NUMA-Aware Memory Coloring for Multicore Real-Time Systems

Xing Pan[a], Frank Mueller[a,*]

[a]*Dept. of Computer Science, North Carolina State University, Raleigh, NC 27519-8206, USA*

Abstract

Non-uniform memory access (NUMA) systems are characterized by varying memory latencies 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 a task's worst-case execution time (WCET). 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. This work is the first to consider multiple memory controllers in real-time systems, combine them with bank coloring, and assess its performance on a NUMA architecture, to the best of our knowledge.

Keywords: memory access, NUMA, real-time predictability
PACS: 07.05.Bx

1. Introduction

Hard real-time systems are control systems consisting of period tasks whose jobs perform sensing, computation and actuation actions and have rigid deadlines. Any deadline is miss may have severe impacts ranging from environmental impacts to loss of life. To ensure deadlines are met, the computational demand of each task needs to be upper bounded so that a schedulability analysis can determine if a given task set is feasible, i.e., schedulable without any deadline miss [1]. Such an upper bound on computation is derived from the worst-case execution time (WCET) of a task, which can be determined analytically or experimentally, where the former requires detailed information about the underlying hardware and results in more rigid bounds while the latter may provide probabilistic bounds based on fewer hardware details [2, 3]. One of the objectives in assessing WCET bounds is to ensure that these bounds are tight — in the sense that they only insignificantly overestimate the true WCET — as tighter bounds result in better actual processor utilization and thus a more cost effective resource usage. The tightness of these bounds is directly linked to execution time variance, which can be caused by program structure (e.g., conditional execution with alternate short and long paths or variable loop bounds), processor architecture (e.g., pipeline stalls due to structural hazards) and memory (e.g., variable latency memory references due to caches and non-uniform DRAM latency. Timing variability can be analyzed experimentally by assessing the variance (or standard deviation) of actual execution times. By reducing the standard deviation of execution times, bounds of execution times can be tightened, which allows the actual processor utilization to be increased. In the following, we will refer to "predictability" as a means to lower the metric for tightness of computational bounds, i.e., a lower variance/standard deviation in computation times of a task or, even better, constant time, implies higher predictability, while a higher variance/standard deviation results in lower predictability. The former is synonymously referred to as "predictable" and the latter as "unpredictable" behavior in the following.

One source of loose computational bounds (unpredictability) are non-uniform memory access (NUMA) architectures, which are comprised of many cores partitioned into sets of "nodes", where each node has its own local memory controller. 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 *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 or initialization on a different core forced allocation on a non-local node for subsequent accesses in

*Corresponding author
Email address: mueller@cs.ncsu.edu (Frank Mueller)

another thread running on another core. (2) Even with a single memory node, conflicts between shared-bank accesses result in unpredictable memory access latencies. As the execution time of tasks has to be conservatively (over-)estimated in real-time systems, system utilization may be low.

Recent work explores the methods to make main memory accesses more predictable. Palloc [4] exploits bank coloring on dynamic random-access memory (DRAM) to allocate memory to specific DRAM banks. Kim et al. [5] 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 [6, 7]. Programmers need to carefully assign colors to each task and manually set the coloring policy for a real-time task set. Furthermore, none of these approaches solve the problem of making memory accesses time predictable for NUMA systems. Some of them require hardware modifications while others do not consider NUMA as a source of unpredictable behavior.

Hard real-time applications tend to allocate and initialize data structures at the beginning of a a task's execution. Each periodic job invocation of a task then utilizes variables of the data structures for their sensing, computation and actuation actions. Such variables may be global variables or heap variables, pre-allocated once at initialization time and subsequently used during each job of a task. Any data structure can be interchangeably assigned to global variables or heap allocation at initialization time, as long as they are preallocated and accessed once as part of the initialization to ensure that virtual memory allocation actually results in reserving physical memory space. This is critical as allocation by itself is lazy, i.e., physical memory is reserved only once data is accessed due to the first-touch policy of virtual memory management within operating systems [8]. Our work handles coloring for global data, heap, stack and code alike while considering the effects of NUMA systems. Notice that even global variables, code segments and stacks are allocated using the same system call (mmap) as heap allocators, except that their allocation occurs during load time in code executed prior to transferring control to application code. In fact, global allocations prior to task invocations are likely to result in significant latency variation without coloring. This is again due to the first-touch policy that, as we will discuss, results in allocations local to the "main" task instead of local to a thread representing a task that is allocated on a specific core.

Memory allocation in programs relies on a system call that invokes the standard buddy allocator of the operating system. It adheres to a "node local" memory policy, which requests allocations from the memory node local to the core the code executes on from an arbitrary bank. 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, the libnuma library is restricted to heap memory placement at the controller level, and it requires explicit source code modifications to make libnuma calls. Furthermore, neither buddy allocation with local node policy nor libnuma are bank aware.

We contribute 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. The objective of CAMC is to tighten the bounds on a task's computation time and allow the utilization of a real-time system to be increased, yet to make minimal changes to the original application code to achieve this objective. 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. This is the first work to comprehensively evaluate memory coloring performance for real-time NUMA systems, to the best of our knowledge.

Summary of Contributions:

• In contrast to prior work for non-NUMA allocations [4] or "local node" policy in buddy allocation without bank awareness, CAMC colors the entire memory space transparent to the application by considering memory node and bank locality together. 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 [9]. It avoids/reduces (i) memory accesses to remote nodes and (ii) conflicts among banks when possible 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.

• We assess the performance of CAMC for Parsec codes on a standard x86 platform, with and without real-time task sets. This allows us to compare CAMC with Linux' standard buddy allocator with "local node" policy and previous coloring techniques.

• We quantify the non-uniform latency between nodes and experimentally show 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 real-

time 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 essential both to facilitate compositional analysis based on single-task analyses and to improving the schedulability of real-time task sets.

• We describe an algorithm for the mapping of physical address bits for AMD processors. Its principles can be applied to any architecture as long as the mapping of physical address bits is documented.

• CAMC automatically assigns memory colors to tasks based on global utilization of memory colors, much in contrast to prior work that requires manual configuration by programmers. 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 in a backwards portable manner by exploiting semantic restrictions of the system call.

The rest of this paper is organized as follows. Section 2 introduces the memory organization of modern multicore CPUs and discusses the problem of memory controller selection and bank sharing. Section 3 presents the design and implementation of controller-aware memory coloring. Section 4 describes the experimental platform and analyzes the overhead of the approach for micro-benchmarks and experiments with the NAS and Parsec benchmarks. Section 5 reviews related work and Section 6 summarizes the contributions.

2. Background

In this section, we briefly describe the organization of a DDR3 SDRAM system and how it services a memory accessing request.

DRAM Organization: Dynamic random-access memory (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, see Fig. 2. 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.

