Coordinate Channel-aware Page Mapping Policy and Memory Scheduling for Reducing Memory Interference among Multimedia Applications

Gangyong Jia, *Member, IEEE*, Guangjie Han, *Member, IEEE*, Aohan Li, and Jaime Lloret, Senior *Member, IEEE*

***Abstract*—In modern multi-core system, memory is shared among more and more concurrently running multimedia applications. Therefore, memory contention and interference is more and more seriously which induces system performance degradation significantly, each thread performance degradation differently, unfairness resource sharing and priority inversion even starvation. In this paper, we propose an approach of coordinating channel-aware page mapping policy and memory scheduling (CCPS) to reduce inter-multimedia application interference in the memory system. The idea is to map the data of different threads to different channels, combining with memory scheduling. The key principles are policies of page mapping and memory scheduling 1) memory address space, thread priority and load balance, 2) prioritize low memory request thread, row buffer hit access and older request. We evaluate CCPS on a variety of mixed single and multi-thread benchmarks and system configurations and compare them to four previously proposed state-of-the-art reducing interference policies. Experimental results demonstrate that CCPS improves performance while reducing energy consumption significantly, moreover, CCPS incurs much lower hardware overhead than current proposed policies.**

***Index Terms*—Memory contention, memory interference, performance, page mapping, memory scheduling, fairness, energy.**

I. Introduction

Multi-core systems have become so prevalent not only in desktops and servers but also in multimedia platforms, which may be considered the norm for modern computing systems. However, modern multi-core systems are designed to allow clusters of cores to share hardware structures, including main memory which is one of the most important shared resources. Although shared resources improve hardware utilization and power effective, there is a fundamental flaw induced by it which threads executing concurrently on a multi-core chip contend with each other to access main memory resulting in significant degradation in system performance, individual thread performance and fairness simultaneously for memory interference among threads.

These degradations from using shared resources, especial sharing memory, display in many phenomenon: 1) locality disturbed more and more with the cores increase, row buffer hit ratio is worse along with the core number is more. Figure 1 demonstrates the row buffer hit ratio decreases with the number of parallel running threads increased. Clearly, with the parallel running threads being more the row buffer hit ration decreases seriously; 2) performance degradation different for individual thread, which depends on both behavior of itself and other concurrently running threads. Figure 2 demonstrates the performance degradation of individual thread after four threads running concurrently relative to run solo. Obviously, the difference of individual thread's performance degradation is serious; 3) unfairness sharing the resources, which can not guarantee the quality of service (QoS); 4) priority inversion, higher priority thread is occupied by less priority for less main memory allocated, moreover, starvation may happen.

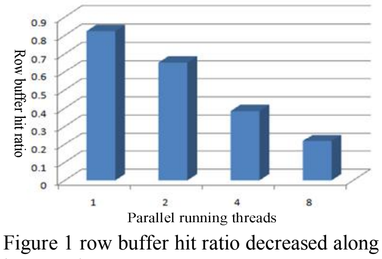


Figure 1 row buffer hit ratio decreased along with the parallel running threads increased

A considerable number of prior works have proposed several different approaches to reduce memory interference among threads for improving system performance, predictable of individual thread performance degradation and fairness. For example, thread scheduling [1-6, 31, 32], leveraging the different characteristics information, has been demonstrated to be able to effectively reduce the memory contention and interference; memory scheduling policies [7-12, 33-35], prioritize the requests of row buffer hit, different applications and so on, which reduces interference; memory

This paragraph of the first footnote will contain the date on which you submitted your paper for review.

Gangyong Jia is now with the Department of Information & Communication Systems, Hohai University, Changzhou, China and the Department of Computer Science, Hangzhou Dianzi University, Hangzhou, China (Email: gangyong@hdu.edu.cn)

Guangjie Han is now with the Department of Information & Communication Systems, Hohai University, Changzhou, China (Email: hanguangjie@gmail.com)

Aohan Li is now with the Department of Signal and Information Processing, Heilongjiang University, Changzhou, China (Email: liaohan1989@gmail.com)

Jaime Lloret is with the Integrated Management Coastal Research Institute, Universidad Politecnica de Valencia, Valencia, Spain (Email: jlloret@dcom.upv.es).

channel/rank/bank partitioning [13, 14]; memory interleaving [15]; and source throttling [16] and so on.

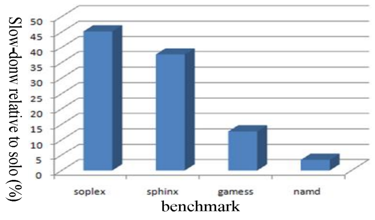


Figure 2 performance degradation of individual thread after 4 threads running parallel

Although these previous proposals are effective, almost all of them have one or more below problems: 1) require non-negligible changes to the existing memory controllers; 2) improve not just only one of the three goals which are system performance, predictable of individual thread performance and fairness, but also at the expense of other two goals; 3) effective of the proposals depends on concurrent running threads, sometimes are good, but sometimes are negative; 4) disturb the priority which frequently appearing priority inversion.

In this paper, we propose an approach to effectively reduce memory interference to improve system performance among concurrently running multimedia applications, predictable of individual thread performance degradation, fairness and power efficiency simultaneously through coordinating channel-aware page mapping policy and memory scheduling (CCPS). Our CCPS partitions memory channels for each core which combines thread group partition and page mapping policy to form one thread group run on one core using a stationary channel, moreover, coordinate with our behavior-aware memory scheduling policy. So, our CCPS reduces interference both from exclusively using channel and memory scheduling.

We implement CCPS in different system configurations and use mixed single and multi-thread benchmarks. Experimental results show CCPS reduces row buffer miss rate and switch overhead than buddy algorithm which improves system performance. Besides, CCPS improves fairness which is mainly because of behavior-aware memory scheduling and channel partition. Moreover, CCPS saves 6.1% of the energy consumption of memory system.

In summary, we make the following contributions:

(1) Allocate each core stationary memory channels, which are enough for performance. Along with the more core number, memory channels are more needed, which we expand modern memory system into more channels.

(2) Based on channel-aware page mapping policy, aggregate physical memory pages of each thread into specific memory channel.

(3) Partition threads into thread groups combining with page allocation policy to form one thread group run on one core using its unique memory channels which memory access is parallel among cores not simultaneous any more.

(4) According to threads' behavior of parallel running, schedule memory request. In this way, interference is reduced further and threads are fairness in performance degradation.

The rest of this paper is organized as follow. In section 2, we introduce the background of DRAM system and related works. In section 3, we present our CCPS in detail. The methodology and metrics are discussed in section 4. We evaluate CCPS in section 5. We conclude in section 6.

II. Background and Related Works

We provide a brief overview of modern memory subsystems, then analyze the OS memory management and relationship between thread performance with allocated channels/banks, finally introduce the related works.

A. DRAM Organization

Figure 3 illustrates the multiple levels of organization of the memory subsystem. To service memory accesses, the memory controller (MC) sends commands to the DIMMs on behalf of the CPU's last-level cache across a memory bus. As shown, recent processors have integrated the MC into the same package as the CPU. To enable greater parallelism, the width of the memory bus is split into multiple channels. These channels act independently and can access disjoint regions of the physical address space in parallel [17].

