A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems

Lei Liu[1,2], Zehan Cui[§1,2], Mingjie Xing[1], Yungang Bao[1], Mingyu Chen[1], Chengyong Wu[1]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]Graduate School of Chinese Academy of Sciences
Beijing, China
{liulei2010, cuizehan, baoyg, cmy, cwu}@ict.ac.cn

ABSTRACT
Main memory system is a shared resource in modern multicore machines, resulting in serious interference, which causes performance degradation in terms of throughput slowdown and unfairness. Numerous new memory scheduling algorithms have been proposed to address the interference problem. However, these algorithms usually employ complex scheduling logic and need hardware modification to memory controllers, as a result, industrial venders seem to have some hesitation in adopting them.

This paper presents a practical software approach to effectively eliminate the interference without hardware modification. The key idea is to modify the OS memory management subsystem to adopt a page-coloring based bank-level partition mechanism (BPM), which allocates specific DRAM banks to specific cores (threads). By using BPM, memory controllers can passively schedule memory requests in a core-cluster (or thread-cluster) way.

We implement BPM in Linux 2.6.32.15 kernel and evaluate BPM on 4-core and 8-core real machines by running randomly generated 20 multi-programmed workloads (each contains 4/8 benchmarks) and multi-threaded benchmark. Experimental results show that BPM can improve the overall system throughput by 4.7% on average (up to 8.6%), and reduce the maximum slowdown by 4.5% on average (up to 15.8%). Moreover, BPM also saves 5.2% of the energy consumption of memory system.

Categories and Subject Descriptors
B.3.1 [Semiconductor Memories]: Dynamic memory (DRAM), Static memory (SRAM); C.4 [Computer Systems Organization]: Performance of Systems-Design studies; D.4.1[Operating System]: Process Management-Scheduling; D.4.2[Operating System]: Storage Management-Main Memory;

General Terms
Management, Performance, Design

Keywords
Main Memory, Multicore, Interference, Data Allocation, Memory Scheduling, Bank, Partition

1. INTRODUCTION
On multicore platforms, DRAM memory system shared by all cores usually suffers from the memory contention and interference problem, which can cause serious performance degradation and unfairness of the overall system. Specifically, modern multicore machines consist of many components, such as processing cores, prefetchers and DMA engines, which can generate memory requests with different characteristics and priorities. For example, different cores can generate memory-intensive and non-intensive requests simultaneously; prefetchers' requests are of low priority and DMA engines' requests are sequential. If memory controllers are unable to distinguish these different requests, interference inevitably occurs.

A number of recently proposed scheduling algorithms [18, 19, 25, 26, 27, 30], leveraging the abovementioned different characteristics information, have been demonstrated to be able to effectively reduce the memory contention and interference. For instance, TCM [18], which classifies threads into memory-intensive group and non-intensive group and uses different policies for the two groups, is shown to exhibit both performance and QoS improvements for overall system.

Although some memory scheduling algorithms are claimed to be easily integrated into memory controllers [18, 19, 22, 25, 29, 30], they usually introduce complex hardware logic and require extra storage in memory controllers to store per core (or per thread) information, which can be an obstacle to the scalability of on-chip core number. Therefore, industrial venders seem to have some hesitation in adopting aggressive scheduling algorithms.

In this paper, we propose a software approach to effectively eliminate the memory contention and interference problem without any hardware modification to memory controllers. Our approach is inspired by two observations that 1) DRAM bank-level conflict is a major reason of the memory contention and interference problem and 2) the demanded bank amount for a thread is limited (typically less than 16 banks).

Intuitively, inter-thread bank-level conflicts can be eliminated by exclusively mapping a thread's data to specific banks. We adopt this basic idea and modify the physical pages allocation of OS memory management subsystem. Therefore the physical pages in specific banks can be exclusively mapped to a specific thread (or core). For example, if OS maps thread 1's data to 4 banks (e.g., bank 0~3) and maps thread 2's data to 12 banks (e.g., bank 4~15), memory controllers will deliver all memory requests of thread 2 to only bank 4~15 and will not affect thread 1 whose memory requests are delivered to only bank 0~3. By doing this mapping,

[§] Co-first author

the bank-level inter-thread memory contention and interference are eliminated. We call this approach Bank-level Partition Mechanism (BPM), which is an extension of page-coloring.

Although the OS page-coloring technique is not new, the distinct advantage of BPM is that it exposes both cache and memory bank-level information to OS. We implement BPM in Linux 2.6.32.15 kernel and run multi-programmed/threaded workloads on 4-core and 8-core real machines to evaluate BPM. Experimental results show that BPM can improve the overall system throughput by 4.7% (up to 8.6%), reduce the maximum slowdown by 4.5% (up to 15.8%) on average. Besides, we find that BPM can also save 5.2% of the energy consumption of memory system.

In summary, we make the following contributions:

(1) We observe that the demanded bank amount for a thread is limited, which means that a thread's performance would not be improved even if more banks are assigned to it. Empirical studies show that 8 ~ 16 banks are enough for one thread.

(2) We propose a new practical page-coloring based Bank-level Partition Mechanism (BPM) to effectively eliminate the memory contention and interference problem without any hardware modification to memory controllers.

(3) We implement BPM in Linux 2.6.32.15 kernel for evaluation. Experimental results show that BPM can improve the overall system throughput by 4.7% (up to 8.6%), reduce the maximum slowdown by 4.5% (up to 15.8%) and save 5.2% of the energy consumption of memory system.

(4) We find that the product of the sum of all threads' memory bandwidth and the standard deviation of each thread's row-buffer locality (i.e., Sum(BW)*Stdev(RBL)) is a good indicator for predicting BPM's performance improvement. This indicator probably can be used to evaluate system performance in a shared environment.

