PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms

Heechul Yun[†], Renato Mancuso[‡], Zheng-Pei Wu[⋆], Rodolfo Pellizzoni[⋆]
[†] University of Kansas, USA. heechul.yun@ku.edu
[‡] University of Illinois at Urbana-Champaign, USA. rmancus2@illinois.edu
[⋆] University of Waterloo, Canada. {zpwu,rpellizz}@uwaterloo.ca

***Abstract*—DRAM consists of multiple resources called banks that can be accessed in parallel and independently maintain state information. In Commercial Off-The-Shelf (COTS) multicore platforms, banks are typically shared among all cores, even though programs running on the cores do not share memory space. In this situation, memory performance is highly unpredictable due to contention in the shared banks.**

In this paper, we propose PALLOC, a DRAM bank-aware memory allocator which exploits the page-based virtual memory system to allocate memory pages of each application to specific banks. With PALLOC, we can dynamically partition banks to avoid bank sharing among cores, thereby improving isolation on COTS multicore platforms without requiring any special hardware support.

We performed an extensive set of experiments to investigate the performance impact of DRAM bank partitioning on two COTS multicore platforms with a set of synthetic and SPEC2006 benchmarks. Our evaluation results demonstrate that DRAM bank partitioning significantly improves isolation and real-time performance.

I. Introduction

In multicore platforms, DRAM is a crucial shared resource. As applications become more memory intensive [1] and processors include more cores, the performance of DRAM becomes more critical for overall system performance [28].

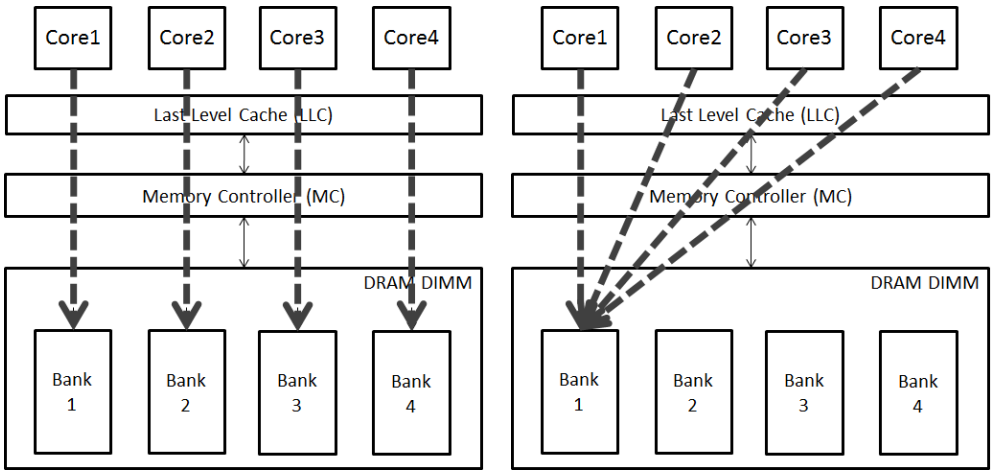
DRAM consists of multiple resources called banks that can be accessed in parallel. Therefore, memory performance in multicore platforms can vary significantly depending on how data are located in the banks and how the banks are shared among the cores at a given time. Figure 1 shows the best and the worst-case memory access scenarios in multicore processors: when all cores are accessing data located in different memory banks (best-case), requests can be processed in parallel. On the other hand, when all cores are accessing data located in the same memory bank (worst-case) at the same time, requests would be delayed due to contention in the bank.

Unfortunately, today's operating systems view DRAM as a single resource and do not consider banks when allocating memory. Therefore, the exact locations of the allocated memory over the banks are unpredictable. Moreover, memory controllers are typically configured to interleave the banks in order to improve bank-level parallelism [9]. This further exacerbates the problem because multiple programs running on different cores at a given time are likely to share banks, even though they do not share memory space.

There have been several proposals to make DRAM more predictable. Predator [2] and AMC [22] DRAM controllers treat multiple banks as a single unit of access, effectively transforming multiple resources into a single one. This makes it possible to apply traditional single resource scheduling algorithms, such as fixed priority or TDMA, to guarantee memory performance. Other proposals employ private banking schemes so that cores can exclusively access their private DRAM banks by hardware design [27], [24]. All these proposals, however, require special hardware modifications and are not directly applicable on today's Commercial Off-The-Shelf (COTS) systems.

In this paper, we present PALLOC, a DRAM bank-aware memory allocator that can allocate memory to specific DRAM banks by leveraging the page-based virtual memory system of modern operating systems. Using PALLOC, a system designer can partition DRAM banks in a flexible manner to improve the quality of performance isolation of a multicore platform. For example, the designer can create a virtual scheduling partition [4] for each core and assign private DRAM banks for each partition. Such private banking scheme effectively eliminates bank sharing among cores without requiring any hardware modification. Partitioning DRAM banks is not free in the sense that processes in a partition cannot use more memory than the size of the allocated DRAM banks, even though the rest of the DRAM banks are not used. However, we argue that the cost of adding more memory can be justifiable for critical real-time systems. Furthermore, because PALLOC can dynamically change bank assignments at runtime, careful bank assignment schemes can greatly alleviate the limited space problem.

Using PALLOC, we performed an extensive experimental study that investigates the performance impact of private banking in two COTS multicore platforms with a set of synthetic and SPEC2006 benchmarks. Our findings are as follows: First, the private banking strategy reduces performance variations (up to 4.4X), and offers better real-time performance on COTS multicore platforms. Second, using a smaller number of memory banks can hurt single thread performance (up to -35%) due to the reduced memory level parallelism (MLP), but the degree of performance degradation is not significant for most SPEC2006 benchmarks. Third, we also investigated the performance impact of cache partitioning in conjunction with DRAM bank partitioning, and found that



(a) Best-case (b) Worst-case

Fig. 1: Best and worst-case memory access patterns on multicore

cache partitioning helps reduce performance variations but at the cost of considerable single thread performance reduction on SPEC2006 benchmarks (-18% on average). Lastly, although space partitioning of DRAM banks (and caches) significantly improves isolation, there are still other shared resources, including the memory bus, that need to be addressed for better performance isolation.

Our contributions are: (1) the design and implementation of a DRAM bank-level partitioning algorithm in recent Linux kernels; (2) the DRAM controller address mapping detection methodology; (3) the detailed experimental performance analysis on two real COTS multicore platforms.

