TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring

Xing Pan, Yasaswini Jyothi Gownivaripalli, Frank Mueller
North Carolina State University, Dept. of Computer Science, Raleigh, NC, USA, Email: mueller@cs.ncsu.edu

***Abstract*—DRAM memory of modern multicores is partitioned into sets, each with its own memory controller governing multiple banks. Accesses can be served in parallel to controllers and banks, but sharing of either between threads results in contention that increases latency, and so do accesses to remote controllers due to the non-uniform memory access (NUMA) design. Above DRAM, a last-level cache (LLC), typically at level 3 (L3), is shared by all cores while L1 and L2 caches tend to be core private.**

This NUMA design inflicts significant variations in execution time for applications with large datasets due to different latencies incurred by remote memory node accesses or contention in LLC and at memory banks/controllers. As a result, single program multiple data (SPMD) applications tend to experience computational imbalance at barriers, which inflicts idle (wait) time for threads that at barriers arrive early and thus impairs effective processor utilization and ultimately performance.

This work contributes a novel memory allocator called *TintMalloc* that colors memory at the LLC, bank, and controller level to ensure locality to the local memory node while reducing contention at the LLC/bank levels in software. After adding one line of code during initialization in each thread, existing applications automatically obtain colored heap space through regular malloc calls.

Experimental results with the SPEC and Parsec benchmarks show that by choosing disjoint colors per thread, locality is increased, contention is decreased, and overall SPMD execution becomes more balanced at barriers than default memory allocation under Linux as well as prior coloring approaches.

***Keywords*-NUMA, caches, memory controller, page coloring**

I. Introduction

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

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

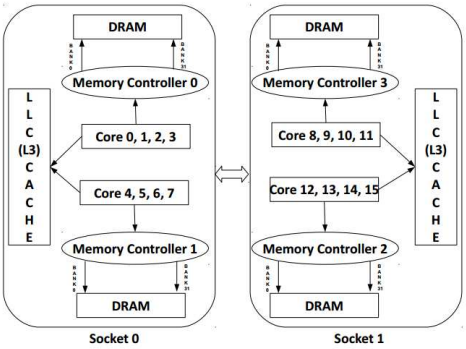

Fig. 1. Architecture of memory and cache on AMD Opteron

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

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

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

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

smaller) and interference is reduced. The approach can keep the runtime of tasks in parallel section more balanced, which reduces idle time and increases core utilization.

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

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

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

Several approaches have been proposed to address contention between shared resources, e.g., scheduling algorithms based on data reference characteristics [1]–[3], cache locality [4]–[7], and page coloring for DRAM partitioning [8]–[10]. Compared to them, our approach not only partitions memory banks and the LLC but also consider the locality of memory controllers. To our knowledge, it is the first paper to (a) color memory controllers and (b) combine memory controller, bank and LLC coloring together. Overall, TintMalloc effectively lowers contention for shared resources and reduces imbalance at boundaries of parallel sections in programs resulting in improved overall performance and core utilization, much in contrast to other allocators.

II. NUMA Memory Architecture

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

A. Caches

Most modern CPU architectures have multiple levels of caches organized hierarchically within a single chip. For example, there are two sockets in our AMD Opteron 6128 system and the cache hierarchy in each socket is shown in Fig. 2. Each core has its private, local L1 and L2 caches and all cores share the LLC. A miss in L1 initiates an access to L2, and a miss in L2 initiates an access to L3. A miss in LLC initiates an access to memory.

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

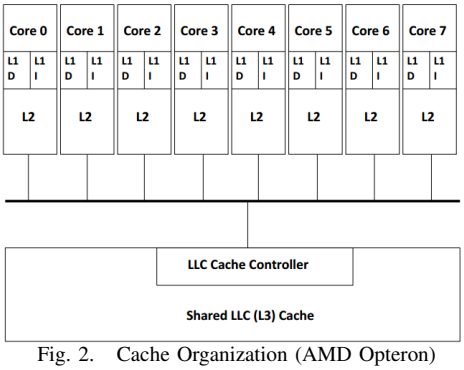

Fig. 2. Cache Organization (AMD Opteron)

B. DRAM Memory

Sets of cores comprise a memory node in NUMA systems. Each such node has one local memory controller as depicted in Fig. 1. For example, the AMD Opteron system used in our experiments has four memory controllers over two sockets with four cores per controller. The DRAM memory behind a controller is organized into channels, ranks, and banks depicted in Fig. 3. On a same controller, accesses to different banks and channels may proceed in parallel, which provides the capability of interleaving memory accesses, thereby improving memory bandwidth/throughput.