The rest of this paper is organized as follow. In Section 2, we introduce the background of DRAM system, and our motivation on this work. In Section 3, we present our BPM in detail. The methodology and metrics are discussed in Section 4. We evaluate BPM in Section 5. In Section 6, we discuss related work and conclude in Section 7.

2. Background and Motivation

2.1 DRAM System

We briefly describe DRAM memory systems and OS memory management mechanism. Our description is based on DDR3 SDRAM systems, and it is generally applicable to some other DRAM types that employ bank/page-mode.

DRAM Organization: Modern memory system consists of multiple independent banks, each of which contains at least one 2-dimensional storage array. Banks can work in a parallel way, hence, memory requests to different banks can be served concurrently [25, 27, 33].

However, since each bank has only one row buffer, only one row is accessible in a bank at any time. Typically, DDR3 chip's row buffer is 1KB~2KB size. Once a request to a bank arrives，if the required row is in the row-buffer, MC[1] can immediately issue a read/write command. Otherwise, a row buffer conflict occurs so that the MC needs to firstly issue a precharge command to write back the content in the row-buffer and then issue an active command to fetch the required row into row buffer before issuing a read/write command. Obviously, row buffer conflict results in longer memory request latency (may more than two times) than the row buffer hit case. Requests from different threads seldom go to same rows, so the row buffer conflict occurs more frequently on multicore platform than single thread computing environment.

Bank-Level Parallelism (BLP) and Bank Sharing: BLP means that multiple banks can serve memory requests concurrently and independently because they are physically independent. BLP can often help make full use of banks and improve memory bandwidth. Hence, memory system usually employs a bank-interleaved address mapping schema to take the advantages of BLP [17, 25, 27, 33, 38], which is meant to share all banks to all cores in a multicore system. Nevertheless, such bank-sharing schema brings interference among threads because one bank may receive memory requests from different cores, which probably have different memory access characteristics. Therefore, the bank conflicts between cores become more and more frequent as the core number increasing.

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 2^{order} 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 (2^{order} physical pages) for that program. Usually the first block of a free list is selected but the corresponding physical pages are undetermined [8].

2.2 Multicore-Posed Challenges and Current Solutions

Multicore architecture poses two major challenges on memory system: 1) Interference. Usually a single thread's memory requests have good locality and can exhibit good row buffer hit rate. But the locality is significantly reduced in a multicore machine wherein multiple threads can issue memory requests to the memory system. Row buffer hit rate also decreases sharply, leading to poor overall system performance. For example, Udipi et al. [37] illustrate that the row buffer hit rate decreases significantly from 1 core (over 60%) to 16 cores (35%). 2) Unfairness. Conventional memory controller's scheduling algorithms (e.g., FR-FCFS [15, 33]) favor processing those memory requests with good row buffer locality in order to improve row buffer hit rate. Therefore, memory intensive applications, which have better locality can obtain higher priority than memory non-intensive applications. For instance, Mutlu et al. [27] illustrate that the slowdowns for some memory non-intensive applications can increase by 7.74X for 4-core system and even 11.35X for 8-core system whereas the memory intensive application experiences the slowdowns of only 1.04X and 1.09X respectively.

One major reason of the two problems is that memory controllers are unable to identify one thread's distinct access pattern from other threads' pattern in a multiple-threaded mixed memory requests stream. To address these challenges, numerous new memory scheduling algorithms have been proposed to effectively reduce the memory contention and interference. For instance, the

[1] MC is short for Memory Controller.

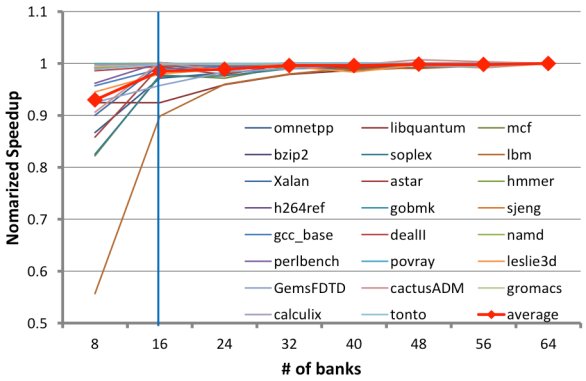

Figure 1. The correlation between application performance and bank amount. The blue line is the "watershed", which indicates all benchmarks can achieve 90% of its maximum performance with only 16 banks.

state-of-the-art scheduling algorithm, Thread Clustering Memory Scheduling (TCM), which classifies threads into memory-intensive group and non-intensive group and uses different policies for the two groups, exhibits both performance and QoS improvements for overall system [18]. Another important reason is DRAM bank-level conflict. As mentioned above, because all banks are shared by all cores, one bank can receive memory requests from different cores with different memory access characteristics. Unfortunately, even the state-of-the-art scheduling algorithms are unable to fully eliminate the interference problem unless banks are not shared among cores. Some partition approaches are proposed in order to eliminate interference at cache-level [23] and channel level [30], but the contention in terms of bank-level still remains. Recent research proposes a bank level partition among multi-programmed workloads [28, 32]. Yet, their work is not deployed in real hardware. Besides, they do not take into account the relationship between cache and bank partition.

Although those solutions are claimed to be easily integrated into memory controllers, they usually introduce complex hardware logic and require extra storage in memory controllers to store per core (or per thread) information [18, 19, 25-27, 30]. For example, TCM [18] requires additional 4 Kbits storage in a memory controller to support 24 cores, additional logic to rank threads and a central meta-controller to gather global information of multiple memory controllers. Besides, because the thread number in a system is usually much more than the core number, there probably exist frequent context-switches, which require a sophisticated hardware thread-behavior monitoring mechanism. Therefore, industrial venders seem to have some hesitation in adopting aggressive scheduling algorithms. Then, a question is raised: ***Can we use a software approach to achieve the similar effect as these hardware solutions do?***

2.3 Our Insights

Intuitively, the inter-thread bank-level conflicts can be fully eliminated by exclusively mapping a thread's data to specific banks. But doing so will reduce the available bank amount for one thread, thus it is important to know how available bank amount influences the thread's performance.

