BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems[1] (Revised 2016-01-01)

LEI LIU, State Key Laboratory of Computer Architecture, ICT, CAS, China;
ZEHAN CUI, State Key Laboratory of Computer Architecture, ICT, CAS, China;
Graduate School, CAS, China
YONG LI, Department of ECE, University of Pittsburgh, USA
CHENGYONG WU, State Key Laboratory of Computer Architecture, ICT, CAS, China

Main memory system is a shared resource in modern multicore machines that can result in serious interference leading to reduced throughput and unfairness. Many new memory scheduling mechanisms have been proposed to address the interference problem. However, these mechanisms usually employ relative complex scheduling logic and need modifications to memory controllers (MCs), which incur expensive hardware design and manufacturing overheads.

This paper presents a practical software approach to effectively eliminate the interference without any hardware modifications. The key idea is to modify the OS memory management system and adopt a page-coloring based bank-level partitioning mechanism (BPM) that allocates dedicated DRAM banks to each core (or thread). By using BPM, memory requests from distinct programs are segregated across multiple memory banks to promote locality/fairness and reduce interference. We further extend BPM to BPM+ by incorporating channel-level partitioning, on which we demonstrate additional gain over BPM in many cases. To achieve benefits in the presence of diverse application memory needs and avoid performance degradation due to resource underutilization, we propose a dynamic mechanism upon BPM/BPM+ that assigns appropriate bank/channel resources based on application memory/bandwidth demands monitored through PMU (performance monitoring unit) and a low-overhead OS page table scanning process.

We implement BPM/BPM+ in Linux 2.6.32.15 kernel and evaluate the technique on 4-core and 8-core real machines by running a large amount of randomly generated multi-programmed and multi-threaded workloads. Experimental results show that BPM/BPM+ can improve the overall system throughput by 4.7%/5.9% on average (up to 8.6%/9.5%) and reduce the unfairness by an average of 4.2%/6.1% (up to 15.8%/13.9%).

Categories and Subject Descriptors: **C.4 [Computer System Organization]**: Performance of Systems-Design studies; **D.4.2 [Operating System]:** Storage Management-Main Memory;

General Terms: Management, Performance, Design

Additional Key Words and Phrases: Main Memory, Multicore, Interference, Memory Scheduling

ACM Reference Format: Lei Liu, Zehan Cui, Yong Li, Chengyong Wu. 2014. BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems. 2014, 32 pages.

[1] Extension of Conference Paper. The additional contributions of this manuscript over the previously published work of L.Liu et al at PACT-2012 include:

1) We analyze and propose DRAM channel-level partitioning and extend BPM to BPM+.
2) We implement the proposed BPM+ in CentOS Linux 5.4 with kernel 2.6.32.15.
3) We evaluate the impact of BPM+ on the system performance and fairness and show additional improvement over BPM.
4) We devise and implement a PMU-based dynamic BPM/BPM+ scheduling mechanism, which works well in practice and brings additional optimization opportunities.

Refer to the Acknowledgements for the grants under which this work is supported.

Author's addresses: L.Liu, Z.Cui, C.Wu, Institute of Computing Technology, Chinese Academy of Sciences; Email: liulei2010@ict.ac.cn; Y. Li, Department of ECE, University of Pittsburgh.

1. INTRODUCTION

Modern multicore machines provide parallel and powerful hardware components to deliver generic computing capabilities for simultaneously running applications. These applications often exhibit diverse and dynamic characteristics in memory accesses, which are serviced by the commonly shared DRAM memory system. This posts significant issues such as memory interferences and resource underutilization/unfairness as a result of blindly sharing memory resource among applications with distinct memory and bandwidth requirements. For example, a memory-intensive application whose memory requests dominate a memory bank/channel can significantly interfere with and cause performance lost for a non-memory-intensive application utilizing that memory bank/channel [Y.Kim and O.Mutlu, 2010; O.Mutlu and T.Moscibroda, 2007 and 2008; S. P. Muralidhara et al 2011].

A number of recently proposed scheduling algorithms [Y.Kim and O.Mutlu, 2010; O.Mutlu and T.Moscibroda, 2007 and 2008; S. P. Muralidhara et al, 2011] that are aware of different application memory characteristics have been demonstrated to be effective for reducing the memory contention and interference. For instance, TCM [Y.Kim and O.Mutlu, 2010], 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 (Quality of Service) improvements for the overall system. Although some sophisticated memory scheduling algorithms are claimed to be easy for integration into memory controllers (MCs) [Y.Kim and O.Mutlu, 2010; G.L.Yuan et al, 2009; O.Mutlu and T.Moscibroda, 2008; C.Natarajan et al, 2004; S. P. Muralidhara et al 2011], they usually introduce relatively complex hardware logic and require extra storage in MCs to store per-core (or per-thread) information.

In this paper, we propose software approaches to effectively eliminate the memory contention and interference problem without any hardware modifications to MCs. Our approach is inspired by three observations that 1) DRAM bank-level conflict is a major source for the memory contention and interference problem and 2) the performance benefit achieved through multi-banking for one thread does not scale beyond a certain number of banks (typically less than 16). 3) Contention and interference among memory channels across threads often cause memory operation thrashing and bandwidth underutilization, even with the prevalent channel-level interleaved access patterns (discussed in Section 3).

Theoretically, exclusively mapping a thread's data to a number of dedicated banks can eliminate inter-thread bank-level conflicts. We adopt this basic idea and modify the physical page allocation policy in the OS memory management subsystem so that physical pages in specific banks are exclusively mapped to a specific thread (or core). For example, if the 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), the MC will deliver all memory requests from thread 2 to only bank 4~15 without affecting thread 1, whose memory requests are delivered to only bank 0~3. By enforcing this mapping, the bank-level inter-thread memory contention and interference are eliminated. We call this approach Bank-level Partitioning Mechanism (BPM), which is an extension of the page-coloring. Other than the bank-level interferences, multiple memory requests issued from different threads directed to the same DRAM channel can also cause in-channel interferences, which not only exacerbate the overall system interference but also defeat any prior method that aims to address only a single aspect (i.e., bank- or channel-level) of the problem. To address the channel-level memory contention and promote more efficient memory operations and utilization, we propose channel-level

partitioning as an extension of BPM, noted as BPM+. BPM+ can be used together with BPM to effectively mitigate multi-thread memory interference issue at different levels and improve performance as well as quality of service (QoS) for a wide range of applications, particularly those with intensive memory accesses such as server and graphic workloads. In scenarios where the concurrently running applications require drastically different memory resources, our partitioning policy can detect each application's memory capacity needs by monitoring the amount of application's potentially required pages in the OS page table and bandwidth requirement through the processor's performance monitoring unit (PMU). Based on the obtained information, our scheme dynamically assigns/adjusts appropriate amount of memory resources (i.e., memory banks and channels) for each application to avoid resource underutilization and unfairness.

One remarkable advantage of our approach is that we cooperatively and dynamically utilize DRAM bank- and channel-level information to mitigate inter-core memory interference, reduce contention and enhance memory access fairness for multicore platforms. We implement BPM and 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/BPM+. Experimental results show that BPM can improve the overall system throughput by 4.7% (up to 8.6%), and reduce the maximum slowdown by 4.5% (up to 15.8%) on average. Moreover, BPM+ can bring additional up to 3% throughput improvement, and it outperforms BPM in many cases. We also find that BPM and BPM+ can save 5.2% energy consumption compared to the existing memory system.

In summary, we make the following contributions:

(1) We observe that the required number of banks for a particular thread does not scale beyond a certain value, which means that the thread's performance would not be improved if more banks were assigned to it. Empirical studies with diverse programs show that 8 ~ 16 banks are enough for one thread.

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

(3) We study the memory request handling mechanisms used in existing MCs with multi-channel support and find that the commonly used address interleaving based approach can post inter-core interference issues. To mitigate the channel-level interference we enhance BPM to BPM+ by introducing a partitioning approach at DRAM channel-level and demonstrate that BPM+ works well with BPM.

(4) To avoid memory bank/channel underutilizations and satisfy programs with dynamically changing memory capacity and bandwidth requirements, we develop a dynamic partitioning module for BPM/BPM+. The dynamic partitioning module monitors each application's memory requirement as well as the bandwidth utilization and allocates/adjusts the memory bank/channel resources accordingly to balance channel utilization, promote fairness and avoid memory limitations for high memory demanding applications. To the best of our knowledge, this work is the first to propose and implement the bank/channel partitioning based on dynamic application memory demands.

(5) We implement BPM and BPM+ in Linux 2.6.32.15 kernel for evaluation. Experimental results show that BPM and BPM+ can stably improve the overall system throughput, reduce the unfairness (represented by maximum slowdown) for most workloads and save energy consumption of the memory system. Moreover, BPM and BPM+ scale well for future multicore systems.

(6) We study the correlation between several micro-architecture factors (i.e., per-core bandwidth, row-buffer locality and DRAM page-policy) and the effectiveness of our BPM/BPM+, and find that our mechanism is promising for prevalent multi-/many-core platforms. Moreover, we devise an accurate indicator for BPM/BPM+'s effectiveness represented as a function of the standard deviation of each thread's row-buffer locality (i.e., Sum(BW)*Stdev(RBL)), which can be used to predict the performance improvement brought by BPM/BPM+.

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 other DRAM systems that employ banking and paging mechanisms.

DRAM Organization: Modern memory system consists of multiple independent banks, each of which contains at least one two-dimensional storage array. Banks can operate in parallel and thus memory requests to different banks can be served concurrently [O.Mutlu and T.Moscibroda, 2007 and 2008; S.Rixner et al, 2000].

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 has a size of 1KB~2KB. Once a request to a bank arrives, if the required row is in the row-buffer, MC can immediately issue a *read/write* command. Otherwise, a row-buffer conflict occurs and the MC needs to issue a *precharge* command to write back the content in the row-buffer and then issue an *activate* command to fetch the required row into the row buffer before issuing the *read/write* command. Obviously, a row-buffer conflict results in memory latency longer than that in the case of a row-buffer hit without any conflicts. For parallel applications, especially multi-programed workloads, requests from different cores rarely go to the same row and thus the row-buffer conflict occurs more frequently on a multicore platform, compared to a single thread computing environment.