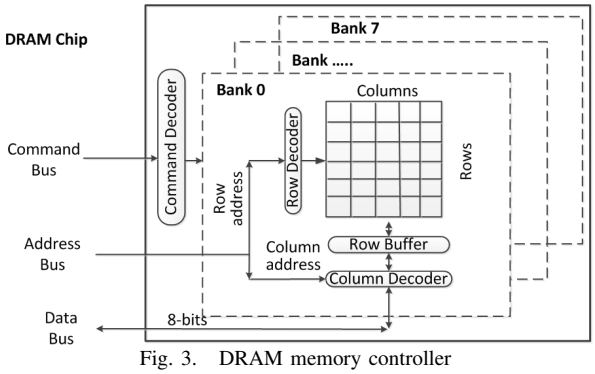

Fig. 3. DRAM memory controller

A DRAM bank array is organized into rows and columns of individual data cells. Upon access to data, the corresponding row is selected and pulled from the array into the row buffer (incurring a row access strobe penalty). Once in the row buffer adjacent data may be accessed with just a column access strobe penalty, which is smaller than the row activation cost. This, spatial locality in the buffer can be exploited (while temporal locality is typically taken core of by the upper-level caches). When a row buffer is replaced, an additional precharge penalty is incurred to update the row in the array with any modified data from the row buffer due to memory writes. DRAM cells are also periodically refreshed by the controller so that they do not lose their data, which also flushes the row buffer.

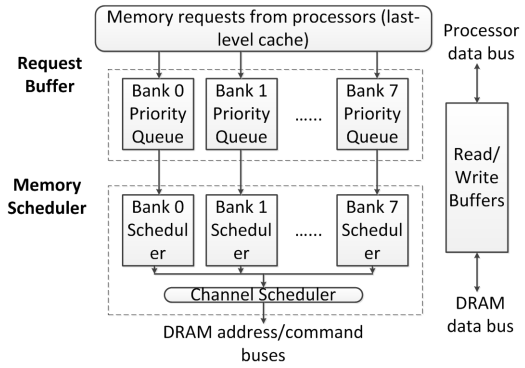

Fig. 4. Logical Structure of DRAM Controller

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

The memory controller governs the activities across banks/channels of local memory arrays. An initial controller queue de-multiplexes requests to the respective parallel sub-components and then issues DRAM commands for row/column accesses. Its operations are subject to timing constraints of banks and buses, which are typically configured at boot time and limited by manufacturing parameters (see Fig. 4). Multi-threaded programs can profit from avoiding resource contention by utilizing memory of the local controller, yet of different banks per thread. Access to the same bank increase latency due to contention, access to remote controllers increase costs due to propagation latencies over the on-chip (or cross-socket) interconnects and potential contention on interconnects and remote controllers/banks. Hence, data placement play a decisive role in ensuring that threads issue local accesses with lower contention and latency penalties instead of remote accesses of higher contention and latencies.

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

III. TintMalloc Design

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

A. Frame Color Selection

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

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

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

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

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

TintMalloc utilizes information on bit-level physical memory mapping. Its design is portable to any platform with known mappings. Our implementation is specific to the AMD Opteron platform, specifically the Opteron 6128 with bit mapping indicated in Fig. 5, where we combine fixed mappings with PCI register information obtained at runtime to determine address translation bits for node/controller (DRAM base/limit and limit system address registers indicate bits 16-20), channel (controller select low register), rank and bank (CS base address registers indicating bit 7 for the rank and bits 15, 16, and 18 for the bank) as well as row/column (bank address mapping register) — see AMD's architectural manuals — to select frame colors. The LLC color (set index) is given by bits 12-16.

TintMalloc is activated in the late phase of booting Linux at which time the bit-level information above is derived from PCI registers. For our Opteron 6128 platform, four memory controllers detected with two channels each, two ranks per

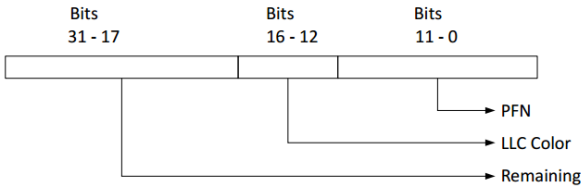

Fig. 5. Cache Color Address Mapping bits (AMD Opteron)

channel and eight banks per rank. This amounts to $2^7 = 128$ banks altogether across all controllers on our platform suitable for coloring. The LLC also has $2^5 = 32$ colors (over 5 bits).