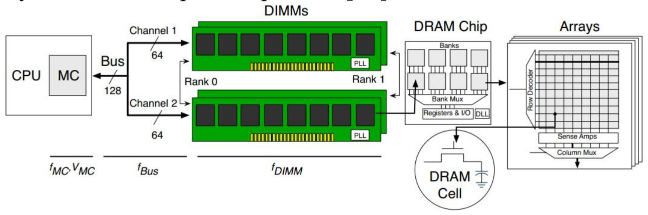


Figure 3 organization of a modern memory subsystem

Multiple DIMMs may be connected to the same channel. Each DIMM comprises a printed circuit board with register devices, a Phase Lock Loop device, and multiple DRAM chips. The DRAM chips are the ultimate destination of the MC commands. The subset of DRAM chips that participate in each access is called a rank. The number of chips in a rank depends on how many bits each chip produces/consumes at a time. Each DIMM can have up to 16 chips, organized into 1-4 ranks.

Each DRAM chip contains multiple banks (typically 8 banks nowadays), each of which contains multiple two-dimensional memory arrays. The basic unit of storage in an array is a simple capacitor representing a bit—the DRAM cell. Thus, in a x8 DRAM chip, each bank has 8 arrays, each of which produces/consumes one bit at a time. However, each time an array is accessed, an entire multi-KB row is transferred to a row buffer. This operation is called an "activation" or a "row opening". Then, any column of the row can be read/written over the channel in one burst. Because the activation is destructive, the corresponding row eventually needs to be "pre-charged", that is, written back to the array.

B. OS Memory Management

Nowadays, Linux kernel's memory management system uses a buddy system to manage physical memory pages. In the buddy system, the continuous 2order pages (called a block) are organized in the free list with the corresponding order, which ranges from 0 to a specific upper limit. When a program accesses an unmapped virtual address, a page fault occurs and OS kernel takes over the following execution wherein the buddy system identifies the right order free list and allocates one block (2order physical pages) for that program. Usually the first block of a free list is selected but the corresponding physical pages are undetermined [21].

In Linux operating system, the default page allocation is using buddy algorithm, figure 4 demonstrates physical pages organization, which allocates the first block of a free list to the request thread. So, a thread's occupying memory may cover all channels/banks of the memory.

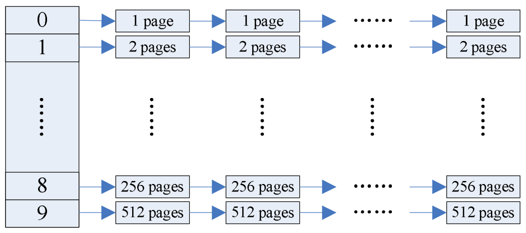


Figure 4 physical memory pages organization of buddy algorithm

C. *Channel/bank Amount Requirement for One Thread*

Buddy algorithm of the Linux operating system takes advantage of parallelism to improve performance. However, the experimental results demonstrate the necessary amount of one thread requirement banks is limited [4].

In order to illustrate the channel/bank requirement of every thread is limited, we perform experiments of comparing performance improvement along with increased channel amount from 1 to 4 channels and bank amount from 8 to 64 banks, which conduct as many as possible benchmarks. Each channel contains 16 banks in the experiment. Figure 5 demonstrates the correlation between each benchmark performance and bank amount, which allocated banks spread all channels. Expectedly, the necessary required amount of banks for one thread is limited, mostly as we find, 16 banks are enough. More than necessary, the performance hardly improved for all threads.

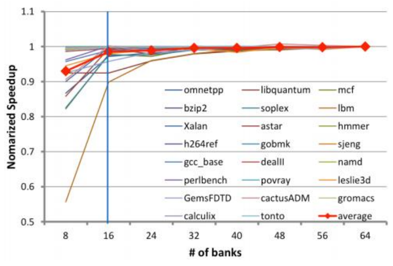


Figure 5 thread performance improves along with bank amount added

Figure 6 shows the correlation between each benchmark performance and channel amount. All banks belong to the allocated channels can be used by the running benchmarks. Obviously, similarly to the figure 5, the necessary required amount of channels for one thread is limited, 1 channel are enough. More than necessary, the performance hardly improved.

And for some reasons like memory dependency and high cache hit rate, a single core is unable to generate enough concurrent memory requests. Nevertheless, buddy algorithm of Linux interleaves memory requests across memory banks for taking the advantages of channel-level/bank-level parallelism, thus a thread's occupying memory may cover all channels/banks of the memory, which largely exceeds its necessary channel/bank amount. Therefore, those threads of occupied across whole memory channels/banks only suffer from memory interference rather than obtain any performance gain.

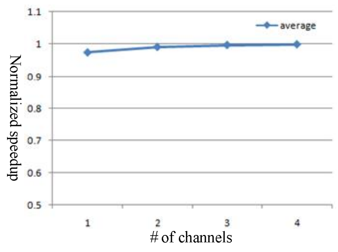


Figure 6 thread performance improves along with channel amount added

Therefore, allocate unique memory channels/banks for each core which can reduce memory interference while not affecting performance.

D. *Related Works*

There are a number of related studies.

Thread Scheduling. Scheduling algorithms aimed to distribute threads to get an even distribution of miss rate among multiple caches are proposed in [22], which avoid severe contention on shared resource of cache, memory controller, memory bus and prefetching hardware. Similar mechanisms are also proposed in [23, 24]. Although these methods can alleviate contention, they hardly eliminate the bank interference among threads.

Channel Partition. Data of different threads are mapped into different channels according to their memory access behavior in [25], which can eliminate the interference between threads at channel level. However, channel partition cannot be applied to system with cache line interleaving policy between channels [25], which limit its applicable scope. Furthermore, there are usually more threads than channels in a system, so some threads have to be assigned to the same channel, which still interference with each other. Besides, channel partition actually partitions the bandwidth of memory system into several portions. Since the total number of portions is limited by channel amount, which is usually small, it is challenging to seek a balance among channels so as to ensure no bandwidth wasted.

Thread-based Memory Scheduling. Memory controllers are designed to distinguish the memory access behavior at thread-level in [16, 18, 19, 25, 27], so that scheduling modules can adjust their scheduling policy at the running time. TCM [18], which dynamically groups threads into two clusters (memory intensive and CPU intensive), and assign different scheduling policy to different group, is the best scheduling policy, which aim to address fairness and throughput at the same time. Yet, this method needs modification to memory controller, and the overhead at running time cannot be neglected.

Row buffer optimization. In [26], frequently accessed data of different rows are dynamically migrated into row buffer, which can improve the row buffer usage and performance; power consumption is also lowered by reducing the operations of precharge and active. In [27], the content in row-buffer will be precharged after 4 times access, which target at the reduction of row-buffer conflicts.

III. CCPS

Our coordinating channel-aware page mapping policy and memory scheduling (CCPS) consists of five components: 1) assigning unique memory channel for each core, 2) binding threads to the running cores, 3) allocating pages of specified channel to each thread, 4) scheduling threads for running, 5) modifying memory scheduling policy for further reducing interference.