Bank-Level Parallelism (BLP) and Bank Sharing: BLP indicates that multiple banks can serve concurrent memory requests because they are largely independent[2]. BLP can often help make full utilization of banks and improve memory bandwidth. Hence, memory system usually employs a bank-interleaved address mapping scheme to take advantage of BLP [C.J.Lee et al, 2009; O.Mutlu and T.Moscibroda, 2007 and 2008; S.Rixner et al, 2000, Z.Zhang et al, 2000], which unselectively shares all banks among all cores in a multicore system. Unfortunately, such a global bank-sharing scheme brings interferences among threads because one bank may receive memory requests from different cores with significantly distinct memory access characteristics. Therefore, the inter-core bank conflicts become more and more frequent as the core number increases.

Multi-channel DRAMs: Multi-channel technologies are employed in modern DRAM systems and supported by many popular microarchitectures including Intel i7 (double/triple-channel) and AMD Opteron 6100 processors (quad-channel). In such systems each channel is equipped with its own row/column control bus and data bus and thus can operate independently to provide higher memory bandwidth and data transfer rate compared to a single channel memory system. To promote high bandwidth utilization and ensure memory accesses are evenly distributed, modern

[2] Multiple banks under the same channel typically have independent decoders, row buffers, etc., but they can still share command and data buses.

MCs finely interleave memory requests from multiple processing cores across all channels.

OS Memory Management: Currently, the Linux kernel's memory management system uses a buddy system to manage physical memory pages. In the buddy system, a power-of-two number of continuous pages (called a block) are organized in a free list in sequential order, ranging from zero to a specific upper limit. When a program accesses an unmapped virtual address, a page fault occurs and the OS kernel takes over the subsequent execution flow where the buddy system identifies the free list in an appropriate order and allocates one block of pages for that program. Usually the first block of a free list is selected but the exact physical pages are undetermined [S.Cho and L.Jin et al, 2006; J.Lin et al, 2009].

2.2 Multicore-posed Challenges and Current Solutions

Multicore architecture poses two major challenges on memory systems:

Interference: Usually a single thread's memory requests have good locality and exhibit a high row-buffer hit rate. However, this high locality can be significantly reduced in a multicore machine where multiple threads issue requests with shuffled memory addresses that break the locality a single thread would see. As a result, row-buffer hit rate decreases sharply, leading to poor overall system performance. For example, Udipi [A. Udipi et al, 2010] demonstrate that the row-buffer hit rate decreases significantly from 1 core (over 60%) to 16 cores (35%).

Unfairness: Conventional MC scheduling algorithms (e.g., FR-FCFS [I. Hur et al 2007, S. Rixner et al, 2000]) are designed in favor of memory requests with good row-buffer locality in order to improve row buffer hit rate. Therefore, memory intensive applications with better locality can obtain higher priority over memory non-intensive applications. For instance, Mutlu et al. [O.Mutlu and T.Moscibroda, 2007] demonstrate that the slowdown for some memory non-intensive applications can increase by 7.74X for a 4-core system and 11.35X for an 8-core system whereas the memory intensive applications only experience a slowdown of 1.04X and 1.09X, respectively.

One major reason of the two problems is that in practice MCs cannot identify distinct memory access patterns from various threads in a multiple-threaded memory request stream and these memory requests are scheduled by the basic FR-FCFS policy, which aims to maximize the utilization of the opened row-buffer in DRAM banks but leaves potential optimization opportunities for system fairness and overall performance [I. Hur et al, 2007; R. Iyer et al, 2007; C.J.Lee, 2009; Y.Kim et al, 2010; O.Mutlu et al, 2007 and 2008]. To address these challenges, many new memory scheduling algorithms have been proposed. For instance, the state-of-the-art scheduling algorithm Thread Clustering Memory Scheduling (TCM) [Y.Kim et al, 2010] brings both performance and QoS improvements for the overall system by identifying the threads' memory patterns and then classifying threads into memory-intensive group and non-intensive group, each of which adopts a different scheduling policy.

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 partitioning approaches are proposed in order to eliminate interference at cache level [J.Lin et al,

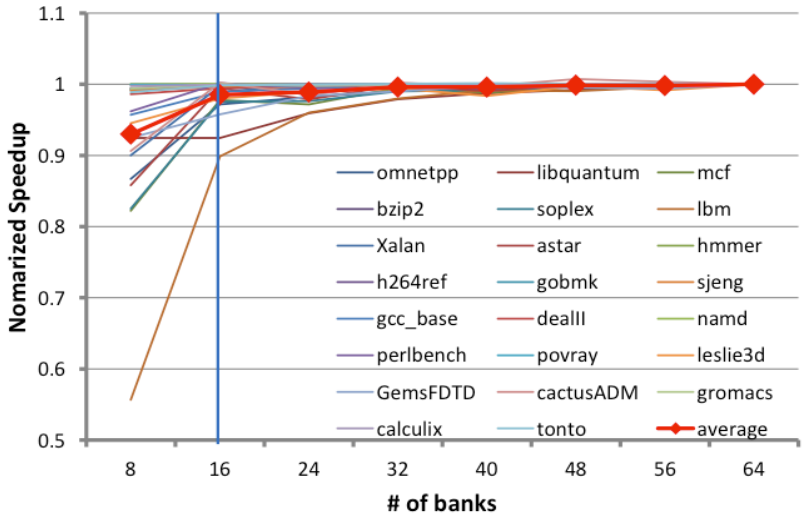


Figure 1. The correlation between application performance and number of banks. The blue line is the "watershed", which indicates all benchmarks can achieve 90% of its maximum performance with only 16 banks.

2008] while leaving the contention at bank-level unaddressed. Recent research proposes a bank-level partitioning among multi-programmed workloads [W.Mi X.Feng et al, 2010; M.K.Jeong et al, 2012]. However, their work is not deployed or tested with real hardware.

In addition to the bank-level conflict, the prevalence of multi-channel memory technology (e.g., DDR, DDR2, SDRAM and DDR3) introduces another level of conflict that further exacerbates the memory interference problem. This is due to the fact that existing MCs in multi-channel memories make a straightforward attempt to promote high channel utilization among multiple cores/threads by finely interleaving (e.g., at 64 bytes granularity) memory requests across multiple memory channels. With such a fine-grained interleaving, MCs virtually deliver memory requests from all running threads to every channel, creating significant contention issue and performance bottleneck, especially when threads' memory requests with disparate characteristics are blended together within one channel. For example, in a typical system where the MCs prioritize memory requests with higher row-buffer locality, memory requests from a thread with streaming access pattern (or other memory-intensive patterns with higher row-buffer locality) are interleaved across all channels and thus all other threads (especially memory-non-intensive ones) suffer from significant interference and unfairness issue. Even MCs in the most recent commodity processor chips (e.g. i7 series) are completely unaware of the channel-level contention and blindly interleave all memory requests among all channels. S. P. Muralidhara et al. [S. P. Muralidhara et al, 2011] propose a channel partitioning approach to address this problem. However, they do not take into account the bank-level conflict problem together and they do not verify their approach on real machines.

Although many of the aforementioned solutions [Y.Kim and O.Mutlu, 2010; G.L.Yuan et al, 2009; O.Mutlu and T.Moscibroda, 2008; C.Natarajan et al, 2004; S. P. Muralidhara et al, 2011] are claimed to be easy and low overhead, they usually introduce considerable amount of storage to collect various information (e.g, per-core

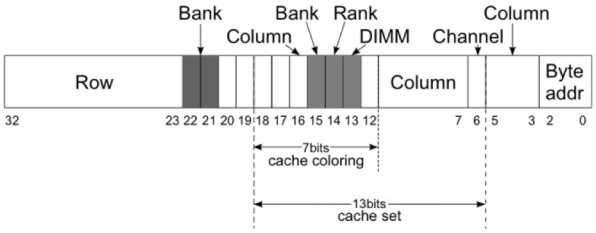


Figure 2. Address mapping policy of our platform (i7-860 8GB DDR3). The bank bits are divided into two separate parts. One is overlapped with cache set bits and the other is independent.

access patterns, row-buffer hit rates, bandwidth requirements, etc.) and expensive hardware to perform adaptive memory request scheduling algorithms. For example, TCM [Y.Kim and O.Mutlu, 2010] requires additional 4K bits storage in a MC to support 24 cores, an extra hardware unit to rank threads and a central meta-controller to gather global information from multiple MCs. Additionally, their method requires a sophisticated hardware for monitoring thread behaviors such as context-switch, which could occur frequently when the thread number in the system is much larger than the core number. Therefore, industrial vendors seem to show some hesitation in adopting these aggressive and hardware-based scheduling algorithms. Hence 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 through exclusively mapping a thread's data to a limited number of banks. However, doing so will reduce the available banks and potentially the bank-level parallelism. Thus, it is important to know how the amount of available banks influences the performance.

We conduct experiments on an Intel i7-860 machine with 64 banks (125MB per bank) to analyze the correlation between the amount of banks and application's performance (details of experimental setup are in Section 4). For each application, we vary the number of available banks from 8 to 64 and observe the performance changes. Figure 1 illustrates the results of 23 benchmarks from SPEC2006 [Standard Performance Evaluation Corporation]. Surprisingly, we find that the necessary amount of banks one program requires is limited, for example 16 banks in our experiments. In other words, 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.