B. Coloring Policy Activation

TintMalloc groups frames of pages with valid page table mappings into separate lists for each of these colors to later serve allocation requests after the boot-up phase via the mmap() system call. We modified mmap() so that a zero-sized request is interpreted as the specification of color(s) by the calling thread for subsequent non-zero sized allocations. More specifically, a set bit 30 of the *protection* argument indicates that the first argument should be interpreted as the color and a mode, where the most significant bits specify the mode to indicate if the color should be set or cleared for memory (controller/bank) or LLC (see Fig. 6). A set color is recorded in the TCB of the corresponding calling thread/process (handled uniformly as a task in Linux). A thread may even call mmap() multiple times to establish a set of "owned" colors.

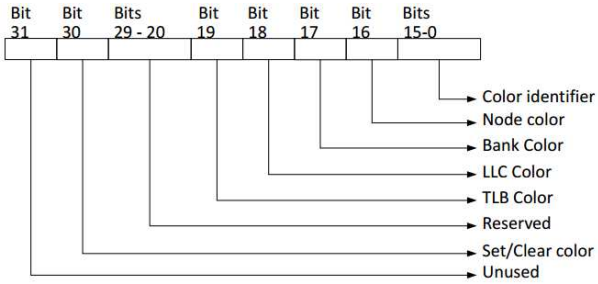

Fig. 6. mmap() color parameter bits identifiers

For example, the following mmap() call adds the LLC color "c" to the LLC colors of the current thread:
int length = 0;
char * A = (char*) mmap(c — SET_LLC_COLOR, length, prot | COLOR_ALLOC, flag, fd, offset);

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

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

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

C. Heap Policies: Linux Buddy Allocations vs. TintMalloc

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

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

TintMalloc maintains a free list and 128*32 color lists simultaneously inside the Linux kernel. Those color lists are defined as a matrix of `color_list[MEM_ID][cache_ID]`. At boot-up, these color lists are empty, all free pages are in the non-colored free list of the buddy allocator. A page fault by a program causes the kernel to invoke `alloc_pages` to find a free page. In the function `alloc_pages`, we disable the "pcp list" and use the function `_rmqueue_smallest` to serve page allocation requests. The colored page selection process is shown in Algorithms 1 and 2.

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

Algorithm 1 Colored Page Selection /* find page of certain size and color */

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

Algorithm 2 create_color_list /*move page from buddy free_list to colored free_lists*/

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

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

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

IV. Experimental Platform

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

V. Experimental Result

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

A. Synthetic Benchmark Results

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

In essence, this benchmark assesses the write latency of DRAM as it inflicts first cold and later capacity misses in L1/L2 caches for LLC coloring, or all caches for controller+bank coloring. Fig. 7 illustrates how one task may access a remote memory node and suffer the remote latency penalty under the buddy allocator. In addition, multiple tasks may share the same memory bank under buddy allocation. When two tasks access this bank at same time (Fig. 8), the second one will populate the row buffer and evict data from first one. This will inflate the memory access time of first task. The same problem also occurs in LLC accesses (Fig. 9). Here, the task's L3 cache miss rate will increase since other tasks evict its data from LLC.

Fig. 10 depicts the execution time of the synthetic benchmark on the y-axis for different coloring policies on x-axis. The shortest execution time is obtained with MEM/LLC, which indicates that both memory and LLC coloring are activated under TintMalloc. With MEM/LLC coloring, each

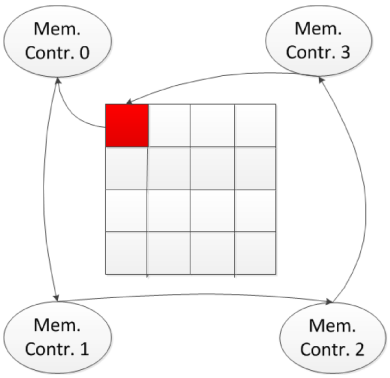

Fig. 7. Access to remote controller (node)

