ABSTRACT

GOWNIVARIPALLI, YASASWINI JYOTHI. Hybrid Cache, Bank, and Controller Aware Coloring for Multicore Real-Time Systems. (Under the direction of Dr. Frank Mueller.)

Real-time systems require tight bounds on the Worst Case Execution Time (WCET) of a task. Modern multicore CPUs provide multiple levels of storing data (caching) and different paths to access memory (banks) that improve the performance but also introduce unpredictability in task execution time. To obtain predictable task execution times, developers have to consider the WCET, which may void the benefits of modern CPUs and may lead to low utilization of resources.

On multicore platforms, caches present a source of unpredictability as standard heap allocated regions do not provide guarantees on the cache set that will hold a particular page translation. This unpredictability can lead to cache misses and consequently inter-task interference resulting in loose bounds on the WCET. Memory access latency varies significantly depending on where the data is located in Non-Uniform Memory Access (NUMA) systems and how banks are interleaved. Data allocations without locality awareness may experience high memory latency so that execution times become highly unpredictable, resulting in loose bounds on the WCET of tasks and overly conservative scheduling with low utilization of modern CPUs.

The goal of this work is to obtain tighter bounds on the WCET of a task while still using the benefits of modern CPUs. The coloring technique is applied for the Last Level Cache (LLC) and memory for this reason. The design reuses the existing Application Programming Interface (API) so that changes are transparent. The interface works with a minimal modifications to existing applications to provide LLC and bank-aware heap allocation.

Synthetic benchmark results indicate that with the integrated LLC+bank coloring, a 26.43% reduction in task execution time (with standard deviation: 0.01) is achieved relative to shared same LLC+bank (worst case) (with standard deviation: 0.39). Parsec standard benchmarks indicate that a 24-89% reduction in execution time can be achieved relative to shared same LLC+bank (worst case). Hardware performance counters indicate that the LLC miss rate can be reduced by 39% with the LLC coloring and by 95.3% with the integrated LLC+bank coloring relative to shared same LLC+bank (worst case). Experimental results indicate that the LLC, memory bank, and memory controller-aware coloring reduces memory latency, avoids inter-task conflicts, and improves timing predictability of real-time tasks.

Hybrid Cache, Bank, and Controller Aware Coloring for Multicore Real-Time Systems

by
Yasaswini Jyothi Gownivaripalli

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

Dr. Xipeng Shen

Dr. Guoliang Jin

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my teachers who taught me the joy in learning

BIOGRAPHY

The author received her B.Tech degree in Computer Science and Engineering from Jawaharlal Nehru Technological University, Hyderabad, India. She worked as a kernel developer in the Defence Research and Development Organization, India. She came to NC State in Fall 2013 as a Master's student in the department of Computer Science. She has been working under the guidance of Dr. Frank Mueller as a Research Assistant since May 2014.

ACKNOWLEDGEMENTS

I would like to thank my adviser Dr. Frank Mueller for his guidance, technical insights, and encouragement. I am thankful to my parents for their constant encouragement and support. I would like to thank Dr. Emerson Murphy-Hill, Dr. Xipeng Shen, and Dr. Guoliang Jin to serve on my committee. I am thankful to my labmates in the Systems Research lab for providing a joyful working environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Multicore processors share resources like the LLC and memory across all the cores. When part/all of the data used by different tasks map to the same sets in LLC, eviction and refill of the cache line may occur for each task. If tasks use the same bank to access memory there will be contention. Both scenarios will lead to variable latency in task execution. A Non-Uniform Memory Architecture partitions sets of cores into a "node" with a local memory controller, such that memory accesses may be resolved locally or via the on-chip interconnect from a remote node and its memory. Each memory node consists of multi-level resources called channel, rank and bank. The banks can be accessed in parallel to increase memory throughput. When tasks running on different cores access memory concurrently, performance varies significantly across cores depending on which node data is located on and how banks are shared. The latency of accessing memory of a remote node is significantly longer than that of accessing memory local to the node. Even for the same memory node, conflicts between shared-bank accesses result in unpredictable memory access latency. The latency is worse when all running tasks map their data to same cache line. As a result, the WCET bound of tasks has to be conservatively (over-)estimated, which may result in low system utilization due to potentially inferior memory performance.

1.1 Hypothesis

We hypothesize that dynamic memory allocation can be controlled in software, so that real-time tasks will not interfere with one another in terms of Last Level Cache (LLC) and memory bank accesses in multicore systems.

1.2 Contributions

We have designed and implemented a coloring approach that affects heap memory allocations at LLC, bank, and controller levels. Two tasks can request colors in six possible ways: 1) different local controllers and different LLC-sets; 2) different local controllers and same LLC-sets; 3) different banks of same controller and different LLC-sets; 4) different banks of same controller and same LLC-sets; 5) same banks and different LLC-sets; and 6) same banks and same LLC-sets. Option one gives the lowest latency and option six the highest for accesses. Programmers can assign one (or more) colored LLC-sets and memory banks exclusively per task. Migration of a task to a different core can be avoided by pinning it to a core. The underlining kernel establishes that heap allocation is done such that LLC-set and memory bank isolation is established across tasks. This effectively shortens the WCET and makes execution more predictable. A novelty of our LLC and memory-aware coloring API is that it does not need any linking of additional libraries. Our approach requires the developer to make a single mmap() call during initialization with an appropriate set of color arguments to enable colored allocation for all subsequent malloc() calls. We have modified the Linux kernel to support colored allocation.

Our design and implementation contributions are:

1. We design and implement a new heap allocator that guarantees isolation between two tasks while accessing LLC.

2. We conduct an experimental evaluation of our heap allocator to demonstrate task isolation realized by LLC coloring.

3. We integrate bank and controller aware coloring with our LLC coloring.

4. We conduct an experimental evaluation of our heap allocator to demonstrate task isolation realized by integrated LLC, bank, and controller aware coloring.

We observed a 25.43% reduction in execution time (with standard deviation: 0.01) of synthetic benchmarks with LLC+bank coloring relative to shared same LLC+bank (worst case) (with standard deviation: 0.39). Parsec standard benchmarks indicate that we can achieve a 24-89% reduction in execution time with LLC+bank coloring relative to shared same LLC+bank (worst case). Hardware performance counters indicate that we can reduce the LLC miss rate by 95.3% with our integrated LLC+bank coloring relative to shared same LLC+bank (worst case).

Chapter 2

Background

Data access latency is affected by the following aspect: (1) whether or not the data is located in the cache; (2) how the memory banks interleave and how much of the accesses contend; and (3) where the data is actually located, i.e., the local memory node or the remote memory node. In this section, we provide a brief description of these three aspects. Our description is based on LLC and DDR3 SDRAM memory systems.

2.1 Cache Organization

Most modern CPU architectures have multiple levels of cache hierarchy within in a single chip. For example, the cache hierarchy in an AMD Opteron 6128 single socket is as shown in Fig. 2.1. Each core has access to a local L1 and L2 cache 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.