Typically, a single core is not likely to generate enough concurrent memory requests due to a 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 all banks in order to take advantage of the bank-level parallelism. In those systems, any program can access all the DRAM banks, largely exceeding the necessary amount a single application requires. As a result, the programs that share all banks suffer from memory interference rather than obtaining any performance gain. In DRAMs with multi-channel support, the memory interference issue is further amplified due to the fine-grained (e.g., cache-block-level) interleaving of memory requests among all channels, which is adopted in many existing MCs to promote high channel bandwidth utilization among multiple cores/threads (see Section 3.2 for details). In such a scenario, memory requests from

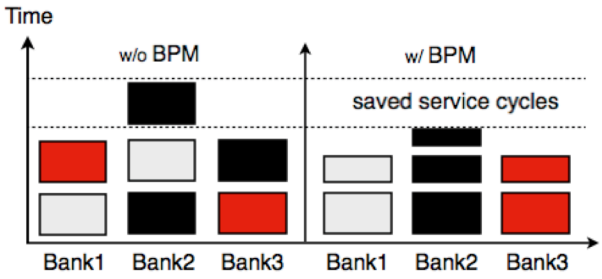


Figure 3. The comparison between interleaved address mapping and BPM. Different colors represent requests from different threads.

any program can be potentially interleaved among all channels, thus multiple concurrently running programs with diverse access patterns can reside within the same channel. Consequently, applications might suffer from more severe interference and contention when all cores share all the channels and banks.

The above insight inspires us that it is feasible to partition banks into several groups and then designates 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 partitioning mechanism (BPM), to effectively eliminate the memory contention and interference problem without any hardware modification to MCs. Moreover, with the concern about channel-level interference and contention, we further extend BPM to BPM+, which partitions channels among threads and reduce the channel-level contention and interference.

3. BANK- AND CHANNEL-LEVEL PARTITIONING MECHANISM

3.1 Bank-level Partitioning Mechanism (BPM)

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

3.1.1 Partitioning Methodology

A physical address contains several bits that are commonly used to denote both OS page index and bank index. 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. Partitioning means that BPM exclusively assigns banks with the same color to a thread such that only that specific thread can access those banks. Note that a thread can have multiple bank colors. Consider the following as a concrete example. Our experimental machine has 8GB DDR3 main memory with 64 banks. Typically the OS page size is 4KB and thus the OS physical page index bits are bit 12~32. Figure 2 illustrates the 5 bank color bits of our platform, i.e., bit 13~15 and bit 21~22. We use these 5 bits to represent 2^5 = 32 colors for the 8G/64-bank system and thus each color corresponds to 2 banks across different channels (bit 6 denotes the channel). The 5 coloring bits in the index (bit 12~32) of a particular OS page designate the color and thus the corresponding memory bank(s) to which the page is assigned. When a thread applies for one page, BPM first checks which bank colors are owned by the thread, and then one of the available bank colors is selected and a corresponding physical page with that color is allocated for the thread.

3.1.2 Advantages of BPM

Although the OS page-coloring technique is well known for its adoption in cache partitioning techniques [J.Lin et al, 2008], the proposed BPM, to the best of our knowledge, is the first attempt to effectively leverage the page-coloring to eliminate DRAM bank interferences in practice. In particular, BPM allows the OS to partition memory space based on the underlying memory bank information. Because some MCs also schedule memory requests at bank level, the OS partitioning effect can be indirectly propagated to MCs.

Figure 3 illustrates the advantage of BPM. The smaller blocks represent row-buffer hits with lower latencies. Assume there are three threads (denoted by different colors) issuing memory requests to DRAM banks. In conventional page allocation without BPM (the left), these requests are delivered to all banks, resulting in many row-buffer conflicts. With BPM (on the right), one thread's memory requests are mapped to its dedicated banks so that row-buffer conflicts are eliminated among threads (cores).

Specifically, BPM brings the following advantages:

(1) BPM is an entirely software approach and thus it is easier to be implemented in modern multi-core systems. Additionally, BPM, as a software approach, is more flexible and offers a wider spectrum of customization compared to hardware approaches. In particular, more partitioning policies can be explored to achieve better performance.

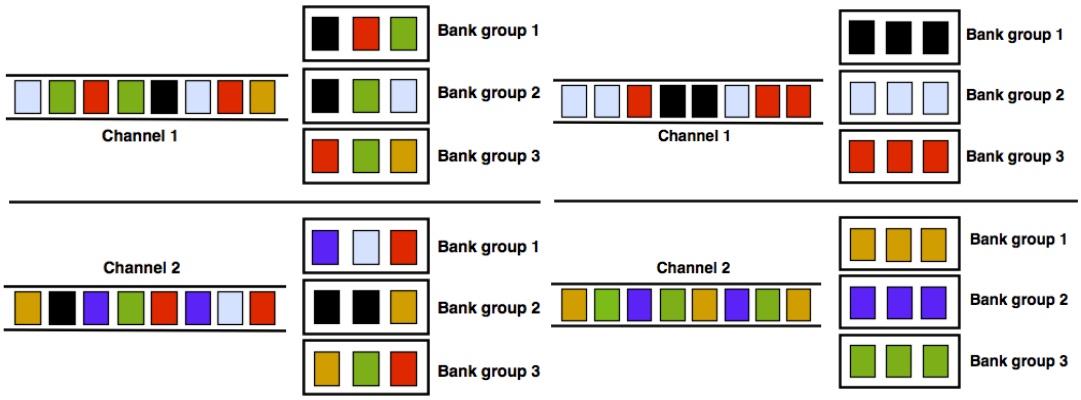
(2) Compared to prior hardware approaches that require additional storage and logic, our OS-based scheme can monitor threads' behavior (e.g., cache miss rate, memory bandwidth, etc.) by leveraging the existing performance counter available in contemporary processors with negligible overhead. This significantly eases the exploration of the partitioning policies.

(3) Moreover, BPM brings opportunities for improving other OS functionalities. For example, the OS process management module can utilize BPM partitioning 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 enhance the VM isolation.

3.2 Channel-level Partitioning and BPM+

The baseline BPM partitions memory requests at the DRAM bank level and is completely unaware of different DRAM channels. Based on our quantitative study (see the following sections for more details), inter-threads memory interference can also frequently occur at channel level due to the way MCs interleave memory requests. Specifically, existing MCs typically use a particular low-order bit (the bit 6 in Figure 2) in the physical address to finely interleave the memory requests among DRAM channels. This fine-grained interleaving mechanism reduces potential memory access hot spots in a particular channel but significantly increases the channel-wide interferences among threads.

Figure 4(a) illustrates a common scenario where memory requests from six threads are finely shuffled in the two channels, each of which contains requests from all the threads and thus suffers from all-to-all thread interferences. In other words, any thread might potentially compete with, or break the locality of, any other threads in the system. The fact that a memory request from a single thread in one channel can be distributed to multiple banks under this channel further increases the likelihood



(a) No partition is used (cache-line interleaving) (b) BMP+ is used (BPM+ includes BPM)
Figure 4. Interference elimination via BMP+.

of the inter-core interferences. For example, the black request in channel 1 is sent to bank group 1 and 2 while the green requests are directed to all of the three banks groups, creating potential contention with the requests from all the other processes in the channel. The issue becomes particularly severe when each channel contains requests from a large number of different cores, which is likely to be the case with the core number scaling and the fine-grained address interleaving. By contrast, Figure 4(b) depicts that in our approach a coarse-grained channel-level partitioning is applied to exclusively split the requests into two groups based on the issuing cores. This reduces the possibility of the all-to-all interference among threads/cores. The channel-level partitioning can be achieved by modifying the channel selection bits in the system BIOS (shift the channel selection bit from the 6th one to the 32nd one) and the OS memory management policy (see Section 3.4).

As Figure 4(b) shows, after the channel partitioning, threads in the two different channels are isolated thus only half of the system's threads could potentially interfere with each other. Combined with our BPM approach, requests from different threads in a particular channel can be further directed to the pre-designated bank groups and thus the interference is entirely eliminated while the parallelism is still preserved within the bank group. We call this combination of bank- and channel-level partitioning approach BPM+.

3.3 Discover Bank Bits by Software Method

In order to employ BPM and BPM+, we need to obtain the memory address mapping information so as to extract the bank bits. The address mapping in a specific memory design can be found in the corresponding vendor's manual. However, address mapping in a MC is not fixed and can be configured by BIOS at boot time. Moreover, a MC can support various address mapping policies. Some hardware tools can be used to derive the address mapping policy, but it is impossible to deploy this approach for a large number of machines in production environments. In order to solve this problem, we propose a practical software method (Algorithm1) to discover the address mapping policy as well as bank selection bits for any particular machine. 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 and the verification results show that our algorithm works well on various platforms. Thus, this

algorithm can be embedded into the OS bootstrap to collect the address mapping information, which can be used for BPM and BPM+ setup. We show a concrete example on an Intel i3 platform in Appendix.