We perform experiments on an Intel i7-860 machine with 64 banks (125MB per bank) to analyze the correlation between bank amount and application's performance (details of experimental setup are in Section 4). For one application, we fix the available banks from 8 banks to 64 banks and see its performance changes. Figure 1 illustrates the results of 23 benchmarks from SPEC2006 [2]. Surprisingly, we find that **the necessary amount of banks one program requires is limited**, for example 16 banks in our experiments. Providing more banks (e.g., all 64 banks) than the necessary amount (e.g., 16 banks) to a program will not yield significant performance improvement.

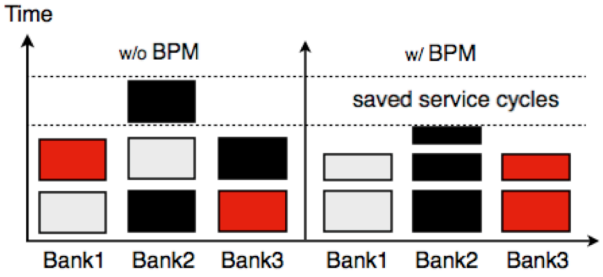

Figure 2. The Comparison between interleaved address mapping and BPM. Different colors mean that the requests are from different threads.

In fact, usually a single core is unable to generate enough concurrent memory requests due to the combination of many factors such as memory dependency, high cache hit rate and limited number of MSHRs. Nevertheless, most modern systems always interleave memory requests across banks in order to take the advantages of bank-level parallelism, thus one program can access all banks, largely exceeding its necessary bank amount. As a result, those programs, which are sharing all banks only suffer from memory interference rather than obtain any performance gain.

This insight inspires us that it is feasible to partition banks into several groups and designate specific bank groups to specific threads so as to eliminate inter-thread bank conflicts. Based on the key insight, we propose a software approach, OS page-coloring based bank-level partition mechanism (BPM), to effectively eliminate the memory contention and interference problem without any hardware modification to memory controllers.

3. Bank-Level Partition Mechanism (BPM)

3.1 Overview of BPM

The key idea of BPM is that OS memory management system uses a page-coloring mechanism to partition banks into several groups and maps each thread (process) to a specific bank group. Consequently, memory controllers can passively schedule memory requests in a thread-cluster (or core-cluster) way, i.e., scheduling one thread's memory requests to pre-specific banks.

3.1.1 Advantages

OS page-coloring technique is a well-known technique for cache partition [23]. But the distinct advantage of BPM allows OS to partition memory space based on the underlying memory bank-level information. Because memory controllers also schedule memory requests at bank-level, the OS partition effect can be indirectly propagated to memory controllers.

Figure 2 illustrates an example. The smaller blocks mean row buffer hits, which result in shorter latencies. Assume there are three threads (different colors) issuing memory requests to DRAM banks. For conventional page allocation without BPM (the left), these requests are delivered to all banks, resulting in many row buffer conflicts. With BPM (the right), one thread's memory

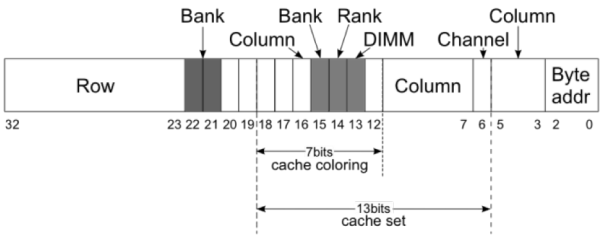

Figure 3. Address Mapping policy of our platform. The Bank bits are divided into two separate parts. One is overlapped with Cache set bits; another is independent.

requests are mapped to its specific bank so that row buffer conflicts are eliminated between threads (processes).

Specifically, BPM brings the following advantages:

(1) BPM is an entirely software approach, so that it is easier to be implemented in modern systems. Since software is more flexible than hardware, more partition policies can be explored to achieve better performance.

(2) It is easier for OS to monitor threads' behavior than hardware, which requires additional storage and logic. Contemporary processors provide powerful performance counters to monitor system behavior, and the state-of-the-art tools such as LiMit [12] are already able to precisely monitor per-thread behavior (e.g., cache miss rate, memory bandwidth etc.) with negligible overhead. BPM can easily leverage this kind of information for exploring various partition policies.

(3) Moreover, BPM can facilitate improving other OS functionalities. For example, OS's process management module can utilize BPM partition information to guide process scheduling. BPM can also be implemented in Virtual Machine Monitor (VMM) to partition memory space for virtual machines in order to improve VM isolation effect.

3.1.2 Principle

A physical address contains several common bits, which denote both OS page index and bank index, so these bits are referred to as ***bank color bits***. For instance, if a physical address has 4 bank color bits, then there are 2^4=16 bank colors. ***Partition*** means that BPM exclusively assigns banks with the same color to a thread such that those banks can be accessed only by the specific thread. Note that a thread can possess multiple bank colors. Here is a concrete example. Our experimental machine has 4GB/8GB DDR3 main memory with 32/64 banks. Usually, the OS page size is 4KB, so the low 12 bits are page offset and the OS physical page index bits are bit 12~32. Figure 3 illustrates the 5 bank color bits of our platform, i.e., bit 13~15 and bit 21~22. Therefore, there are 32 colors, and each color represents 1 bank for 4GB memory or 2 banks for 8GB memory.

Each OS page index (bit 12~32) also contains 5 bank color bits which designate one color, thus each physical page belongs to one bank color. When a thread applies for one page, BPM first checks which bank colors are possessed by the thread, then picks a bank color, and finally allocates a physical page with the color for the thread.