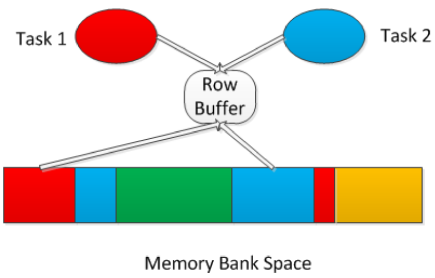

Fig. 8. Access conflict in same bank

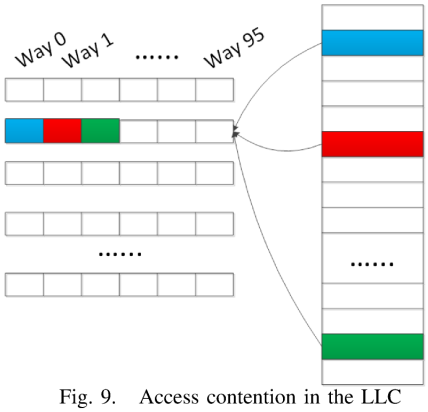

Fig. 9. Access contention in the LLC

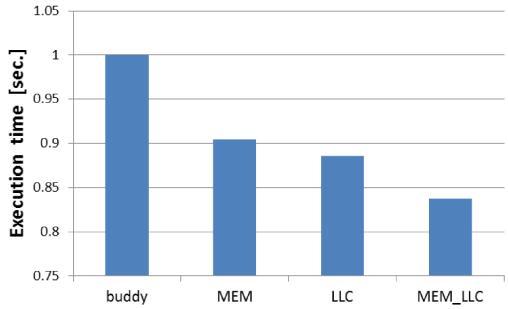

Fig. 10. micro_result

Algorithm 3 Measure idle time

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

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

B. Standard Benchmark Results

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

The OpenMP programs contain data-intensive parallel tasks with implicit barriers at the end of each parallel section. Due to runtime differences, threads executing faster have to wait for the slower ones at a barrier. We measured the benchmark's runtime, total idle time, runtime per thread, and idle time per thread. The `idle time` indicates a thread's wait time at barriers (see Algorithm 3).

To show the impact of memory controllers, we vary the numbers of threads, their memory nodes, and the level of parallelism. There are a total of five configurations: `16_threads_4_nodes`, `8_threads_4_nodes`, `8_threads_2_nodes`, `4_threads_4_nodes` and `4_threads_1_nodes`. For `16_threads_4_nodes`, we utilize all 16 cores and 4 memory controllers. 16 threads are pinned to different cores and share 4 controllers (nodes). For `8_threads_4_nodes`, 8 threads are pinned to 8 different cores and each pair of them accesses a common controller (their local memory controller) disjoint from other pairs. For example, 8 threads are pinned to cores 0,1,4,5,8,9,12,13. In this case, tasks pinned to core 0 and core 1 access local controller 0. For `8_threads_2_nodes`, 8 threads are pinned to 8 cores from 2 nodes, e.g., cores 0-7. For `4_threads_4_nodes`, we select one core from each node and pin 4 threads to them, such as cores 0,4,8 and 12. For `4_threads_1_nodes`, 4 threads are pinned to 4 cores on the same node, e.g., cores 0-3.

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

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

We compare TintMalloc to the standard buddy allocator of Linux (no coloring) and bank-level partitioning (BPM) [10]. BPM partitions memory banks and the LLC but it does not consider the memory controller in dynamic allocations.

`MEM+LLC(part)` and `LLC+MEM(part)` are different than `MEM+LLC` coloring. For example, there are a total of 32

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

We compared `MEM+LLC` coloring, standard buddy allocation, previous work (BPM) and the best result from `MEM, LLC, MEM+LLC(part) and LLC+MEM(part)`. The results are shown in Figures 11, 12 , 13 and 14.

Fig. 11 shows normalized benchmark runtimes for these approaches relative to the standard buddy allocator. We observe that `MEM+LLC` coloring results in shorter runtimes than buddy and previous work (BPM) for all six benchmarks. For example, for 16 threads and 4 memory nodes, our approach reduces the average runtime by up to 29.84% over standard buddy allocation (for SPEC/lbm). We observe that some benchmarks' performance enhancements exceed than of our synthetic benchmark. This is caused by additional spatial locality resulting in cache hits for these codes. The synthetic benchmark, in contrast, cannot benefit from spatial locality at all since only one access occurs per cache line. The error bar shows the maximum and minimum runtime of each benchmark over 10 repeated experiments. We also observe that our approach reduces the deviation of runtime, i.e., it reduces the variance of execution time, which helps increase computational balance in parallel sections at barriers. In addition, the previous work (BPM) always results in longer runtimes than our coloring approach and the standard buddy allocator. This is because BPM only partitions memory banks and LLC but does not indicate a memory controller. In this case, tasks may access remote memory nodes and have to pay the remote access penalty. Of the different coloring configurations, `16_threads_4_nodes` experiences the largest performance boost over the 6 benchmarks. This is because more tasks increase the probability to access a remote memory, which results in more memory bank and LLC contention. Fig. 12 also indicates that a benchmark's idle time can be reduced by our coloring approach. For `16_threads_4_nodes`, our MEM+LLC coloring results in up to 74.3% lower idle time compared to standard buddy allocation due to more balanced computation (less runtime variation). In fact, we observe a correlation between idle reduction and benchmark runtimes across experiments.