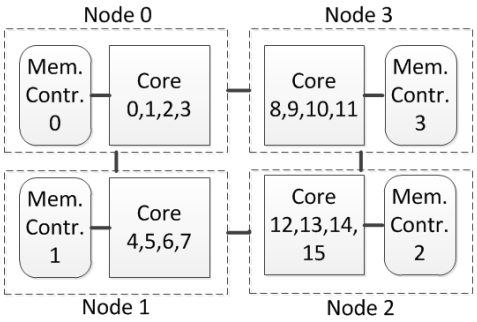

Fig. 1: 16 Cores, 4 Memory Controllers and Nodes

When 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., under bank contention the memory access time and bank miss ratios increase as access latencies fluctuate.

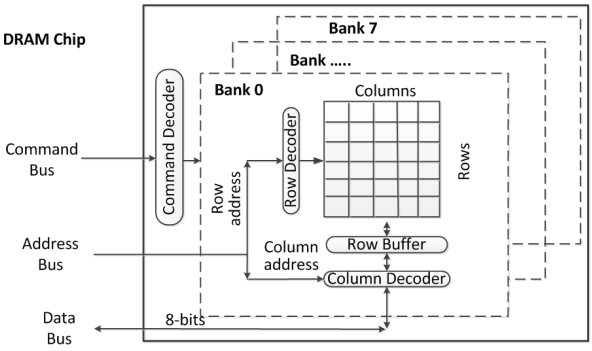

Fig. 2: DRAM Device Organization

Memory Controller: Cache misses within the last-level cache (LLC) of a processor are forwarded to the memory controller, which acts as a mediator between the LLC and the DRAM devices. The memory controller translates read/write memory requests into corresponding DRAM commands and schedules these commands while satisfying timing constraints of DRAM banks and buses, see Fig. 3. 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 of 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, proper placement of data can increase the overall memory bandwidth and decrease its latency due to node locality when remote memory accesses are avoided, i.e., both performance and predictability can be improved due to more uniform and shorter latencies. In particular, if remote node references can be completely avoided, the upper bound on access latencies can be tightened.

As briefly outlined in the last section, allocation and initialization of all global variables within the context of the initial (main) task results in all data being local to that task's core due to the aforementioned first-touch policy. Subsequent tasks allocated to other cores on different nodes consequently suffer remote access latencies when ac-

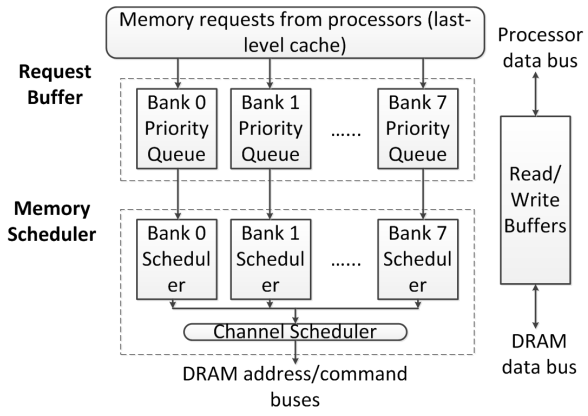

Fig. 3: Logical Structure of DRAM Controllers

cessing these global variables. One method to address this problem is to perform global allocations in the context of the specific task that later accesses this data. But this would require restructuring of the initialization phase of applications. Alternatively, the main task may indicate at allocation time which node a specific data structure should be allocated on. Our work follows this latter idea as it requires less program restructuring, i.e., the initialization code only needs to be augmented by systems calls to indicate which node the following allocations should be associated with.

3. Controller-Aware Memory Coloring (CAMC)

3.1. Motivation

Upon the first reference (touch) of a memory page during application execution, a page fault is raised by the memory management unit. This fault results in a trap, which invokes an operating system (OS) handler. The handler makes the page accessible at a new map location in physical memory if it was a legal memory address or raises a user signal in case of an illegal memory access. The OS maintains a list of available physical memory frames from which pages are dealt out in FIFO order for such new page mappings.

After a physical memory frame is allocated to an application (on first memory touch), such a frame can be accessed again until the application frees it or terminates. When a variable or instruction is accessed, its virtual addresses are translated into physical ones based on the page table maintained by the OS. An access request for a physical address will be sent to the corresponding memory controller. The DRAM memory controller then translates the physical address into a DRAM address to identify channel ID, rank ID, bank ID, row, and column. With those parameters, the DRAM cell that contains the requested data can be located. As discussed in Sec. 2, to serve the memory request, a row with the respective DRAM cell is loaded into the bank's row buffer and suffers row buffer hit/miss delay.

Accesses to memory are uniform in latency in a single node/memory controller system. However, a NUMA system with multiple nodes causes memory access latencies and contention to vary. 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 much of the accesses contend; and (3) how memory banks interleave.

A memory node consists of multiple memory banks, each of which can be accessed in parallel. Multiple tasks may share the *same* memory bank and contend for the *same* bank's row buffer under Linux' buddy allocation. As shown in Fig. 4, Tasks 1 and 2 are accessing the same DRAM bank and compete with one another on their respective memory request, i.e., task 1 requests the red data (left side arrow) while task 2 tries to access blue one (right side). In this case, task 2 replaces the row buffer that had been populated by task 1 and thereby evicts the data of task 1 in the row buffer. This subsequently increases memory access times and bank miss ratios if task 1 proceeds to access memory close by (spatial locality) so that access latencies fluctuate due to alternating row buffer hits and misses.

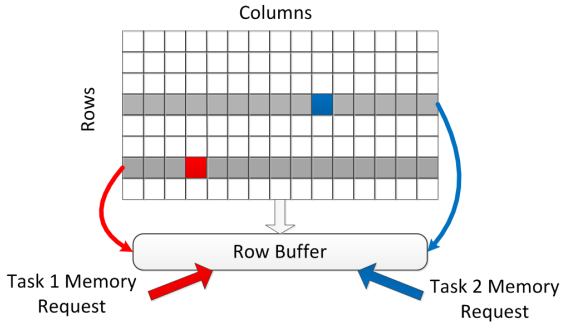

Fig. 4: Memory Bank Contention

We design Controller-Aware Memory Coloring (CAMC) to avoid remote memory node accesses and reduce bank contention. CAMC 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. The bank color, bc, of a physical frame is determined as

bc= ((node×NC+channel)×NR+rank)×NB+bank,

where node (controller), channel, (memory) rank, and (memory) bank are specific to this physical frame and

NC the number of channels per memory controller,
NR the number of ranks per channel,
NB the number of banks per rank.

Example: In a system with two nodes, two channels per node, two ranks per channel, and four banks per rank,

if we access node 1, channel 0, rank 1 and bank 3, $bc = ((1 \times 2 + 0) \times 2 + 1) \times 4 + 3 = 19$.