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 1000000 times)
5. END FOR
6. The latency will be easily clustered into two groups
7. Put the group with higher latency into Row{}  //higher latency is caused by row buffer miss
8. The left parts with relative lower latency are put into Remain{}  //lower latency is caused by row buffer 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 1000000 times)
6. END FOR
7. The latency will be easily clustered into two groups
8. Put the group with higher latency into Column{} //row buffer miss leads to higher latency
9. The left parts are put into Remain{}  //mapped to different banks. Bank parallelism causes lower latency
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> IN {<u,v> | ∀u∈Remain{}∧∀v∈Remain{}}   // Note: u != v
2. DO
3.     Generate 2 memory requests, one's u and v bit are both 0, and another request's u and v bit are both 1
       The other bits are identical except one bit in Row{} is different
4.     Access the two addresses (uncached) in turn and record the latency (repeat at least 1000000 times)
5.  END FOR
6.  The latency will be easily clustered into one or two groups
7.  IF there is only one group THEN
8.     XOR is not employed
9.  ELSE
       // XOR makes <0,0> and <1,1> denote a specific bank, and thus higher latency is caused by row buffer miss
10.    For each <u,v> pair in the group with higher latency, put "u XOR v" into BANK{} and delete u,v from Remain{}
11. END IF
12. Put Remain{} into BANK{}
13. Output BANK{}
END
```

3.4 Augmenting BPM and BPM+ with Dynamic Adjustments

The prior sections only present BPM/BPM+ with static partitioning policy in which each of the running application is assigned to a pre-determined amount of memory bank/channel resources. However, in practice, a system could encounter challenges and complicated cases that cannot be appropriately handled by the static approach. For example, static partitioning is not suitable for the cases where multiple programs require dynamically changing amount of memory that cannot be easily determined statically. If an application requires much more memory than the amount allocated by the static BPM, system performance could suffer due to excessive IO transactions as a result of the utilization of swap region. BMP+ is also limited by several factors such as the channel utilizations of various running applications (detailed in Section 5.8). This section presents our study of limitations of the static BPM/BPM+ and motivates the design for BPM/BPM+ with dynamic adjustments.

3.4.1 Dynamic Bank Partitioning for BPM

Dynamic BPM is useful when an application requires more memory than the amount that the static BPM assigns to it. In such a case, we extend BPM to dynamically adjust the resource allocation and assign more memory to threads with higher memory demands through page re-coloring mechanism. Since page re-coloring incurs significant overheads [J.Lin et al, 2008], the dynamic adjustment process in BPM is only triggered when certain application experiences high memory pressure. Note that in many cases dynamic re-coloring means that applications with high memory demands can "borrow" unused pages from applications with low memory requirements.

To determine whether an application's memory requirement can be satisfied with the bank-level partitioning, BPM uses an OS level low overhead online profiling module that scans the application's page table[3] and calculates the **r**equired **n**umber of **p**ages (RNP), which is the maximum number of pages an application would touch at runtime. BPM then compares RNP with the actual **n**umber of **p**ages in the allocated **b**anks (BNP) for the application, which can be easily calculated based on the memory configuration information (i.e., memory capacity, number of banks, page size, etc.). BPM calculates RNP and BNP for all the running applications. Dynamic adjustment decision is made based on a comparison of the two values: if RNP is larger than BNP for one application, BPM can potentially "borrow" RNP-BNP pages from one or more applications in which BNP is larger than RNP. In reality, since partitioning mechanism prevents two or more applications from sharing the same bank, BPM only adjusts memory resource at a coarse granularity in most cases. In our experiments, we define SUM(RNP) = $\sum_{i=0}^{N-1} RNP_i$ as the amount of pages used by all threads (N is the number of threads). Obviously, if SUM(RNP) × Page_Size (i.e., 4KB in our system) is smaller than the memory capacity, it is likely that no I/O overhead will be incurred due to swap, and BPM can allocate appropriate amount of banks to applications according to their demands by dynamically adjusting. We further define NICE=BNP–RNP for each running application to represent the amount of potential resource that could be borrowed by other applications. The higher the NICE value is, the more resource one application can give to others. BPM sorts applications according to their NICE values to determine the "lenders" and "borrowers". Note that BPM guarantees that the adjustment does not hurt any

[3] We actually scan the virtual memory area (VMA) structure of the program and use the start and end address of each memory area to calculate the total required virtual memory to avoid expensive traversal over the entire page table.

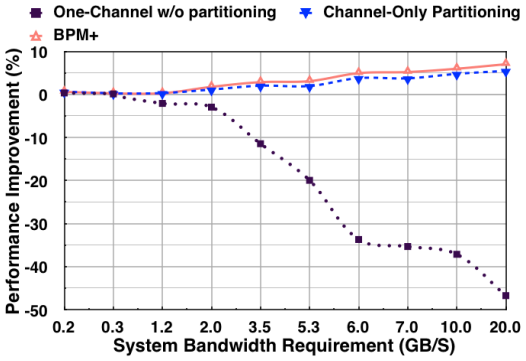


Figure 5. The correlation between bandwidth demands and performance changes caused by different policies. (This figure includes tens of workloads. The baseline is channel interleaving).

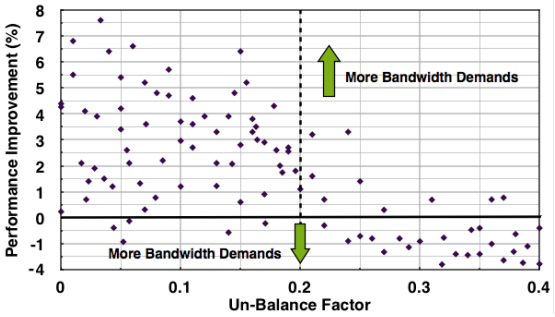


Figure 6. The correlation between unbalance factors and performance changes caused by channel partitioning across nearly 100 workloads after channel partitioning. (Each dot denotes a workload)

application (i.e., causing BNP–RNP<0). We conduct experiments and demonstrate that dynamic bank partitioning could significantly improve BPM effectiveness and memory utilizations under high memory pressure environments. In most case, adequate memory resource is provided and thus, dynamic adjustment does not happen frequently. Our experiments show that the average overhead is below 1.4%. Under certain extreme cases where memory pressure is high (e.g., most running applications exhibit high memory footprint), the coarse-grained bank-level adjustment is not suitable because the number of available banks is limited. Under these circumstances, in order to avoid resource underutilization and I/O swap, dynamic BPM is able to allow two or more applications to share the same bank or bank group.

3.4.2 Dynamic Channel Partitioning for BPM+

When Channel Partitioning is Beneficial? We conduct experiments to study the correlation between the system bandwidth requirement and the effectiveness of different partitioning approaches. Shown in Figure 5, as the bandwidth increases (x-axis), the two approaches with channel partitioning (channel-only and BPM+) exhibit similar performance increasing trend. However, when the overall bandwidth requirement is below 2GB/s, the two approaches bring no obvious improvement (<1%). Thus, based on our experiments that average several tens of different cases, we conclude that channel partitioning is more effective when the system bandwidth requirement is beyond a certain threshold (2GB/s in DDR3-1600 with i7 CPU with channel-level interleaving configuration. i.e., 1GB/s per-channel). Notably, when entire bandwidth demand is above 2GB/s, the overall performance suffers tremendously if all running applications share only one channel (one-channel w/o partitioning represented in the purple curve). Thus we conclude that 2GB/s (i.e., 1GB/s per-channel) is an important channel bandwidth threshold in our platform (with channel-level interleaving), beyond which the system needs to use more bandwidth resources from multiple channels to avoid significant performance degradation, indicating channel partitioning might potentially benefit performance.

From another aspect, to reveal how channel partitioning affects the performance under different scenarios we run experiments of hundred of different workload combinations to find the key factor(s) that can impact the effectiveness of channel partitioning. We found that the unbalance between the two memory channels shows

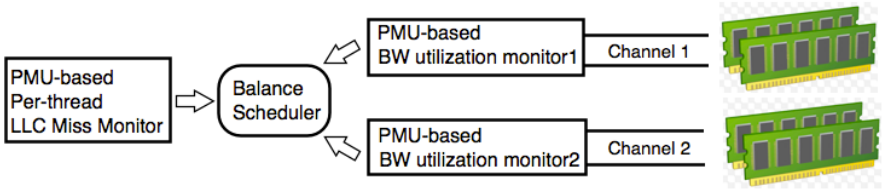


Figure 7. The framework of dynamic BPM+.

a great influence on the performance of channel partitioning. To quantify this influence we define a metric named **Un-balance Factor**, defined as |BUC1 – BUC2| / MIN (BUC1, BUC2), where BUC1 and BUC2 are the **B**andwidth **U**tilizations of the two **C**hannels. It directly reflects the difference of bandwidth utilization between different channels. Figure 6 draws a distribution of the tested workloads (dots in the figure) with different performance gains and unbalanced factors. As can be seen from the figure, workloads with small unbalance factors (left part of Figure 6) exhibit higher performance gains. As the unbalance factor grows larger the performance gain begins to drop. When the unbalance factor is above 0.2 (which means one channel's total memory requirement is 20% more than the other), many workloads experience a negative performance improvement. The above observations indicate that channels should be partitioned evenly. In addition to the unbalance factor, the absolute bandwidth utilization also affects performance. On the same unbalance factor, workloads with higher bandwidth utilization (workloads towards the two ends along the vertical axis in Figure 6) tend to exhibit larger performance impacts from the channel partitioning. ***Thus, we conclude that channel partitioning works best under low unbalance factor and high bandwidth utilization.*** The above conclusion motivates our design for BPM+ with dynamic adjustment and channel balancing, as described next.