3.2 Cache Partition

Figure 3 illustrates that bit 13~15 are used for both bank coloring and cache coloring. Therefore, the last level cache with physical index is also partitioned into 2^3 = 8 groups. As a result, we could gain the benefits from both cache partition and bank partition. On the other hand, since there are two extra bits (21~22) for bank coloring, BLP can also be orthogonal to cache partition.

Algorithm1: Discover Bank Bits
Input: The address bits; **Output:** BANK{}, which contains all bank bits.

```
BEGIN
/* STEP 1: Detect row address bits */
/*Based on the idea that row miss causes larger latency*/
1. FOR each bit x IN address bits
2. DO
3.     Generate 2 memory requests, one's x bit is 0, and
       another's x bit is 1;
4.     Access the two addresses (uncached) in turn and record the
       latency (repeat at least 10000000 times);
5. END FOR
6. The latency will be easily clustered into two groups;
7. Put the group with higher latency into Row{}  // row miss
8. The left parts are put into Remain{}  // row hit
9. Call Step 2

/*STEP 2: Detect column address bits*/
/*Based on the idea that bank parallelism outperforms row miss */
1. FOR each bit y IN Remain{}
2. DO
3.     Choose an x from Row{};
4.     Generate 2 requests, one's x and y bit are both 0, and
       another x and y bit are both 1;
5.     Access the two address (uncached) in turn and record
       the latency (repeat at least 10000000 times);
6. END FOR
7. The latency will be easily clustered into two groups;
8. Put the group with higher latency into Column{} //row miss
9. The left parts are put into Remain{}  //mapped to different banks
10. IF there is no XOR policy THEN
11.    Put Remain{} into BANK{}
12.    Output BANK{}
13. ELSE
14.    Call STEP 3
15. ENDIF

/*STEP 3: Detect XOR Policy (Optional)*/
/*Many MCs employ XOR to improve performance*/
1. FOR each pair <u,v>, u,v IN Remain{}
2. DO
3.     Generate 2 memory requests, one's u and v bit are both 0, and
       another's u and v bit are both 1; the other bits are identical;
4.     Access the two address (uncached) in turn and record the latency
       (repeat at least 10000000 times);
5. END FOR
6. The latency will be easily clustered into one or two groups;
7. IF there is only one groups THEN
8.    XOR is not employed
9. ELSE
10.    Put the group of <u,v> pairs with higher latency into BANK{},
       Delete all u, v in that group from Remain{}  //row miss
11. END IF
12. Put Remain{} into BANK{}
13. Output BANK{}
END
```

3.3 Discover bank bits by software method

In order to employ BPM, we need to obtain the memory address mapping information so as to extract the bank bits. We can look up address mapping in vendors' manuals. However, address

mapping in a memory controller is not fixed and can be configured by BIOS at boot time. A memory controller always supports various address mapping policies. Some hardware tools can be used (such as HMTT [5]) to figure out address mapping policy, but it is impossible to deploy this approach for massive normal machines. In order to solve this problem, we propose a practical software method (**Algorithm1**) to discover the address mapping policy as well as bank bits on any machines.

The approach is based on two observations that: 1) the latency of row buffer misses is much longer than the latency of row buffer hits (refer to STEP 1); 2) concurrent accesses to two different banks (BLP) still result in lower latency than row buffer conflict within a bank (refer to STEP 2). We verify our algorithm by a hybrid memory trace tool (HMTT) [5], which is able to monitor memory signals over memory buses. The verification results show that our algorithm works well on various platforms. Thus, this algorithm can be embedded into OS boot phase to collect the address mapping information, which can be used for BPM setup.

3.4 Implementation

We implement BPM in Linux kernel 2.6.32.15. The kernel uses a buddy system to manage the free physical pages, which are organized as different orders (0~11) of free lists (refer to Section 2.1). We modify the original free list organization into a hierarchy way: for each order of free page list, we re-organize the free pages to form 32 colored free lists according to the 5 bank bits. Each process has its own colors (i.e., a group of banks). When a page fault occurs, the OS kernel will search a colored free list and allocate a page for the process. This is transparent to applications so that programmers do not need to modify programs.

For multi-programmed workloads, bank colors are assigned to each program. For multi-threaded workloads, we enhance OS kernel with new APIs for programmers to perceive the underlying bank colors so that programmers can map threads' data into different colors on their demand.

The overhead of the searching color operation is negligible. Experimental results show that the average kernel time accounts for only 0.3% for all workloads.

4. Methodology and Metrics

4.1 Hardware and software platform

We conduct our experiments on a machine, which has 4 cores, 2.8GHz Intel Core i7-860 processor with a shared 8MB 16-way associative last level cache and 8GB DDR3 main memory. The processor incorporates Hyper-Threading technology, so we can run 8 threads concurrently.

We use CentOS Linux 5.4 with kernel 2.6.32.15. The memory system is 8GB with 64 banks (each bank is 125MB). There are 5 bank bits, so the memory is divided into 32 colors, and each color represents 2 banks bundled together across dual-channel. In our experiments, colors are statically assigned to processes/threads at their starting time. Modern multi-thread servers often have enough memory for their running threads [11]. Therefore, we also disable OS swap. Moreover, for most experimental workloads, the memory capacity of 0.5~1GB (4~8 banks) is enough.

We use Perfmon2 [1] and its corresponding libpfm library to access the performance counters to gather architectural information, such as memory bandwidth and last level cache miss rate. For memory system, because DIMMs are directly plugged into motherboard and it is difficult to measure memory power consumption, we adopt an in-house hardware tool [10], which consists of a wrapper card for each DIMM. The wrapper card is plugged into motherboard's DIMM slot and memory power consumption can be measured precisely via the sensors embedded into the wrapper card. It should be noted that the wrapper card does not affect the memory access at all.