The first component is soft static (Sec 3.1), one core is assigned one memory channel, which the correspondence between the core and memory channel is unchanged except for special circumstances like one core shuts down. The second component is proceeded for one time when a new thread is created (Sec 3.2). According to memory address space, thread priority of the thread and load balance of the system, bind the thread to a core, which mostly run the thread on the core. Any time when access to a missing page, the third component is evoked (Sec 3.3). Based on the core of thread running, allocate pages belonging the core to the thread. Both component four and five only adjust the current policies (Sec 3.4 and 3.5), our memory scheduling policy prioritizes low memory request thread, row buffer hit access and older request, which further reduces interference and improves performance.

A. Assign Unique Memory Channel for Each Core

In section 2.3, we have demonstrated one memory channel is enough for most threads. So, in this paper, we assign each core one memory channel.

Most modern memory system is usually packaged as DIMMs, each of which usually contains 1 or 2 ranks and 8 banks. A memory system can contain multiple channels, and each channel is associated with 1 or 2 DIMMs which only has 64 memory banks. 4 channels/64 memory banks can only be assigned for most 4 cores if each core occupying unique 1 channel/16 banks, demonstrated like in figure 7. Core 0, 1, 2 and 3 occupy channel 0, 1, 2 and 3 respectively. Every core hardly accesses unoccupied memory unless accessing operating system memory.

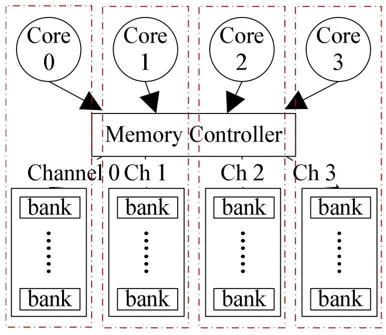


Figure 7 one unique channel/64 banks assigned to one core

If there are more than four cores in the system, some cores need to share one channel/16 banks. Cores sharing one channel called a core group. Cores belonging different core groups occupy different channels/banks. Figure 8 demonstrates two cores share a channel and four core groups occupy different four channels. Core 0 and 4 forming group 0 occupy channel 0, core 1 and 5, 2 and 6, 3 and 7 form group 1, 2 and 3 respectively and occupy channel 1, 2 and 3 uniquely. Cores of the same group are still contending and interfering with each other. For reducing interference among the same group, we intergrade memory scheduling policy, which will introduce below.

In section 2.1, we have presented each DIMM can have up to 16 chips, each DRAM chip contains multiple banks (typically 8 banks nowadays), which is more 128 banks, even 256 or 512. And if remaining only four channels in the system, every channel will contain more than 32 memory banks, which means one channel is more than necessary for one core. We can partition memory banks belonging one channel for cores of the same core group which reduces interference among cores of the same core group. Figure 9 presents a conceptual example showing the performance benefits of memory banks partition belonging the same channel which reduces the interference among cores of the same core group. Figure 9(a) and 9(b) show characteristic examples of what will happen with conventional memory assignment policy (both core 0 and 4 occupy all banks spread the whole channel which shares some banks) and with banks partition in the same channel (where core 0 and 4 occupy different banks of the same channel), respectively. In the first case, requests from both core 0 and 4 are interrupted each other in banks of channel 0 (see Fig 9(a)). As a result, both core 0 and 4 stall more time for increasing row buffer miss. In contrast, if two cores' data are mapped to different banks of the same channel as shown in figure 9(b), both core 0 and 4 are not interrupted by the other, reducing the interference from the other core to speed up progress.

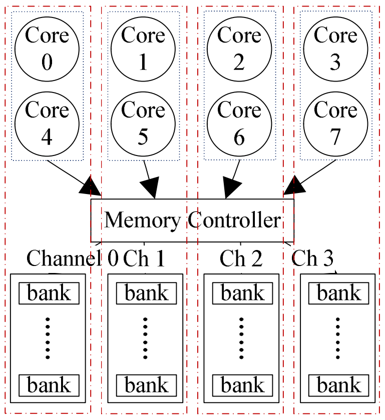
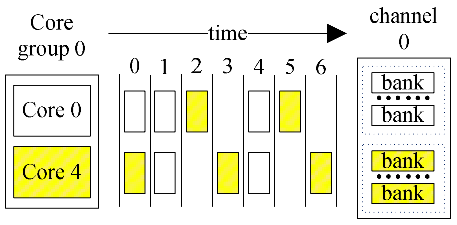


Figure 8 some cores share a channel and different core groups occupy different channels

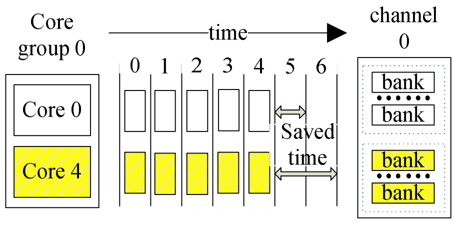
In this paper, we adopt both the most modern memory system architecture consisting of 4 channels and 64 banks and memory architecture containing 4 channels and 256 banks. Every core is assigned 16 banks within one channel. In order to optimize memory power efficiency, assigned 16 banks are limited in less memory ranks for rank is the smallest physical unit for power management. If there are no memory accesses to some memory ranks for cores being idle, these ranks can be set low power mode.

B. Bind Threads to the Running Cores

All cores in the system are assigned 16 banks within one channel respectively, which hardly exceeds their own assigned region. Therefore, threads need to be bind with one unique core, use memory region of the unique core and run on that core, seldom migration among cores, which take advantages of preventing frequently flushing cache, TLB, memory and so on.



(a) Conventional memory assignment policy



(b) Bank partition belonging one channel

Figure 9 conceptual example showing the performance benefits of banks partition belonging the same channel

However, in the operating system, thread migration has some advantages, like: 1) dynamic load distribution is possible in multiprocessing systems to balance the load on the different cores, by migrating threads from overloaded cores to less loaded ones; 2) fault resilience can be achieved in such systems, by migrating threads from cores that may have experienced a partial failure or are likely to fail completely in the immediate future; 3) improved system administration can be achieved by migrating threads from cores that are about to be shut down or otherwise made unavailable; 4) resource sharing is possible on a grid, by migration of a thread to a specific core that is equipped with a special hardware device, large amount of free memory or some other unique resource.

Although our binding threads to running cores policy seldom migrating threads among cores, we can take not less than advantages of the thread migration. Our binding threads to cores method combines policies of memory address space, thread priority and load balance simultaneously. Based on load balance policy, we can satisfy dynamic load distribution. If some cores fail, our method can migrate threads from failure cores to others according our adopted policies to meet fault resilience and system administration. Introducing memory address space policy, bind threads of sharing memory address space to the same core, not only sufficing the resource sharing but also improving utilization efficiency. Moreover, thread priority policy decreases the average response time to improve real-time.

Algorithm 1 demonstrates the process of a new thread binds to one unique core. Every time a new created thread is bind to a core according to three policies of memory address space, thread priority and load balance simultaneously.

Algorithm 1: process of a new thread binds to one core

```
Thread T is the new thread
begin
1: check whether binding to exist cores;
2: if so
3:     bind T to one of those cores;
4: else
5:     find one core based on priority and load balance;
6: return;
End
```

Firstly, check whether there are threads binding to some cores sharing memory address space with the new emerging one. If exist one core and no 5 threads more than the smallest load core, bind the new thread to one of those cores. Switching between threads of sharing memory address space avoids replacing the TLB and cache and increases row buffer hit ratio, which both have advantages in performance improvement.

Secondly, if no existing thread shares memory address space with new thread or existing one core but too many threads binding to that core (5 threads more than other cores), find another core according both thread priority and load balance.