The remaining sections are organized as follows: Section II provides background on DRAM operation and discusses the problem of DRAM bank sharing in multicore platforms. Section III presents the design and implementation of PALLOC. Section IV describes the evaluation platforms and the implementation overhead analysis. Section V presents the evaluation results with a set of synthetic benchmarks, while Section VI details the results with SPEC2006 benchmarks. Section VII discusses related work. Finally, we conclude in Section VIII.

II. Background and Problems

Modern DRAM memory systems are composed of a memory controller and memory devices. The controller handles requests from CPUs or DMAs and memory devices store the actual data. The device and controller are connected by a command bus and a data bus. The controller typically has a front-end and a back-end. The front-end receives requests, keeps track of the status of the device, and generates a set of memory commands required to handle each request. The back end handles command arbitration, ensure that all timing constraints are satisfied, and issues commands to the device. Modern memory devices are organized into *ranks* and each rank is divided into multiple *banks*, which can be accessed in parallel provided that no collisions occur on either buses. Each bank comprises a *row-buffer* and an array of storage cells organized as *rows*[1] and *columns*. Furthermore, some systems have multiple memory *channels*, each of which can be operated independently. Typically, multiple channels are configured to interleave at the cache-line granularity to improve average throughput.

In order to access the data stored in a DRAM row, an *Activate* (*ACT*) command must be issued to load the data into the row buffer first before it can be read or written. Once the data is in the row buffer, a *CAS* command (read or write) can be issued to retrieve or store the actual data. If a second request wishes to access a different row from the same bank, the row buffer must be written back to the array with a *Pre-charge* (*PRE*) command first before the second row can be activated. Finally, since DRAM storage contains capacitors, the device must be periodically restored to full voltage level, a periodic *Refresh* (*REF*) command must be issued to all ranks and banks. The result of REF is that all row buffers are written back to the data array (i.e., all row buffers are empty after refresh).

Due to hardware limitations, the memory device takes time to perform different operations and therefore timing constraints between various commands must be satisfied by the controller. The operation and timing constraints of memory devices are defined by the JEDEC standard [13]. The key results of these timing constraints are: 1) the latency for accessing a closed row is much longer than accessing a row that is already open. 2) Different banks can be operated in parallel since there are no long timing constraints between banks. In particular, the nominal device throughput can only be achieved by simultaneously reading or writing in multiple banks at once.

Due to these reasons, most mid-high profile COTS DRAM controllers employ an *interleaved bank* strategy with some form of per-bank queueing. Under this scheme, consecutive memory blocks in physical address space, typically of the size of a memory page, are allocated to different banks. Once

[1]DRAM *rows* are also referred to as *'pages'* in the literature; we use the term rows to avoid confusion with a virtual memory page.

a request of a core reaches the front-end of the memory controller, its physical address is used to identify the targeted bank and then the corresponding commands are entered in a queue for that bank. Since modern cores can issue a large number of outstanding memory requests, with this mechanism even a single application can simultaneously open and access multiple rows in different banks, thus increasing its average memory throughput.

Unfortunately, the discussed strategy has two major shortcomings when applied to real-time systems. First of all, there is no guarantee that an application will indeed be able to access multiple banks: if the OS is unaware of bank interleaving, in the worst case it could allocate all memory pages for that application to the same bank, resulting in much increased memory latency compared to the average case. This dependency on runtime decisions by the memory allocator can be a significant potential source of unpredictability.

Second, even if the application memory is spread over multiple banks, throughput can be significantly degraded in a multicore system due to the effects of bank sharing. Since banks are interleaved, any core in the system can access any bank. If two applications running in parallel on different cores access two different rows in the same bank, they can force the memory controller to continuously pre-charge the row buffer and open a new row every time an access is performed. This loss of row locality can result in a much degraded row hit ratio and thus a corresponding latency increases for both applications. Furthermore, it creates a dependency between applications running on different cores, which can greatly complicate timing analysis. Finally, notice that in the worst-case both issues can be simultaneously manifested, resulting in the worst-case scenario of Figure 1, where all cores access the same bank at the same time.

To avoid the bank sharing problem while still allowing for bank-level parallelism in a multicore system, existing work in the area of predictable real-time controllers has proposed to employ a private bank scheme, where each core is assigned either one or a set of exclusive banks [24], [27]. Since under these schemes banks are not shared, cores cannot interfere with each other by closing rows opened by another core. As a matter of fact, interference is mostly limited to data bus contention, resulting in better worst case latency bounds [27]. Unfortunately, the discussed solutions suffer from two main limitations: first, COTS memory controllers do not typically support private bank allocation. Hence, partitioning banks in hardware would require modifications to existing memory controllers, which is undesirable in COTS-based systems. Second, hardware bank partitioning can be highly inflexible, since banks must be statically allocated to cores. In the next section, we will describe our software-based solution, PALLOC, which solves the problem of bank sharing without having aforementioned problems of hardware based solutions.

III. PALLOC: DRAM Bank-Aware Kernel Memory Allocator

PALLOC is a kernel-level memory allocator that exploits page-based virtual-to-physical memory translation to selectively allocate memory pages of each application to the desired DRAM banks. The goal of PALLOC is to control applications' memory locations in a way to minimize memory performance unpredictability in multicore systems. As discussed in the previous section, such unpredictability can be minimized by eliminating bank sharing among parallel executing applications. Unlike hardware based approaches [27], [24], however, PALLOC is a software based solution, which is fully compatible with existing COTS hardware platforms and transparent to applications (i.e., no need to modify application code.)

```
/* return a free page frame from the selected banks */
struct page *palloc_find_page(bankmap)
{
    for (bank <- bankmap) {
        if (!empty(bank_bins[bank]) {
            page = pop(bank_bins[bank])
            return page;
        }
    }
    return NULL;
}

/* return a free page frame (4KB) */
struct page *__rmqueue_smallest(...)
{
    freelist <- free pages
    bankmap <- selected banks

    /* search page from bank cache */
    page = palloc_find_page(bankmap);
    if (page)
        return page;

    /* build bank cache & search page */
    for(page <- freelist) {
        bank = addr_to_bank(page);
        push(bank_bins[bank], page);
        page = palloc_find_page(bankmap);
        if (page)
            return page;
    }
    return NULL;
}
```