4.2 Benchmarks

We use the SPEC CPU2006 benchmarks for evaluation. We compile each benchmark using gcc 4.1.2 with -O3 optimizations. From these benchmarks, we randomly generate multi-programmed workloads each of which contains 4/8 applications (due to space limitation, we omit a table depicting all workloads' characteristics). We employ a multi-threaded benchmark streamcluster from PARSEC 2.1 [7]. We use the notion of "Miss Per Kilo-Instruction (MPKI) > 1" to define memory-intensive applications. We use ref input size for the SPEC benchmarks and the native for PARSEC.

In our experiments, for each benchmark in 4-programmed workloads, each program has 8 colors (i.e., 16 banks/2GB); for 8-programmed workloads, each program has 4 colors (i.e., 8 banks/1GB).

4.3 Metrics

We use ***Weighted Speedup*** [18] (WS) to measure system throughput and use ***Maximum Slowdown*** (MS) [18] for fairness. We also report ***Improvement*** compared with the normal environment without BPM.

$$\text{Weighted Speedup(WS)} = \sum \frac{RUNTIME_{ALONE}}{RUNTIME_{SHARED}}$$

$$\text{Maximum Slowdown(MS)} = Max\left\{\frac{RUNTIME_{SHARED}}{RUNTIME_{ALONE}}\right\}$$

$$\text{Improvement\_ws} = \frac{WS_{BPM} - WS_{No-BPM}}{WS_{No-BPM}}$$

$$\text{Improvement\_fairness} = \frac{MS_{No-BPM} - MS_{BPM}}{MS_{No-BPM}}$$

5. Results

5.1 Overall system performance

In this experiment, we use static uniform partition policy. Figure 4 combines the improvement of both the fairness and system throughput across all 20 workloads. According to Figure 4, BPM can improve weighted speedup by 4.7% on average, and reduce the maximum slowdown (unfairness) by 4.5% on average.

The improvement of Weighted Speedup is steady for all workloads and no workload's performance degrades. We firstly study these 4-programmed workloads (workload 11~20 in Figure 4). BPM can achieve a maximum weighted speedup by up to 5.9% for workload_11, which consists of 462.libquantum (MPKI = 50, RBL[2] = 99.22%), 403.gcc (MPKI = 0.4), 447.dealII (MPKI = 0.5), 444.namd (MPKI = 0.3). For the four benchmarks in workload_11, 462.libquantum is obviously a memory intensive application, specifically a stream-like program (RBL = 99.22%), while the other three are memory non-intensive. In normal configuration without BPM, where memory accesses are interleaved across banks, 462.libquantum causes substantial row buffer conflicts due to its stream characteristics, affecting the other three applications'

[2] RBL (Row Buffer Locality) is equivalent to row buffer hit rate.

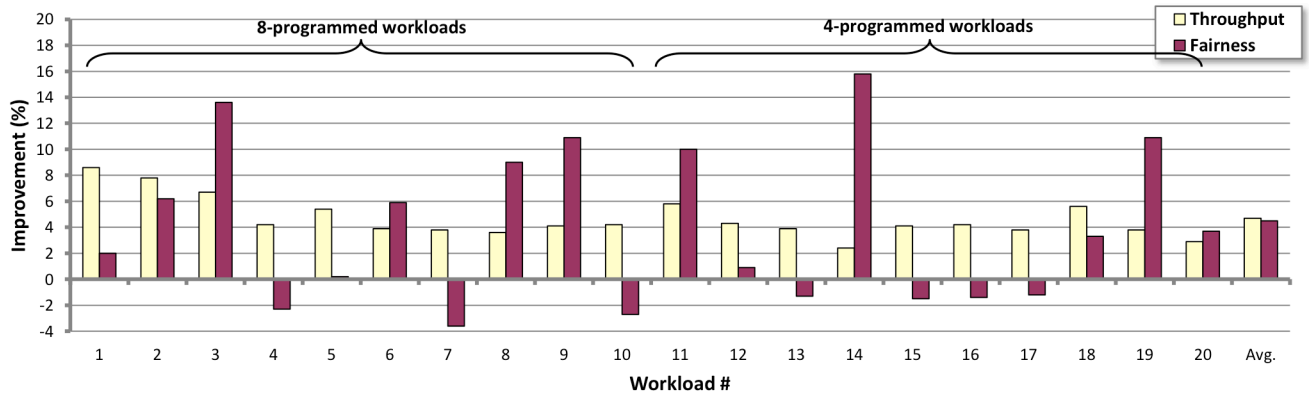

Figure 4: The Overall system performance of BPM across 20 workloads. The x-axis denotes the workload number, and the y-axis shows both the improvement of fairness and system throughput. Baseline is the conventional Linux kernel without BPM.

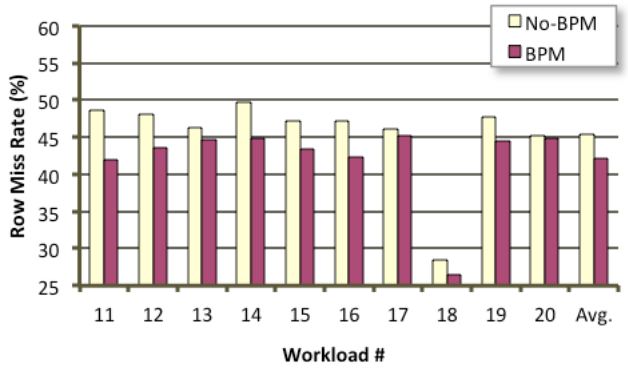

Figure 5. Row buffer miss rate across 10 workloads

row buffer hit rate. Figure 5 shows the row buffer miss rate is almost 50% for workload_11. However, when using BPM, each application has 8 dedicated bank-colors (16 banks) and their memory requests can only be delivered to those banks. According to Figure 1, the performance for the four benchmarks decrease very slightly from 64 banks to 16 banks (92% for 462.libquantum and 99% for others). Since the memory interference is fully eliminated (the overall row buffer miss rate reduces by about 10% in Figure 5), the overall system performance is improved. In 8-programmed workloads, we find the workload_1 can achieve the maximum improvement by 8.6%. Wokload_1 also includes 462.libquantum, and it is a heavy memory-intensive workload (all programs' MPKI > 2). The more intensive the threads are, the more interference on banks there would be. Thus, BPM would exhibit better improvements by totally eliminating the bank-level interference in such an environment.