CAMC optimizes the physical memory frame selection process based on this partitioning to provide a private memory space for each task on their local memory node in order to make memory access latency stable and predictable. But in practice, it is hard to avoid all remote accesses as tasks run concurrently and may incur complex memory reference patterns, e.g., due to data sharing. 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 absence of controller/bank conflicts and supports locality.

3.2. Address Mapping for Page Coloring

In the following, we explain how to read PCI information (Algorithm 1), which provides the controller ranges (Tab. 1) and bit indices for ranks (Algorithm 1, sequential within the range of a controller) and banks (bits in Fig. 5). This suffices to calculate and store the color, *bc*, for each allocated frame in the page table (last paragraph of this subsection).

A physical address is translated by CAMC to a DRAM address and then mapped 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 to any documented mapping. Algorithm 1 illustrates how we query PCI registers and determine the bits that translate physical addresses to DRAM locations on the AMD platform (with PCI register addresses documented in the architecture manual). We denote the physical address of a memory frame as "phyaddr", while nodes[n] represents the PCI information for each memory node. We then use this mapping information to color every memory page.

Table 1: Range for Each Memory Controller

	range of physical address
Controller 0	0x0 – 0x227FFFFFF
Controller 1	0x228000000 – 0x427FFFFFF
Controller 2	0x428000000 – 0x627FFFFFF
Controller 3	0x628000000 – 0x827FFFFFF

The range of its physical address identifies the memory controller/node of a frame. E.g., on the AMD Opteron 6128 processor with a total of 32 GB physical memory, a page's physical address is in a range between 0x0 and 0x827FFFFFF, where each controller serves a subrange (see Tab. 1), which is used to determine the node ID if the address resolves to the current node's range (lines 3-6) or to search for an address resolution in the next node (line 29). If channels are interleaved, phyaddr is modified to compensate (lines 7-10). Channel and rank ID bits are indicated by the "DRAM Controller Select Low Register" (lines 11-12) and "DRAM CS Base Address Registers" (lines 15-22), 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 (line 23). Next, we identify the bank, row, and column bits based on the "DRAM Bank Address Mapping Register" (lines 24-25 and Fig. 5).

Bits	Hardware Info.		bank ID			Row and Column Address															
1000b	2GB	1GB, x4	16	15	14	Row	x	x	17	30	29	28	27	26	25	24	23	22	21	20	19	18
						Col.	x	x	x	x	13	AP	12	11	10	9	8	7	6	5	4	3

Fig. 5: DRAM Bank Address Mapping Register (AMD Opteron)

Algorithm 1 Translate Address

```
1: INPUT: phyaddr, nodes[n]
2: for i=0...n-1 do
3:     DramBase=getPCI(DRAM Base Registers)
4:     DramLimit=getPCI(DRAM Limit Registers)
5:     if DramBase ≤ phyaddr ≤ DramLimit then
6:         NodeID = i
7:         if Node Interleaving == True then
8:             temp = getPCI(Swap Interleaved Region Base/Limit Register)
9:             Modify phyaddr to remove node interleaving based on temp
10:        end if
11:        temp=getPCI(DRAM Controller Select Low Register)
12:        Translate phyaddr to ChannelID based on temp
13:        Modify phyaddr to remove channel interleave
14:        for CS = 0...7 do
15:            CSBase=getPCI(DRAM CS Base Address Registers)
16:            CSEn=CSBase & 0x00000001
17:            CSMask=getPCI(DRAM CS Mask Registers)
18:            /*Get the specified bits of PCI register according processor document*/
19:            CSBase=CSBase & 0x1FF83FE0
20:            CSMask=(CSMask | 0x0007C01F) & 0x1FFFFFFF
21:            if (CSEn != 0) and ((phyaddr & ∼CSMask) == (CSBase & ∼CSMask)) then
22:                RankID = CS
23:                Modify phyaddr to remove masked bits and get normalized address for this rank
24:                DramAddrMap = getPCI(DRAM Bank Address Mapping Register)
25:                Translate phyaddr to BankID, RowID and ColumnID based on DramAddrMap
26:            end if
27:        end for
28:    else
29:        continue
30:    end if
31: end for
```

Our coloring mechanism is triggered upon boot-up of 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 ($4 \times 2 \times 2 \times 8 = 128$ banks in total). After boot-up and page color initialization, the system groups the entire memory space into 128 colors and records in the page table which color a page belongs to.

3.3. CAMC User Interface

Once the system is fully booted, it 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 the per-task memory requirement, with a default of one color per 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 "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);
```

3.4. Memory Policy Configuration

A backward-compatible enhancement of the `mmap()` call registers (adds) the current user_id to the coloring_user list in the kernel after CAMC is activated. 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, a command-line coloring request indicates the path of a binary. We check this execution path of new tasks to determine if this task should be colored.

After CAMC activation, the user_id and execution_path of tasks are 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 other words, coloring constraints are recorded globally within the kernel (slab) allocator for all tasks that have registered colors via mmap and are subsequently used on each task's allocation request.

In addition, a coloring flag, using_color, is set in the task_struct by the 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. No software/application source code or hardware architecture modifications are needed for CAMC. Fig. 6 depicts the configuration of the coloring memory policy.

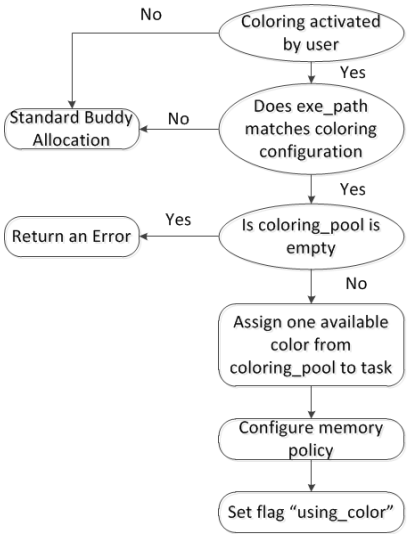

Fig. 6: Program Flow to configure memory coloring

A table records 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 should we run out of a task's pre-allocated colors.

CAMC is activated when a new task (process) is created by a fork() system call. Such a fork() results in sharing of the parent process' pages (with read-only permission) between the child and parent processes following the copy-on-write (COW) paradigm of Linux. 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, pages for the binary indicated as a parameter are allocated in place of the original parent pages. CAMC ensures that the entire memory space of the child is colored by configuring the coloring memory policy in the `do_exec` function.

3.5. Page Allocation Design

We next detail how page coloring is provided for both interleaved and non-interleaved modes. Subsequently we

show in Algorithm 2 how separate free lists are maintained per color and dealt out upon being requested via mmap(). When the free list of a certain color is empty, unused "high order" memory regions are broken into 4KB sized order-0 regions and added to the free list by color (derived by their physical address, see Algorithm 3).

The Linux buddy allocator has been augmented to implement CAMC. We only handle $order = 0$ allocations in CAMC when the current task has its coloring flag set (line 3) 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., $order = 0$). Common kernel internal allocation requests for obtaining a page frame are thus handled by CAMC.

CAMC also supports channel interleaving for multi-channel memory architectures. With channel interleaving, one page is spread evenly across channels at cache line granularity to increase memory throughput.

The interleaving boundary is related to the size of cache line and determined by the 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 the same time though multiple channels. Via memory coloring, we still guarantee isolation and predictability.

Algorithm 2 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 require-
    ments
 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 3 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
```

Once the memory policy is configured, one needs to determine which page to select at a page fault. This process is shown in Algorithms 2+3. 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 (line 3).