Fig. 2: PALLOC implementation

Figure 2 shows simplified pseudocode of the proposed allocator. For simplicity, here we assume that the kernel maintains a single free page list, `freelist`, although the actual implementation deals with multiple freelists with different sizes (more details later in this section). Here, `__rmqueue_smallest()` is the main allocator function called by the kernel. For user-level applications, this is called when a page fault occurs. Instead of simply returning the head of the freelist, PALLOC maintains a set of lists—`bank_bins`, one per DRAM bank—to quickly find a page from the selected banks, `bankmap`. If a page is found in one of the bins for the banks, it returns the page and removes the entry from the bin. If such page is not found, however, PALLOC checks every page in the freelist iteratively, until it finds a matching page. In the process, unmatched pages are removed from the freelist and inserted into the corresponding bins, to decrease the overhead of future allocations. While this

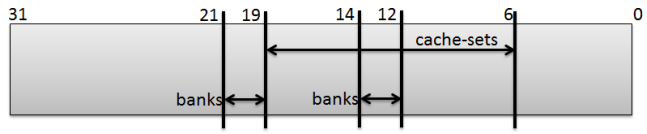
greatly reduces the average overhead, PALLOC may still need to traverse the entire freelist in the worst-case. As such, we do not intend to target "real-time" memory allocation. Rather, we are interested in application's real-time performance *after* memory allocation is completed. Nevertheless, we provide detailed overhead analysis in Section IV-C that shows reasonable allocation time performance of our implementation. An important requirement for PALLOC is the ability to determine the bank address for a given physical address, abstracted by `addr_to_bank()` in the pseudocode. We describe ways to identify this mapping information in Section IV-B.

We implemented PALLOC in Linux. The standard kernel memory allocator used in Linux is based on the buddy algorithm [16]. The buddy algorithm aggregates blocks (buddies) of contiguous pages of exponential size (order) [2] and arranges them in a set of free-lists (a list for each order). On a memory request, the allocator returns the head of a free-list with a matching size. We extend the buddy allocator to implement the algorithm in Figure 2. PALLOC only handles the order 0 (4KiB) page allocation, while the original buddy allocator handles the rest (order 1 or above). Because user-level memory allocations are eventually performed at the page fault handler with the page granularity (4KiB) [3], PALLOC can properly control bank assignments for user-level applications. The most common kernel internal allocation requests (getting a page frame) are also handled by PALLOC. Note that freeing pages is handled by the original buddy allocator so that the kernel maintains the buddy structure. This also allows the kernel to aggregate the freed pages to make a bigger page. In this way, the kernel can keep both bank-aware order 0 pages and bigger pages at the same time.

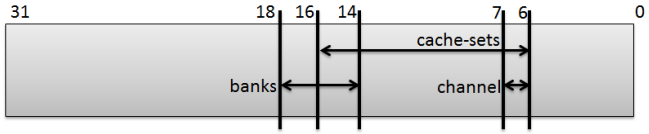
Finally, banks are configured through the CGROUP interface in Linux. Once a CGROUP partition is created, it provides a file interface to describe desired DRAM banks for the partition. The setting can be modified at runtime and the modified setting is immediately applied to all subsequent memory allocations in the partition. If the assigned banks are used up, in our current implementation, the standard kernel action—trying to reclaim from the page cache and swapping to the disk—will be taken even if other banks are free. For better safety, we can use the standard CGROUP memsize limit controller so that the OOM killer can kill off tasks in the CGROUP partition.

One potential issue in using PALLOC is shared memory regions, such as shared program text (disk cache pages). When a process executes in a CGROUP partition, the pages that are already allocated elsewhere, for example, the *bash* program text pages, will be shared instead of reallocating the pages. Therefore, these pages may not conform to the DRAM bank setting of the CGROUP. In such case, a page-migration scheme to migrate existing pages to different banks can be considered, similar to NUMA page migration [17].

[2]An order N block refers to a block of 2^N contiguous memory pages.
[3]We currently do not consider huge page allocation via mmap interface.



(a) Intel Xeon 3530 + 4 GiB DDR3 DIMM (16 banks)



(b) Freescale P4080 + 2x2 GiB DDR3 DIMM (32 banks)

Fig. 3: Memory address mappings

IV. Evaluation Setup

In this section, we present details on the hardware and software platform used in our evaluation. In particular, we discuss how to derive the bank mapping employed by the memory controller, and we provide overhead analysis for our Linux-based PALLOC implementation.

A. Hardware Platform

The primary platform used in this paper is a quad-core Intel Xeon W3530 based desktop computer. The processor has private 32K-I/32K-D (4/8 way) L1 cache, a private 256 KiB (8 way) L2 cache for each core and a shared 8MiB (16 way) L3 cache. The memory controller (MC) is integrated in the processor and supports 1333 MHz DDR3 memory with the maximum theoretical transfer rate of 10.6 GB/s. The computer has a single-channel dual-rank 4 GiB PC10666 DDR3 DIMM module, which has 16 DRAM banks (8 banks per rank). We disabled hardware prefetchers and the turbo-boost feature to improve predictability.

We also conducted some experiments on a Freescale P4080 platform, which features an eight-core PowerPC processor. Each core has a private 32K-I/32K-D (8/8 way) L1 cache, a private 256 KiB (8 way) L2 cache and shares two 1 MiB (32 way) L3 caches (cache-line interleaved). The MC supports 1333 MHz DDR3 and the platform has two 2 GiB DDR3 DIMM modules in a dual-channel configuration, each of which has 16 DRAM banks (for a total of 32 banks).

B. DRAM Controller Address Mapping

A prerequisite of PALLOC is the exact knowledge of the address mapping of the DRAM controller. The physical address of a memory request is translated by the memory controller and mapped onto the physical structures of the DRAM module: columns, rows, banks, and ranks. There are many possible mapping schemes for a given hardware platform. For example, one configuration could assign two most-significant bits of a physical address to the "rank" address of the DRAM module, while one could assign them to "bank" address instead.

Unfortunately, the exact address mapping information is typically not publicly available, which is indeed the case

TABLE I: Page allocation overhead

size (MiB)	average (ns) palloc	buddy	max (ns) palloc	buddy	alloc pages
1	219	21	11,764	137	256
2	198	22	12,067	131	512
4	170	26	12,234	192	1,024
8	161	26	24,136	343	2,048
16	276	27	283,443	423	4,096
32	252	26	292,399	344	8,192
64	232	29	306,041	800	16,384
128	232	31	279,477	614	32,768
256	226	32	352,113	1,557	65,536
512	225	37	572,935	1,024	131,072

of our Intel Xeon platform. Therefore, we experimentally determined the mapping information of the Xeon platform. We use a specifically written micro-benchmark that traverses a linked list over the physical address space (by mapping the `/dev/mem` in Linux). The list has 32 entries and it is engineered in such a way that the distance between two successive entries is 8 MiB. The distance is large enough to span through all the memory banks in the system, so that each entry in the list will be placed in exactly the same bank, under the assumption that banks are interleaved [6]. Note also that each entry is located at the exact same cache-set. Therefore, iterating the 32 entries exhausts the 16 cache-ways, forcing to access DRAM all the time. While a first instance of the benchmark is running on a core, we execute a second instance on a different core. Note that, in the latter instance, the addresses of all the entries in the list are shifted by b bits (i.e., offset = $1 \ll b$). If the shifted addresses are mapped to the same memory banks as the first instance, then the measured memory performance will drop due to bank conflicts. Conversely, if the addresses are mapped to a different bank, the measured memory performance will be higher since no bank conflict will be experienced. By varying b, we identify the address bits that are associated with memory banks.