Dynamic BPM+ Design: BPM+ with dynamic channel adjustment requires the knowledge of the bandwidth requirement for each running workload. However, precisely measuring the bandwidth usage on a per-thread basis is not possible when more than one threads share the same memory channel. To address the above issue and achieve dynamic BPM+, we developed a per-channel bandwidth utilization monitor implemented by Intel PMU to obtain the channel utilization information and a per-thread last level cache (LLC) miss monitor to estimate the memory bandwidth requirement for each thread. The LLC miss and channel utilization information is fed into a balance scheduler to dynamically adjust/migrate (implemented through page re-coloring) threads across memory channels to achieve better bandwidth balance. Since the lower the unbalanced factor is the more performance gains BPM+ can bring, as demonstrated by Figure 7, the key point is to minimize the difference of bandwidth utilization between two channels. For instance, our approach assigns one thread with high bandwidth demands into channel 1 and three threads with relative lower bandwidth into channel 2 on a 4-core 2-channel platform. When adjusting channel balance, the balance scheduler considers the thread with the smallest memory footprint dictated by the LLC miss monitor as the migration candidate to avoid high page re-coloring overheads. In reality, our balance scheduler does not adjust the channel assignment frequently due to two reasons: first, dynamic channel adjustment requires page re-coloring, which incurs high overheads; second, in many cases once an application is running stably its bandwidth requirement does not change frequently. In summary, with the abovementioned dynamic partitioning mechanism, dynamic BPM+ accomplishes partitioning in two steps. In the first step, BPM+ tries to partition applications over the memory channels so that the unbalance

factor is minimized. In the second step the DRAM banks are partitioned dynamically among the applications under each channel based on their memory requirements, as described in Section 3.4.1. The scheduling routine scans the system in every 10s and collects the memory usage information (i.e., bandwidth unbalance factor, memory footprint of each application, etc.). Based on an online analysis of the collected information, our system determines whether or not to invoke the scheduling process. On our platform with the SPEC2006 workloads, 10s is an appropriate scheduling frequency that is enough to collect the workloads' memory utilization information while incurring low overhead under most cases. In practice, users can adjust the scanning interval according to their workloads' features and system needs.

3.5 Implementation of BPM/BPM+

We implement BPM and 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 structure: for each order of free page list, we re-organize the free pages to form 32 colored free lists according to the five bank bits. Each process has its own colors (i.e., a group of banks). When a page fault occurs, the OS kernel searches a colored free list and allocates a page with target color 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 the OS kernel with new APIs to expose the underlying bank colors to programmers so that they can map threads' data into different colors based on their demands. The overhead of the color searching operation is negligible (0.3% on average).

4. METHODOLOGY AND METRICS

4.1 Hardware and Software Platform

We conduct our experiments on a machine with four 2.8GHz Intel Core i7-860 processors sharing 8MB 16-way associative LLC. The processor incorporates Hyper-Threading technology thus we can run 8 threads concurrently. We use CentOS Linux 5.4 with kernel 2.6.32.15. The memory system is 8GB DDR3 with 64 banks (each bank is 125MB). There are five bank bits (i.e.,13,14,15,21,22) dividing the memory into 32 colors and each color represents two banks bundled together across the two channels. In our experiments, colors are statically assigned to processes/threads when they are created. Modern multicore servers often have enough memory for their running threads [G.Dhiman et al, 2009]. Therefore, we also disable the OS swap to avoid unexpected overhead. Moreover, for most of the experimental workloads, the memory capacity of 0.5~1GB (4~8 banks) is enough. We use Perfmon2 [Hewlett-Packed Development Company] and its corresponding libpfm library to access the performance counters to gather architectural information such as memory bandwidth and LLC miss rate. For memory system, because DIMMs are directly plugged into the motherboard and it is difficult to measure memory power consumption, we adopt an in-house hardware tool [Z.Cui et al, 2011] that consists of a wrapper card for each DIMM. The wrapper card is plugged into the motherboard's DIMM slot and the 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.

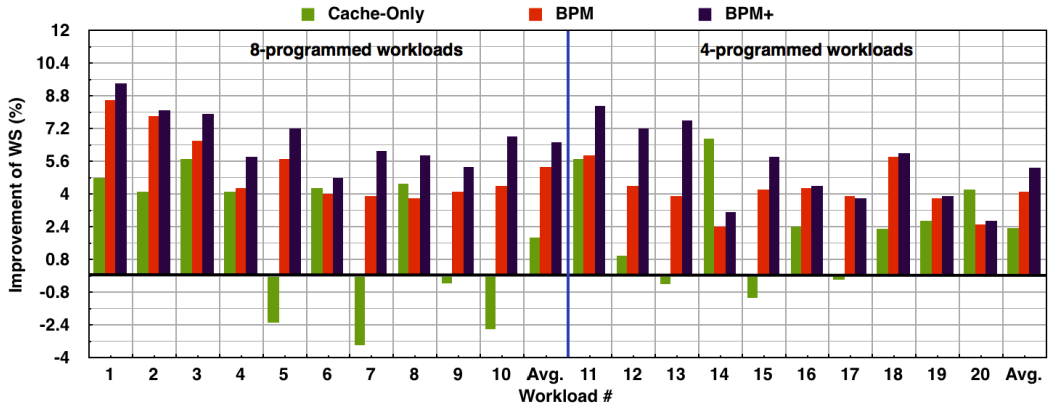


Figure 8. Overall system performance improvement.

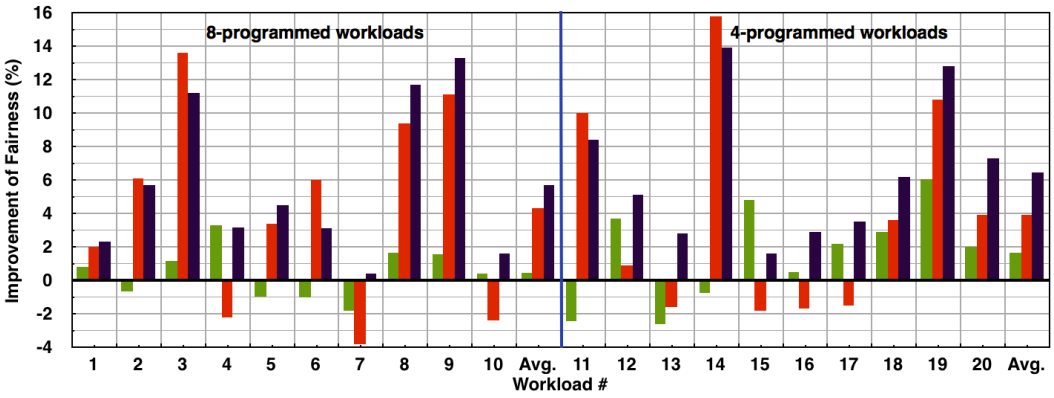


Figure 9. Fairness improvement.

4.2 Benchmarks

We use the SPEC CPU2006 [Standard Performance Evaluation Corporation] benchmarks for evaluation. We compile each benchmark using gcc 4.4.3 with -O3 optimizations. From these benchmarks, we randomly generate multi-programmed workloads, each of which contains 4 or 8 applications. We employ a multi-threaded benchmark streamcluster from PARSEC 2.1 [C.Bienia et al, 2008]. We use the notion of "Miss Per Kilo-Instruction (MPKI) > 1" to define memory-intensive applications. We use the recommended input size for SPEC benchmarks and the native input for PARSEC. In the first step of our experiments, we manually balance the bandwidth utilization among channels and assign DRAM banks to different threads statically. For BPM, each application in a 4-programmed workload is assigned 8 colors (i.e., 16 banks/2GB) and each one in an 8-programmed workload is assigned 4 colors (i.e., 8 banks/1GB). In the second step of the experiments, we enable the dynamic BPM/BPM+ and test them using on-line randomly generated diverse workloads.

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 and BPM+.

$$\text{Weighted Speedup(WS)} = \sum \frac{RUNTIME_{ALONE}}{RUNTIME_{SHARED}}; \text{Maximum Slowdown(MS)} = Max\left\{\frac{RUNTIME_{SHARED}}{RUNTIME_{ALONE}}\right\}$$

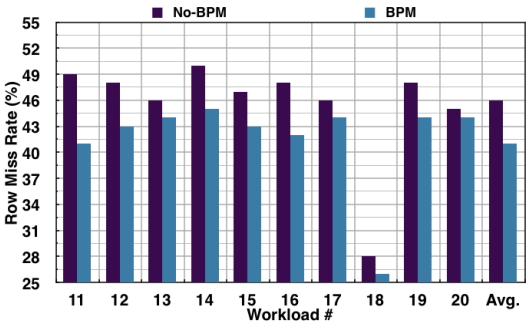


Figure 10. Row-buffer miss rate across 10 workloads between BPM and No-BPM platforms.

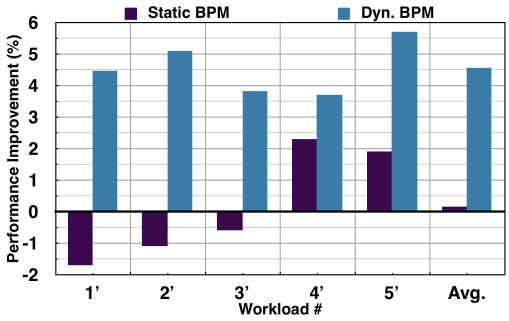


Figure 11. Dynamic BPM's performance in memory-limited cases.

5. RESULSTS

5.1 Overall System Performance

For performance evaluation we compare the proposed BPM and BPM+ with the cache-only partitioning approach over 20 randomly selected workloads (Note that in this experiment, we mannually balance the workloads' bandwidth based on the off-line profiling results). All the three schemes are normalized to a baseline system with no partitioning. Figure 8 shows that BPM and BPM+ bring steady speedups for all the tested workloads and no workload's performance degrades compared with the baseline system. On average, BPM can improve the weighted speedup by 5.4% and 4.1% for 8- and 4-programmed workloads, respectively. BPM+ brings an additional performance improvement of 1.2% and 1.1% over BPM for the 8- and 4-programmed workloads, respectively.

Now let us consider the 4- and 8- workloads seperately. For the 4-programmed workloads (i.e., workloads 11~20 in Figure 8), BPM achieves a weighted speedup of 5.7% for workload 11, which contains 462.libquantum (MPKI = 50, RBL[4] = 99.22%), 403.gcc (MPKI = 0.4), 447.dealII (MPKI = 0.5) and 444.namd (MPKI = 0.3). Among these 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 not memory intensive applications. In the baseline system where memory accesses are interleaved across all the DRAM banks, 462.libquantum causes substantial row-buffer conflicts due to its stream characteristic and thus results in drastic increases in the row-buffer hit rates for the other three applications. Figure 10 shows the row buffer miss rate reaches nearly 50% for workload 11. On the other hand, BPM assigns each application with 8 dedicated bank-colors (16 banks) only to which the application's memory requests can be delivered. According to Figure 1, restricting these benchmarks to dedicated banks incurs negligible performance degradation (8% for 462.libquantum and 1% for others). As a result, the overall system is improved as memory interference is fully eliminated (the overall row buffer miss rate also reduces by about 10% in Figure 10).

Among the 8-programmed workloads we find that workload 1, which also includes the memory intensive benchmark 462.libquantum, achieves a maximum performance improvement of 8.6%, compared to the 5.9% achieved by the 4-programmed counterpart (i.e., workload 11). In general, BPM and BPM+ bring higher performance gains with the 8-programmed workloads. This is because the 8-programmed

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

workloads create more contentions from a larger number of concurrently running programs. BPM+ performs slightly better for 8-programmed workloads since it introduces another level (i.e., channel-level) of partitioning that effectively partitions the potentially interfering memory requests. From the above analysis we draw a conclusion that our approaches can scale well for future multicore platforms with worsening interference scenarios.

5.2 Fairness Evaluation

For the fairness measure (shown in Figure 9), workload 11 exhibits 10% improvement and workload 14 sees an even higher improvement of 15.8%. A careful look into the constitutions of the two workloads finds that workload 11 comprises 462.libquantum, 403.gcc, 444.namd and 447.dealII (MPKI = 0.5) while workload 14 shares the first three benchmarks with its fourth one replaced by 456.hmmer (MPKI = 5.7). Obviously, 456.hmmer issues much more (i.e., more than 10 times) memory requests than 447.dealII thus it creates severe interference and fairness issues for workload 14 when running on the baseline system without BPM. This is due to the fact that the MCs in the baseline system usually give higher priority to memory requests with good row-buffer locality and thus memory intensive applications, which are more likely to exhibit good locality, can obtain higher priority over memory non-intensive applications. In the worst case, memory requests from memory intensive applications are always serviced first and this can starve applications with poor locality in a certain time window. By contrast, BPM isolates these applications with conflicting memory access patterns within their dedicated bank groups and thus effectively eliminating this unfairness. On average, BPM improves the fairness by 4% over the baseline system and 2.3% over the cache-only partitioning approach for the 4-programmed workload. BPM brings slightly higher fairness improvement for the 8-programmed workloads.

