Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors

Wei Mi, Xiaobing Feng, Jingling Xue, Yaocang Jia

Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors

Wei Mi[1,2], Xiaobing Feng[1], Jingling Xue[3], Yaocang Jia[1,2],

[1] Key Laboratory of Computer System and Architecture. Institution of Computing Technology
[2]Graduate University of Chinese Academy of Sciences
[3]School of Computer Science and Engineering. University of New South Wales
[1]{miwei, fxb, jiayaocang}@ict.ac.cn, [3]jingling@cse.unsw.edu.au

Abstract. DRAM row buffer conflicts can increase the memory access latency significantly for single-threaded applications. In a chip multiprocessor system, multiple applications competing for DRAM will suffer additional row buffer conflicts due to interthread interference. This paper presents a new hardware and software cooperative DRAM bank partitioning method that combines page coloring and XOR cache mapping to evaluate the benefit potential of reducing interthread interference. Using SPECfp2000 as our benchmarks, our simulation results show that our scheme can boost the performance of the most benchmark combinations tested, with the speedups of up to 13%, 14% and 8.06% observed for two cores (with 16 banks), two cores (with 32 banks) and four cores (with 32 banks).

Keywords: Row Buffer Locality, Cache Locality, Address Mapping.

1 Introduction

The DRAM memory system is a critical shared resource among multiple cores in a chip multiprocessor system. In a multi-programmed workload, multiple applications (i.e., threads) competing for DRAM will impede each other's progress due to interthread interference. Accesses from one thread can cause row buffer conflicts, bank conflicts and data/address bus conflicts to accesses from other threads. Therefore, uncontrolled interthread interference can significantly degrade overall system performance.

DRAM row buffer conflicts, i.e. row misses occur when a sequence of DRAM accesses to different rows go to the same DRAM bank, causing much higher access latency than row hits. For a single-core system, some bitwise XOR address mapping schemes [1][2] and memory scheduling policies like FR_FCFS (First Ready First-Come-First-Serve) [3] can be employed by the memory controller to reduce row buffer conflicts. In addition, bitwise XOR cache mapping schemes [4][5] can also be incorporated into the last-level cache to reduce both row buffer conflicts and last-level cache misses [1].

For a multi-core system, existing techniques for reducing row buffer conflicts appear to be all hardware-based, focusing mostly on improving memory scheduling

Supported by the National Basic Research 973 Program of China under Grant No.2005CB321602, and the National Natural Science Foundation of China under Grant No.60736012.

policies [6][7][8][9][10][11][12]. By leveraging the solutions for single cores [1][2][3], these hardware-based schemes attempt to optimize a multitude of objectives, including row buffer utilization, memory efficiency, DRAM throughput, fairness and QoS. But they suffer from a limited scope (when done purely in hardware) and are constrained by the conflicting nature of multiple objectives. In addition, the row latency reduction techniques through improving row hit rates become less effective due to the increased bank contention [6].

In this paper, we address the problem of reducing intrathread and interthread row buffer conflicts for running multi-programmed workloads on multi-core systems that keep the last-level cache private to each core (e.g., AMD Athlon). We introduce for the first time a static software-hardware cooperative DRAM bank partitioning scheme to reduce both kinds of conflicts, thereby improving overall system performance. By partitioning the DRAM banks among the applications, interthread interference is reduced or controlled in a more deterministic manner. This allows potentially existing techniques [6][7][8][9][10][11][12] to be applied more effectively. We have evaluated our scheme using the SPECfp2000 benchmarks on two- and four-core systems by using the cycle-accurate x86 full system CMP simulator FeS2 [13] (Section 5). Our preliminary results show that our scheme can boost the performance of the most benchmark combinations experimented with, with the speedups of up to 13%, 14% and 8.06% observed for two cores (with 16 banks), two cores (with 32 banks) and four cores (with 32 banks). respectively.

2 Background and Motivation

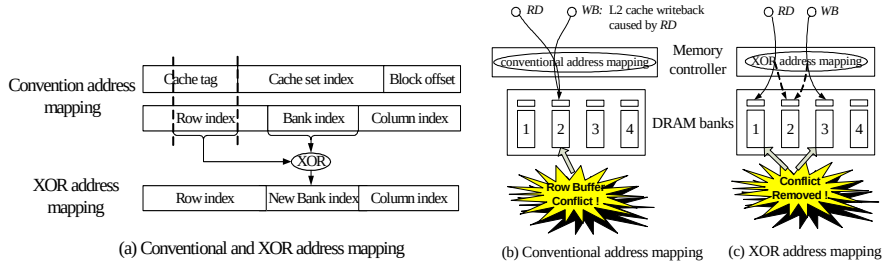
An SDRAM system consists of multiple banks that can be accessed in parallel. Each DRAM bank is a DRAM cell array organized in rows and columns. Each bank contains a row buffer used to cache the data in the most recently accessed row. The row buffer size is usually 2/4KB (i.e., one-half page/one page). The latency of a memory access can vary greatly depending on whether the access is a row (buffer) hit or miss. Normally, a DRAM access for a bank is realized in three stages, *precharge, row access and column access* [14]. There are two modes for DRAM accesses: *open-page* and *close-page*. In the open-page mode, if the next access to the same bank goes to the same row (a *row hit*), only column access is necessary. If the next access is a *row miss* (*row buffer conflict*), however, the precharge does not start until after the request has arrived. The close-page mode allows the precharge to start immediately after the current access.