On the other hand, the P4080 platform provides detailed documentation about the exact physical address-to-DRAM mapping information.

Figure 3 shows the identified memory mappings of both platforms. The *banks*, *channel*, and *cache-sets* show address bits for DRAM banks, MC channels, and L3 cache index, respectively.

C. Software Implementation Overhead Analysis

We implemented PALLOC on the standard Linux 3.6.0 kernel for the Intel Xeon platform. Kernel modifications to implement PALLOC are small and easily portable: mostly by replacing `__rmqueue_smallest()` in `mm/page_alloc.c`. We also ported it to the Freescale's custom Linux 3.0.6 kernel for the P4080 platform.

Due to the design of PALLOC, the page allocation time can vary widely depending on the availability of free pages for the selected DRAM banks. To measure the allocation overhead, we instrumented the allocator to collect time spent on the allocator, while running a synthetic program that allocates and accesses a specified amount of memory.

Table I shows the average and the worst-case per-page allocation time of PALLOC and the unmodified buddy allocator. For PALLOC, we run the benchmark on a CGROUP partition with four DRAM banks assigned to it. Therefore, the maximum allocatable memory space is 1024 MiB (256 MiB per DRAM bank). On average, PALLOC adds less than 200ns overhead for each page allocation, compared to the buddy allocator. The worst-case allocation time is, however, noticeably higher than the buddy allocator, especially as the total allocation size grows. This is because, as the number of allocated pages increases, the number of remaining pages that fall in the selected banks becomes increasingly low, even though there are free pages in other DRAM banks. Note that the experiment is unfavorable for PALLOC because all its per-bank free lists (bins) are initially empty. In real multi-programmed environment, however, the lists can be cached by memory requests generated from other processes in different partitions, thereby increasing PALLOC's overall efficiency. Furthermore, it is important to note that the buddy allocator can also suffer significantly longer allocation delays when the system-wide free pages are low, forcing the kernel to reclaim pages from the kernel page cache and, eventually, to swap pages to disks. For this reason, it is a standard practice to allocate memory pages at the process initialization phase and outside the critical real-time regions in order to make sure that the allocation procedures are never invoked while processing time-sensitive critical sections.

We also investigated the allocation overhead using the SPEC2006 benchmarks and found that the total allocation overhead is lower than 0.4% of the total execution time in the most memory allocation intensive benchmark. Therefore, we consider that the overhead of PALLOC is acceptable.

V. Results with Synthetic Benchmarks

In this section, we investigate the performance impact of using private DRAM banks, through PALLOC, with a set of synthetic benchmarks. We also investigate the architectural differences in DRAM controllers of the two hardware platforms we used in this paper.

A. Samebank vs. Diffbank

In this experiment, we use a synthetic benchmark *Latency* [29] to illustrate the performance impact of using private DRAM banks. The benchmark is a pointer-chasing application using a randomly shuffled linked-list. The two successive memory instructions that access the elements of the linked list contain read after write (RAW) dependency. Hence, the second instruction cannot proceed and must stall until the first one is completed. In other words, the benchmark can generate only one outstanding memory request at a time. The size of the linked-list is configured to be two times bigger than the size of the LLC (last level cache), in order to make sure all memory requests result in cache-misses. Since the list is randomly shuffled over a big memory area, successive memory requests are likely to target a different DRAM row. Therefore, Latency is used to measure worst-case memory performance in terms of latency and bandwidth. By running multiple instances of the

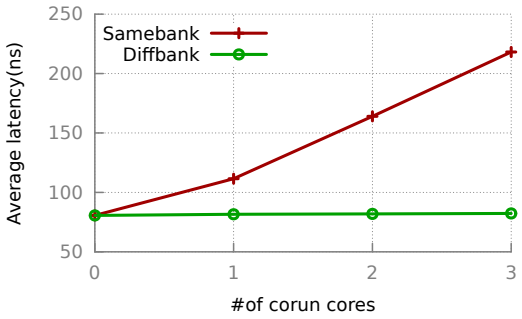


Fig. 4: Samebank vs. Diffbank on Intel Xeon.

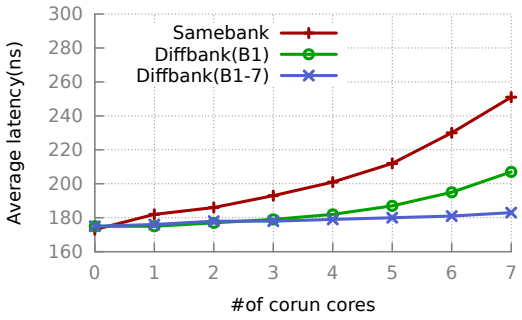


Fig. 5: Samebank vs. Diffbank(s) on Freescale P4080.

Latency benchmark, we now show the performance impact of using private DRAM banks.

The experiment setup is to run a Latency instance on Core0 while varying the number of co-running instances from 0 to 3 (one instance per core). We repeat the experiment in two memory settings. In *Samebank*, all the memory pages of all Latency instances are allocated in Bank0. In *Diffbank*, Core0 uses Bank0 and the other cores use Bank1.

Figure 4 shows the average memory access latency of the Latency benchmark running on Core0. The X-axis shows the number of co-running cores, each of which executes a separate instance of the Latency benchmark. Note that the average access latency is increased significantly in Samebank. This is because all memory requests are serialized in Bank0, i.e. they all result in *bank conflicts*. In Diffbank, however, the memory requests from Core0 are serviced in parallel with memory requests generated from other cores, because they are accessing a different memory bank. As a result, the instance of the Latency benchmark running at Core0 does not suffer any noticeable performance reduction. This demonstrates the potential of private banking in providing performance isolation.