2.2 DRAM Organization

Modern day DRAM systems are organized as a group of controllers, where each controller is associated with a set of cores as shown in Fig. 2.2. Every memory controller has multiple resources, namely rank, bank and channel, where each rank consists of multiple banks, and banks can be accesses in parallel. Average throughput can be improved by interleaving access between multiple channels. For each bank, a row buffer and a storage array is organized into rows and columns. If multiple tasks access (write or read) the same bank, they contend for the row buffer. The data loaded by one task may be evicted by the other tasks, i.e., the latency of memory accesses will be increased if multiple threads access the same bank concurrently. For each bank, a row buffer and a storage array is organized into rows and columns. When a

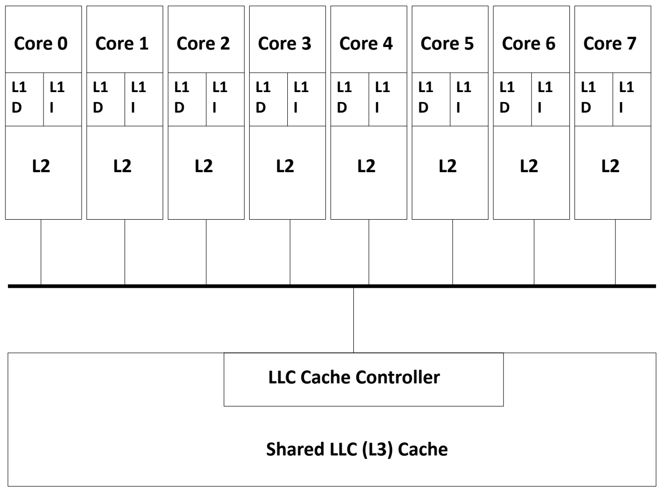

Figure 2.1: Multicore Cache Organization

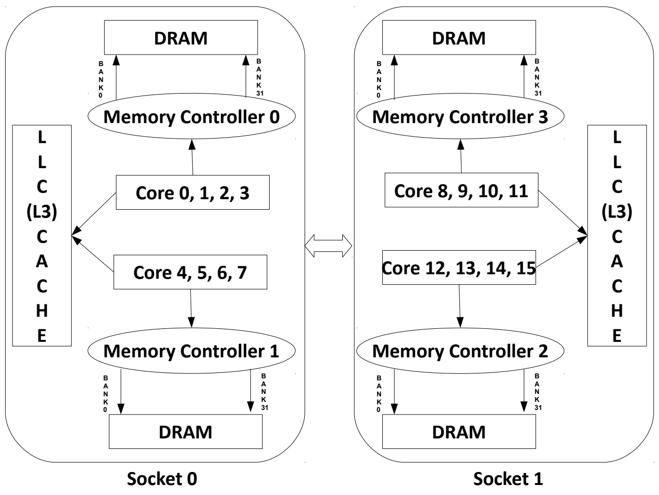

Figure 2.2: AMD Opteron 6128

memory request is issued, the data is first loaded into the row buffer through an Activate (ACT) command. If the data has previously been placed in the row buffer, it can be read or written by a column access strobe (CAS) command. If a second request wishes to access a different row from the same bank, the system first needs to write the data in the row buffer back to the array with a Pre-charge (PRE) command before loading the second row into the row buffer. Finally, a periodic Refresh (REF) command must be issued to all banks and, as a side effect, data in the row buffer is written back to the data array.

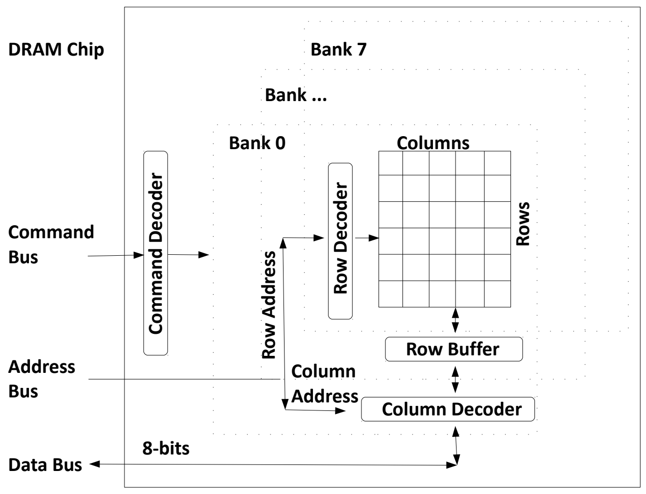

Figure 2.3: DRAM Device Organization

2.3 Memory Controller

The memory controller translates memory requests (read/write) to corresponding DRAM commands (Fig. 2.4). It acts as a mediator between the LLC of a processor and the DRAM devices. In an architecture with multiple memory controllers, a memory (DRAM device) that is directly connected to the controller (local memory) will have the shortest access latency. A memory access from one controller to memory of another controller (remote memory) incurs additional cycles of memory load penalty compared to local memory access. This is due to interconnect access latency. Local memory access has lower latency and is also contention free from interconnect access synchronization delays. It is beneficial to place the data locally as much as possible and avoid remote memory access. In practice, it is hard to completely avoid such accesses to remote controllers as a task's usage of data is complex and most of the times tasks run concurrently. This leads to unpredictability in task execution times depending on where the data is placed. Without localization of memory references, we have to always consider remote accesses, which will result in overly pessimistic WCET bounds and low utilization of system resources.

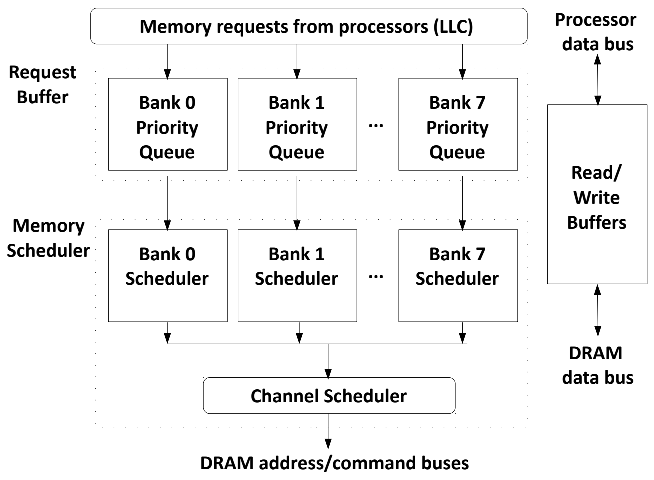

Figure 2.4: Structure of DRAM Controllers

Chapter 3

Design

3.1 Kernel design

Our design is based on the number of sets available in LLC, and the number of banks available within a memory controller. We assign llc_color and bank_color to each physical page based on the LLC-set and the memory bank to which the physical page belongs to (Fig. 3.3).
The LLC color of a physical page is determined as follows:

$$NLLC = S/(W * L) \tag{3.1}$$

$$B = \{lg(P)...lg(L + lg(NLLC) - 1\} \tag{3.2}$$

$$PFN = physical\_address >> lg(P) \tag{3.3}$$

$$llc\_color = (PFN)\&(2powNLLC) \tag{3.4}$$

where
NLLC - Number of LLC colors,
B - bits to identify LLC color of page,
PFN - page frame number,
S - LLC size,
W - LLC ways,
L - LLC line size,
P - page size.