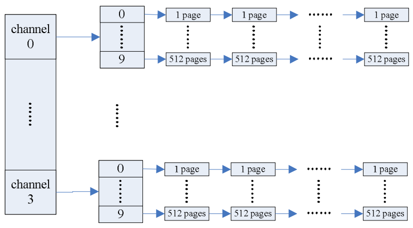
Thirdly, if exist more than one core which has threads sharing memory address space with the new thread, find one of these cores based on both thread priority and load balance.

Policies of thread priority and load balance choose three smallest load cores firstly, queue the three cores based on average thread priority decreasing. Compare the new thread's priority with every core's average priority of the three cores. If the new thread has more priority than the lowest average priority of the three cores, bind the new thread to the core of lowest average priority; else, bind the new thread to the core of highest average priority.

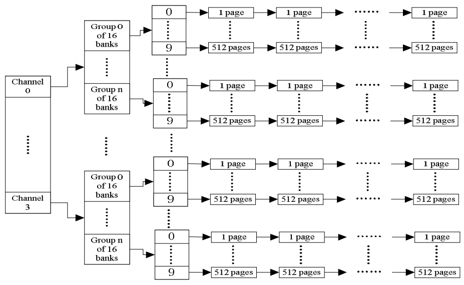
C. Channel-aware Page Mapping

In section 2.2 has introduced buddy algorithm which is the most used memory management method of current operating system, allocating pages for each thread spreading all channels/banks of the whole memory. However, every core can only use one channel (even part banks within one channel) in our CCPS for reducing interference among parallel running threads in different cores. So, we proposal channel-aware page mapping policy which maps pages according thread's binding core. Allocate pages for a thread within memory channel/banks assigned to its binding core, limiting one channel/16 banks.

Therefore, physical pages are organized not only according free block size but also inserting channel (even bank) information. Figure 11(a) demonstrates physical memory pages organization containing channel information which is enough for the most modern memory system architecture consisting of 4 channels and 64 banks. But for memory architecture containing 4 channels and 256 banks, bank information is necessary to partition banks within one channel. Figure 11(b) shows physical memory pages organization consisting both channel and bank information additionally. The most difference with the default organization of buddy algorithm is free block is organized channel/bank and free size, which the whole memory subsystem is partitioned regions.



(a) physical pages organization inserting channel information



(b) physical pages organization inserting both channel and bank information

Figure 11 physical pages organization for different memory architecture

After organizing the free blocks, when page request from one thread, our channel-aware page mapping policy checks the core request page before identifying the right order free list and allocating one block (2order physical pages). Algorithm 2 describes how our channel-aware page mapping policy allocates a free page after requiring.

```
Algorithm 2: channel-aware page mapping policy
Thread T accesses an unmapped virtual address, OS
kernel maps pages
begin
1: find the core C which T binds to;
2: according to the C, find assigned channel $id_c$ (and bank
   $id_b$);
3: find the free lists of $id_c$ (and $id_b$);
4: search the suitable free block based on buddy algorithm
   within $id_c$ (and $id_b$);
5: allocate a block for T;
6: return;
End
```

Firstly, determine which core requests the page for restricting the region of allocating. Through assigned channel and banks of the core, locate the region in the memory.

Secondly, identify the right order free list within the located region and allocate one block (2order physical pages) for that thread.

D. Schedule Threads for Running

Every thread binds to one core, each core has many threads. All threads on one core are organized into an rb-tree according each thread's vruntime, which is the same with the default Completely Fair Schedule (CFS) of Linux. So, there are N rb-tree groups in the system (N is also the core number). Figure 12 demonstrates the framework of every core bind by some threads and assigned one channel/16 banks with the most modern memory system architecture consisting of 4 channels and 64 banks. For memory architectures containing 4 channels and 256 banks or more, we only need insert bank information additionally within channel. Commonly, threads run on their bind core and use assigned memory for that core, seldom exceed.

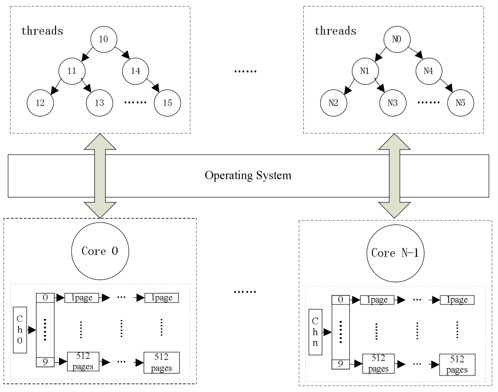


Figure 12 the framework of every core bind by some threads and assigned one channel/16 banks

The schedule policy for every core is almost the same with Completely Fair Schedule (CFS) of Linux operating system, which each thread is located in the rb-trees according its vruntime. Vruntime of every thread is related with its priority and its waiting time for running. Choose the thread of smallest vruntime to run next for each time. The mainly differences to CFS are regarding to migration:

1. Prevent calling the load_balance service of the kernel as much as possible. After calling the load_balance, threads will be migrated from one core to the other, which means the occupied memory also needs to migrate, but memory migration is costly.

2.When one core is shut down, all threads bind to it are migrated to other cores according our policy of binding threads to cores (introduced in section 3.2).

Kernel threads run on all cores, which is the different with user threads.

E. Modify Memory Scheduling Policy

Our channel-aware page mapping policy mainly solve memory interference problem through accessing different memory channel/bank for parallel running threads. But if parallel running threads access the same channel/bank (two cores access the same channel/bank in the most modern memory system architecture), our channel-aware page mapping policy can not deal with them.

In order to solve the memory interference among threads of accessing the same channel/bank, we propose a memory scheduling policy of prioritizing low memory request thread, row buffer hit access and older request.

Under many experiments, we observe very low memory request threads will seriously interfer by other threads. But giving priority to run over other threads, they do not cause significant slowdowns to other threads. Figure 13 shows the performance different degradation with/without prioritizing low memory request thread when benchmarks of lbm and namd parallel running. If not giving priority to the low memory request thread of namd which interfers seriously by lbm and degrades performance significantly. Else, both lbm and namd threads' performance degrades slightly to improve the system throughput. Low memory request threads, like namd, seldom generate memory requests and prioritize these requests which enable the threads to quickly continue with long computation periods and utilize their cores better without disturbing other threads significantly, thereby significantly improving system performance.

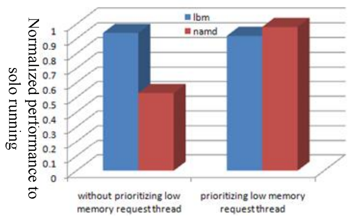


Figure 13 performance different degradation with/without prioritizing low memory request thread

Moreover, we observe prioritizing row buffer hit access can shorten memory access time than in-order memory scheduling policy in sharing channel/bank circumstance. Figure 14 (left) and (right) demonstrate the result sequence of the in-order memory access scheduling and prioritizing row buffer hit access memory scheduling respectively. The x-axis shows the scheduling order from left to right and y-axis shows the execution clock cycles from top to bottom. Finish the example memory access trace takes 36 clock cycles for in-order scheduling and 30 cycles for prioritizing row buffer hit access scheduling which is 16.67% shorter than in-order scheduling. Obviously, from figure 14 we can take advantages of the prioritizing row buffer hit access memory scheduling.