Figures 13 and 14 indicate the runtime and idle time, respectively, spent by each thread in parallel sections. We observe that difference in runtime between the fastest and slowest thread under standard buddy allocation is larger than under our TintMalloc. For example, for SPEC/lbm benchmark in the `16_threads_4_nodes` configuration, the difference in maximum thread running time and minimum thread running time for buddy allocator is 4.38 times larger than that of the MEM+LLC coloring approach. In addition, the maximum thread runtime under MEM+LLC coloring is 30.77% smaller than for standard buddy. The maximum thread idle time of the lbm benchmark is also reduced by 75% under MEM+LLC coloring compared to buddy allocation. The results show that TintMalloc effectively results in more balanced parallel program execution and enhance performance at the same time.

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

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

In addition, we obverse that Parsec/blackscholes has the least performance improvement of the six benchmarks. Of all TintMalloc coloring solutions, MEM+LLC(part) is the best one and it reduces the runtime by 3.6% compared to buddy allocation for 16 threads on 4 nodes. This happens because blackscholes reads a large mount of input data and is less memory intensive. Furthermore, the large fraction of the master thread's runtime prevents further performance enhancements since the master thread suffers from more restrictive memory allocation due to coloring. The larger the serial portion on the master thread is, the smaller will be the benchmark's performance enhancement.

The result also indicates that the MEM+LLC coloring approach is not always the best: For the Parsec/freqmine benchmark in the `16_threads_4_nodes` configuration, LLC+MEM(part) coloring outperforms MEM+LLC coloring in this case. This is because MEM+LLC coloring partitions the entire memory and LLC space, which restricts the overall memory space. This restriction increases the number of LLC