Fig. 3.1 shows the bits used for identifying the LLC color of a physical page on AMD

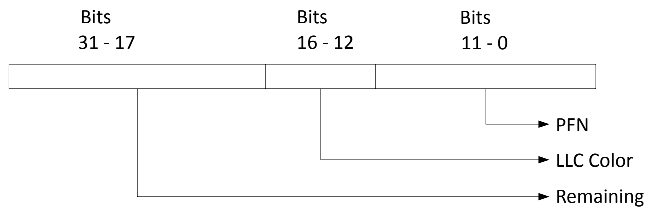

Figure 3.1: LLC color identifier on AMD Opteron 6128

Opteron 6128 (Magny core) [1].

The bank color of a physical page is determined as follows:

$$bank\_color = channel * NC + rank * NR + bank * NB \tag{3.5}$$

where
channel - channel to which the physical page belongs,
rank - memory rank to which the physical page belongs,
bank - memory bank to which the physical page belongs,
NC - number of channels available within a memory controller,
NR - number of ranks available within a memory controller,
NB - number of banks available within a memory controller.
Channel, rank, and bank information is obtained from AMD Opteron 6128 (Magny core) [1].

3.2 Application Programming Interface (API) design

3.2.1 Normal API

void *mmap (start, length, prot, flags, fd, offset);

3.2.2 Color API

void *mmap (color_arg, 0, COLOR_ALLOC, ...)
color_arg: Used to represent the color identification bits as shown in Fig. 3.2.
size: Zero is used to mark that this is a special call for coloring
prot: COLOR_ALLOC is used to mark colored allocation for this task

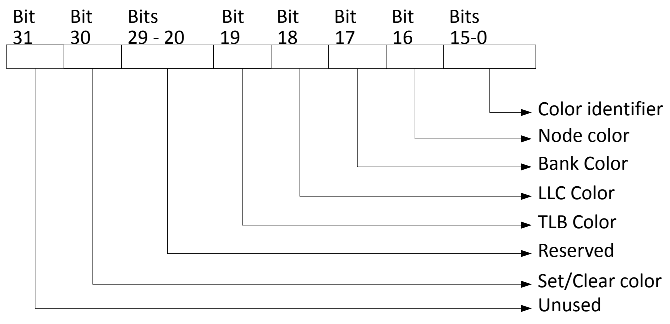

Figure 3.2: mmap() color_arg: Bit identifiers

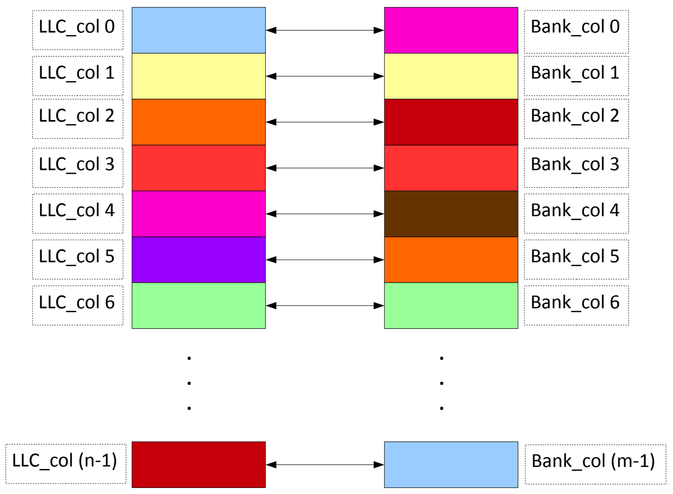

Figure 3.3: Coloring LLC and Memory

Chapter 4

Implementation

In our design, the Application Programming Interface (API) for colored allocation is provided through the mmap() system call. Using mmap() instead of a new system call reduces the developer efforts of linking with a new library and it does not require an extra system call in the kernel. For colored allocation, the address is used to pass the color information(set/clear color, coloring level: LLC/memory/both, color identifier), and the length argument of mmap() is passed as zero. We differentiate a mmap() colored request call from a regular call by interpreting the length argument in the kernel. In regular Linux, when the kernel observes the length as zero, it returns an error to the application. In our colored implementation, when the kernel identifies the length argument as zero, it considers it as a colored call and marks in the task struct of the process that it is requesting a colored allocation. The kernel stores the requested LLC and bank color(s) in *task_struct* of the process. All subsequent calls to malloc() will be served by our colored allocation strategy in the kernel. In summary, when an application needs a colored page, it issues an mmap() call with the colored information as the first argument (address) and zero as the second argument(length). Fig. 4.1 demonstrates the interpretation done in the kernel when the mmap() system call is called by the application. When the application requires more memory than can be satisfied by a single memory bank, multiple banks needed to be requested by making multiple mmap() calls.

LLC and bank coloring is implemented by modifying the page allocation strategy in the Linux kernel. For experimental evaluations, we have modified Linux kernel version 2.6.32.27, but our implementation does not have any version dependency. Our implementation does not require any hardware features, this differs from other implementations [15]. During the booting phase, the LLC color and bank color of each page is stored in the *struct page* data structure of kernel, in *page*$\rightarrow$*llc_color*, and *page*$\rightarrow$*bank_color*, respectively. The number of LLC colors is same as the number of available sets in the LLC. So, the LLC color of page is determined by the bits that identify the LLC set to which the physical frame maps. The bank color is identified by channel, rank and bank information obtained from Peripheral Component Interconnect (PCI) registers/BIOS. Coloring is provided as a configuration option in the Linux kernel, which can be selected using the standard Linux kernel configuration.

The Linux kernel page allocation strategy is modified to support colored allocation requests. In our implementation, the Linux kernel maintains pages in a corresponding *color_list*. It is a two-dimensional array supporting a combination of LLC and bank colors. When the task requests an LLC color B and a bank color A, color_list[A][B] is searched. If a free page is found, it is returned. If the list is empty, the kernel will try to pull pages from buddy allocation lists and add them to the respective *color_list*. Once the required entry with matching LLC and bank color is found, it will be returned. If the task is requesting for only LLC coloring, then the page with a matching LLC color is returned without considering its bank color. Similarly, if the task is requesting only bank coloring then the page with a matching bank color is returned without considering its LLC color. If the task has requested for multiple LLC/bank colors, the first matching page is returned. The overhead of building color lists will be imposed only when the corresponding list is empty. Therefore, subsequent requests for allocation will be faster.

The Buddy allocator in the Linux kernel maintains pages in different order lists, i.e., in an array of size *MAX_ORDER*. The 0th element of the array points to a list of free page blocks of size 2^0 or 1 page, the 1st element points to a list of 2^1 or 2 pages, and similarly the *MAX_-ORDER* points to 2^{MAX_ORDER1} number of pages. This eliminates the need of a larger block to be split to satisfy a request where a smaller block would have sufficed. The page blocks are maintained on a linear linked list via page→list. In our implementation, the colored allocation and freeing is done only for order-0 lists i.e., we allocate/free only one page at a time because most application requests result in a single page being allocated as part of serving a page-fault.