Considering fairness of accessing memory among parallel running threads sharing channel/bank, we also prioritize older request. If some memory requests grab by low memory request from threads and row buffer hit access for waiting T memory cycles, these requests can be executed immediately. In this way, one memory request will not starve and parallel running threads will be more fairness in accessing memory.

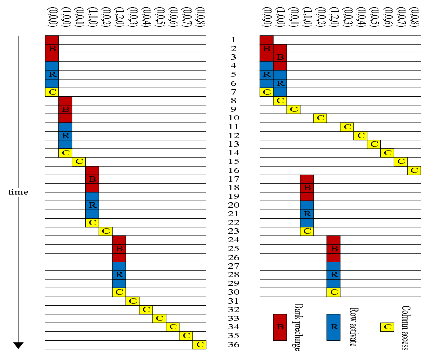


Figure 14 in-order memory scheduling (left) and prioritizing row buffer hit access memory scheduling (right)

In this paper, we implement the memory scheduling policy of prioritizing low memory request thread, row buffer hit access and older request simultaneously in circumstances of some parallel running threads sharing channel/bank.

F. Bandwidth Partition

After forming CCPS of parallel accessing memory for each core, the mainly contention is from bandwidth. If the bandwidth allocates to each parallel running thread fairly, the performance differently degradation and fairness problems induced by shared memory will mitigate.

There are N parallel running threads in the N-core system, using T1, T2, … , TN represent each thread. And through performance management unit (PMU), we can count committed instructions and access memory numbers/last level cache misses of each thread, which represented by INi and Mi respectively for thread i. B stands for the total bandwidth. Therefore, we need to determine B1, B2, … , BN for each thread in order to fairly share. Moreover, the performance degradation of each thread is (INi+Bi)/(INi+Mi), which Bi is smaller than Mi for contending bandwidth.

So, for random i and j which 1⩽i, j⩽N, must be

$$(IN_i+B_i)/(IN_i+M_i) = (IN_j+B_j)/(IN_j+M_j) \quad (1)$$

$$(B_1+ B_2+ \dots + B_N) = B \quad (2)$$

Through (1) and (2), we can get each Bi, which allocates for each thread. After that, the process of bandwidth partition finishes.

IV. EXPERIMENTAL SETUP

We use MARSSX86 [28] as the base full-system architectural simulator to run Linux 2.6.31 and extend its memory part with DRAMSim simulator to simulate DDRx DRAM systems in the details. Table 1 shows the major simulation parameters of the eight core with one memory controller for the most modern memory system architecture, and most parameters are the same for memory architecture containing 4 channels and 256 banks, which only has more banks.

In order to evaluate our CCPS, we simultaneously run different combinations of selected from sysbench [30], SPEC2000 and SPEC2006. In table 2, the number-appname notation is the number of threads of the application with the name of appname for sysbench; for SPEC2000 and SPEC2006 workload, it is the number of copies of the application with the name of appname. After conducting experiment to get each benchmark's memory access characteristic, we classify the benchmarks into different categories: memory-intensive and memory-non-intensive. From mix1 to mix9 in table 2, workloads are less and less memory-intensive.

Table 1 Processor and memory configurations

Feature	value
CPU cores	four/eight/sixteen cores
L1 I/D cache (per core)	16KB, 2-way
L2 cache (shared)	64KB
Cache block size	64bytes
Memory configuration	4 GB, 4 channels, 8 ranks, 8banks per rank

Evaluation Metrics. We measure system throughput using weighted speedup and fairness using maximum slowdown.

$$\text{weighted speedup} = \sum_i \frac{IPC_i^{spared}}{IPC_i^{alone}} \qquad (2)$$

$$\text{maximum slowdown} = \max_i \frac{IPC_i^{alone}}{IPC_i^{spared}} \qquad (3)$$

Table 2 Workload description

mix	Sysbench, SPEC2000 and SPEC2006
mix1	18-sysbench cpu, 3-povray, 3-tonto, 3-calculix, 3-perlbench, 3-namd, 3-wrf
mix2	18-sysbench cpu, 3-perlbench, 3-namd, 3-wrf, 3-dealII, 3-gcc, 3-sjeng
mix3	18-sysbench cpu, 3-dealII, 3-gcc, 3-sjeng, 3-gobmk, 3-gromacs, 3-h264ref
mix4	9-sysbench cpu, 9-sysbench memory, 3-gobmk, 3-gromacs, 3-h264ref, 3-bzip2, 3-hmmer, 3-astar
mix5	9-sysbench cpu, 9-sysbench memory, 3-h264ref, 3-bizp2, 3-hmmer, 3-astar, 3-cactus, 3-omnetpp
mix6	9-sysbench cpu, 9-sysbench memory, 3-hmmer, 3-astar, 3-cactus, 3-omnetpp, 3-xalanc, 3-sphinx3
mix7	18-sysbench memory, 3-cactus, 3-omnetpp, 3-xalancbmk, 3-sphinx3, 3-gems, 3-lbm
mix8	18-sysbench memory, 3-xalancbmk, 3-sphinx3, 3-gems, 3-lbm, 3-soplex, 3-leslie3d
mix9	18-sysbench memory, 3-gems, 3-lbm, 3-soplex, 3-leslie3d, 3-libquantum, 3-mcf

V. EXPERIMENTAL RESULTS

In this section, we first examine if the CCPS improves the system performance. Then, we analyze system fairness. Finally, we show the power reduction and sensitivity of our CCPS.

A. System Performance Analysis

We compare CCPS's performance against for four methods, CFS, TCM, BPM and IMPS. CFS is the default method of the Linux operating system, which taken as the standard for others; TCM [1] is one of the best previous thread scheduling method for trading off between performance and fairness; BPM [36] is one of the best policy for reducing memory contention and interference through partitioning memory banks; IMPS [6] is one of the best policy to combine the interference reduction benefits of both the system software page mapper and the memory request scheduling hardware. IMPS is the most like to our CCPS, we will detail analyze our advantages to these methods below, especially to IMPS.

Figure 15 demonstrates the performance improvement of four methods normalized to CFS in 8-core but with different memory architectures, one is the most modern memory system architecture consisting of 4 channels and 64 banks and the other contains 4 channels and 256 banks. Obviously, from the figure, we can easily see CCPS is better than other methods in both memory architectures and the more banks in the memory subsystem the better performance improvement of our CCPS. In order to demonstrate the scalability of our CCPS with cores number increased, we compare the performance in different core numbers. So, besides 8-core circumstance, we also demonstrate 4-core and 16-core circumstances in the figure 16, and all circumstances contain both memory architecture. Our CCPS's performance improvement proportional to the number of cores is better than other methods, moreover, in the 256 banks circumstance, our CCPS behaves much better. With more banks in the memory subsystem, every channel has more banks for parallel access to obtain better performance for each core. Therefore, CCPS is scalable to more cores for prevalent multi-core system which core number is more and more. In the figure, the number-core-number notation represents the core number and bank number respectively.