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 (line 5). 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 from the corresponding colored free list and hands it to the user (line 7). Otherwise, the kernel traverses the general buddy free list (line 10) and returns the first page with a matching color for this task (line 7). Any pages with non-matching colors encountered during the traversal are added to the corresponding color lists by calling the `create_color_list` function (line 12). 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 (Algorithm 3). If new page is left (memory of this color exhausted), then an error will be indicated by returning NULL (line 16), which is the conventional manner of indicating an allocation has failed. For example, with 128 colors and no interleaving, a higher order region of n pages, where n is divisible by 128 (since it has 2^{12} pages or a multiple thereof) will be distributed round robin over the 128 free lists. If a task requires large amounts of memory (more than 1/128 in the example), it requests more than one color so that page requests can be fulfilled by choosing any page of the assigned colors.

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 on a per-task basis for a specific bank and memory controller.

4. Evaluation Framework and Results

This section provides details on the hardware platform and benchmarks for experiments before reporting results for our approach in comparison with previous work plus multicore results for multi-threaded benchmarks and real-time experiments to assess our coloring method.

4.1. Hardware Platform

A two-socket SMP with AMD Opteron 6128 (Magny Cours) processors with eight cores per socket (16 cores altogether) comprises our experimental platform. 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 as in Fig. 1 but 8 memory controllers total instead of just 4 in the figure), 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. Eight-way parallel access is support to these banks.

4.2. CAMC vs. Buddy with Local Node Policy

Let us 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 and repeat experiments multiple times.

We use large strides to defeat hardware prefetching and allocate a large address space to inflict capacity misses in all caches so that we can assess the performance of memory controller coloring. 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 and so on.

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 perform allocations for different memory controllers. Table 2 indicates the average memory access latency and standard deviation over a sequence of accesses per memory controller, where the task is running on core 1 (to reduce interference since core 0 often serves interrupts as per Linux policy). Since controller 0 is local to core 1, memory at controller 0 is accessed locally while that at other controllers is accessed remotely (see Fig. 1). We observe 60% lower memory access latency and 33% smaller deviation for local over remote references, simply due to resolving all references locally with our allocator.

Table 2: Access Latency of each Controller

	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 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.

4.2.2. CAMC vs. Buddy Allocation with Local Node Policy

A comparison of the cost of CAMC and buddy allocation with "local node" policy is given next. While all memory is allocated to the local controller 0, tasks access disjoint banks under coloring while bank conflicts can and will occur under buddy allocation. 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 3 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. CAMC accesses disjoint private banks per thread on the same controller while buddy shares memory controller and banks among the threads.

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

Table 3: Cost of CAMC 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

4.2.3. CAMC Overhead

The overhead of CAMC allocations normalized to standard buddy allocation is depicted in Table 3. CAMC imposes no overhead over buddy allocation during regular program execution. But during initialization, CAMC 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. To amortize the overhead of initialization, it typically suffices to make a single allocation call for coloring.

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.

4.3. NAS Parallel Benchmark Results

We investigated the performance impact of CAMC for IS, the only C code from the NAS Parallel Benchmark (NPB) suite [10]. The NPB suite is designed to evaluate the performance of parallel kernels. We investigate the OpenMP version of IS. IS is an integer sort application with many random memory accesses. We configured IS to run with 4 OpenMP threads. Each thread is bound to a different CPU core. The benchmark dynamically allocates about 32 MB of memory per thread before accessing it. The normal buddy allocator may inflict conflicts when accessing memory while CAMC ensures that neither remote memory nor shared memory bank accesses are issued by these threads.

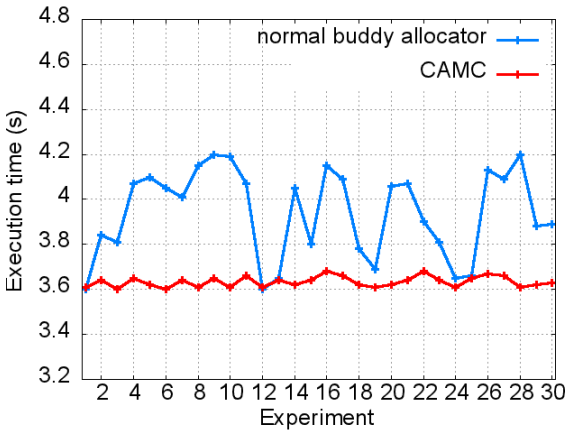

Fig. 7: NPB OpenMP IS Result for 4 Threads

Fig. 7 depicts the overall execution time (y-axis) for a sequence of experiments (x-axis), each allocating their own 32MB data and then accessing it. We make the following observations. Upon default buddy allocation (blue/upper curve), the execution time of the IS OpenMP benchmark fluctuates between 3.6-4.2 seconds in consecutive runs. The fluctuation is due to (1) data (especially stack, static, and instruction segments) placed on remote memory controllers relative to cores bound to threads, and (2) contention at the memory bank level for shared data accesses. With CAMC (red/lower curve), the execution time is nearly constant at 3.65 secs. Notice that the best-case timing for buddy is even lower than that under coloring, though not by much. This is due to the overhead of maintaining 128 separate queues for coloring.

For real-time scheduling, a tight but safe (conservative) bound on the execution time is important to ensure good system utilization. CAMC provides a much tighter bound than buddy allocation (lower standard deviation), i.e., more predictable real-time behavior. And CAMC has up to 12.4% higher performance due to reduced conflicts and memory access latency.

Observation 5: Not just multi-threaded programs under Pthreads (see synthetic benchmark result), but also OpenMP programs benefit from CAMC.