The Linux kernel page management is modified to add the freed page back to either the color list or to the regular buddy lists. If the task has requested colored allocation and frees the page in colored mode then the page is added to the corresponding color list. If the task has requested colored allocation and frees the page *after* clearing the colored mode or if the task has not requested colored allocation (i.e. a regular task) then the page is freed using the general Linux freeing policy. In our implementation, the changes to the Linux kernel are minimal as the coloring information of a physical page is stored in Page Table Entry (PTE) and task-related coloring information is stored in Task Control Block (TCB) data structures.

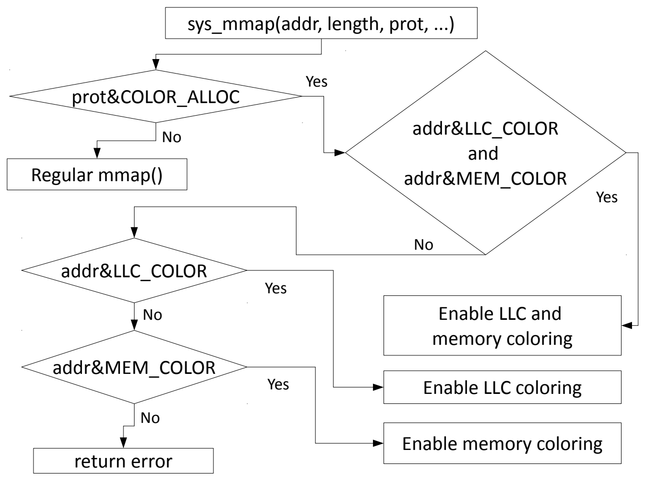

Figure 4.1: mmap() API and control flow for coloring

Chapter 5

Architecture

We investigated the LLC and memory organization of contemporary architectures. The ARM Tegra3 (Kayla) [2] has a smaller associative LLC (8 ways), one/two memory banks, which made is hard to provide memory isolation on this platform. The AMD Opteron 6128 (Magny core) [1] has a higher associative LLC (96 ways), multiple memory banks (32) per controller, and four memory controllers.

The platform used for running our Linux colored implementation and experiments is a two socket SMP with AMD Opteron 6128 (Magny core) processors [1]. Each socket has two memory controller nodes, and each memory controller accesses four local cores. In total, the system has sixteen cores. Each core has a private L1 cache (64KB I-cache and 64KB D-cache) and a private L2 cache (512KB unified). All 8 cores of a socket share an L3 cache (12MB unified), which is the LLC. There are two nodes per socket and nodes are connected via Hypertransport as shown in Fig. 2.2. The core frequency is between 800MHz-2GHz with a governor that selects 2GHz once a CPU-bound task starts running. There are four memory controllers, two per socket (per 8 cores), connected via Hypertransport at a link speed of 1.8GHz with different distances to controllers (1 hop to the local controller, 2 hops to the other controller within a socket and 3 hops to controllers of the other socket). There are two channels per memory controller, two ranks per channel and eight banks per rank, i.e., 128 banks altogether. All banks can be accessed in parallel.

Chapter 6

Results

We assess the performance of LLC aware allocation, LLC plus bank aware allocation, and LLC plus controller aware allocation using synthetic micro-benchmarks and standard benchmarks from Parsec [4] and NAS Parallel Benchmarks [3].

6.1 Synthetic Benchmarks

We assessed the access latency to different controllers with a synthetic benchmark. We bind the synthetic program to a specific core in one controller and try to access memory across different controllers in the system. Fig. 6.1 depicts access latency (y-axis) for different access types (x-axis). We observed that the access latency to memory for the local controller (Local-samesocket) is the lowest, followed by the remote controller in same socket (Remote-samesocket). The memory access latency to controllers in a different socket (Remote-diffsocket) is the highest.

Using a synthetic micro-benchmark we allocated a large amount of memory to a task. We created four tasks, each pinned to a different core, three tasks run in the background and one task runs in the foreground. The purpose of the background tasks is to create conflicts for the foreground task by accessing various memory locations. We assigned banks and LLC sets based on the application requests and we measured the time required to complete the foreground task. We ran a synthetic benchmark to observe the latency in accessing shared banks and shared LLC sets. Fig. 6.2 depicts the overall execution time (y-axis) across our different coloring techniques for a sequence of ten experiments (x-axis). We observed that the most conflicts are created when all tasks share LLC sets and banks (shared LLC+bank) and task execution time is the highest (29 seconds, standard deviation 0.39). The foreground task has lower execution time (27 seconds, standard deviation 0.01) when tasks access different LLC sets (LLC coloring). Execution time for the foreground task is lower (23 seconds, standard deviation 0.32) when tasks access different banks (bank coloring). The foreground task has the lowest and bounded

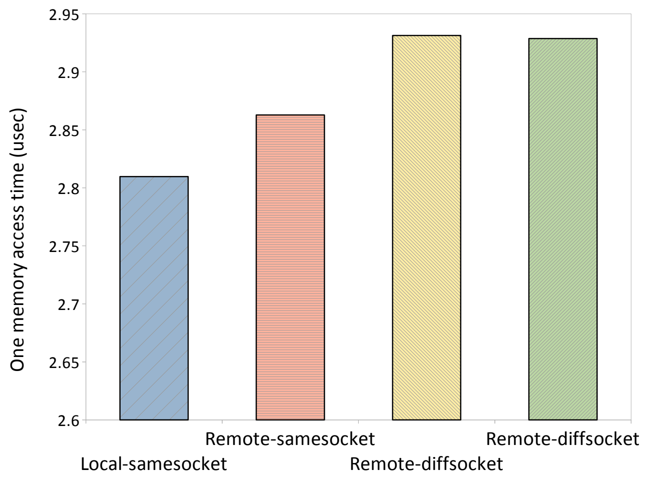

Figure 6.1: Latency as a function of distance to memory controller

execution time (20.6 seconds, standard deviation 0.01) when all tasks access not only different banks but also different LLC sets (LLC+bank coloring). We observed a 26.43% improvement with our LLC+bank coloring over shared LLC+bank along with lower and tightly bounded execution time with our coloring techniques.

We ran the same benchmark to observe LLC coloring benefits along with controller-aware allocation, in controller-aware coloring (controller) each task is assigned to a different core in a different controller. This ensures that tasks experience the least conflicts as there is less contention at the hardware level. Fig. 6.3 depicts the overall execution time (y-axis) across our different coloring techniques for a sequence of ten experiments (x-axis). We observe that controller allocation in Fig. 6.3 has a lower execution time than our LLC+bank coloring in Fig. 6.2. This is because controller-aware allocation inherently avoids queuing contention at the controller and contention for banks. Controller-aware allocation still has LLC conflicts. We also applied our LLC coloring on top of controller-aware coloring (LLC+controller coloring). We observe that our LLC+controller coloring gives lower execution time (1.63 seconds, standard deviation 0.01) than controller-aware allocation (1.66 seconds, standard deviation 0.01). Our LLC+controller's allocation worst case is same as controller-aware allocation for two of the ten runs of the experiment. This is due to the overhead of maintaining multiple color lists for coloring.