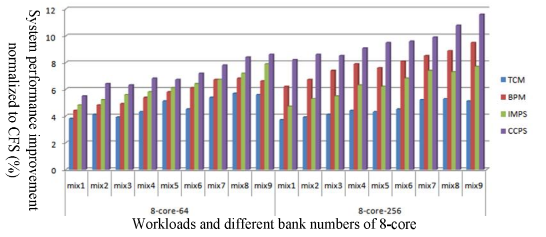


Figure 15 system performance improvement in different workloads and bank numbers

Besides of the core and bank number changing, with the workloads switch from mix1 to mix9, which is less and less memory-intensive, CCPS like other methods is more and more effective in improving overall system performance.

CCPS improves 6.3%, 8.4%, 7.1%, 9.6%, 7.9% and 10.8% system performance on average comparing to the CFS in 4-core, 8-core and 16-core with both memory architectures respectively across 9 workloads. The system improvement is mainly from independent memory access for each core, changing from simultaneously to parallel in accessing shared memory, which solves the memory contention and interference. And another advantage of CCPS is more thread switching between sharing memory address space which reduces the switching overhead. Moreover, combining our memory scheduling policy of prioritizing low memory request thread, row buffer hit access and older request. Besides above three advantages, removing load_balance of the kernel service is also beneficial in improving performance.

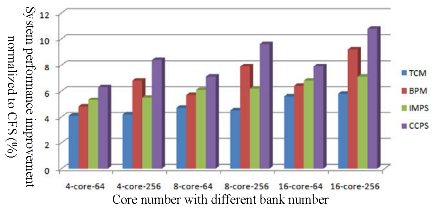


Figure 16 average system performance improvement in different core and bank numbers

Compared to TCM, one of the best performance previous thread scheduling methods, CCPS combined channel aware page allocation, independent accessing memory for each core and our memory scheduling further to improve the overall system performance, 2.2%, 4.2%, 2.4%, 4.8%, 2.5% and 5% more performance than TCM in 4-core, 8-core and 16-core with both memory architectures respectively.

BPM is one of the best performance previous partition memory banks for each thread methods, which also behaves better in 256 banks circumstances with all core number. But our CCPS combined independent accessing memory for each core, memory scheduling and thread scheduling additionally also further to improve the overall system performance, 1.4%, 1.6%, 1.5%, 1.7%, 1.5% and 1.7% more performance than BPM in 4-core, 8-core and 16-core with both memory architectures respectively.

IMPS is the most similar method with our CCPS, except the channel aware page allocation and independent accessing memory for each core of our CCPS. Our CCPS is 1%, 2.9%, 1%, 3.4%, 1.1% and 3.7% more performance than IMPS.

Then, we analyze each of the three advantages in improving system performance.

1) Row Buffer Miss Rate Reduction

Combining channel aware page allocation, independent memory accessing for each core, our memory scheduling policy of prioritizing low memory request thread, row buffer hit access and older request with thread scheduling to improve system performance, we detailed analyze each part of CCPS in improving performance. Firstly, we analyze the reducing row buffer miss rate through memory accessing from simultaneously to parallel and our memory scheduling in multi-core.

Figure 17 demonstrates the reduced row buffer miss rate normalized to CFS of four different methods across 9 workloads in all circumstances of 4 and 8 cores. From the figure, we can obviously see our CCPS reduces more row buffer miss rate comparing to other three methods. This is the one major reason of our CCPS improves more system performance.

CCPS reduces 5.7%, 8.2%, 5.8% and 8.4% more row buffer miss rate compared to CFS on average in circumstances of 4 and 8 cores with both memory architectures respectively, and reduces 1.4%, 3.8%, 0.8% and 2.3% more comparing to TCM, comparing to BPM which reduces 0.7%, 0.8%, 0.6% and 0.9%, and comparing to IMPS which reduces 0.6%, 2.3%, 0.5% and 1.7%. Obviously, With the more cores and more banks, CCPS behaves better and better in reducing row buffer miss rate, which is conserve to TCM, BPM and IMPS, reducing row buffer miss rate slightly better along with increasing cores. In the figure, sometimes we even notice BPM is worse when in 8-core than in 4-core, which is mainly because the effective of BPM in reducing memory interference is bad after cores increasing. Therefore, TCM is better than BPM in 8-core-64 circumstance even though being worse on less cores and 256 banks.

CCPS is not affected by the core numbers, with cores are more and more in the multi-core, even in the many-core system, our CCPS behaves well scalability.

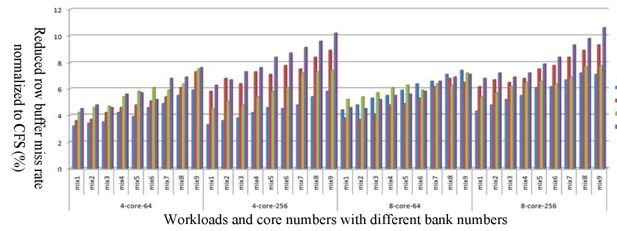


Figure 17 the reduced row buffer miss rate across 9 workloads in four circumstances

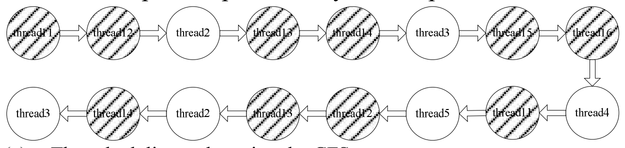
Partition threads into thread groups, and bind one thread group to one core occupying one memory channel, this independent memory accessing mode operates parallel from simultaneously which reduces the memory contention and interference seriously. This part of reducing row buffer miss rate through independent memory accessing is the mainly partial of system improvement.

2) Switch Overhead Reduction

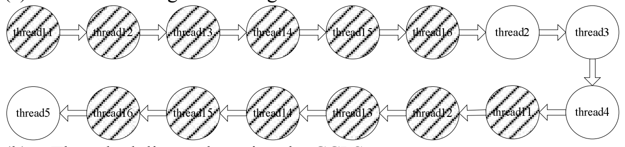
In this section, we analyze the reducing switch overhead through thread scheduling part. After threads partitioned, threads of sharing memory address space are partitioned into the same group which running in the same core and using the same channel. The cost of switching between sharing memory address space threads is much lower.

Figure 18 demonstrates the difference of scheduling order between CFS and CCPS. Figure 18(a) shows the real running sequence using CFS on one core. The slashed threads are multi-threaded threads. Without slashed threads are single-threaded threads. In figure 18(a) threads without slashed are more scattered among single-threaded threads than in figure 11(b) using CCPS. In figure 18(b), sharing memory address threads are always scheduling sequence without inserting other threads to reduce cost.

Table 3 demonstrates the average ratio switch between sharing memory address space in 4-core circumstances. The more ratio of switching between sharing memory address space, the more switch overhead reduced and better performance improvement. From table 3, we can easily to see CCPS is much more in ratio of switching between sharing memory. CCPS is 82.6% more than CFS on average across the 9 workloads, and 83%, 82% and 75% more than TCM, BPM and IMPS respectively. And on other circumstances from 4 to 16 cores with both memory architectures, the improved ratio of switching between sharing memory is likely to 4-core. This part of reducing switch overhead through thread scheduling is also one of the important partial of system improvement.