For the Fairness metric, workload_11 exhibits 10% improvement and workload_14 improves by even 15.8%. We find that workload_14 comprises of 462.libquantum, 456.hmmer (MPKI = 5.7), 403.gcc, 444.namd. The only difference between workload_11 and workload_14 is replacing 447.dealII (MPKI = 0.5) with 456.hmmer. Obviously, when workload_14 runs on normal environment, there will be more memory interference because 456.hmmer issues more memory requests than 447.dealII. As mentioned before, current memory controllers favor processing those memory requests that have good row buffer locality, thus memory intensive applications, which are more likely to have good locality, can obtain higher priority than

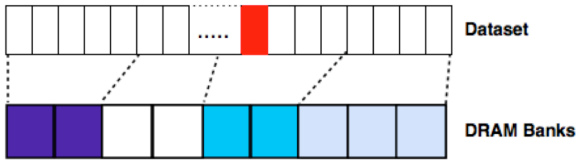

Figure 6: Thread Level Coloring. The above array is mapped to different bank colors. The colored rectangle in dataset represents shared data.

memory non-intensive applications. The more memory intensive applications are there, the less opportunities do non-intensive applications get their memory requests been served. Therefore, the unfairness problems always occur in modern multicore systems. Obviously, BPM can effectively eliminate this unfairness.

It should be noted that there are several workloads exhibiting worse fairness, i.e., workload_13, workload_15, workload_16 and workload_17. We find that those workloads have a common benchmark 429.mcf (MPKI = 99.8), which is an extreme memory intensive application. But the interesting thing is that when reducing bank amount from 64 to 16, unlike 462.libquantum whose performance decreases by 8%, the performance of 429.mcf almost does not decrease (only 2%). Therefore, BPM can improve 429.mcf's performance even more than other non-intensive applications, leading to a slight unfairness. For 8-programmed workloads, the average improvement of system throughput is 5.3% (up to 8.6%), which are slightly better than 4-programmed workloads (4.1%). This implies that BPM is able to exhibit better performance improvements in a worse interference scenario.

5.2 Multi-threaded workload

In practical, many servers are used to run multi-threaded workloads. We use streamcluster of PARSEC [7] to evaluate BPM. Its coloring scheme is nearly the same as that of multi-programmed workloads. We use Native dataset (200000 *5 points) as input in our experiment. For a stream of these input points, they are divided into N chunks according to core number, the first N-1 chunks contains the same amount of points, while the N*th* chunk collects the rest points. Because streamcluster itself is a typical data parallelism computing multi-threaded program, we could partition the dataset in a straightforward way (Figure 6). We get performance gains by 1.7% and 2.3% on 4/8-thread separately. The improvement is less than that of multi-programmed workloads because there is too much shared data among threads

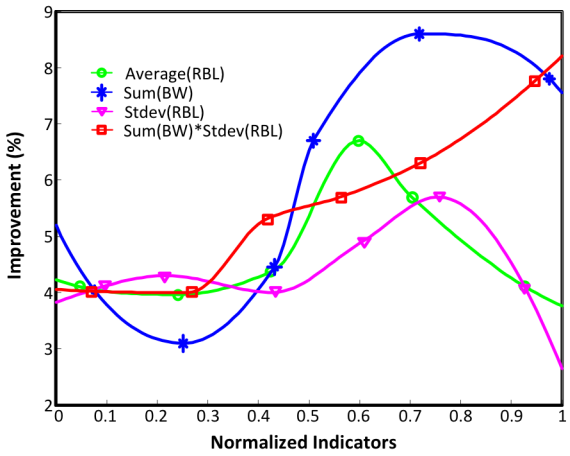

Figure 7. The correlation of BPM improvements and four indicators.

(the colored rectangle in dataset in Figure 6). In our straightforward partition, the shared data belongs to blue banks. When other threads access the shared data, inter-thread bank conflicts occur. There are two ways to improve multi-threaded applications: 1) designing a better partition policy and 2) leveraging a dynamic color adjustment mechanism. We will do further study on these issues in the future work.

5.3 What affects the BPM?

In this subsection, we study the correlation between workloads' characteristics and performance improvements. We investigate four indicators derived from memory bandwidth (BW) and row buffer locality (RBL) of individual benchmarks, which are collected by perfmon [1] when the benchmarks are running alone. Given a workload, we calculate the following four indicators: 1) The indicator Average(RBL) is the weighted average of the 4/8 programs' RBL, where BW is the weight. This indicates the overall row buffer locality of the workload. 2) The indicator Sum(BW) is the sum of the 4/8 programs' BW. This indicates the intensity of the workload. 3) The indicator Stdev(RBL) is the weighted standard deviation of the 4/8 programs' RBL. This indicates the difference of locality among programs. 4) Sum(BW)*Stdev(RBL) is the combination of the two indicator stated before.

