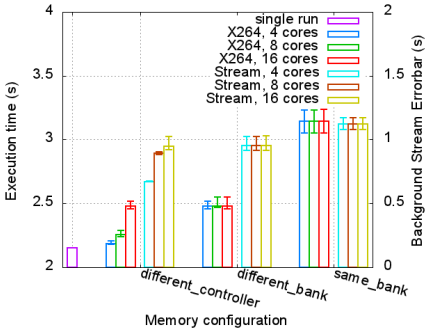

Fig. 5. Runtime of one X264 and 3/7/15 Stream Tasks

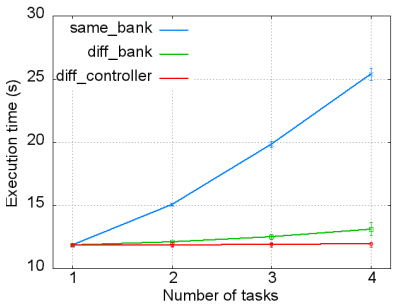

Fig. 6. Palloc: Avg. Memory Latency for Controller/Bank/no Coloring

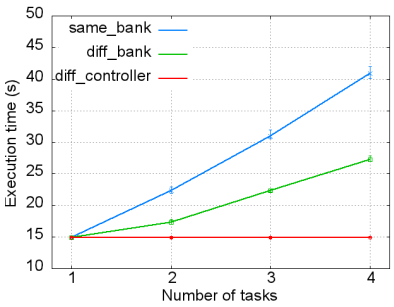

Fig. 7. Alternating Strides for Controller/Bank/no Coloring

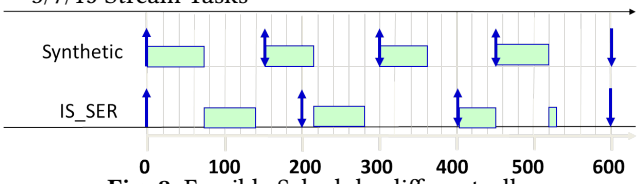

Fig. 8. Feasible Schedule: diff-controller

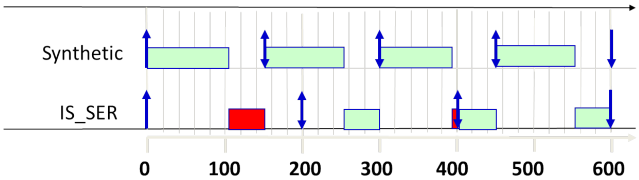

Fig. 9. Deadline Misses (red) for same-bank

Fig. 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 [22] utilizing our synthetic benchmark (striding back and forth with increasing offsets) under the same setup as for the Palloc latency benchmark. Fig. 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 [22] 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 6: 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.

5 Related Work

The performance of multithreaded programs on NUMA multicores system has been studied extensively [3, 7, 11, 13, 14, 17, 23]. Scheduling or page placement has been proposed to solve the data sharing problem in NUMA system [5, 8, 10, 15, 20]. 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 [1, 4, 6, 9, 16, 22]. Awasthi et al. [1] 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. [16] 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. [9] 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 [22] 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. [19] combine cache and bank coloring to obtain tight timing predictions. Mancuso et al. [12] 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.

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.

Acknowledgment

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

References

[1] Manu Awasthi, David W Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.

[3] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[4] Micaiah Chisholm, Bryan C Ward, Namhoon Kim, and James H Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *IEEE Real-Time Systems Symposium*, 2015.

[5] Pengcheng Huang, Georgia Giannopoulou, Rehan Ahmed, Davide B. Bartolini, and Lothar Thiele. An isolation scheduling model for multicores. In *IEEE Real-Time Systems Symposium*, 2015.

[6] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2014.

[7] Renaud Lachaize, Baptiste Lepers, Vivien Quéma, et al. Memprof: A memory profiler for numa multicore systems. In *USENIX Annual Technical Conference*, 2012.

[8] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C Sevcik. Locality and loop scheduling on numa multiprocessors. In *International Conference on Parallel Processing*, 1993.

[9] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[10] Zoltan Majo and Thomas R Gross. Matching memory access patterns and data placement for numa systems. In *International Symposium on Code Generation and Optimization*, 2012.

[11] Zoltan Majo and Thomas R Gross. (mis) understanding the numa memory system performance of multithreaded workloads. In *International Symposium on Workload Characterization*, 2013.

[12] Renato Mancuso, Rodolfo Pellizzoni, Caccamo Marco, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *Euromicro Conference on Real-Time Systems*, 2015.

[13] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccnuma via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 2010.

[14] Collin McCurdy and Jeffrey Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *International Symposium on Performance Analysis of Systems & Software*, 2010.

[15] Takeshi Ogasawara. Numa-aware memory manager with dominant-thread-based copying gc.

[16] Xing Pan, Yasaswini J. Gownivaripalli, and Frank Mueller. Tintmalloc: Reducing memory access divergence via controller-aware coloring. In *International Parallel and Distributed Processing Symposium*, 2016.

[17] Rodolfo Pellizzoni and Heechul Yun. Memory servers for multicore systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2016.

[18] Xiao Zhang Sandhya Dwarkadas Kai Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conference*, 2009.

[19] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *International Conference on Computational Science and Engineering*, 2013.

[20] Bryan C. Ward. Relaxing resource-sharing constraints for improved hardware management and schedulability. In *IEEE Real-Time Systems Symposium*, 2015.

[21] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *IEEE Real-Time Systems Symposium*, 2013.

[22] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2014.

[23] Heechul Yun, Rodolfo Pellizzoni, and Prathap Valsan, Kumar. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Euromicro Conference on Real-Time Systems*, 2015.

[24] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*, 2013.