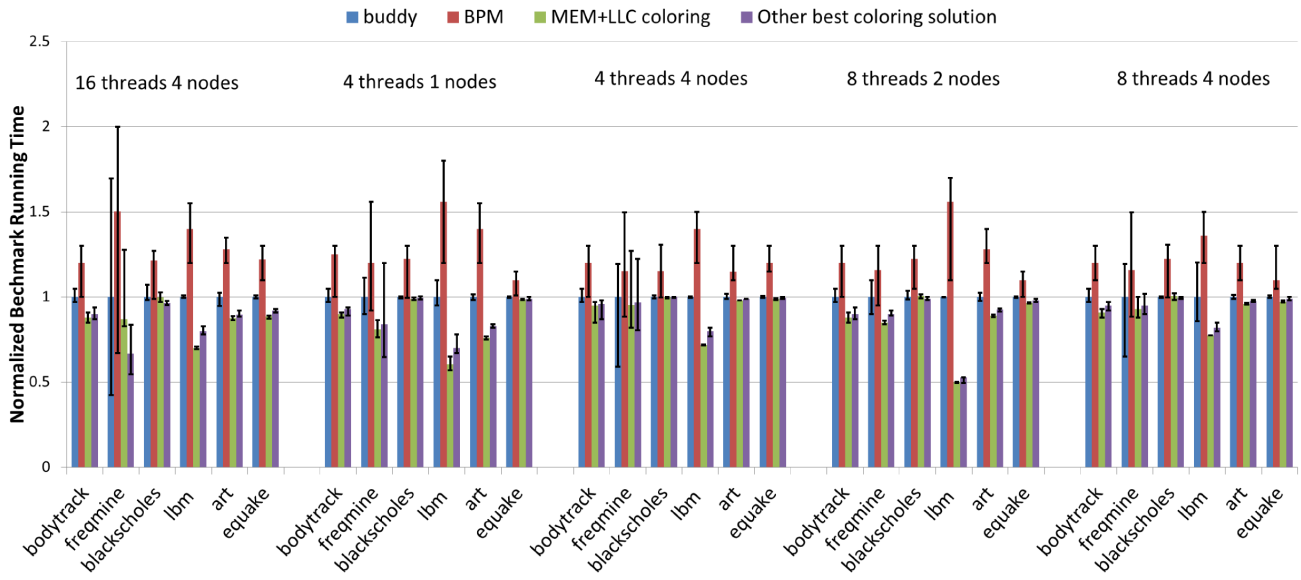

Fig. 11. Benchmark running time

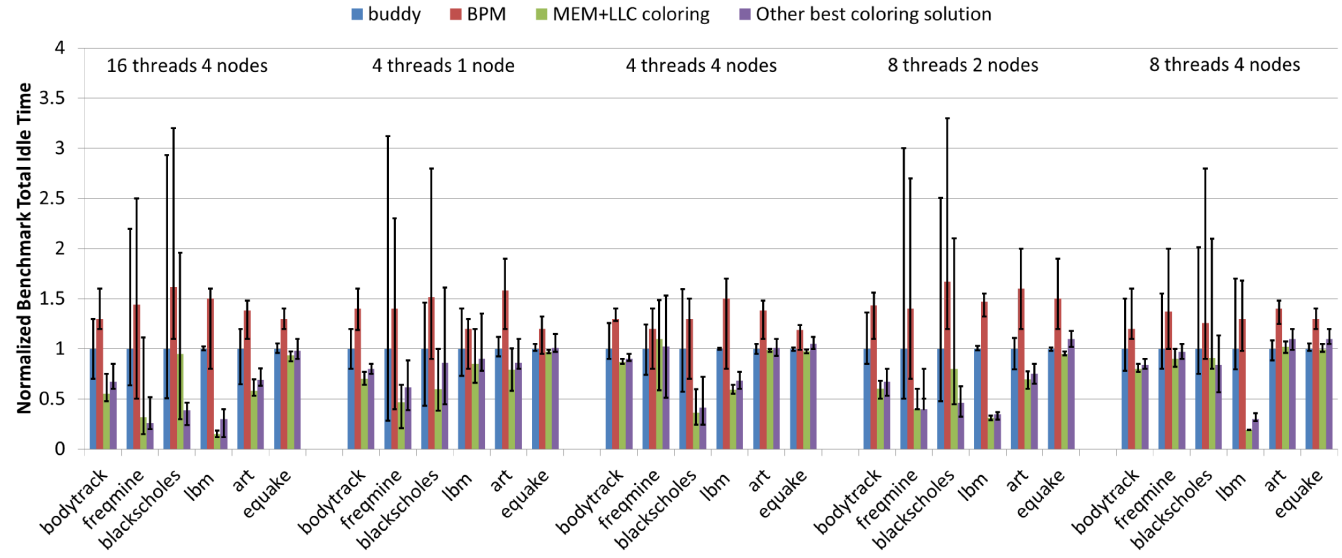

Fig. 12. Benchmark idle time

cache conflict misses. Hence, the LLC+MEM(part) coloring method is an interesting trade-off for memory coloring without restrictions. Sometimes, "partly coloring", like LLC+MEM(part) or MEM+LLC(part), may result in better performance than "fully coloring", such as MEM+LLC.

VI. Related Work

Blagodurov et al. [11] and McCurdy et al. [12] describe performance problems that NUMA can present for multithreaded applications and investigate their causes. Marathe et al. [13] propose a profiler to optimize data placement of multithreaded programs via hardware-generated memory traces. Lachaize et al. [14] use profiling to help programmers understand why and which memory objects are accessed remotely and allow them to choose efficient application-level optimizations for NUMA systems. Majo et al. [15] study which factors limit the performance of multithreaded programs on modern NUMA multicores and describe source-level techniques to address these problems. Yun et al. [16] propose a new parallelism-aware memory interference delay analysis for parallel requests. TintMalloc address these problems by reducing memory access divergence on NUMA systems.

Several scheduling algorithms [1]–[3] have been proposed to reduce memory contention. ATLAS [2] proposes a thread scheduling algorithm that optimizes the service order of threads periodically based on the amount of service they have attained from the memory controllers so far to improve system throughput. Kim et al. [2] dynamically divide threads with similar memory access behavior into two separate clusters (memory-non-intensive or memory-intensive) and employ different memory request scheduling policies in each cluster. Muralidhara et al. propose channel partitioning, which maps the data of applications that are likely to severely interfere with each other to different memory channels [3]. In [17], the frequently accessed data from different rows are dynamically