Figure 7 illustrates four curves, which represent the correlation between the improvements of BPM and the four indicators respectively. To fit them into one figure, we normalized the value of all the four indicators into range (0, 1). Besides, there are 6 points on each curve; each point represents one workload or the average of multi workloads, which have close indicator values. According to the figure, none of the indicators can match the improvements of BPM perfectly except ***Sum(BW)*Stdev(RBL)*** – as the indicator increases, the improvements of BPM also increase steadily. Actually, we can use ***Sum(BW)*Stdev(RBL)*** to indicate the interference degree of a multi-programmed workload. The more interference is there, the more improvement can be achieved by BPM.

5.4 Page-Policy and Power

There are two page policies in memory system, open-page policy and close-page policy. Usually, open-page policy has better performance than close-page policy. But recent studies show

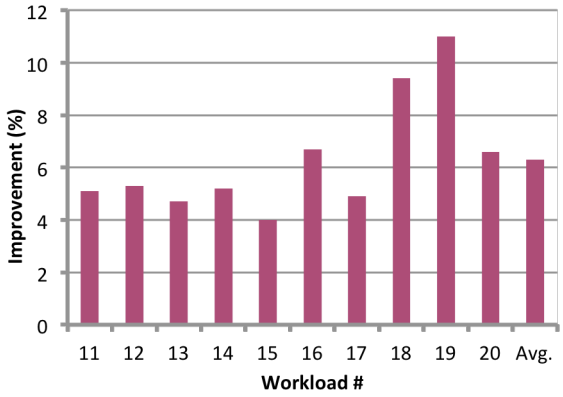

Figure 8. Improvement of weighted speedup of Open-Page policy with BPM over Close-Page policy.

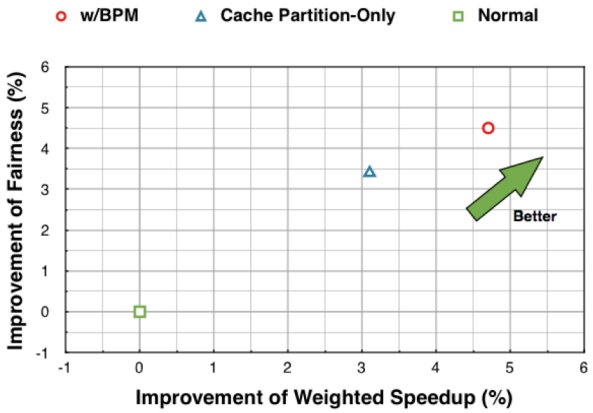

Figure 9. BPM vs. Cache-Partition-Only across 20 workloads on average (Toward right top is better)

that the row-buffer locality in multicore systems is sharply decreased to a lower level [35, 37]. Therefore, some server machines have to compromise to adopt close-page policy. Our experiments show that BPM can revive open-page policy in multicore systems. In our experiments, we change the page policies of the experimental machine and measure the system throughput improvement. Figure 8 shows that open-page with BPM outperforms close-page by 6.3% in terms of weighted speedup. This implies that if we partition banks appropriately, open-page policy can still be employed in heavily threads computing environment.

The active operation is the most power-consuming operation in the DRAM system [3, 37], because it has to move an entire row from array to a row buffer. BPM can lower the power consumption of DRAM because of the reduced row buffer conflict miss rate (as illustrated in Figure 5). As mentioned in 4.1, we measure the power consumption by real hardware, so we can get the real value of power savings on memory system. Our experimental results show that BPM with open-page policy can save up to 5.2% of memory power consumption, better than the configurations without BPM.

5.5 Comparison between Bank and Cache

As mentioned in Section 3.2, there are 3 common bits for both bank partition and cache partition and 2 extra bits for bank-partition only. This mechanism allows us to evaluate the effect of

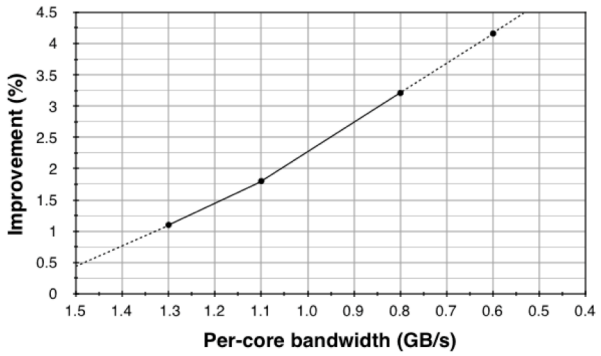

Figure 10. The correlation of BPM improvements and per-core Bandwidth

different partition schemes on real machines. Figure 9 illustrates the comparison between cache-partition-only and BPM, which embraces cache partition. When only adopting cache partition with 8 colors (3 bits), both system throughput and fairness are improved slightly (3.1% and 3.4%). Furthermore, when the two extra bits are used to form 32 bank colors (still 8 cache colors), the performance is further improved (4.7% and 4.5%), which proves that BPM is orthogonal to cache partition.

5.6 The Correlation between BPM improvements and Per-core bandwidth

Off-chip memory bandwidth is limited by the pin count of micro-processor chip and thereby is considered as the major bottleneck of the scalability of on-chip core number [6, 34]. Since the core number is still increasing, memory bandwidth per core is decreasing, which causes more and more serious interference. In order to evaluate the influence of different per-core bandwidth on our BPM approach, we emulate different bandwidth scenarios by varying memory frequency from 1333 to 800 MHz so that the per-core bandwidth decreases form 1.3GB/s to 0.8GB/s.

Figure 10 illustrates that the correlation of performance improvements and per-core bandwidth is negative: BPM performs better when per-core bandwidth is less. In fact, our previous experiments also provide evidences from another perspective. For example, when we enable Hyper-Threading on the experimental machine, the per-thread memory bandwidth halves, but the overall system throughput still improve from 4.1% (4-programmed) to 5.3% (8-programmed). Therefore, BPM is a promising approach for future many-core architecture that arguably has even less per-core bandwidth.

6. Related work

There are a number of related studies.