(a) The scheduling order using the CFS on one core



(b) The scheduling order using the CCPS on one core

Figure 18 scheduling order on one core

Table 3 Ratio of switching between sharing memory in 4-core circumstance

	CFS	TCM	BPM	IMPS	CCPS
mix1	26%	22%	26%	25%	42%
mix2	24%	23%	25%	27%	46%
mix3	25%	25%	27%	26%	47%
mix4	19%	18%	19%	21%	37%
mix5	19%	20%	18%	20%	36%
mix6	21%	21%	20%	20%	34%
mix7	25%	26%	25%	26%	45%
mix8	27%	25%	23%	27%	46%
mix9	25%	27%	26%	25%	48%

3) Advantage of our Memory Scheduling

In this paper, we combine the memory scheduling of prioritizing low memory request thread, row buffer hit access and older request, which reduces the row buffer miss rate to improve throughput while retaining fairness and reducing response time as much as possible.

The aspect of reducing row buffer miss rate, we have analyzed in the 5.1.1 section, in this section, we mainly analyze the response time. Fairness will compare in the below section.

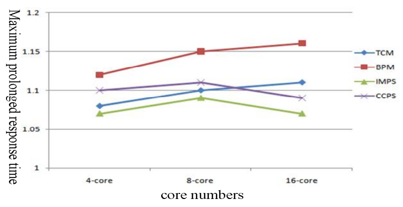


Figure 19 maximum prolonged response time

Figure 19 demonstrates the average response time comparing among four methods normalized to CFS, which adopt maximum prolonged response time as the evaluation metric, the more maximum prolonged response time the worse ability in real-time response. From the figure, we can see our CCPS is not the best method but also not the worst one. Some methods, like TCM and BPM, prolong response time proportion to the core number. But our CCPS and IMPS do not prolong response time with core number increased, which is a well feature for scalability.

B. Fairness Analysis

Figure 20 demonstrates the maximum slowdown of three methods in 4-core and 8-core with both memory architectures circumstances. More maximum slowdown, the worse fairness is. So, from the figure, we can easily find both TCM and IMPS is better than our CCPS and BPM, but our CCPS is better than BPM. BPM is the worst in the fairness for not considering fairness among threads with only the goal of reducing interference. Although our CCPS is worse than TCM and IMPS, the experimental results show acceptable.

The maximum slowdown of TCM is 4.5%, 4.6%, 4.5% and 4.7% on average respectively in four circumstances. The maximum slowdown of BPM is 10.4%, 10.9%, 10.7% and 11.2% on average respectively in four circumstances. The maximum slowdown of IMPS is 4.6%, 4.5%, 4.8% and 4.6% on average respectively in four circumstances. And our CCPS is 5.9%, 6.5%, 6.3% and 6.8%, which is also scalable.

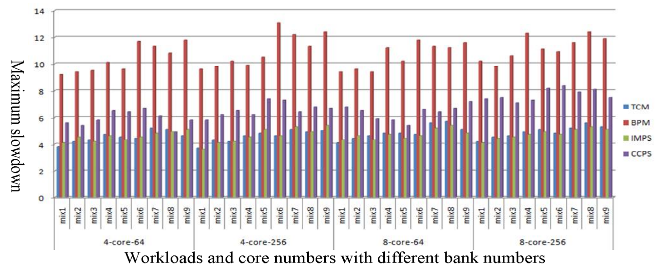


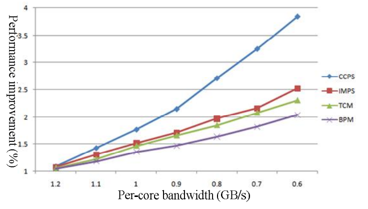
Figure 20 maximum slowdown of four methods in four circumstances

C. Bandwidth Sensitivity Analysis

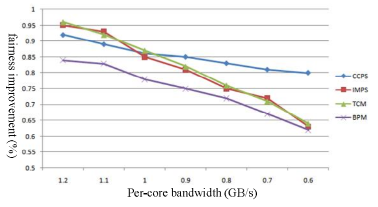
Although the core number is still increasing, memory bandwidth for per-core is decreasing because of off-chip memory bandwidth is limited by the pin count of micro-processor chip, which is considered as the major bottleneck of the scalability of on-chip core number. With the less and less bandwidth of each core, more and more interference needed to be relieved seriously. In order to evaluate the effectiveness of CCPS under different extreme condition, we emulate different bandwidth scenarios for per-core by decreasing from 1.2GB/s to 0.6GB/s.

Figure 21 illustrates the memory bandwidth sensitivity of CCPS to the per-core bandwidth comparing with TCM, BPM and IMPS normalized to max bandwidth. Figure 21(a) clearly shows the negative correlation of performance improvement and per-core bandwidth of all four methods, which means all of TCM, BPM, IMPS and CCPS is better in performance improvement along with increasing per-core bandwidth. But our CCPS is better than other three methods proportion to the reduced per-core bandwidth, which means our CCPS is more robustness in performance than other three methods under extreme condition of bandwidth.

Figure 21(b) shows the correlation of fairness and per-core bandwidth. Proportion to the decreased bandwidth of per-core, CCPS can maintain the fairness. Although other methods can also maintain, our CCPS behaves more non-sensitive. Therefore, our CCPS is also more robustness in fairness than other two methods under extreme condition of bandwidth.



(a) correlation of performance improvement and per-core bandwidth



(b) correlation of fairness and per-core bandwidth
Figure 21 bandwidth sensitivity

D. Power Reduction of CCPS

The active operation is the most power-consuming operation in the DRAM system, because it has to move an entire row from array to a row buffer. CCPS can lower the power consumption of DRAM because of the reduced both row buffer conflict miss rate and switching overhead (as illustrated in Figure 18 and table 3 respectively). We measure the power consumption by simulator, so we can get the value of power savings on memory system. Our experimental results show that CCPS with open-page policy can save up to 5.9% of memory power consumption, better than the configurations without CCPS.

VI. CONCLUSION

In this paper, we propose a CCPS approach, coordinating channel-aware page mapping policy, memory scheduling and thread scheduling, which map the data of different threads to different channels based on memory address space, thread priority and load balance and prioritize low memory request thread, row buffer hit access and older request. We evaluate CCPS on a variety of mixed single and multi-thread benchmarks and system configurations and compare them to four previously proposed state-of-the-art reducing interference policies. Experimental results show CCPS reduces both row buffer miss rate and switch overhead which improves system performance than modern memory management approaches while reducing energy consumption nearly 5.9%. Moreover, CCPS incurs much lower hardware overhead than current proposed policies.

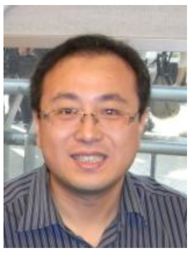
ACKNOWLEDGMENT

This work was supported by the National Science Foundation of China under grants (No. 61003077, No. 61100193). Zhejiang provincial Natural Science Foundation (No. LQ14F020011).