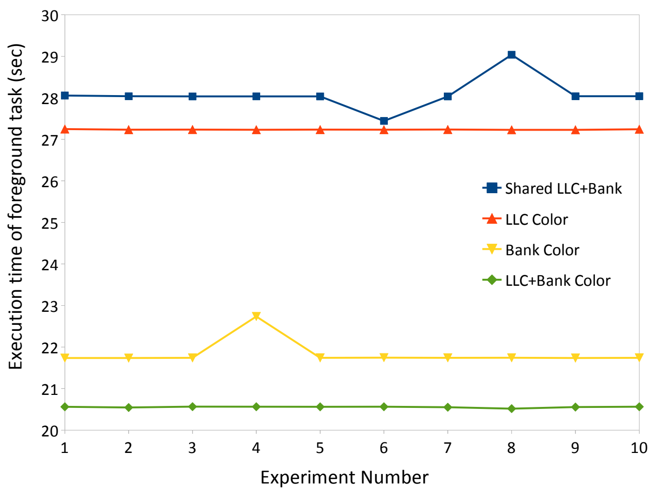

Figure 6.2: Synthetic benchmark with LLC and memory(bank) coloring

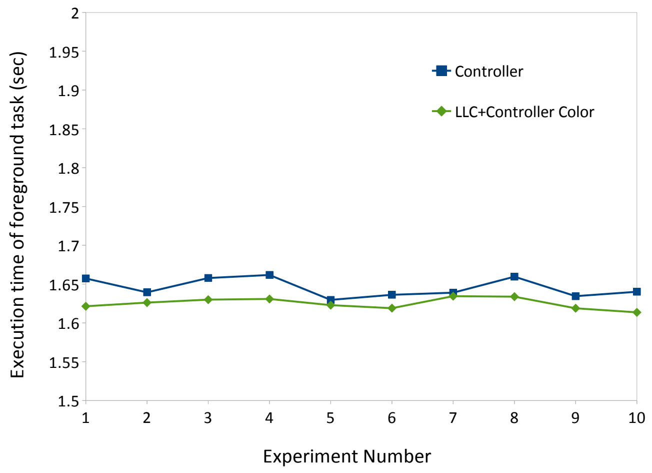

Figure 6.3: Synthetic benchmark with LLC and controller coloring

6.2 Standard Benchmarks

We have modified standard benchmarks to replace static allocation with dynamic allocation. We also pin a task/thread to a core to avoid task/thread migration effects.

6.2.1 NAS Parallel Benchmark Results

We observe performance benefits of LLC coloring, bank coloring, and LLC+bank coloring over shared LLC+bank coloring using the NAS Parallel Benchmark (NPB) suite [3]. The NPB suite

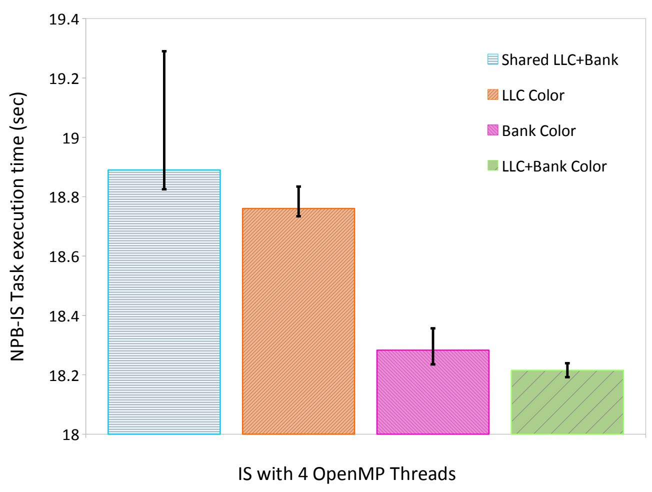

Figure 6.4: NPB-IS benchmark

is a set of programs designed to evaluate the performance of parallel execution. We investigate the OpenMP version of IS with four OpenMP threads. IS is written in C. The other NAS codes are Fortran benchmarks and do not work with heap allocation. IS is an integer sort application with many random memory accesses. Each of the four OpenMP threads is pinned to a different CPU core and dynamically allocates memory per thread. In the worst case, the normal buddy allocator can assign all threads the same banks and LLC sets, which would inflict conflicts when accessing memory. Our LLC-aware and bank-aware coloring ensures that neither remote memory nor shared memory bank accesses are issued by these threads. Fig. 6.4 depicts the average execution time of the program over ten runs for different allocation schemes (y-axis) along with error bars denoting the maximum and minimum execution time. We observe that the execution time for the first run is higher compared to the next runs. This is due to the initial overhead for creating color lists. The worst case execution time of our LLC+Bank coloring (18.2 seconds) is less than the best case execution time of shared LLC+bank model (18.8 seconds). For real-time scheduling, a tight bound on the execution time is important to guarantee good system utilization, which we achieve with our coloring technique with less variance in execution time across different runs as indicated by the standard deviation in Table. 6.1. LLC+bank aware allocation also utilizes system resources better, as this avoids LLC conflicts and interleaves banks across tasks. We make the following observations: The execution time of shared LLC+bank (worst case of buddy allocation) is significantly higher than that of LLC coloring and bank coloring. With integrated LLC+bank coloring we not only decrease the execution time, but also bound it more tightly with less variance.

Table 6.1: NPB-IS

	standard deviation
Shared LLC+Bank	0.14
LLC Color	0.03
Bank COLOR	0.06
LLC+Bank COLOR	0.01

6.2.2 Parsec Benchmark Results

We investigate the performance and predictability for the PARSEC benchmark suite [4]. The Parsec suite focuses on emerging workloads and is said to be representative of next-generation shared-memory programs for chip-multiprocessors. We have created a multi-task workload where one or more 'memory attackers' run in the background to assess their interference on memory latency for a foreground task similar to prior work [16], [11]. E.g., if there are four tasks in the experiment, one of them (the foreground task) is a Parsec benchmark and the others (background) are memory attackers. Fig. 6.5 shows an average execution time of the Parsec task over ten runs along with error bars denoting the maximum and minimum execution time. Shared LLC+bank (worst case of buddy allocation) has the highest latency and maximum variance as indicated by the error bars. For comparison, we have included the single run of a Parsec program ('parsec without attackers'). It uses the general buddy allocator and does not have memory attackers in the background. We observe that fluidanimate has the highest gain due to coloring: shared LLC+bank (58 seconds, standard deviation 0.18) compared to LLC+bank coloring (6.5 seconds, standard deviation 0.01) i.e., a 88.79% improvement in execution time. Swaptions has the lowest gain due to coloring: shared LLC+bank (4.2 seconds, standard deviation 0.04) compared to LLC+bank coloring (3.2 seconds, standard deviation 0.01), i.e., a 23.81% improvement in execution time. Our observation is, with LLC+bank aware allocation we can reduce the task execution time, achieve better system utilization, and provide tighter bounds/less variance on execution time.