B. Understanding DRAM Controller Architecture

Having the capability to control memory bank placement using PALLOC allows us to investigate DRAM controller's characteristics. In particular, we are interested in the queueing structure of the DRAM controller, as it has profound impact on DRAM performance isolation. According to [10], the most common queueing structures are 1) a shared global queue and 2) per-bank queues, but this information is rarely publicly available on COTS components.

In this experiment, we reverse engineer such information using PALLOC. The experiment setup is as follows. Each core executes one *Latency* instance. Core0 always accesses Bank0. For co-running cores, there are three cases: (1) In *Samebank*, they access Bank0 (the same as Core0); (2) In *Diffbank(B1-7)*, they access Bank1-7; (3) In *Diffbank(B1)*, they access Bank1.

Figure 5 shows Core0's average latency as a function of the number of other cores on the P4080 platform. Similar to Figure 4, Samebank shows the worst performance. Interestingly, however, Diffbank(B1-7) shows better performance than Diffbank(B1), even though Core0 always accesses its own Bank0 in both cases. From the result, we can infer that P4080's MC is unlikely to feature per-bank queues, because if each bank had its own queue, then Diffbank(B1) would show the same performance as Diffbank(B1-7). Instead, it is likely to have a single global queue. Then, the better performance of Diffbank(B1-7) can be explained as it can process multiple requests in parallel (faster) using more banks.

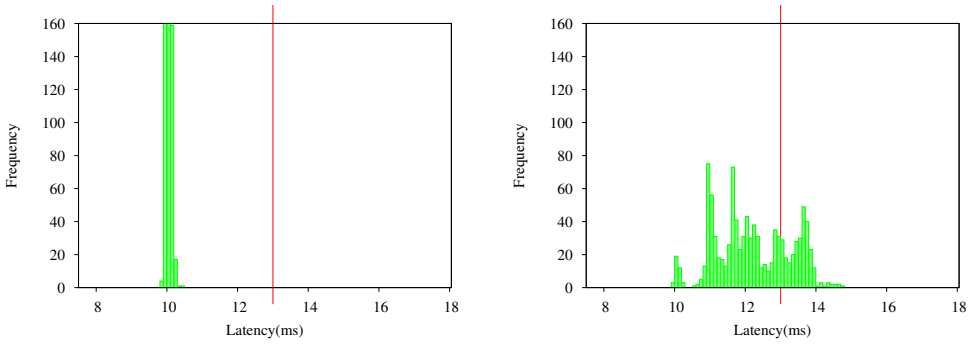
We repeat the same experiment on the Intel Xeon platform. Unlike the P4080 platform, we found that the results of Diffbank(B1) and Diffbank(B1-7) are almost identical in the Xeon platform. Therefore, the figure for this experiment looks similar to Figure 4. This result suggests the Xeon's MC has per-bank queue and/or support request re-ordering mechanism that favors open banks over closed banks.

In summary, the two platforms have different queueing structures based on our experiments. For P4080, even if we partition DRAM banks, the shared queue is still being contended among cores, which may result in poor performance isolation. On the other hand, Xeon can benefit better from DRAM bank partitioning as contention at the memory controller queue would be eliminated.

C. Real-Time Performance Impact

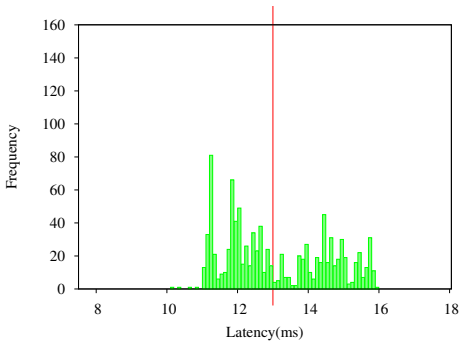
In this experiment, we explore the real-time performance impact of using private DRAM banking in a realistic scenario, modeling a simple real-time data acquisition system. The system is composed of two tasks: a periodic real-time task and a non-real-time task. The real-time task periodically reads a memory region and performs basic processing. The computation must be completed within a given deadline to avoid data corruption, since the hardware can overwrite the buffer with incoming data. The non-real-time task is responsible for analyzing and post-processing the data acquired by the real-time task.

We simulate the scenario on the Intel Xeon platform as follows. First, we create a synthetic real-time task *HRT* that periodically reads a chunk of main memory. Jobs are released with a 20 ms period (50Hz) and have to be finished within a deadline of 13 ms. We schedule the task with the SCHED_FIFO real-time scheduling policy. The HRT task is engineered to be insensitive to the shared L3 cache, so that all its memory requests result in cache-misses. Second, for the non-real-time data processing task, we use the standard

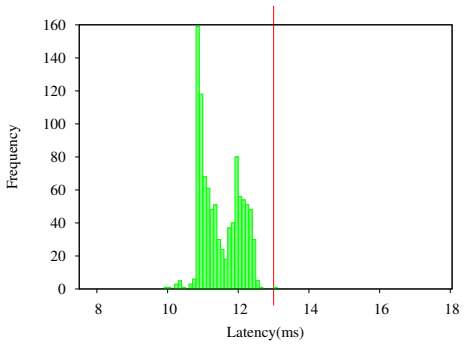


(a) Buddy(solo)

(b) Buddy



(c) Samebank



(d) Diffbank

Fig. 6: HRT runtime distribution on Intel Xeon. Core0-HRT, Core1-Xserver(none for (a))

TABLE II: HRT task runtime statistics on Intel Xeon.

Conf.	average (ms)	99 pct. (ms)	deadline miss(%)	stdev.
Buddy (solo)	10.0	10.2	0	0.08
Buddy	12.2	14.3	27.9	1.05
Samebank	13.1	15.8	44.5	1.47
Diffbank	11.5	12.5	0.1	0.56

TABLE III: HRT task runtime statistics on Freescale P4080.

Conf.	average (ms)	99 pct. (ms)	stdev.
Buddy (solo)	20.1	20.2	0.03
Buddy	48.7	49.6	0.28
Samebank	75.1	75.6	0.21
Diffbank	28.9	29.8	0.09

X-window server, which generates memory traffic whenever it updates the screen. To make the X-server update the screen, we keep printing text strings (the standard output of the HRT program) on a gnome-terminal. The X-server runs with the SCHED_OTHER scheduling policy.

The two desired goals of the system are (1) to meet the deadline of the HRT task and (2) to provide a high frame-rate for the X-server. The former goal is more important than the latter. We use the deadline miss ratio and the execution time distribution to gauge the first goal. We use the CPU utilization of the X-server to measure the latter goal.