It should be noted that there are several workloads exhibiting degraded 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 extremely memory intensive application. An interesting observation is that when reducing bank number from 64 to 16, unlike 462.libquantum whose performance decreases by 8%, the performance of 429.mcf only decreases by 2%. Therefore, BPM improves 429.mcf's performance more than other non-intensive applications at the expense of slightly higher unfairness. BPM+ performs well and improves the fairness slightly since the aggressive programs such as 429.mcf are mapped into a different channel so that they cannot interfere with other programs and create unfairness.

In general, BPM+ benefits fairness and also outperforms BPM for both 4- and 8-programmed workloads on average. This is due to the similar reason that leads to the weighted speedup improvement of BPM+, as analyzed in Section 5.1. Notably, we observe that there are some workloads (workload 2, 3, and 6) for which BMP brings better fairness improvement than BPM+. This observation leaves us the space to choose an appropriate partitioning approach for fairness optimization in different cases. Figure 8 also illustrates the comparison between BPM/BPM+ and the cache-only partitioning. When the cache partitioning is used (with 8 colors in our case), both system throughput and fairness are improved slightly (around 3%). However, cache partitioning does not perform as well as BPM/BPM+ for most of the tested workloads and in certain cases it can hurt the performance (e.g., workload 5, 7 in Figure 8). On the contrary, BPM and BPM+ perform well in all cases. Thus, we come up to the conclusion that BPM and BPM+ are promising techniques to mitigate fairness issues for future multicore computing environment.

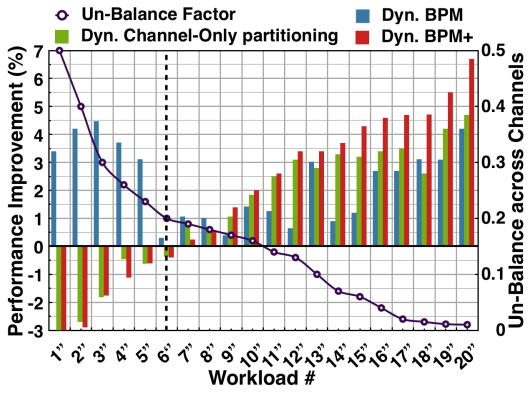


Figure 12. Dynamic Partitioning's performance under different unbalance factors.

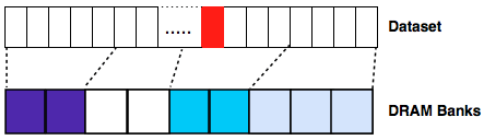


Figure 13. Thread level coloring. The above array is mapped to different bank colors. The colored rectangle in dataset represents shared data. For these streaming-like multithreaded workloads, we partition the dataset in a straightforward way. But the memory requests issued from threads to the shared data (red rectangle in this figure) would impact the effectiveness of partitioning.

5.3 Experimental Results of Dynamic Scheduling

To test BPM with dynamic bank-level partitioning, we generate five 8-programmed workloads with heavy memory demanding benchmarks (each workload contains at least one benchmark that requires more than 500MB memory) and have them run on a platform with limited memory capacity (i.e., 4GB). Figure 11 presents the results. As BPM evenly partitions DRAM banks among the running benchmarks, I/O storage overhead is incurred for benchmarks whose memory demands exceed the initially assigned amount. Therefore, in the static BPM where the dynamic bank partitioning is disabled, the overall system performance degrades in many cases (workload 1', 2' and 3'), or with insignificant improvement (workload 4', 5') due to the I/O latency offsets the performance gains brought by eliminating bank-level conflicts. Dynamic BPM addresses this problem by allocating more resources to benchmarks with high memory demands on-the-fly while eliminating bank-level interferences and thus leads to an average of 4.6% performance improvement for the overall system.

Figure 12 reports the performance improvements achieved by dynamic channel-only partitioning, BPM and BPM+ under different unbalance factors across 20 randomly online generated workloads with at least 4 benchmarks. These workloads are not the same ones as in previous experiments. As clearly shown in the figure, there are six workload combinations (numbered 1" to 6") with relatively high unbalance factors (> 0.2) on which channel partitioning brings negative performance impact up to 3%. This impact diminishes as the two memory channels exhibit a more balanced partitioning. BPM (no channel partitioning) can bring the performance gains in these cases because of the fine-grain cache-line interleaving across multi-channel, which leverages the bandwidth utilization. Thus, in such cases BPM should be used instead of BPM+. The 7th and 8th workloads see a very close performance difference over the scheme without channel partitioning (i.e., BPM) and the schemes with channel partitioning (channel-only partitioning and BPM+). Workloads 6", 7" and 8" have relatively small unbalance factors (<0.2) but their performance are minimally affected (within ±1%) by various partitioning schemes due to the low memory bandwidth requirements of these workloads. As shown in Figure 5 and Figure 6, if the memory bandwidth requirement is around 2 GB/s (each channel provides around 1 GB/s with the channel interleaving), there will be no obvious difference (<0.5% on average) between BPM/BPM+ and Channel-Only partitioning, and the overall system performance improvement is less than 1% on average.

Starting from the 9th workload, channel partitioning begins to bring more gains than the bank partitioning. This happens since the fine-grained channel-level memory request interleaving brings more severe interferences than the bank-level interferences and the channel partitioning eliminates the inference without creating a resource under-utilization problem (both the two channels are almost equally utilized). As the channels are more balanced for the workloads to the right in Figure 12, the channel partitioning brings more performance benefits than the bank-only partitioning. From Figure 12 we conclude that the dynamic mechanism (BPM or BPM+), by considering applications' memory requirements and channel utilizations, could potentially lead to better memory utilizations and achieve optimal performance gain under different scenarios while avoiding off-line profiling and limitations in a static approach (i.e., incurring I/O storage).

As mentioned in previous sections, during the dynamic scheduling process, our scheduler first tries to balance the bandwidth utilization through re-assigning some threads to an underutilized channel and then adjusts the amount of banks for each thread according to the actual memory usage. We find it is a cost-effective approach in practice since it can be easily deployed and brings performance benefits without complex scheduling logic in most cases. Nevertheless, for the workloads from 1" to 6", it is impossible to balance the two channels through coarse-grain scheduling. This is because that these workloads contain some applications whose bandwidth demands are significantly higher than the bandwidth sum of all the other threads. To handle these cases, our scheduler could assign memory resource from both of the two channels for the applications with extremely high bandwidth demands. However, as our experimental system is only equipped with a dual-channel memory, doing so will inevitably incur serious channel-level interference. In such cases, we can enable BPM-only partitioning to bring performance gain, as shown in Figure 12. Although the channel partitioning seems not desirable in above cases (workload 1" to 6" on our dual-channel platform), it would be useful on high-end servers with 4- or more channels, which provide more channel resources thus leaving more optimization space and flexibility for channel partitioning (more discussion are in Section 5.8: BPM+ orthogonally supports other policy).

5.4 Multi-threaded Workload

In practice, many servers are used to run multi-threaded workloads. We use streamcluster of PARSEC to evaluate BPM/BPM+. Its coloring scheme is nearly the same as that of the 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 the core number, the first N-1 chunks contains the same amount of points, while the Nth 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 (in Figure 13). We achieve performance gains by 1.7% and 2.3% with 4/8-thread respectively. The improvement is less than that of multi-programmed workloads, because of the large amount of shared data among threads (the colored rectangle in dataset in Figure 13). In our straightforward partitioning, the shared data belongs to the blue banks. When other threads access the shared data, inter-thread bank conflicts occur. Unlike our approach, a recent research attempt [H. Park et al, 2013] tries to optimize the memory accesses of multi-threads by interleaving all the memory requests across all banks. For streamcluster, their approach works better than BPM (around 15% improvement for 8-thread cases). But for other workloads their approach shows

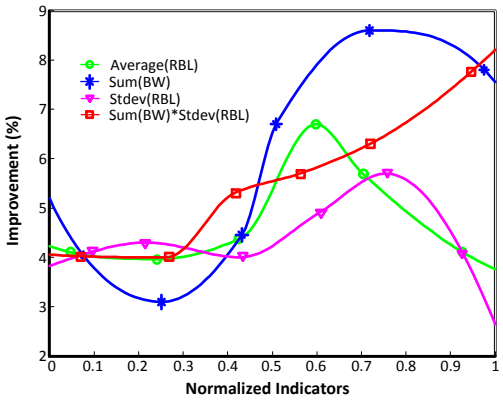


Figure 14. The correlation between BPM/BPM+ improvement and four indictors

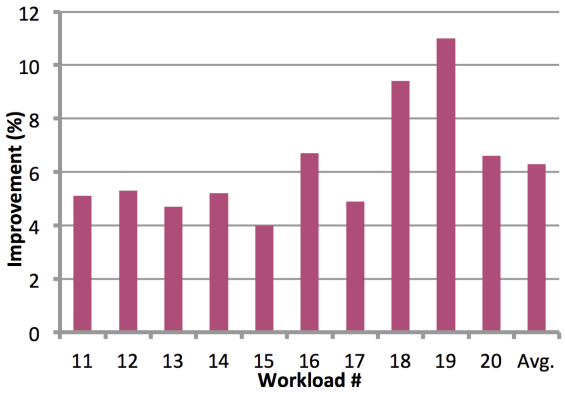


Figure 15. Performance improvement of open-page policy w/ BPM over close-page policy

moderate or even negative performance improvements in some cases. In our future work, we will compare our approach with H. Park's effort greater depth and explore more effective optimization mechanism for multi-threaded workloads. Our potential directions include designing a better partitioning policy and leveraging a dynamic color adjustment mechanism.

5.5 What Affects the BPM/BPM+?

In this section, 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. 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 shows the intensity of the workload. 3) The indicator Stdev(RBL) is the weighted standard deviation of the 4/8 programs' RBL. This represents the difference of locality among programs. 4) Sum(BW)*Stdev(RBL) is the combination of the two indicator stated before. Figure 14 illustrates four curves, which represent the correlation between the improvements of BPM and the four indicators respectively. To fit them into one figure, we normalize the value of all the four indicators within range (0, 1). Besides, there are 6 points on each curve and each point represents the average of multi workloads, which have close indicator values. According to the figure 14, none of the indicators matches the improvement trend of BPM perfectly except the ***Sum(BW)*Stdev(RBL)*** – as the indicator increases, the improvements of BPM also increase steadily. This metric indicates that memory interference is positively correlated to memory access intensity and divergence of application' locality. The heavier the interference is in the system and the higher the potential row-buffer locality the application might exhibit, the more improvement BPM can achieve. We verify this indictor in most of the BPM+ effective cases and the experimental results show that the indictor works well in predicting the performance benefits from BPM+. This is because in the cases where bandwidth utilizations are in balance (or near balance), the primary reasons that lead to the performance gains in BPM and BPM+ are the same (i.e., eliminating the interference and improving the row-buffer locality).