We observe the performance improvement for multi-threaded applications by using our LLC+bank coloring. We used the OpenMP version of Parsec's blackscholes application and created experiments with 1, 4, 8, and 16 threads, where each thread is pinned to a different core. We do not have any attackers in this experiment. Fig. 6.6 depicts an average execution time of the Parsec task over ten runs along with error bars denoting the maximum and minimum execution time. We present execution time (y-axis) across our different coloring techniques for different number of threads (x-axis). We observe that one thread has the highest execution time, as it is equivalent to running a non-threaded version of the application. Four threads have lower

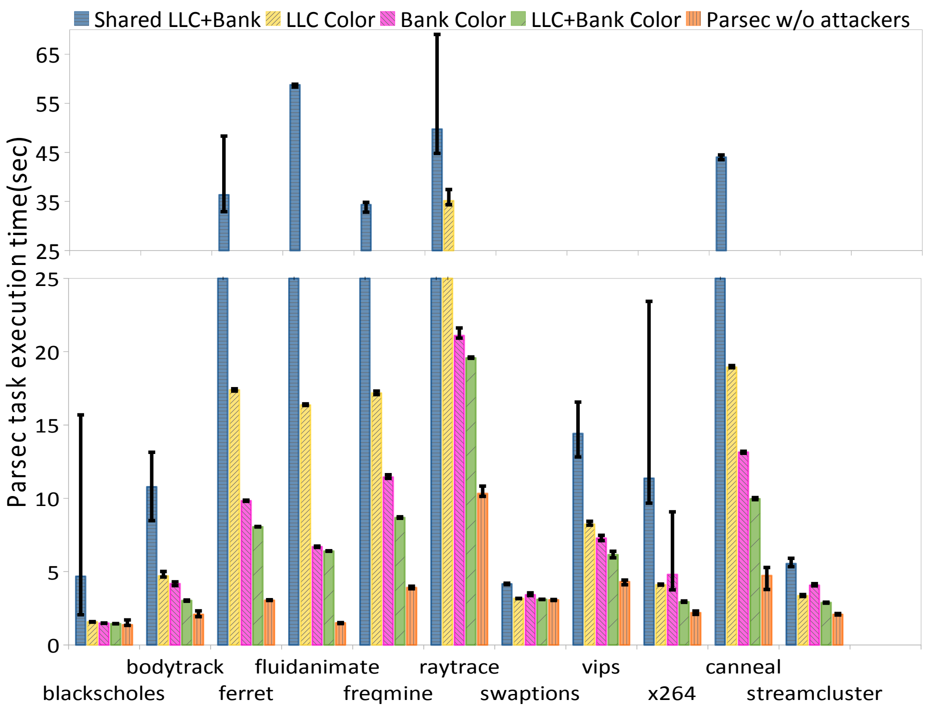

Figure 6.5: Parsec with memory attackers

execution time than one thread. This is due to parallelization and because each thread accesses a different LLC and bank, which avoids LLC conflicts and interleaves bank accessing. Eight threads have lower execution time than four threads. This is again because of parallelization and less contention for LLC sets and banks. We observe that our coloring provides better results with increased number of threads, as we avoid conflicts among threads by allocating different LLC(s) and bank(s) to each thread. We observe that when we increase the number of threads to sixteen, performance does not improve compared to four/eight threads. This is because of thread synchronization requirements and also because of inter-controller access delays while accessing a different socket. As shown in Fig. 2.2, eight cores are in one socket and the other eight cores are in another socket. Sixteen threads require more access synchronization involving multiple controllers. The result is a smaller improvement in execution time than four/eight threads. We have observed the inter-controller access latency using our synthetic benchmark in Fig. 6.1. We observe that our coloring technique has less variance across different runs, as indicated by the standard deviation in Table. 6.2. We conclude that our LLC+bank coloring achieves benefits for multi-threaded applications, but we shall also consider hardware access delays while allocating resources to tasks.

We investigate LLC misses with our LLC coloring and LLC+bank coloring using Parsec's

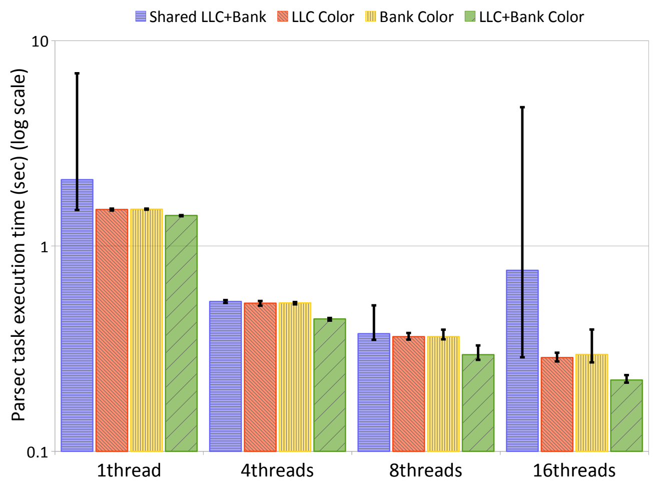

Figure 6.6: Multi threaded Parsec (Blackscholes)

Table 6.2: Multi threaded Parsec

1thread	standard deviation
Shared LLC+Bank	1.7
LLC Color	0.01
Bank COLOR	0.01
LLC+Bank COLOR	0.01
4threads	standard deviation
Shared LLC+Bank	0.01
LLC Color	0.01
Bank COLOR	0
LLC+Bank COLOR	0
8threads	standard deviation
Shared LLC+Bank	0.05
LLC Color	0.01
Bank COLOR	0.01
LLC+Bank COLOR	0.01
16threads	standard deviation
Shared LLC+Bank	1.4
LLC Color	0.01
Bank COLOR	0.03
LLC+Bank COLOR	0.01

x264 benchmark. The setup for this experiment is same as for the memory attacker model explained earlier. Three stream attackers run in the background and x264 runs in the foreground.