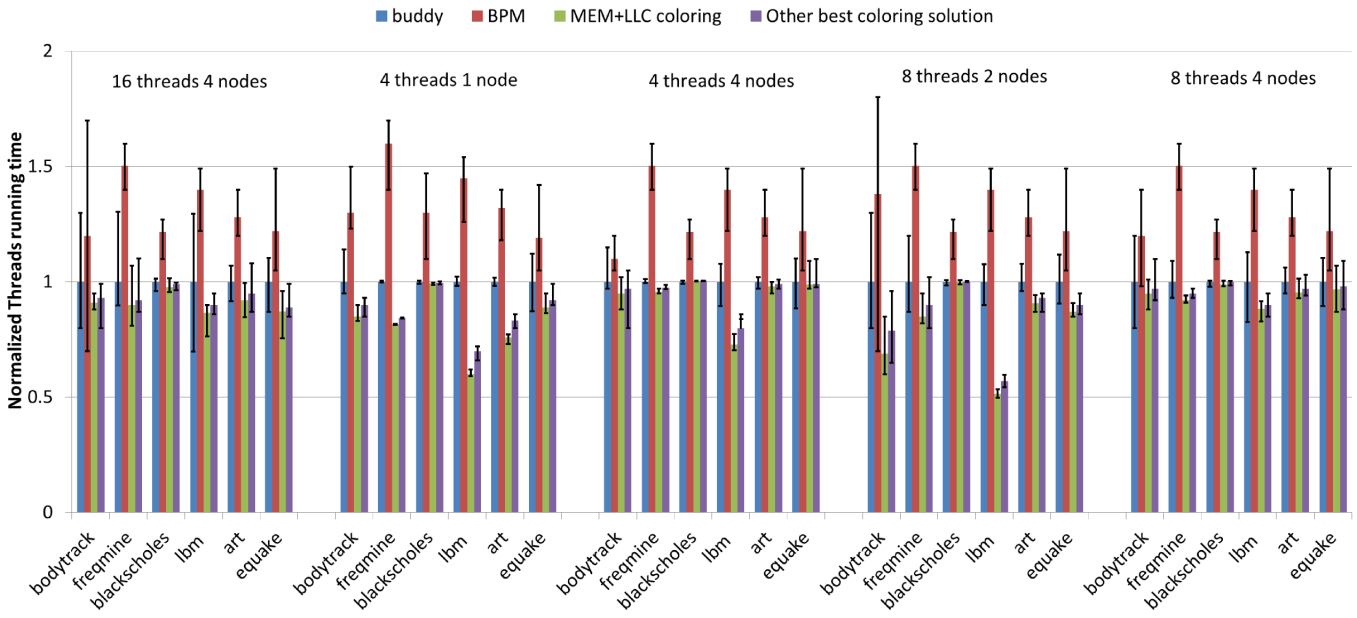

Fig. 13. Threads running time

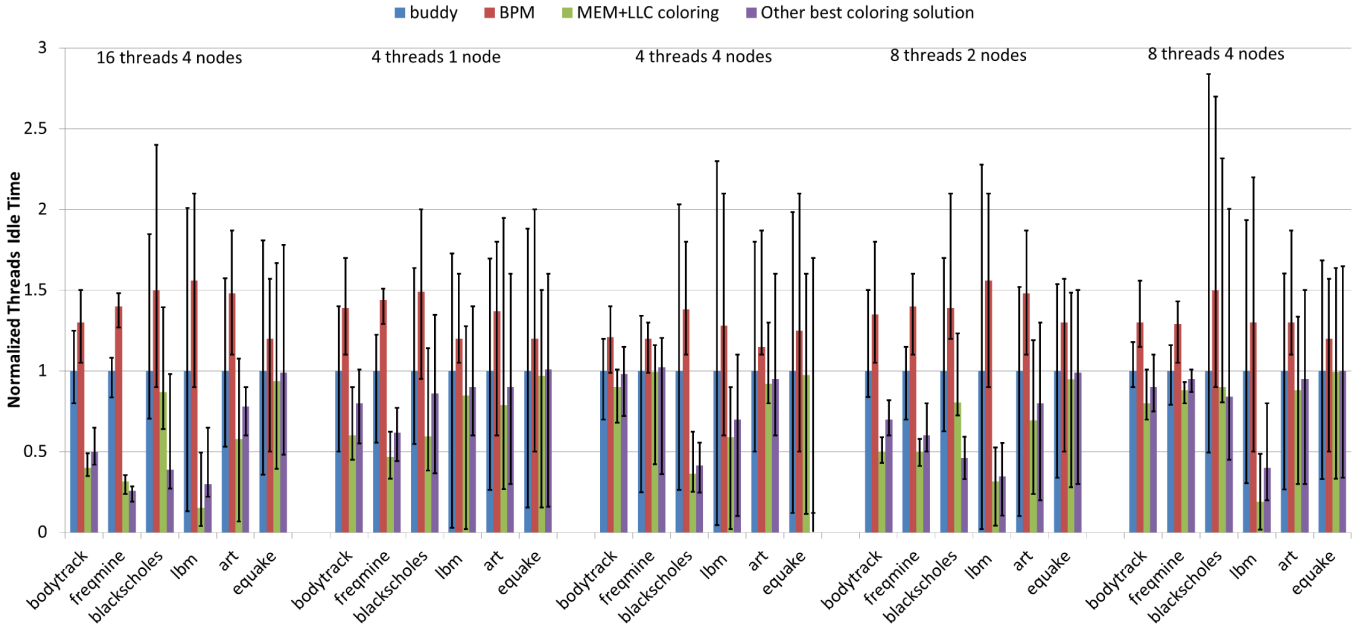

Fig. 14. Threads idle time

migrated into the row buffer in memory, which can increase the memory row buffer usage and system performance. In contrast to TintMalloc, memory access contention cannot be removed entirely for all cases with their scheduling algorithms.