REFERENCES

[1] Y. Kim, M. Papamicheal and O. Mutlu. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In MICRO-43, 2010.
[2] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In HPCA-16, 2010.
[3] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In ISCA-35, 2008.
[4] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In USENIX Security, 2007.
[5] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In MICRO-40, 2007.
[6] S. Prashanth et al. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In Micro-44, 2011.
[7] R. Ausavarungnirun et al. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In ISCA, 2012.
[8] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In HPCA, 2010.
[9] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In MICRO, 2010.
[10] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In MICRO, 2007.
[11] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In ISCA, 2008.
[12] K. J. Nesbit et al. Fair queuing memory systems. In MICRO, 2006.
[13] M. K. Jeong et al. Balancing DRAM locality and parallelism in shared memory CMP systems. In HPCA, 2012.
[14] S. P. Muralidhara et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In MICRO, 2011.
[15] D. Kaseridis et al. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In MICRO, 2011.
[16] E. Ebrahimi et al. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In ASPLOS, 2010.
[17] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In ASPLOS, 2011.
[18] V. Cuppu, B. Jacob, B. Davis, T. Mudge. High-performance drams in workstation environments. IEEE Transactions on Computer 50 (11) (2001) 1133–1153.
[19] B. Davis. Modern dram architectures. Ph.D. thesis, Department of Computer Science and Engineering, University of Michigan, 2001.
[20] R. Crisp. Direct rambus technology: the new main memory standard. In: Micro-30: Proceedings of the 30rd annual ACM/IEEE International Symposium on Microarchitecture, 1997, pp. 18–28.
[21] S. Cho, and L. Jin. Managing Distributed, shared L2 Caches through OS-Level page Allocation. *In MICRO-39,* 2006.
[22] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In ASPLOS - XV, 2010.
[23] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy Efficient Computing in Virtualized Environments. In Proceedings of International Symposium on Low Power Electronics and Design. In ISLPED-2009.
[24] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In Micro-41, 2008.
[25] S. Prashanth et al. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In Micro-44, 2011.
[26] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro- Pages: Increasing DRAM Efficiency with Locality-Aware. In ASPLOS- 2010.
[27] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the many-core Era. In MICRO-44, 2011.
[28] Patel, Avadh, et al. MARSSx86: a full system simulator for x86 CPUs. In DAC, 2011.
[29] Z. Zhang, Z. Zhu, X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In MICRO'33: Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture, 2000, pp. 32–41.
[30] Kopytov, A. SysBench: a system performance benchmark. http://sysbench.sourceforge.net/index.html. 2004.
[31] Gangyong Jia, Xi Li, Jian Wan, Liang Shi, Chao Wang. Coordinate Page Allocation and Thread Group for Improving Main Memory Power Efficiency. In Hotpower'13.
[32] Gangyong Jia, Xi Li, Jian Wan, Chao Wang, Dong Dai, Congfeng Jiang. Coordinate Task and Memory Management for Improving Power Efficiency. In ICA3PP'13.
[33] Xi Li, Gangyong Jia, Chao Wang, Xuehai Zhou, Zongwei Zhu. A Scheduling of Periodically Active Rank of DRAM to Optimize Power Efficiency. First Workshop on Highly-Reliable Power-Efficient Embedded Design (HARSH) in conjunction with HPCA'13, 2013.
[34] Gangyong Jia, Xi Li, Chao Wang, Xuehai Zhou, Zongwei Zhu. Memory Affinity: Balancing Performance, Power, Thermal and Fairness for Multi-core Systems. IEEE Conference on Cluster Computing, Beijing, China, Sep.24-28, 2012.
[35] Xi Li, Gangyong Jia, Yun Chen, Zongwei Zhu, Xuehai Zhou. Share Memory Aware Scheduler: Balancing Performance and Fairness. ACM/IEEE the 22th Great Lakes Symposium on VLSI (GLSVLSI). 2012.
[1] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, Chengyong Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In PACT'12, 2012.

Gangyong Jia is currently an Assistant Professor of Department of Computer Science at Hangzhou Dianzi University, China. He received his Ph.D. degree in Department of Computer Science from University of Science and Technology of China, Hefei, China, in 2013. He has published over 20 papers in related international conferences and journals. He has served as a reviewer of Microprocessors and Microsystems. His current research interests are power management, operating system, cache optimization, memory management. He is a member of IEEE.



Guangjie Han received the Ph.D. degree from Northeastern University, Shenyang, China, in 2004. From 2004 to 2006, he was a Product Manager for the ZTE Company. In February 2008, he finished his work as a Postdoctoral Researcher with the Department of Computer Science, Chonnam National University, Gwangju, Korea. From October 2010 to 2011, he was a Visiting Research Scholar with Osaka University, Suita, Japan. He is currently a Professor with the Department of Information and Communication System, Hohai University, Nanjing, China. He is the author of over 130 papers published in related international conference proceedings and journals, and is the holder of 55 patents. His current research interests include sensor networks, computer communications, mobile cloud computing, and multimedia communication and security.

Dr. Han has served as a Cochair for more than 20 international conferences/workshops and as a Technical Program Committee member of more than 70 conferences. He has served on the Editorial Boards of up to 16 international journals, including the *International Journal of Ad Hoc and Ubiquitous Computing, Journal of Internet Technology* and *KSII* Transactions *on Internet and Information Systems.* He has served as a Reviewer of more than 50 journals. He received the 2014 Second International Conference on Computing, Management, Computing, Communications and IT Applications Conference and Telecommunications and International Conference on Communications and Networking in China Best Paper Awards. He is a member of the Association for Computing Machinery.

Aohan Li is currently pursuing M.S degree in signal and information processing at Heilongjiang University, China. She received her B.S degree in electronic information engineering from Heilongjiang University, China, in 2012. Her current research interests are wireless sensor networks and cognitive radio networks

Jaime Lloret received his M.Sc. in Physics in 1997, his M.Sc. in electronic Engineering in 2003 and his Ph.D. in telecommunication engineering (Dr. Ing.) in 2006. He is a Cisco Certified Network Professional Instructor. He worked as a network designer and administrator in several enterprises. He is currently Associate Professor in the Polytechnic University of Valencia. He is the head of the research group "communications and remote sensing" of the Integrated Management Coastal Research Institute and he is the head of the "Active and collaborative techniques and use of technologic resources in the education (EITACURTE)" Innovation Group. He is the director of the University Expert Certificate “Redes y Comunicaciones de Ordenadores”, the University Expert Certificate “Tecnologías Web y Comercio Electrónico”, and the University Master "Digital Post Production". He is currently Chair of the Internet Technical Committee (IEEE Communications Society and Internet society). He has authored 12 books and has more than 240 research papers published in national and international conferences, international journals (more than 80 with ISI Thomson Impact Factor). He has been the co-editor of 15 conference proceedings and guest editor of several international books and journals. He is editor-in-chief of the international journal "Networks Protocols and Algorithms", IARIA Journals Board Chair (8 Journals) and he is associate editor of several international journals. He has been involved in more than 200 Program committees of international conferences and in many organization and steering committees. He led many national and international projects. He is currently the chair of the Working Group of the Standard IEEE 1907.1. He has been general chair (or co-chair) of 19 International workshops and conferences (chairman of SENSORCOMM 2007, UBICOMM 2008, ICNS 2009, ICWMC 2010, eKNOW 2012, SERVICE COMPUTATION 2013, COGNITIVE 2013, and ADAPTIVE 2013, and co-chairman of ICAS 2009, INTERNET 2010, MARSS 2011, IEEE MASS 2011, SCPA 2011, ICDS 2012, 2nd IEEE SCPA 2012, GreeNets 2012, 3rd IEEE SCPA 2013, SSPA 2013 and local chair of MIC-WCMC 2013). He is co-chairman AdHocNow 2014, MARSS 2014, SSPA 2014 and GreeNets 2014, and local chair IEEE Sensors 2014. He is IEEE Senior and IARIA Fellow.