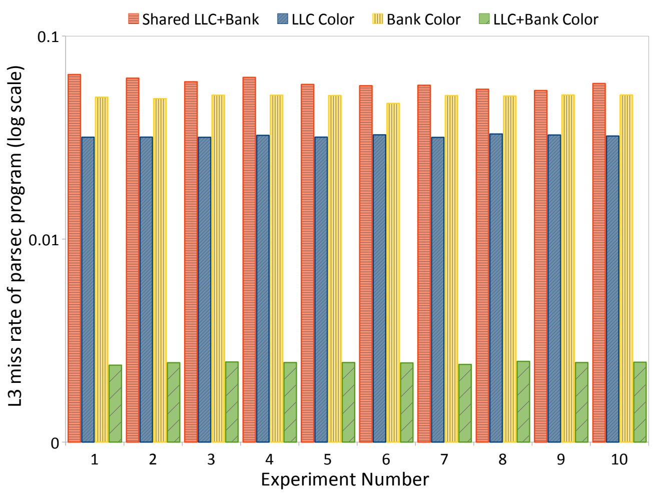

Figure 6.7: LLC (L3) miss-rate of Parsec (x264)

We measure the LLC miss rate of x264 using LIKWID [14], an open source software that we have configured for the AMD Opteron. Fig. 6.7 depicts Parsec's x264 task execution time in log scale (y-axis) across our different coloring techniques for a sequence of ten runs (x-axis). We observe nearly zero variance across different runs in all coloring techniques. We observe that the LLC miss-rate is the highest when all tasks share LLC+bank. The LLC miss-rate reduces with LLC coloring as the attackers and x264 use different LLC sets, i.e., we observed a 39% reduction in the LLC miss rate compared to shared LLC+bank. Only with bank coloring, we observed that, we do not achieve much improvement in the LLC miss rate as tasks contend for same cache sets. Our integrated LLC+bank coloring has the lowest LLC miss rate for x264 compared to shared LLC+bank as it avoids LLC conflicts and interleaves banks across attackers and the x264 code, i.e., we observed a 95.3% reduction in the LLC miss rate.

6.2.3 Real-time Experiment with Malardalen Benchmarks

For the experiments described in this section, we use benchmarks from the Malardalen suite [6] to show task isolation. We modified the benchmarks so that they use heap allocation instead of statically allocated data. We have also increased the size of the programs, as the original data sets are too small to conduct our experiments. We have created a multi-threaded program with four OpenMP threads, each thread pinned to a different core and assigned a different real-time priority, where each thread executes one of the Malardalen suite programs. All the threads use the SCHED_FIFO real-time scheduling policy of the Linux kernel. We used select (select k-th largest number), compress, st (stat), and bs (binary search) of the Malardalen suite. Fig. 6.8 depicts task execution time of each Malardalen program (averaged over ten runs) (y-axis)

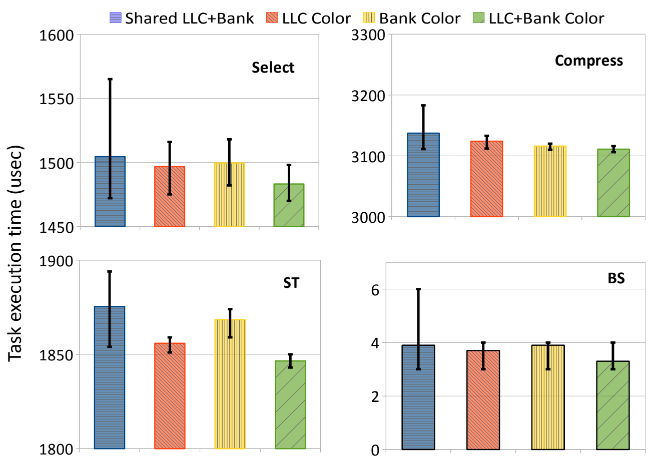

Figure 6.8: Malardalen programs as RT-tasks

along with error bars denoting the maximum and minimum execution time across our different coloring techniques (x-axis). We observed that rt-tasks have low execution time and low variance with our coloring techniques compared to shared same LLC+bank. Our LLC+bank coloring has lower execution time for all real-time tasks and also tighter bounds on the WCET as indicated by the standard deviation in Table. 6.3.

Table 6.3: Malardalen RT-Tasks

Select	standard deviation
Shared LLC+Bank	28.85
LLC Color	12.88
Bank COLOR	11.58
LLC+Bank COLOR	8.48
Compress	standard deviation
Shared LLC+Bank	21.05
LLC Color	7.64
Bank COLOR	3.09
LLC+Bank COLOR	3.02
ST	standard deviation
Shared LLC+Bank	10.39
LLC Color	2.73
Bank COLOR	6.13
LLC+Bank COLOR	2.27
BS	standard deviation
Shared LLC+Bank	0.99
LLC Color	0.48
Bank COLOR	0.32
LLC+Bank COLOR	0.48

Chapter 7

Related work

Puaut et al. [12] proposed a compiler approach to make paging more predictable. The main idea of the paper is to identify *page in* and *page out* points of virtual pages at compile time. This method relies on the static knowledge of the possible references between virtual pages of tasks. However, the focus of their work is to make *demand paging* more predictable while ours is on task isolation with respect to cache and memory.

Compiler-directed page coloring proposed by Bugnion et al. [5] involves three key phases. First, a compiler creates a summary of array references and communicates this information to the run-time system. The run-time system then uses machine-specific parameters like the cache size to generate a preferred color for the physical frame. The operating system then uses this color as a hint in a best effort to honor them. This technique is applicable to physically indexed caches. Our technique does not require profiling or modifications to the compiler.

Software cache partitioning is related to our idea. This is commonly known as cache coloring. The main idea of this technique is to color physical frames such that two frames of different color will not map to the same cache set. Liedtke at al. [8] proposed OS-controlled cache partitioning. Mancuso et al. [9] used memory profiling to identity hot pages in virtual memory. Then, the kernel subsequently allocates physical frames to these pages, such that there are no cache conflicts. Ward et al. [15] proposed cache locking and cache scheduling for the last level caches. Their scheme treats cache ways as resources that must be acquired by tasks before beginning their execution. The technique requires support for hardware cache locking from the processor. Our approach does not depend on any architectural features like cache locking.

TLSF [10] is an approach to support dynamic memory allocation in real-time systems. The main idea is to provide constant time dynamic memory allocation and deallocation. CAMA [7] builds upon TLSF to incorporate cache awareness. CAMA can allocate dynamic memory in constant time and can also guarantee the cache set that will hold this allocated memory. PALLOC [16] is a DRAM bank-aware memory allocator. It ensures that tasks running concurrently on different cores do not access physical memory that maps to the same DRAM bank. Thus, PALLOC reduces DRAM bank-level interference between tasks. Though these techniques enable real-time tasks to use dynamic memory, they provide only either cache or bank coloring, but not both. Our approach works for reduced interference across the tasks at both cache and memory level accesses.

Suzuki et al. [13] proposed coordinated bank and cache coloring, but they do not consider NUMA systems. In our work, along with integrated cache and bank coloring, we also demonstrate cache and controller-aware coloring. Their work is based on the Linux/RK kernel on an Intel platform, where the application reserves a part of physical memory for exclusive use. Our work is based on Linux on the AMD platform and we do not pre-allocate any memory. So even with dynamic allocation, we observe lower and more tightly bounded execution times. In their work, algorithm(s) decide the color that is going to be allocated to a task. In our work, we give the flexibility to the application to choose the color(s) with very minimal changes to application (one mmap() call). Our approach is simpler and easy to port across different versions of Linux kernels and hardware architectures. Controller-aware bank coloring [11] proposes bank and controller-aware allocation. Our work is an extension as it supports combined LLC, bank, and controller-aware allocation.