Several groups proposed scheduling or page placement to solve the data sharing problem. Li et al. [18] present a loop scheduling algorithm to exploit data locality and dynamically balance the load. Majo et al. [19] use program-level transformations to eliminate remote memory accesses. Ogasawara et al. [20] propose an online method for identifying the preferred NUMA memory nodes of objects during garbage collection. Broquedis et al. [21] propose a hierarchical approach named FORESTGOMP for the execution of OpenMP threads on multicore platforms. To maintain a good balance of threads, FORESTGOMP dynamically generates structured trees out of OpenMP programs and collects relationship information about threads and data. However, all these approaches introduce overhead and cannot completely eliminate data sharing.

Techniques to increase cache data locality that reduce shared memory contention are proposed in [4]–[7]. Majo et al. [4] describes two scheduling algorithms, maximum-local optimizes for maximum data locality and N-MASS reduces data locality to avoid the performance degradation caused by cache contention. Cho et al. [5], Perarnau et al. [6] and Suh et al. [7] employ software page coloring to partition shared caches for concurrently running threads, which eliminates the contention between threads and hence reduces conflicts at the cache level. DRAM partitioning is another scheme to reduce shared

memory contention for parallel execution on multicore platform. The basic idea of using DRAM organization information in allocating memory at the OS level is explored in recent work [8], [9]. Awasthi et al. [9] examine the benefits of data placement across multiple memory controllers in NUMA systems. They introduce an adaptive first-touch page placement policy and dynamic page-migration mechanisms to reduce DRAM access delays in multiple memory controllers system but do not consider bank effects, nor do they reduce cache conflicts. Mi et al. [22] develop a hardware/software co-design for bank coloring using address bit selection (XOR) but do not exploit virtual to physical address mapping purely in software as TintMalloc does. Palloc [8] is a DRAM bank-aware memory allocator that provides performance isolation on multicore platforms by reducing conflicts between interleaved banks. Liu et al. [10] modify the OS memory management subsystem to adopt a page-coloring based LLC and bank level partition mechanism (BPM), which allocates specific LLC and DRAM banks to specific cores (threads). In contrast, our TintMalloc approach not only partitions memory banks and the LLC but also consider the locality of memory controllers, which is unprecedented in this combination.

VII. Conclusion

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

We assess TintMalloc in a number of experiments on a multicore platform with microbenchmarks as well as SPEC and Parsec OpenMP codes. Experimental results indicate that TintMalloc makes parallel tasks more balanced and enhances parallel system performance by reducing overall runtime and idle time at barriers.

Acknowledgment

This work was supported in part by NSF grants 1239246, 1525609, and 0958311.

References

[1] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 65–76.

[2] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *International Symposium on High Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[3] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 374–385.

[4] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *International Symposium on Memory Management*, vol. 46, no. 11. ACM, 2011, pp. 11–20.

[5] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *MICRO-39*. IEEE Computer Society, 2006, pp. 455–468.

[6] S. Perarnau, M. Tchiboukdjian, and G. Huard, "Controlling cache utilization of hpc applications," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 295–304.

[7] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[8] H. Yun, R. Mancuso, Z.-P. Wu, and 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.

[9] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and 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.

[10] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and 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.

[11] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 557–558.

[12] C. McCurdy and 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.

[13] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-directed page placement for ccnuma via hardware-generated memory traces," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, pp. 1204–1219, 2010.

[14] R. Lachaize, B. Lepers, V. Quéma *et al.*, "Memprof: A memory profiler for numa multicore systems." in *USENIX Annual Technical Conference*, 2012, pp. 53–64.

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

[16] H. Yun, R. Pellizzoni, and P. Valsan, Kumar, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *Euromicro Conference on Real-Time Systems*, 2015.

[17] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing dram efficiency with locality-aware data placement," *ASPLOS*, vol. 45, no. 3, pp. 219–230, 2010.

[18] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *International Conference on Parallel Processing*, vol. 93, 1993, pp. 140–147.

[19] Z. Majo and 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.

[20] T. Ogasawara, "Numa-aware memory manager with dominant-thread-based copying gc," in *Conference on Object Oriented Programming Systems, Languages and Applications*, 2009, pp. 377–390.

[21] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of openmp applications for multicore architectures," in *International Parallel & Distributed Processing Symposium*. IEEE, 2010, pp. 1–10.

[22] W. Mi, X. Feng, J. Xue, and Y. Jia, "Software-hardware cooperative dram bank partitioning for chip multiprocessors," in *Network and Parallel Computing*. Springer, 2010, pp. 329–343.