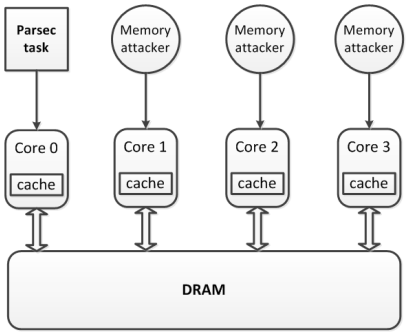

Fig. 8: Mixed: 1 Parsec code + up to 3 memory attackers

4.4. System Performance

Performance and predictability for the PARSEC benchmark suite featuring multithreaded programs are investigated next with "simlarge" inputs [11]. In the experiment, we create a multi-task workload where several "memory attackers" run in the background (as non real-time tasks) to assess their interference on memory latency for a foreground (real-time) task similar to prior work [5, 4]. We call these background tasks the "memory attackers", represented by instances of the stream benchmark. Fig. 8 depicts an example with 4 tasks, one (foreground task) is a Parsec benchmark and the others (background) are memory attackers.

4.4.1. Performance

Next, the runtime of shared vs. private (isolated) bank allocation (different controllers and different banks) is compared. 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*, foreground task and attackers allocate pages from their private bank and private local controller for full task isolation via CAMC.

The experimentally determined WCETs for all Parsec benchmarks with background attackers (bars 1-3) and without (bar 4) are depicted in Fig. 9. 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*. The ferret benchmark gets the largest performance enhancement (28.9%) and the fluidanimate benchmark the smallest one (6.2%) for bank-level coloring (*diff_bank*). In contrast, the canneal benchmark gets the largest performance enhancement (41.7%) and swaptions the smallest one (11.2%) for controller-level coloring (*diff_controller*).

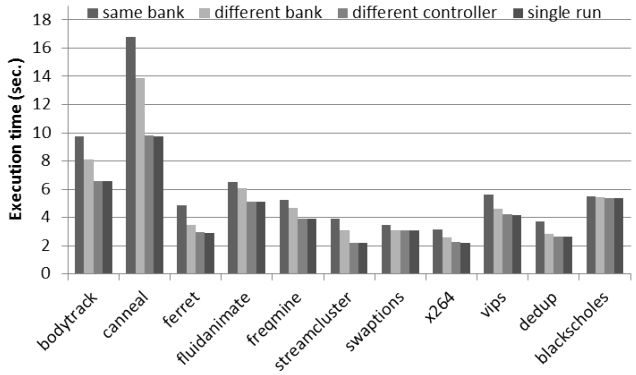

Fig. 9: Parsec: diff. controller/diff. bank/shared bank/single

In all 3 cases, the execution time is relatively predictable (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. Notice that LCC coloring would have removed LLC contention.

Observation 6: 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.

The average WCETs of the 3 memory attackers under 3 memory configurations are shown in Fig. 10. The error bars show the maximum and minimum runtime of 3 Stream benchmarks. The results indicate that *diff_controller* gets a 40% and *diff_bank* a 14.8% performance enhancement over *same_bank*. Hence, not only foreground tasks (from the Parsec suite), background memory attackers (the Stream benchmark) also improve in performance under CAMC.

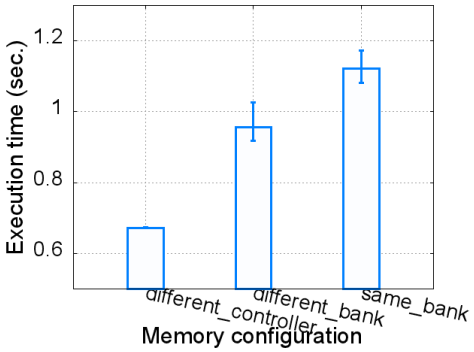

Fig. 10: Background Attacker (Stream): Diff. Controller/Bank/Shared Bank

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

4.4.2. Predictability

By assigning memory space appropriately to each task, we can also make a task's execution time more predicable. In this section, we investigate the predictability impact of controller-aware memory coloring for Canneal and X264 with Stream_cluster instances in the background (attacker) since these are the most memory-bound codes. To this end, we compare the execution times under CAMC and buddy allocation for the Canneal benchmark as the number of memory attackers and the allocated memory size change. All tasks are bound to different cores (0,1,2,3) that belong to one memory controller.

Table 4: Execution Time of Canneal: 1+3 Attackers

Allocator	Canneal+1 Attacker	Canneal+3 Attackers
Buddy	12.278 secs	17.665 secs
CAMC	9.712 secs	9.811 secs

Table 4 indicates the average execution time of Canneal in seconds for one and 3 "attackers" (both averaged over 10 runs). We observe that the time decreased, i.e., the performance increased for CAMC compared to buddy by 20.9% and 44.46% for 1 and 3 attackers, respectively. Furthermore, the execution time of Canneal remains stable under CAMC since background attackers always access other memory controllers, i.e., Canneal does not suffer from memory access contention with other tasks. With increasing numbers of attackers, the execution time gap between CAMC and buddy allocation widens since each attacker adds more controller/bank conflicts.

Table 5 depicts the average execution time over ten repeated runs under CAMC and buddy allocations for different amounts of allocated memory (100K, 200K and 400K elements for simsmall, simmedium and simlarge inputs,

Table 5: Canneal: Performance Sensitivity to Allocation Size

Allocator	small size	medium size	large size
Buddy	8.45 secs	17.72 secs	22.3 secs
CAMC	8.41 secs	10.32 secs	11.12 secs

respectively). Here, Canneal co-runs with three memory attackers. We observe that CAMC and buddy result in nearly identical runtimes for small allocation sizes. But performance under CAMC increases as the allocation sizes become larger and more background attackers are present due to additional contention between the tasks at controller/bank levels.

We next assess the X264 benchmark in terms of predictability for varying numbers of attackers (Stream benchmark). We dynamically change the number of memory attackers running in the background so that 1-3 attackers are active at any given time. Fig. 11 depicts the execution time of X264 (y-axis) over a sequence of experiments (x-axis). We observe that execution is quite uniform under private bank allocation with CAMC (red/lower curve) while it fluctuates more frequently (is less predictable) for shared banks under buddy allocation (blue/upper curve). Buddy's shortest time is about 45% of its longest time creating a wide range of variation. In contrast, shortest and longest times of CAMC differ by just 5.8%. Also, coloring results in a performance enhancement of 47% over buddy allocation for the longest times.

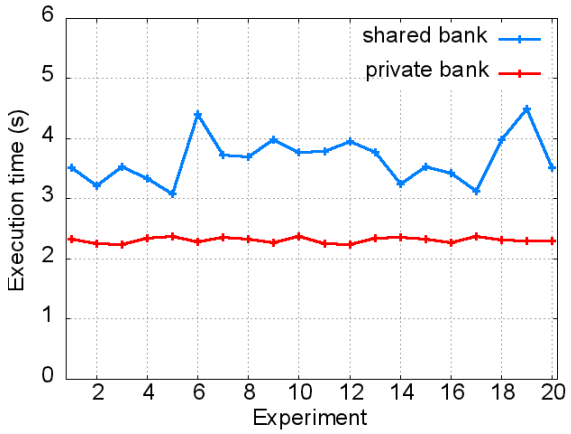

Fig. 11: Runtime: X264 Coloring vs. Buddy

Observation 7: CAMC enhances performance and makes task execution more predictable across different input sizes and numbers of background tasks, which provides single-core equivalence for real-time systems assigned to disjoint memory controllers.