Thread Scheduling. Scheduling algorithms DI and DIO proposed in [39] aimed to distribute threads to get an even distribution of miss rate among multiple caches, which avoid severe contention on cache, memory controller, memory bus and prefetching hardware. Similar mechanisms are also proposed in [11, 21]. This method can alleviate contention, but cannot eliminate the bank interference among threads.

Cache Partition. Either hardware based cache partition [13, 14, 31, 36] or software page coloring based cache partition [4, 8, 23, 24] are employed to partition shared cache to concurrent running threads, which can eliminate the interference between multi-threads and hence reduce conflict at cache level. However, other resources such as MC, memory bus, and DRAM are also shared and confronted with contention and interference.

Channel Partition. Data of different threads are mapped into different channels according to their memory access behavior in [30], 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 [30], 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 non-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 [35], 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 [20], the content in row-buffer will be precharged after 4 times access, which target at the reduction of row-buffer conflicts.

Comparison with BPM. To the best of our knowledge, this is the first work that implements and evaluates bank level partition in reality. Our work is quite different from the previous work [28, 32]. First, we do not need to modify hardware, and our mechanism can be deployed on any Linux platforms. Second, our page-coloring mechanism takes both cache and bank bits information into account, which can eliminate both cache-level and bank-level conflicts, and ensure the fairness between threads.

7. Conclusion

We present Bank-level Partition Mechanism (BPM), a new approach to eliminate the interference between threads and improve the overall system performance. BPM achieves this goal by assign different group of banks to different threads to eliminate inter-thread bank-level interference. This leads to the reduction of row buffer misses as well as the energy consumption of memory system. To the best of our knowledge, BPM is the first bank level partition that is implemented on real machines and can be used in real multicore computing environment.

Our experimental evaluations show that BPM can improve system throughput and reduce unfairness due to the elimination of interference between threads. Our analysis also shows that BPM can be an effective mechanism on future manycore platforms on which per-core bandwidth is decreasing.

8. Acknowledgments

We would like to thank the anonymous reviews for their valuable feedback. Lei Liu, Mingjie Xing and Chengyong Wu are supported by the National Natural Science Foundation of China

under grants No. 60873057, 60921002 and 61033009; the National High Technology Research and Development Program of China (863 Program) under grants No. 2012AA010902; the National Basic Research Program of China (973 Program) under grant No. 2011CB302504; and the National Science and Technology Major Project of China under grants No. 2009ZX01036-001-002 and 2011ZX01028-001-002. Zehan Cui, Yungang Bao and Mingyu Chen are supported by the National Natural Science Foundation of China under grants No. 60903046, 60921002, 60925009, 61003062. the National Basic Research Program of China (973 Program) under grants No.2011CB302502.

9. References

[1] Hewlett-Packed Development Company. Perfmon project. http: //www.hpl.hp.com/research/linux/ perfmon.

[2] Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006/CINT2006/.

[3] N. Aggarwal et al. Power Efficient DRAM Speculation. In HPCA-14, 2008.

[4] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. In ACM SIGOPS Operating Systems Review 43(2): 56-65, 2009.

[5] Y. Bao et al. HMTT: A Platform Independent Full-System Memory Trace Monitoring System. In SIGMETRICS-08, 2008

[6] S. Beamer et al. Re-Architecting DRAM Memory Systems with Monolithically Integrated Silicon Photonics. In ISCA-37, 2010.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton Univ., Jan. 2008.

[8] S. Cho, and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level page Allocation. In MICRO-39, 2006.

[9] J. Carter, IBM Power Aware Systems. Personal Correspondence, 2011.

[10] Z.Cui, Y.Zhu, Y.Bao and M.Chen. A Fine-grained Component-level power measurement method. In PMP,2011.

[11] 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.

[12] J. Demme et al, Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In ISCA, 2011

[13] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In HPCA-8, 2002.

[14] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. In Journal of Supercomputing, 28(1), 2004.

[15] I. Hur and C. Lin. Memory scheduling for modern microprocessors. ACM Transactions on Computer Systems, 25(4), December 2007.

[16] R. Iyer et al, QoS policy and architecture for cache/memory in CMP platforms. In SIGMETRICS-07, 2007.

[17] C. J. Lee et al. Improving memory bank-level parallelism in the presence of prefetching. In MICRO-42, 2009.

[18] Y. Kim, M. Papamicheal and O. Mutlu. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In MICRO-43, 2010.

[19] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In HPCA-16, 2010.

[20] 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.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In Micro-41, 2008.

[22] G. L. Yuan et al. Complexity effective memory access scheduling for many-core accelerator architectures. In MICRO-42, 2009.

[23] 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 HPCA-14, 2008.

[24] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In RTAS-3, 1997.

[25] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In ISCA-35, 2008.

[26] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In USENIX Security, 2007.

[27] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In MICRO-40, 2007.

[28] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. In Proc. the 2010 IFIP Int'l Conf. Network and Parallel Computing (NPC), Sep. 2010.

[29] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-Processor Environment. In Proceedings of WMPI, 2004.

[30] S. Prashanth et al. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In Micro-44, 2011.

[31] M. K. Qureshi, and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In MICRO-39, 2006.

[32] M. K. Jeong, D. H. Yoon et al. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In HPCA-18, 2012.

[33] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In ISCA-27, 2000.

[34] B. Rogers et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In ISCA-42, 2009.

[35] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware. In ASPLOS-2010.

[36] H. S. Stone, J. Turek, and J. L. Wolf. Optimal Partitioning of Cache Memory. In IEEE Transactions on Computers, 41(9), 1992.

[37] A. Udipi et al. Rethinking DRAM design and organization for energy-constrained multi-cores. ISCA, June 2010.

[38] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In MICRO-33, 2000.

[39] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In ASPLOS-XV, 2010.