Chapter 8

Future scope

Our current approach reduces conflicts across tasks at the LLC and at the memory bank level. We can apply a similar approach to provide isolation to the Translation Look-aside Buffer (TLB) to support a conflict free allocation at the TLB-LLC-Memory level. Another scope of extension would be to compare our benefits of multi-level coloring across different architectures, Intel, ARM, etc. Our implementation is not tied to architectural features, i.e., we can easily provide portability across different architectures if we have the information required to identify TLB sets, cache sets and memory banks.

Chapter 9

Conclusion

This work contributes the design and implementation of LLC coloring and the integration of LLC, bank, and controller coloring for multicore real-time systems. The new allocator comprehensively considers LLC, memory bank, and controller to color while allocating memory to a task, without requiring hardware modifications. Developers have to include our header file and add a few initialization calls to an application to request specific LLC and bank colors for the entire program. Our kernel modifications are easily portable across different kernel versions and different architectures provided the information to identify LLC sets and memory banks. 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 our approach in a number of experiments on a multicore platform with microbenchmarks, NPB, Parsec, and Malardalen codes to investigate its performance and predictability impact. Synthetic benchmark results indicate that with the integrated LLC+bank coloring, a 26.43% reduction in task execution time (with standard deviation: 0.01) is achieved relative to shared same LLC+bank (worst case) (with standard deviation: 0.39). Parsec standard benchmarks indicate that a 24-89% reduction in execution time can be achieved relative to shared same LLC+bank (worst case). Hardware performance counters indicate that the LLC miss rate can be reduced by 39% with the LLC coloring and by 95.3% with the integrated LLC+bank coloring relative to shared same LLC+bank (worst case). Experimental results indicate that the LLC, memory bank, and memory controller-aware coloring reduces memory latency, avoids inter-task conflicts, and improves timing predictability of real-time tasks. With the above results, we show that our hypothesis holds by demonstrating that dynamic memory allocation can be controlled in software, such that real-time tasks will not interfere with one another in terms of Last Level Cache (LLC) and memory bank accesses in multicore systems.

REFERENCES

[1] AMD Bios and kernel developers guide (bkdg) for amd family 15h models 00h-0fh processors. http://developer.amd.com/wordpress/media/2012/10/31116.pdf, 2010.

[2] ARM Tegra3 Kayla Platform. http://www.nvidia.com, 2013.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[5] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. In *ACM SIGPLAN Notices*, volume 31, pages 244–255. ACM, 1996.

[6] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2010.

[7] Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. Cama: A predictable cache-aware memory allocator. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 23–32. IEEE, 2011.

[8] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 213–224. IEEE, 1997.

[9] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

[10] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.

[11] Xing Pan and Frank Mueller. Controller-Aware Memory Coloring for Multicore Real-Time Systems. *Real-Time Systems Symposium (RTSS), IEEE*, 2015.

[12] Isabelle Puaut and Damien Hardy. Predictable paging in real-time systems: A compiler approach. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 169–178. IEEE, 2007.

[13] Nobuhiro Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 685–692. IEEE, 2013.

[14] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: Lightweight performance tools. In *Competence in High Performance Computing 2010*, pages 165–175. Springer, 2012.

[15] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 157–167. IEEE, 2013.

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

and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th, pages 155–166. IEEE, 2014.

APPENDIX

Appendix A

Appendix-A

We investigate the performance and predictability for the PARSEC benchmark suite [4]. We have created a multi-task workload where several 'memory attackers' run in the background to assess their interference on memory latency for a foreground task similar to prior work [16], [11]. We call these background tasks the 'memory attackers', represented by one or more instance of the stream benchmark [16]. E.g., if there are four tasks in the experiment, one of them (the foreground task) is a Parsec benchmark and the others (background) are memory attackers. Fig. A.1 shows an average execution time of parsec task over ten runs along with error bars. We have six experimental models: 1) shared LLC+bank, where all the three background tasks and foreground Parsec task share LLC(s) and bank(s); 2) LLC coloring, where all the three background tasks and foreground Parsec task share bank(s) but access different LLC(s); 3) Bank coloring, where all the three background tasks and foreground Parsec task share LLC(s) but access different bank(s); 4) LLC+bank coloring, where all the three background tasks and foreground Parsec task access different bank(s) and LLC(s); 5) Parsec without attackers, where we run standalone Parsec program with buddy allocation and without any attackers in the background; and 6) No color(buddy), where all the three background tasks and foreground Parsec task use the general buddy allocation. We observe that for the most of Parsec applications the buddy allocator (No color(buddy)) gives fairly good performance for an average of ten runs. But we can not guarantee that it would be always predictable, and in the worst case the performance would be equal to shared LLC+bank. We also observe with the buddy allocation, variance and standard deviation in execution time is higher than our colored allocation.

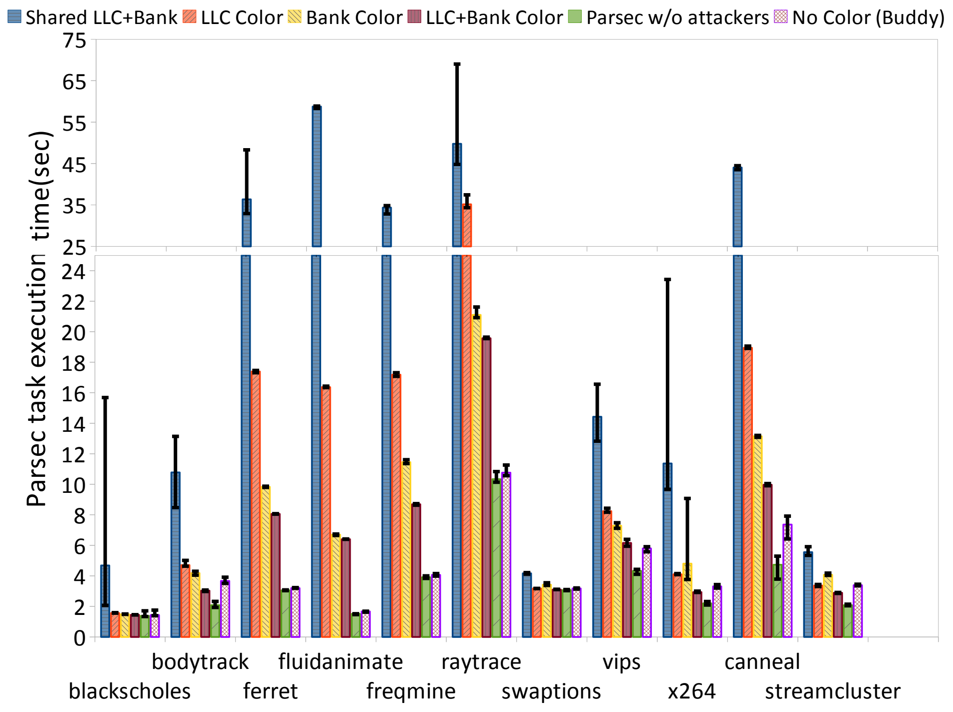

Figure A.1: Parsec with memory attackers (including buddy allocator with attackers)