4.4.3. Multiple Cores

The next experiment executes 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. 12 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, both for our 16-core case and the other schemes.

Observation 8: CAMC results in superior performance for multicore 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

The next experiments evaluate CAMC under rate-monotonic 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 6) 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. 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.

Table 6: 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

Fig. 15 shows the corresponding real-time extended 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 based on verification of schedulability via response-time analysis. In contrast, the `same-bank` configuration does not provide isolation between Tasks 1+2 and the background tasks (Stream), which causes deadlines to be

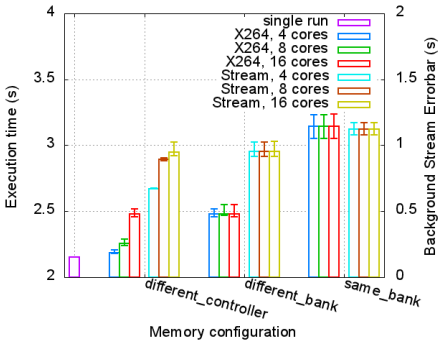

Fig. 12: Runtime of one X264 and 3/7/15 Stream Tasks

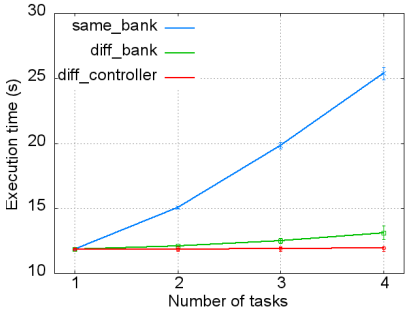

Fig. 13: Palloc: Avg. Memory Latency for Controller/Bank/no Coloring

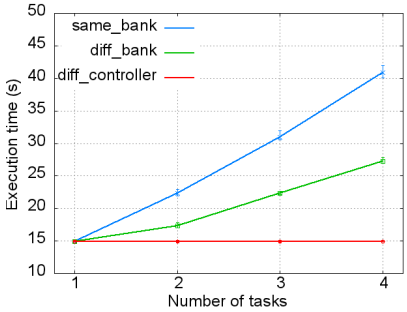

Fig. 14: Alternating Strides for Controller/Bank/no Coloring

missed. Fig. 16 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 (first box of IS_SER) indicates this deadline miss.

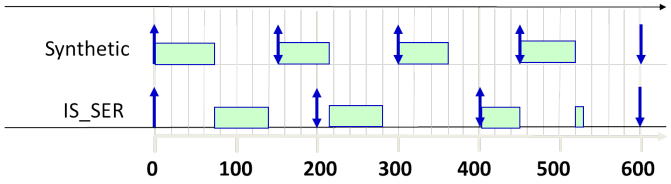

Fig. 15: Feasible Schedule: diff-controller

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 holds for policy that causes interference and not just buddy allocation.

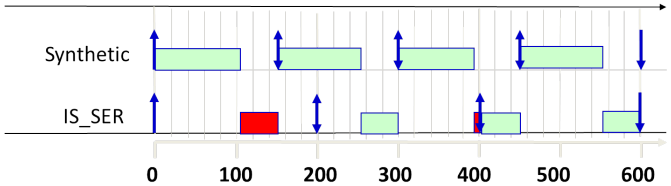

Fig. 16: Deadline Misses (red) for same-bank

Observation 9: In experiments, 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).

The observed execution times (avg. over 100 runs, min./max. and standard deviation) for tasks 1 and 2, respectively, are depicted in Tables 7+ 8 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 real-time extended Gantt chars. While the any of the other policies always meet deadlines, Table 9 quantifies the deadline miss rates for `same-bank` and `diff-bank`.

Table 7: 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 8: 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 9: Deadline Miss Rates

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

4.6. Influence of Coloring on Bank-Level Parallelism

We next experiment with MatMult, which multiplies two matrices, A and B, and stores the result in matrix C, to illustrate its dependence on the number of access streams. We compare MatMult under buddy and CAMC allocation. For the latter, we also varied the numbers of banks (1-3) allotted to a task.

Fig. 17 shows that Matmult's execution time increases as the number of banks decreases. Its execution time is

about 15.1 seconds for buddy allocation (using 128 banks), but increases to 15.37 seconds when only one bank is available. Matmult has 3 access streams (one per array: A,B,C). If we use the performance for 3 banks as a reference, then execution time increases by 1.22% when restricting it to just 1 bank — and time increases by only 0.6% for 128 banks (entire memory space), i.e., 3 banks nearly suffice to reach the bank level parallelism of this algorithm.

Fewer memory banks are available for an application under CAMC than buddy (see Section 3). This experiment shows that CAMC exploits most of the bank-level parallelism as long as the application's bank-level parallelism does not exceed the number of available banks.

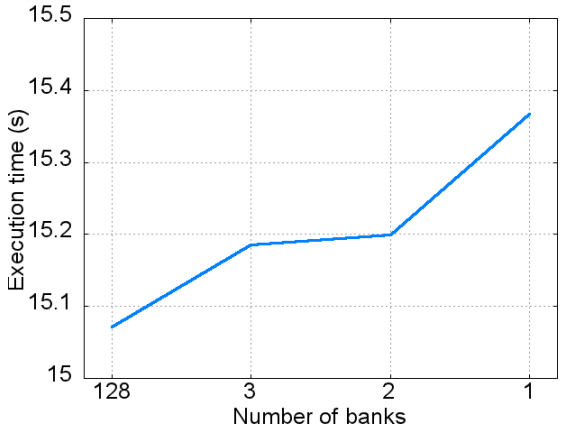

Fig. 17: Time for MatMult Allocating 128/3/2/1 Banks

Observation 10: Application sensitivity to memory parallelism allows the selection of the number of banks under CAMC so that single-core equivalence can be provided without significant performance penalty as opposed to buddy.

4.7. Latency Comparison with Prior Work

The performance of our approach is next compared to Palloc [4], 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.

4.7.1. Latency Benchmark

We utilize Palloc's latency benchmark [4, 12], which iterates through a randomly shuffled linked list whose size is twice that of the last-level cache (LLC) size. When following the references of the linked list, a subsequent memory request must wait until the previous request finishes due to data dependencies. Such "pointer chasing" serializes memory accesses. As the linked list is randomly permuted over a large memory area, hardware prefetching does not kick in and caches will suffer capacity misses because the working set size is much larger than the LLC size. Thus, this latency benchmark represents a "bad" memory access pattern, but not *the* worst case as we will show.