We repeat the experiment on four different settings. In both *Buddy(solo)* and *Buddy*, we use the standard buddy allocator, hence utilizing all 16 DRAM banks. In *Samebank*, both HRT and the X-server are assigned to the same eight banks - Bank0-7. In *Diffbank*, HRT uses Bank0-7 while the X-server uses Bank8-15. In Buddy(solo), we only run HRT on Core0, while in the other settings, we also run the X-server on Core1.

Figure 6 shows the runtime distribution of HRT. Each figure is plotted using 1000 samples collected over 20 seconds. The X-axis shows the observed execution time (ms) while the Y-axis shows the number of samples observed in the time range. Table II also shows relevant statistics for the same experiment.

In Buddy(solo), the runtime distribution of HRT is very stable at around 10 ms and the 99 percentile runtime is 10.2 ms. When the X-server is co-scheduled in Buddy, however, the average and the 99 percentile runtimes are increased to 12.2 ms (up by 22%) and 14.3 ms (up by 40%) respectively, resulting in 28% deadline violations. In Samebank, the average and the 99 percentile runtimes are further increased to 13.1 ms and 15.8 ms, respectively, resulting in 45% deadline misses. In Diffbank, however, the average and the 99 percentile runtimes are decreased to 11.5 ms and 12.4 ms, respectively, and only one deadline violation is observed. This result shows that isolating DRAM banks for critical tasks can significantly improve the real-time performance of the system.

a) Result on P4080: We also performed a similar experiment on the P4080 platform. In this experiment, however, instead of running the X-server (as we do not have a working X-server on the P4080 platform), we use *Bandwidth* benchmark [29], which simply accesses a big array without RAW dependency (hence generating large memory traffic). Similar

benchmark	bandwidth (MB/s)	RSS (MiB)	average IPC	memory intensity
470.lbm	3158	409	0.88	
462.libquantum	3124	64	0.83	
437.leslie3d	2346	123	0.76	
433.milc	2313	523	0.80	High
482.sphinx3	1649	40	1.11	
450.soplex	1211	108	0.70	
434.zeusmp	1122	502	1.13	
483.xalancbmk	798	110	0.63	
436.cactusADM	702	623	0.93	
403.gcc	618	196	1.02	
473.astar	378	325	0.66	Medium
471.omnetpp	203	173	1.09	
447.dealII	136	6	1.61	
481.wrf	131	570	1.89	
400.perlbench	124	147	1.50	

TABLE IV: SPEC2006 characteristics on Intel Xeon

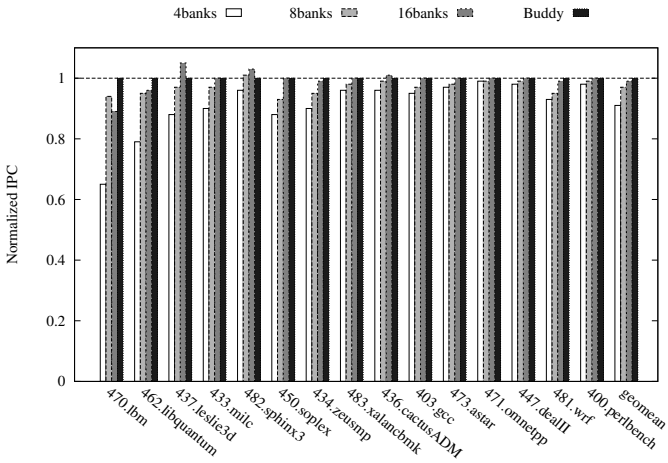


Fig. 7: Performance impact of private DRAM banks on unicore

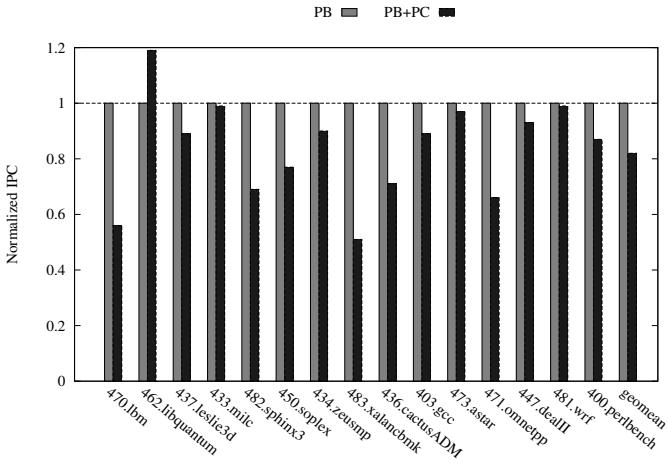


Fig. 8: Performance impact of private DRAM banks (PB) and private cache (PC) on unicore.

to the previous experiment on Intel Xeon, in *Samebank*, all benchmarks are assigned to use the same Bank0. In *Diffbank*, the seven instances of Bandwidth are assigned to all DRAM banks except Bank0. Table III shows that the Diffbank configuration offers superior real-time performance.

VI. Results with SPEC2006

In this section, we investigate the performance impact of partitioning DRAM banks and caches with the SPEC2006 benchmark suite on the Intel Xeon platform. We investigate both solo performance (single core) and co-run performance (four cores) in the presence of heavy memory contention.

Table IV shows the characteristics of SPEC2006 benchmarks used in this paper. RSS and IPC denote Resident Set Size and Instruction-Per-Cycles respectively. The benchmarks are sorted in descending order of the average memory bandwidth usage (MB/s). We excluded benchmarks that are either CPU intensive (i.e., bandwidth $<$ 100 MB/s) or allocate too much memory space (i.e., RSS $>$ 750MiB) for the purpose of our evaluation.

A. DRAM Bank Partitioning

In this experiment, we investigate the performance impact of DRAM bank partitioning to the single thread performance.

We first introduce a bitmap notation to denote DRAM banks mapped to the partition. As shown in 3(a), the Intel Xeon platform has 16 DRAM banks that are addressed by four bits in the physical address: bits 20, 19, 13, and 12. We denote the selected banks by concatenating these four bits in order. For example, `[000X]` specifies two DRAM banks in which the address bit 20, 19, and 13 are all zero, and the bit 12 is either 0 or 1. Likewise, `[0XXX]` indicates eight DRAM banks in which the address bit 20 is zero and all remaining three bits (bit 19, 13, 12) are either 0 or 1.

The experiment setup is as follows. We create a CGROUP partition and assign a subset of DRAM banks to it. We then run each SPEC2006 benchmark in the partition for 10 seconds and measure the average IPC. We repeat the experiment in four different bank assignments. In *Buddy*, we do not use PALLOC. Hence, all 16 memory banks are utilized by the standard buddy allocator; In *4banks*/*8banks*/*16banks*, banks are selected by `[00XX]`/`[0XXX]`/`[XXXX]` , respectively.