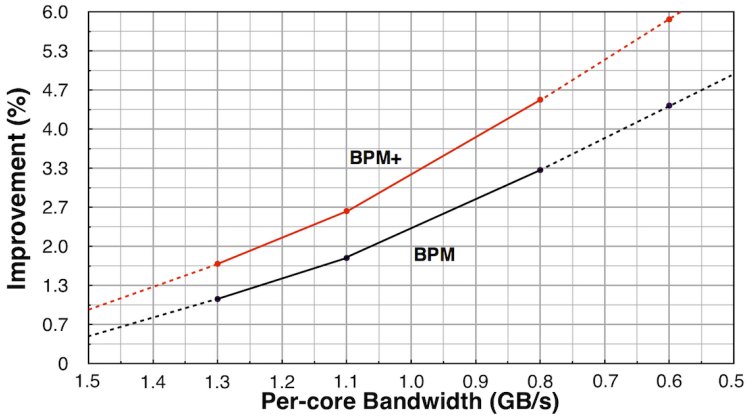


Figure 16. The correlation of BPM/BPM+ improvements and per-core bandwidth across 34 workloads on average.

5.6 Page-policy and Power

The open-page policy is designed to favor memory accesses to the same row of memory bank by keeping the row in row-buffer and maintaining a row of data for consecutive accesses, while the close-page policy is designed to favor random accesses by writing back (*precharge*) the data in row-buffer to DRAM bank after every time access. Usually, MCs aim to achieve good row-buffer locality, therefore open-page policy has better performance than close-page policy in general. But recent studies show that the row-buffer locality in multicore systems is sharply decreased to a lower level [K. Sudan et al, 2010; A. Udipi et al, 2010]. Therefore, in practice, some servers have to compromise to use the close-page policy. Our experiments show that BPM/BPM+ can revive the 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 15 shows that open-page with BPM/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 [N. Aggarwal et al, 2008; A. Udipi et al, 2010], because it has to move an entire row from array to a row-buffer. BPM/BPM+ can lower the power consumption of DRAM because of the reduced row buffer conflict miss rate (as illustrated in Figure 15). As mentioned in 4.1, we measure the power consumption by real hardware [Z. Cui et.al, 2011], so we can get the real value of power savings on memory system. Our experimental results show that BPM/BPM+ with open-page policy can save up to 5.2% of memory power consumption, better than the configurations without BPM/BPM+.

5.7 The Correlation between BPM/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 [S. Beamer et al, 2010; B. Rogers et al, 2009]. Since the number of cores is still increasing, memory bandwidth of each core is decreasing, causing more and more serious interference. To evaluate the influence of different per-core bandwidth upon BPM/BPM+, we emulate different bandwidth scenarios via varying memory frequency from 1333 to 800 MHz so that the per-core bandwidth decreases form 1.3GB/s to 0.8GB/s. This experiment included over 30 workloads, and we average them in different cases. Figure 16 illustrates the correlation of weighted speedup

improvements (denoted as improvement and shown in the vertical axis in Figure 16) and per-core bandwidth is negative: BPM/BPM+ performs better when per-core bandwidth is lower. 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 decreases, but the overall system throughput still improve from 4.1%/5.2% (4-programmed) to 5.3%/6.5% (8-programmed). Therefore, BPM and BPM+ are promising approaches for future many-core architecture that arguably has even less per-core bandwidth. This figure also shows that BPM+ achieves more performance gain over BPM (the BPM+ curve increases slightly sharper than the BPM curve along the pre-core bandwidth axis) as bandwidth contention becomes more severe. This is because BPM+ reduces the all-to-all contention among all applications in every channel.

5.8 Discussion

I/O storage: When the total memory requirement of any running program does not exceed the assigned memory capacity, both BPM and BPM+ could potentially bring significant performance benefits. However, once an application requires higher memory capacity than the system can offer, BPM and BPM+ will hurt the system performance since there are additional I/O transactions generated to access the swap region due to the limited memory space. We conducted experiments with benchmarks from SPEC2006 under limited memory capacity. The results show that once the I/O storage is used, the performances gains brought by BPM/BPM+ are offset by the long I/O transaction latency. Under such circumstances the overall improvement is only around 1% (significantly less than the 5% achieved in normal cases) and sometimes even below zero. Thus, we draw conclusion that a system should avoid such cases (incurring I/O storage) in practice when adopting the proposed techniques.

BLP might hurt BPM/BPM+: Another factor that impacts our mechanism is the bank-level parallelism (BLP). We found that if the 470.lbm benchmark is included in the workload, BPM and BPM+ will hurt the performance. Actually, as shown in Figure 1, 470.lbm suffers the most significant slowdown when the number of banks it can use is below 16. Specifically, when this benchmark is assigned with only 16 to 8 banks, its' performance lost is 10% to 40%, which is much more than other benchmarks (1%~7% on average). This is because 470.lbm exhibits higher bank-level parallelism than other benchmarks. Based on this study we devise a 10% rule as a guideline for using BPM/BPM+ appropriately: when the workload contains one or more benchmarks that suffer more than 10% performance lost with bank partitioning, our approach should be carefully used. In addition to the above factors, the unbalance factor can also affect BPM+, as we have discussed in the prior sections.

BPM+ orthogonally supports other policy: Although BPM+ confines a thread's memory accesses in a particular channel, we provide APIs for users to assign more memory resource to the thread in other channels. The APIs are useful and can be used to balance bandwidth utilizations across multiple channels when one channel's bandwidth utilization is significant higher than another. We have tried to use these APIs in some experiments (workload 16~20 in Figure 8) and find it difficult to always bring performance gains on our two-channel platform due to the following two reasons. First, channel re-balancing incurs channel-level interference as modern MCs do within multicore platforms (discussed before). Second, a thread's bandwidth requirement might not be appropriately switched to another channel due to the diverse memory patterns. However, it is useful for machines with 4- or more

channels, which allow a certain high-bandwidth thread to use more than one channel while still providing multiple channels for interference elimination through other channels. Furthermore, we believe that there is a design space for utilizing this API under different cases and we will study this topic in our future work.

Would channel-level partitioning penalize any threads with more bandwidth demands? It seems intuitively that the channel-level partitioning might over penalize a thread that could otherwise use multiple channels in the standard channel interleaving approach. However, from our experiments we observed some counter-intuitive yet explainable phenomenon. First, in multi-threaded programs, allowing each thread to use multiple channels will inevitably bring serious contention at both channel and bank level, offsetting the advantages of the channel-level interleaving in many cases. Second, we find that when many threads are running together, the average bandwidth that one thread will use does not increase significantly. Thus, we conclude that channel partitioning among threads may potentially benefit the overall performance in practice. To this end, a natural follow-up question is: Is channel partitioning always beneficial? To address this question, we introduce a new metric called unbalance factor to show whether or not our mechanism brings positive impact. In practice it is possible that certain applications may require much more memory bandwidth than others. Our dynamic system handles this by giving these applications more resources (channels or banks).

6. RELATED WORK

Partitioning in Memory Hierarchy: Both hardware based cache partitioning [G. E. Suh et al, 2004; M. K. Qureshi et al, 2006; H. S. Stone et al, 1992] and software page-coloring based cache partitioning [R. Azimi et al, 2009; S. Cho et al, 2006; J. Lin et al, 2008; J. Liedtke et al, 1997] are proposed to partition shared caches among threads to mitigate the inter-thread interference. DRAM bank partitioning was first proposed in [W. Mi et al, 2010], and then explored by several attempts [M. K. Jeong et al, 2012 and L.Liu et al, 2012]. The BPM [L.Liu et al, 2012] is the first effort that implements the bank partitioning in a real multicore system, and reveals the overlapped bits (O-bits) in physical address that index both LLC sets and DRAM banks. O-bits can be used to partition LLC and banks simultaneously. This motivates the work in [N.Suzuki et al, 2013] to develop a coordinated bank and cache coloring method. Additionally, the work in [Linux/RK] offers an open source kernel that also supports bank partitioning. The effort in [S. P. Muralidhara et al, 2011] first propose the channel partitioning approach, which maps data of different threads to different channels according to their memory access behaviors. However, they do not provide a thorough discussion upon the partitioning approach that combines the channel as well as DRAM bank at the same time, and the balance utilization issue among multiple channels. Additionally, this method is not implemented in a real machine.

MC Optimization: MCs are designed to distinguish memory access behavior at thread-level in [R. Iyer et al, 2007; Y. Kim et al, 2010; O. Mutlu et al, 2007 and 2008]. In these approaches, the scheduling policy can be adjusted at run time based on application characteristics. TCM [Y. Kim et al, 2010], aims to address fairness and throughput at the same time by dynamically grouping threads into two clusters (memory intensive and non-intensive) and assigning different scheduling policies to different groups. The effort in [Kyle J. Nesbit et al, 2006] designs a memory scheduler within MC based on network fair queuing scheduling algorithms (offer guaranteed service), improving both fairness and throughput. Moreover, the state-of-the-art approach MISE [L. Subramanian et al, 2013] estimates the application

slowdowns accurately by considering the rate of severed memory requests and their priorities in MC. Based on the estimation approaches are proposed to optimize memory system throughput and fairness. These methods need modifications to MC, and hence, in many cases, incur hardware design and manufacturing overheads.

Software-Hardware Cooperative Memory Scheduling Optimization in User-level: Scheduling algorithms DI and DIO proposed in [S. Zhuravlev et al, 2010] aimed to distribute threads to achieve an even distribution of miss rate among multiple caches to avoid severe contention on cache, MC, bus and prefetching hardware. Similar mechanisms are also proposed in [G. Dhiman et al, 2009, R. Knauerhase et al, 2008]. Additionally, some efforts in [X. Zhang et al, 2009 and E. Ebrahimi et al, 2010] employ thread execution throttling and memory resource utilization throttling approach to achieve high throughput or fairness. These user-level approaches often employ specific performance counters and can potentially be used in conjunction with BPM/BPM+ aim to alleviate memory interferences.