One instance of the latency benchmark is placed on core 0 (the "foreground" load) and up to 3 latency benchmark instances are co-run in the background (cores 1-3). The actual number of background tasks varies (0-3), just as in prior work [4].

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. 13 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, CAMC provides higher predictability as the error bars are the smallest for *diff_controller*.

4.7.2. Synthetic Benchmark

Palloc [4] and CAMC are further contrasted under our synthetic benchmark (striding back and forth with increasing offsets) under the same setup as for the Palloc latency benchmark. Fig. 14 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 [4] 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. PCI registers provide the information for address bit selection of coloring in a portable manner.

4.7.3. SPEC Benchmark

We also compare our memory controller coloring with Palloc [4] by utilizing 11 of the 15 SPEC2006 benchmarks evaluated in their work (as 4 of them would not build in our Linux environment). In the experiment, we run SPEC2006 programs in two settings. First, we run one SPEC2006 program by itself on core 0, called `Solo setting`. Second, we run three 470.lbm program instances on cores 1-3 while executing one SPEC2006 program on core 0. This setting is called `Corun setting`. These are the same configurations of the SPEC2006 experiments in the Palloc work [4].

Fig. 18 shows the slowdown ratios of SPEC2006 under our memory controller coloring. Slowdown ratios equal Solo IPC (Instructions per cycle) divided by Corun IPC. The higher the slowdown ratio the more conflicts occur and the lower the performance is. The best case of a slowdown ratio is 1, which means memory access conflicts have been eliminated. The figure shows that the slowdown ratio is almost exactly one under our memory controller coloring but much larger under normal buddy allocation. In comparison to the results reported for Palloc [4], our approach obtains better performance through controller-aware memory coloring and, more significantly, single core equivalence as the slowdown is 1.

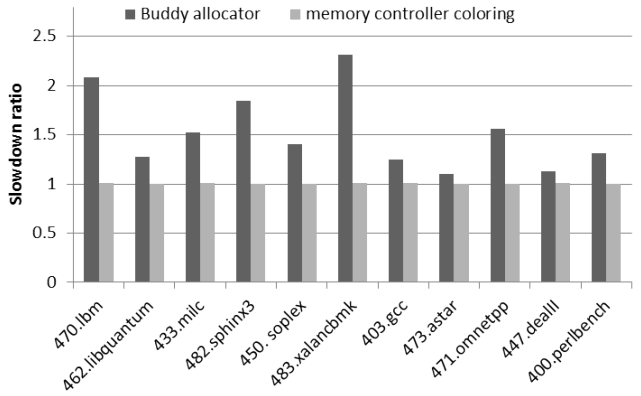

Fig. 18: Slowdown Ratios: SPEC2006 Program on Core 0 (x-axis) plus 3 470.lbm Instances on Cores 1-3

Observation 11: 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 [13, 14, 15, 16, 17, 18, 19]. Blagodurov et al. [13] and McCurdy et al. [14] describe the performance problems of NUMA for multithreaded applications and investigate their causes. Marathe et al. [15] propose a profiler to optimize data placement of multithreaded programs via hardware-generated memory traces. Lachaize et al. [16] use profiling to understand why and which memory objects are accessed remotely and so one can choose efficient application-level optimizations for NUMA systems. Majo et al. [17] study which factors limit the performance of multithreaded programs on NUMA multicores and describe source-level transformations to address these problems. Yun et al. [18] present a new parallelism-aware worst-case memory interference delay analysis for multiple parallel requests. To analyze memory delays in multicore real-time systems, a memory server under partitioned fixed-priority scheduling is introduced [19].

Scheduling or page placement has been proposed to solve the data sharing problem in NUMA system [20, 21, 22, 23, 24]. Li et al. [20] present a loop scheduling algorithm to exploit data locality and dynamically balance the load. Majo et al. [21] use program-level transformations to eliminate remote memory accesses. Ogasawara et al. [22] propose an online method for identifying the preferred NUMA memory nodes of objects during garbage collection. Ward [23] designs synchronization algorithms for buses and caches subject to sharing constraints in real-time systems. Huang et al. [24] propose a scheduling model for mutually exclusive execution among different task classes that avoids inter-class interference. 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 [25, 5, 4, 26, 27, 28]. Verghese et al. [29] increase data locality of multithreaded programs by page migration and page replication across processors. But they still face the problem of data sharing, and their data movement/copying incurs significant overhead. Chisholm et al. [28] present criticality-aware optimization techniques for shared LLC areas allocated for an MC-scheduled multicore system. Our work not only considers cache partitioning but also partitions DRAM for MC multicore real-time system. Awasthi et al. [27] 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. [25] 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. [26] 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 [4] 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 ac-

cesses. 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.

Reineke et al. [30] propose a PRET DRAM controller that partitions the memory space based on the internal structure of the DRAM chip in order to eliminate contention caused by sharing DRAM resource. In contrast, our work designs a software solution that can be applied to commodity DRAM controllers. Kim et al. [31, 32] discuss the hardware Isolation for mixed-criticality system and multicore real-time systems. Chisholm et al. [33] utilize the hardware management to provide strong isolation guarantees to higher-criticality tasks with respect to DRAM banks and the LLC. Kim et al. [5] present techniques to provide a tight upper bound on the worst-case memory interference in a COTS-based multicore systems. They explicitly model the major resources in a DRAM system and analyze the worst-case memory interference delay between tasks running in parallel. Suzuki et al. [34] combine cache and bank coloring to obtain tight timing predictions. Mancuso et al. [9] 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. Yun et al. [35] propose a software scheduler to reduce memory contention and to satisfy schedulability constrains. Gomony et al. [36] present a real-time multi-channel memory controller architecture and an algorithm to map clients to channels while minimizing bandwidth utilization. Alhammad et al. [37] design a global scheduling algorithm for sporadic real-time tasks that efficiently co-schedules cores and DMA activities to increase predictability. Our approach not only colors memory spaces, such as channels, but also banks and controllers. It improves both performance and timing predictability.

6. Conclusion

A novel controller-aware memory coloring allocator for real-time systems, CAMC, has been designed and implemented. 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. This work assesses the real-time predictability of DRAM partitioning on NUMA architectures, which is unprecedented. It thus develops a methodology to partition resources for exclusive access (one core per memory controller) for single core equivalence of real-time tasks. This can facilitate WCET analysis as the sharing of NUMA resources need not be considered in static WCET predictions while still providing sound timing bounds for real-time schedulability analysis. An open problem remains how all cores (instead of just one core per memory controller) could be utilized while providing single core equivalence. Such methods currently lack hardware support, such as rate-based or prioritized guarantees within memory controllers.