Figure 7 shows the normalized IPC of SPEC2006 for each bank assignment scheme. Overall, the number of banks is generally positively correlated with the performance, although not always. This is because the probability of achieving more memory level parallelism (MLP) is higher when more banks are used to allocate the same amount of memory. Because modern out-of-order processors can generate multiple outstanding memory requests at a time, a single threaded program can benefit from the parallelism. For example, 470.lbm's performance improves by 29% by increasing banks from four to eight. For most benchmarks, however, the performance impact of partitioning DRAM banks is modest: the difference between 4banks and Buddy is 9% on average.

B. Cache Partitioning

In this experiment, we investigate the performance impact of cache partitioning, coupled with DRAM bank partitions, to

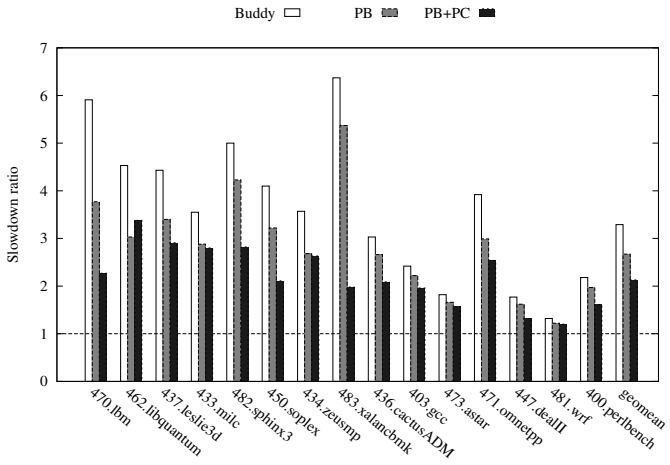


Fig. 9: Slowdown ratios of SPEC2006. Core0=X-axis, Core1-3=470.lbm×3.

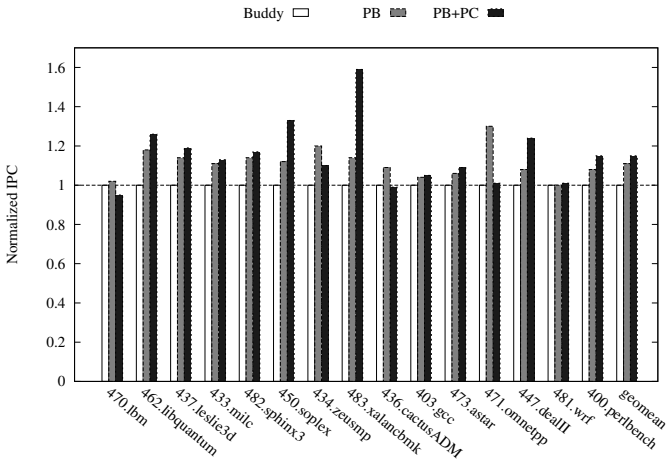


Fig. 10: Normalized IPC of SPEC2006. Core0: X-axis, Core1-3: 470.lbm×3.

the single thread performance.

In most processors, L2 and L3 are indexed by the physical address. For example, our Xeon processor has 16-way 8 MiB (512 KB/way) shared L3 cache in which the cache-set is determined by using address bits [18:6] of the physical address (L2 cache uses [14:6]). Note that two bits are overlapped with the lower part of DRAM bank bits [13:12] (see Figure 3(a)). This means that selecting bits [13:12] to partition DRAM banks has a side effect of partitioning cache-sets. To investigate the performance impact of such cache-partitioning, we repeat the experiment described in the previous subsection, but in two different bank (and cache) assignments. In *PB*, we use four DRAM banks, selected by `[00XX]`. In this setting, each task can use the entire cache; In *PB+PC*, however, DRAM banks are selected by `[XX00]` that addresses only $1/4$ of the L2&L3.

Figure 8 shows the normalized performance of the SPEC2006 benchmarks under PB and PB+PC (normalized to PB). As expected, partitioning cache PB+PC generally negatively affects to the performance, compared to partitioning DRAM banks only (PB). One exception is 462.libquantum, which shows better performance under PB+PC. One reason of this behavior is that the benchmark is completely cache insensitive unless the cache size is bigger than its working set size [11]. Since the 8 MiB L3 cache on our Intel Xeon is smaller than its working set size, cache partitioning does not negatively impact its performance. Overall, the single thread performance of SPEC2006 benchmarks is 18% better, on average, in PB (full cache) than in PB+PC ($1/4$ cache).

C. Performance Isolation and Throughput

In this experiment, we investigate the degree of performance isolation achieved by partitioning DRAM banks (and cache) in the presence of memory intensive co-runners.

The basic setup is as follows: We first run the subject SPEC benchmark on Core0 for 10 seconds and measure the average IPC—i.e., *Solo IPC*. Next, we launch three 470.lbm instances on the other cores (Core 1, 2 and 3) to generate memory traffic, then repeat the experiment to measure the average IPC of SPEC2006 on Core0—i.e.,*Corun IPC*. We repeat the whole experiments in three different memory allocation schemes. In *Buddy*, we use the standard Linux Buddy allocator; in *PB*, DRAM banks are equally partitioned to each core so that it can exclusive access 4 banks (out of 16 total banks), but the caches are not partitioned— Core0=`[00XX]`, Core1=`[01XX]`, Core2=`[10XX]`, and Core3=`[11XX]`; In *PB+PC*, both DRAM banks and the caches are partitioned to each core—Core0=`[XX00]`, Core1=`[XX01]`, Core2=`[XX10]`, and Core3=`[XX11]`.

To quantify the degree of performance isolation of each memory allocation method, we use the slowdown ratio metric, which is defined as follows.

$$Slowdown\ ratio = \frac{Solo\ IPC}{Corun\ IPC} \tag{1}$$

If the slowdown ratio is high, it indicates poor performance isolation. On the other hand, if the ratio is close to one, it means stronger performance isolation.

Figure 9 shows the slowdown ratios of the SPEC2006 benchmarks under the three memory allocation schemes. On average, partitioning DRAM banks and caches considerably improves performance isolation: The average slowdown ratio of Buddy is 3.29, while the slowdown ratios of PB and PB+PC are 2.67 and 2.13, respectively. Note, however, that even in PB+PC where both DRAM banks and caches are partitioned, the slowdown ratio is far from the ideal unitary value. One possible reason is that the memory data bus is still being shared by all the cores. Therefore, the memory bandwidth can become a bottleneck, especially considering the fact that the co-running benchmark, the 470.lbm, is a highly memory bandwidth intensive one.