Row-buffer Optimizations: In [K. Sudan et al, 2010], frequently accessed data from different rows are dynamically migrated into the row buffer to improve row buffer usage and performance. Row buffer power consumption can be also saved due to reduced amount of the *precharge* and *active* operations. In [D. Kaseridis et al, 2011], the row-buffer is precharged upon every four accesses to reduce row-buffer conflicts. Moreover, it modifies the address mapping policy so that the memory access requests are better scattered across different banks. Similarly, the effort in [H. Park et al, 2013] attempts to randomize all the memory accesses through the coarse-grained page-level interleaving reducing the potential row-buffer conflicts.

Physical Page Management Optimization: A recent research effort [H. Park et al, 2013] observes that applications with random physical page accesses tend to outperform those with regular page access patterns in many cases. Because a randomized page access pattern might reduce inter-thread interferences on the row-buffer. Based on this, they propose an approach (M^3) that interleaves memory requests from the same application across all banks as randomly as possible. M^3 performs particularly well for multi-threaded workloads with streaming memory accesses. Their experiments show that for *sreamcluster* M^3 achieves above 15% performance improvement in 8-threaded case. However, M^3 is not always effective and may hurt the overall performance, as can be concluded from the reported results. Another recent work [R. Das et al, 2013] proposes application-to-core mapping policies to reduce memory interference in multicore platforms, and it also modifies the physical page allocation and replacement routine in operating system to enforce the application clustering and mapping. These efforts aim to reducing the memory interference via new physical page management mechanism in operating system.

Comparison with BPM/BPM+: The effort in [S. P. Muralidhara et al, 2011] proposes an application feature aware channel-level partitioning approach. Another similar attempt [M. K. Jeong et al, 2012] proposes a bank-level partitioning and a sub-ranking mechanism to eliminate the bank-level interferences and compensate for the reduced BLP by increasing the number of independent banks, respectively. Our work is quite different from the previous studies in the following two aspects. First, to the best of our knowledge, this is the first work that implements and evaluates both bank- and channel-level partitioning upon real machines. Second, our approach does not modify the hardware and can be applied to any Linux platforms. The approach in [H. Park et al, 2013] works well for some multi-threaded workloads but may brings minimal (around 1%) or even negative (down to -3%) performance in

some cases. Moreover, their work does not test multi-programmed workloads. In the contrast, BPM and BPM+ perform well for diverse workloads (both multi-threaded/programmed) in general and do not degrade system throughput in most of our experiments (except the workloads whose total memory requirements exceed the system's memory capacity). Moreover, our elastic mechanism supports diverse dynamic partitioning and scheduling policies that meet the requirements in practice.

7. CONCLUSION

We present bank-/channel-level partitioning mechanisms (BPM/BPM+) to mitigate the interferences among threads and improve the performance in multicore systems. BPM/BPM+ achieves this goal by assigning different group of banks to different threads to eliminate inter-thread memory interferences. The proposed approaches bring a considerable reduction of row buffer misses as well as the energy consumption of memory system. BPM/BPM+ is the first bank-/channel-level partitioning that is implemented on real machines and supports dynamic partitioning. Our experimental evaluations show that our approaches can improve system throughput and reduce unfairness due to the mitigating of interference among threads. We further show that BPM/BPM+ is effective in future many-core platforms with decreasing per-core bandwidth.

8. ACKNOWLEDGMENTS

We thank the reviews for their valuable feedback. Lei Liu, and Chengyong Wu are supported by 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; the National Natural Science Foundation of China under grants No. 60873057, 60921002, 60925009, 61033009 and 61202055.

9. REFERENCES

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

Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006

Linux/RK. https://rtml.ece.cmu.edu/redmine/projects/rk

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

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.

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

C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton Univ, Jan. 2008.

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

Z.Cui et al. A Fine-grained Component-level power measurement method. In PMP-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.

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

R. Das, R. Ausavarungnirun, O. Mutlu et al, Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In HPCA-2013.

E. Ebrahimi et al. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In ASPLOS-2010.

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

G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache memory. In Journal of Supercomputing, 28(1), 2004.

I. Hur and C. Lin. Memory Scheduling for Modern Microprocessors. ACM Transactions on Computer Systems, 25(4), December 2007.

R. Iyer et al. QoS policy and Architecture for Cache/Memory in CMP platforms. In SIGMETRICS-07, 2007.

C. J. Lee et al. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In MICRO-42, 2009.

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

Y. Kim et al. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple MCs. In HPCA-16, 2010.

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.

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

G. L. Yuan et al. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In MICRO-42, 2009.

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.

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

L. Liu et al. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In PACT-21, 2012.

O. Mutlu and T. Moscibroda. Parallelism-aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In ISCA-35, 2008.

T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of Memory Service in Multi-core Systems. In USENIX Security, 2007.

O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In MICRO-40, 2007.

W. Mi et al. Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors. In Proc. the 2010 IFIP Int'l Conf. NPC, Sep. 2010.

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

Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, James E. Smith: Fair Queuing Memory Systems. In MICRO-39, 2006

S. P. Muralidhara et al. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In Micro-44, 2011.

H. Park et al. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems. In ASPLOS-2013.

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.

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

S. Rixner et al. Memory Access Scheduling. In ISCA-27, 2000.

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

K. Sudan, et al. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware. In ASPLOS-2010.

N.Suzuki, et al. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In ICESS-2013.

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

L. Subramanian et al. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In HPCA-2013.

A. Udipi et al. Rethinking DRAM Design and Organization for Energy-constrained Multi-cores. In ISCA-2010.

X. Zhang, S. Dwarkadas, K. Shen. Hardware Execution Throttling for Multi-core Resource Management. In USENIX ATC-2009.

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.

S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS-2010.

Appendix: An example of Algorithm 1.
Platform: i3-2100T (Sandy Bridge); Memory: 2GB, one DIMM, two ranks
Reserve 1.5~2GB for address mapping test; Step 1, test all bits, the latencies are
Bit 3 ~ Bit 5, latency = 69 ns; Bit 6, latency = 71 ns; Bit 7, latency = 70 ns; Bit 8 ~ Bit 9, latency = 69 ns; Bit 10, latency = 71 ns; Bit 11, latency = 69 ns; Bit 12, latency = 71 ns; Bit 13, latency = 83 ns; Bit 14, latency = 84 ns; Bit 15, latency = 90 ns; Bit 16, latency = 83 ns; Bit 17, latency = 83 ns; Bit 18, latency = 84 ns; Bit 19, latency = 90 ns; Bit 20, latency = 84 ns; Bit 21~ Bit 28, latency = 98 ns.
Select 98ns as the first threshold, so bit 21~28 are classified to row address
Step 2, choose bit21 as row address, test remain bits, which is bit 3~20, latencies are
Bit 3 ~ Bit12, latency = 98 ns; Bit 13, latency = 83 ns; Bit 14, latency = 84 ns; Bit 15, latency = 90 ns; Bit 16 ~ Bit 18, latency = 84 ns; Bit 19, latency = 90 ns; Bit 20, latency = 84 ns.
Select 98 as the second threshold so bit 3~12 are classified to column address
Step 3, remaining bits are 13~20, test all combinations:
Bit 13 and Bit 14, latency = 84 ns; Bit 13 and Bit 15, latency = 90 ns; Bit 13 and Bit 16, latency = 84 ns; Bit 13 and Bit 17, latency = 98 ns; Bit 13 and Bit 18, latency = 84 ns; Bit 13 and Bit 19, latency = 90 ns; Bit 13 and Bit 20, latency = 84 ns; Bit 14 and Bit 15, latency = 92 ns; Bit 14 and Bit 16, latency = 83 ns; Bit 14 and Bit 17, latency = 83 ns; Bit 14 and Bit 18, latency = 98 ns; Bit 14 and Bit 19, latency = 92 ns; Bit 14 and Bit 20, latency = 83 ns; Bit 15 and Bit 16, latency = 92 ns; Bit 15 and Bit 17, latency = 92 ns; Bit 15 and Bit 18, latency = 92 ns; Bit 15 and Bit 19, latency = 98 ns; Bit 15 and Bit 20, latency = 92 ns; Bit 16 and Bit 17, latency = 83 ns; Bit 16 and Bit 18, latency = 84 ns; Bit 16 and Bit 19, latency = 92 ns; Bit 16 and Bit 20, latency = 98 ns; Bit 17 and Bit 18, latency = 84 ns; Bit 17 and Bit 19, latency = 93 ns; Bit 17 and Bit 20, latency = 83 ns; Bit 18 and Bit 19, latency = 93 ns; Bit 18 and Bit 20, latency = 83 ns; Bit 19 and Bit 20, latency = 93 ns.
Select 98ns as the third threshold, so XOR bits are
Bit13 xor Bit17; Bit14 xor Bit18; Bit15 xor Bit19; Bit16 xor Bit20