Acknowledgment

This work was supported in part by NSF grants 1239246, 1329780, and 1525609. It extends an earlier conference paper [38] by (1) detailing description of how our CAMC policy is designed and implemented, (2) complementing descriptions with illustrative examples, (3) providing architectural details for mapping pages to frames based for a sample AMD platform with tables for illustration, (4) conducting new experiments to contrast local and remote memory access latencies on our base system, (5) assessing the performance and predictability of multi-threaded NAS Parallel Benchmark code IS under our coloring policy, (6) furter investigating the predictability impact of controller-aware memory coloring for the Parsec benchmarks in detail, (7) conducting a study on the influence of coloring on bank-level parallelism, (8) comparing the latency Palloc and CAMC for SPEC benchmarks, (9) expanding related work to include more recent as well as additional fundamentally related publications and highlighting differences from CAMC.

References

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

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom, The worst-case execution time problem — overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems 7 (3) (2008) 1–53.

[3] G. Bernat, A. Colin, S. Petters, Wcet analysis of probabilistic hard real-time systems, in: IEEE Real-Time Systems Symposium, 2002, pp. 279–288.

[4] H. Yun, R. Mancuso, Z.-P. Wu, R. Pellizzoni, Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2014, pp. 155–166.

[5] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, R. R. Rajkumar, Bounding memory interference delay in cots-based multi-core systems, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2014, pp. 145–154.

[6] Z. P. Wu, Y. Krish, R. Pellizzoni, Worst case analysis of dram latency in multi-requestor systems, in: IEEE Real-Time Systems Symposium, 2013, pp. 372–383.

[7] X. Z. S. D. K. Shen, Hardware execution throttling for multi-core resource management, in: USENIX Annual Technical Conference, 2009.

[8] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, A. Cox, NUMA policies and their relation to memory architecture, in: Architectural Support for Programming Languages and Operating Systems, 1991, pp. 212–221.

[9] R. Mancuso, R. Pellizzoni, C. Marco, L. Sha, H. Yun, Wcet(m) estimation in multi-core systems using single core equivalence, in: Euromicro Conference on Real-Time Systems, 2015, pp. 174–183.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, The NAS Parallel Benchmarks, The International Journal of Supercomputer Applications 5 (3) (1991) 63–73.
URL `citeseer.ist.psu.edu/article/bailey94nas.html`

[11] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: PACT, 2008, pp. 72–81.

[12] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2013, pp. 55–64.

[13] S. Blagodurov, S. Zhuravlev, A. Fedorova, A. Kamali, A case for numa-aware contention management on multicore systems, in: International Conference on Parallel Architectures and Compilation Techniques, 2010.

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

[15] J. Marathe, V. Thakkar, F. Mueller, Feedback-directed page placement for ccnuma via hardware-generated memory traces, Journal of Parallel and Distributed Computing 70 (12) (2010) 1204–1219.

[16] R. Lachaize, B. Lepers, V. Quéma, et al., Memprof: A memory profiler for numa multicore systems., in: USENIX Annual Technical Conference, 2012.

[17] Z. Majo, T. R. Gross, (mis) understanding the numa memory system performance of multithreaded workloads, in: International Symposium on Workload Characterization, 2013, pp. 11–22.

[18] H. Yun, R. Pellizzoni, P. Valsan, Kumar, Parallelism-aware memory interference delay analysis for cots multicore systems, in: Euromicro Conference on Real-Time Systems, 2015, pp. 184–195.

[19] R. Pellizzoni, H. Yun, Memory servers for multicore systems, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2016, pp. 97–108.

[20] H. Li, S. Tandri, M. Stumm, K. C. Sevcik, Locality and loop scheduling on numa multiprocessors, in: International Conference on Parallel Processing, 1993, pp. 140–147.

[21] Z. Majo, T. R. Gross, Matching memory access patterns and data placement for numa systems, in: International Symposium on Code Generation and Optimization, 2012, pp. 230–241.

[22] T. Ogasawara, Numa-aware memory manager with dominant-thread-based copying gc.

[23] B. C. Ward, Relaxing resource-sharing constraints for improved hardware management and schedulability, in: IEEE Real-Time Systems Symposium, 2015, pp. 153–164.

[24] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, L. Thiele, An isolation scheduling model for multicores, in: IEEE Real-Time Systems Symposium, 2015, pp. 141–152.

[25] X. Pan, Y. J. Gownivaripalli, F. Mueller, Tintmalloc: Reducing memory access divergence via controller-aware coloring, in: International Parallel and Distributed Processing Symposium, 2016, pp. 363–372.

[26] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, C. Wu, A software memory partition approach for eliminating bank-level interference in multicore systems, in: International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 367–376.

[27] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, A. Davis, Handling the problems and opportunities posed by multiple on-chip memory controllers, in: International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 319–330.

[28] M. Chisholm, B. C. Ward, N. Kim, J. H. Anderson, Cache sharing and isolation tradeoffs in multicore mixed-criticality systems, in: IEEE Real-Time Systems Symposium, 2015, pp. 305–316.

[29] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, Operating system support for improving data locality on cc-numa compute servers, in: Architectural Support for Programming Languages and Operating Systems, 1996, pp. 279–289.

[30] J. Reineke, I. Liu, H. D. Patel, S. Kim, E. A. Lee, Pret dram controller: Bank privatization for predictability and temporal isolation, in: CODES+ISSS.

[31] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, F. D. Smith, Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2016, pp. 149–160.

[32] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, F. D. Smith, Allowing shared libraries while supporting hardware isolation in multicore real-time systems, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2017, pp. 223–234.

[33] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, F. D. Smith, Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems, in: IEEE Real-Time Systems Symposium, 2016, pp. 57–68.

[34] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, R. Rajkumar, Coordinated bank and cache coloring for temporal protection of memory accesses, in: International Conference on Computational Science and Engineering (CSE), 2013, pp. 685–692.

[35] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memory access control in multiprocessor for real-time systems with mixed criticality, in: Euromicro Conference on Real-Time Systems, 2012, pp. 299–308.

[36] M. D. Gomony, B. Akesson, K. Goossens, A real-time multi-channel memory controller and optimal mapping of memory clients to memory channels, ACM Transactions on Embedded Computing Systems (2015) 25:1–25:27.

[37] A. Alhammad, S. Wasly, R. Pellizzoni, Memory efficient global scheduling of real-time tasks, in: IEEE Real-Time Embedded Technology and Applications Symposium, 2015, pp. 285–296.

[38] X. Pan, F. Mueller, Controller-aware memory coloring for multi-core real-time systems, in: Symposium on Applied Computing, 2018.