Finally, the low slowdown ratios of PB+PC are partly caused by relatively low solo performance compared to PB (see Section VI-B and Figure 8.) To better understand performance tradeoffs, Figure 10 shows corun performance of the

benchmarks. In this figure, although PB+PC provides better performance over PB, the difference is only 4%. Therefore, depending on system requirements, one may favor PB over PB+PC as it has 18% single thread performance advantage on average, while providing a similar degree of co-run performance.

VII. Related Work

The basic idea of using DRAM organizational information in allocating memory at the OS level is explored in several recent work [23], [12], [14]. In [23], the goal is to maximize throughput by enforcing randomness in the page frame allocation decision. This approach works when the number of banks is much larger than the number of cores as the probability of bank-conflicts would be low. It does not, however, eliminate bank-conflicts hence the worst-case behavior is still the same. Our work is different in that we focus on performance isolation and partition banks to completely eliminate inter-core bank-conflicts. In [12], the main goal is to reduce power consumption by arranging page allocations in part of DRAM to allow the unused part of DRAM to make transitions to low-power states. While their application level coloring scheme can also be used for performance isolation, their kernel level memory allocator does not naturally support bank (and rank) interleaving modes, which are necessary for bank-level partitioning. The work in [14] is most similar to our work as it also partitions DRAM at the bank level amongst cores. The differences are it focuses on throughput and the evaluation is based on a simulator while our work focuses on performance isolation and is based on actual implementation on two real hardware platforms.

There are several predictable memory controller proposals that are closely related to our work [24], [27], [2], [22], [7]. The work in [24] and [27] both use the private banking scheme similar to our work. Because each core accesses its own banks, interferences due to bank sharing are eliminated. They differ in that the controller in [24] uses close page policy with TDMA scheduling while the work in [27] uses open page policy with FCFS arbitration. Both approaches, however, require hardware support to partition the banks from the memory controller, while our work is implemented entirely in software on COTS multicore platforms. AMC [22] and Predator [2] utilize interleaved bank and close page policy. Both approaches treat multiple memory banks as a single unit of access which effectively transforms multiple resources into a single resource. They differ in that AMC uses a round-robin arbiter while Predator uses the credit-controlled static-priority (CCSP) arbitration [3], which assigns priority to requestors in order to guarantee minimum bandwidth and provide a bounded latency. Again, these proposals require hardware modifications to the existing COTS systems. On the other hand, our work works on the existing COTS systems.

Cache partitioning has been studied in both hardware [15], [21] and software [18], [19], [30], [25], [5], [26], [20] perspectives. In software based approaches, commonly known as cache coloring, the basic idea is similar to ours in the sense that the physical locations, in this case cache-sets, are considered in assigning memory objects. The first such work can be found in [18]. More recently, Mancuso et al. proposed a combination of coloring and lockdown mechanisms to perform deterministic allocation of frequently accessed memory areas in cache, protecting them from external interference [20]. Ward et al. proposed an OS level cache scheduling technique. In their work, the colors of accessed pages of an observed task is tracked at an OS level, so that individual blocks of cache can be considered as shared resources whose access has to be serialized, thus allowing to study the problem from a scheduling point of view [26]. Herter et al. proposed CAMA, a user-level memory allocator for better WCET analysis and deterministic time allocation [8]. Our work focuses on DRAM banks but can be used in conjunction with cache partitioning to provide better isolation as we demonstrated in our evaluation.

VIII. Conclusion

We presented PALLOC, a DRAM bank-aware memory allocator, for performance isolation on multicore platforms. It exploits the page-based virtual memory system to allocate memory pages on specific DRAM banks. This allows us to configure systems in a way to minimize bank sharing among concurrently executing applications.

Using PALLOC, we performed an extensive set of experiments to investigate the performance impact of the private DRAM banking strategy on two COTS hardware platforms with a set of synthetic and SPEC2006 benchmarks. Our finding is that private DRAM banking significantly improves the quality of performance isolation and real-time performance on COTS multicore platforms. However, we also found that partitioning DRAM banks (and caches) is still far from ideal performance isolation due to contention in other shared resources, including the memory bus, in the memory hierarchy.

To achieve better performance isolation, we plan to incorporate bandwidth management techniques [29], together with the space partitioning techniques presented in this paper.

Acknowledgements

We thank our shepherd Harvey Tuch for his useful comments and feedback. We also thank the anonymous reviewers for their valuable comments. This research is supported in part by NSF CNS 1302563, by ONR N00014-12-1-0046, by NSF CNS 1035736, by NSF CNS 1219064 and by Lockheed Martin Corporation.

References

[1] The Growing Importance of Big Data and Real-Time Analytics. Technical report, Intel Corp., 2012.

[2] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2007.

[3] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2008.

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 1999.

[5] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In *European Conf. on Computer systems (EuroSys)*. ACM, 2011.

[6] D. Eklov, N. Nikolakis, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: quantitative characterization of memory contention. In *Parallel architectures and compilation techniques (PACT)*, 2012.

[7] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed tim-criticality memory controllers. In *Design, Automation and Test in Europe (DATE)*, 2013.

[8] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2011.

[9] Intel. *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[10] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[11] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation: a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *Intel Corporation, VSSAD*, 2007.

[12] M. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. Doshi. A framework for application guidance in virtual memory systems. In *Virtual Execution Environments (VEE)*. ACM, 2013.

[13] JEDEC. DDR3 SDRAM Standard JESD79-3F, July 2012.

[14] M. Jeong, D. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2012.

[15] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 111–122. IEEE, 2004.

[16] C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[17] C. Lameter. Local and remote memory: Memory in a Linux/NUMA system. In *Linux Symposium*, 2006.

[18] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium (RTAS)*. IEEE, 1997.

[19] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*. IEEE, 2008.

[20] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[21] J. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 57–68. ACM, 2007.

[22] M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4):86–90, 2009.

[23] H. Park, S. Baek, J. Choi, D. Lee, and S. Noh. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-core, Multi-bank Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '13, pages 181–192, New York, NY, USA, 2013. ACM.

[24] J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

[25] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 2008.

[26] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[27] Z. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-Requestor Systems. In *Real-Time Systems Symposium (RTSS)*, 2013.

[28] A. Wulf and A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[29] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[30] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European Conf. on Computer systems (EuroSys)*, 2009.