Like the prior work on improving row buffer locality, this work can be beneficially applied when the open-page mode is used. In this case, when an access is a row hit, the data required is already cached in the row buffer. The data can be directly operated on in the row buffer without the precharge and row access operations, reducing the DRAM access time by half or more compared to when an access is a row miss [8][12][14].

For a single-core system, a conventional address mapping scheme allocates consecutive data blocks to consecutive memory banks using a modular mapping function, i.e., memory interleaving. Zhang et al. [1] found that the resulting address mapping symmetry between the last-level L2 cache and DRAM is a significant source

of row buffer conflicts. Such symmetry refers to the fact that the bank index bits are usually part of the L2 cache set index bits. As a result, L2 cache conflicts or writebacks usually lead to row buffer conflicts. They proposed to use an XOR address mapping scheme implemented in the memory controller to break the address mapping symmetry. By XORing the bank index bits and a portion of cache set index bits, as shown in Figure 1(a), the data blocks are permuted (or distributed evenly) across the memory banks. Without the XOR address mapping, the read *RD* and the L2 cache writeback *WB* caused by *RD*, as shown in Figure 1(b), will result in a row miss because they have the same L2 cache set index. In addition to address mapping, memory scheduling policies [3][6][7][8][9][10][11] are also effective in reducing row buffer conflicts by prioritizing row hit accesses over others. Furthermore, the XOR cache mapping schemes [4][5] can also reduce simultaneously both row buffer conflicts and the last-level cache misses [1].

This work is the first to apply page coloring in combination with a bitwise XOR cache mapping scheme to manage DRAM bank partitioning to reduce both intrathread and interthread row buffer conflicts while not increasing bank conflicts unduly. Our software-hardware cooperative scheme can improve row buffer locality effectively for a multi-programmed workload and may potentially enable existing techniques [7][8][9][10][11][12] to better optimize other objectives such as fairness and QoS in future.



(a) Conventional and XOR address mapping (b) Conventional address mapping (c) XOR address mapping

Fig. 1. Memory controller address mapping

3 DRAM Bank Partitioning

Our proposed approach is to use page coloring to map the DRAM accesses from different applications to different banks (if possible) in order to reduce interthread row buffer conflicts. Simultaneously, we also apply a bitwise XOR cache mapping scheme at the last-level, i.e., L2 cache to reduce intrathread row buffer conflicts and cache misses.

3.1 Page-Coloring-based Bank Partitioning

We use the classic OS page-coloring [15] to partition the DRAM banks for a multi-programmed workload. In the OS page coloring, all physical pages are divided into groups with all pages in the same group being labeled a distinct color. When a new physical page is requested by an application, the OS will allocate a page whose color

is in the set of colors assigned to the application. If an application has no colors in common with other applications, and in addition, if different colored pages are mapped to different DRAM banks, then there are no row buffer conflicts among the accesses from different applications. As a result, all interthread row buffer conflicts are avoided. For two applications that share some pages and thus some banks, their interthread row buffer conflicts can be reduced if the number of their shared pages, i.e., banks can be reduced. Figure 2 illustrates the page-coloring-induced bank partitioning for a 16-bank DRAM. The number of colors available is determined by the number of the bank index bits that are not part of the page offset.

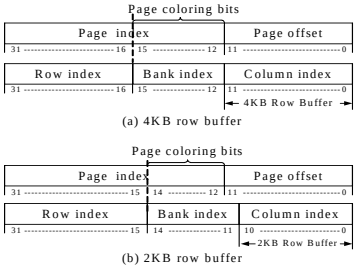


Fig. 2. Page coloring for partitioning a 16-bank memory

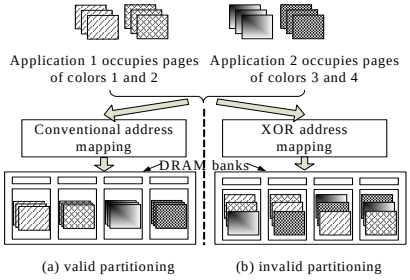


Fig. 3. Conventional and XOR address mapping

3.2 Bitwise XOR L2 Cache Mapping

Once the banks are partitioned across the applications, interthread row buffer conflicts are reduced. How do we also reduce intrathread row buffer conflicts? How do we also improve cache utilization since the last-level cache is also partitioned unexpected by page coloring? Figure 3 illustrates why conventional and XOR address mapping schemes implemented in the memory controller are not helpful in reducing intrathread row buffer conflicts in the presence of page coloring. As shown in Figure 3(a), the conventional address mapping allocates consecutive data blocks to consecutive DRAM banks so that different colored pages are mapped to different DRAM banks. Its main advantage is that the static bank partitioning results obtained by page coloring are preserved but its main drawback is that the address mapping symmetry problem discovered in [1] is prevalent. On the other hand, as shown in Figure 3(b), the XOR address mapping scheme [1] suffers from the opposite problem: it can break the address mapping symmetry and thus reduce row buffer conflicts significantly, but it reshuffles colored pages to banks and thus disables the partitioning effect done by page coloring.

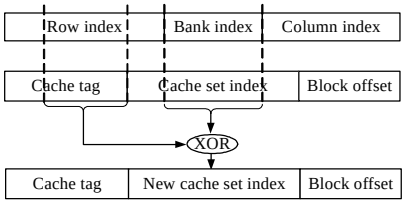
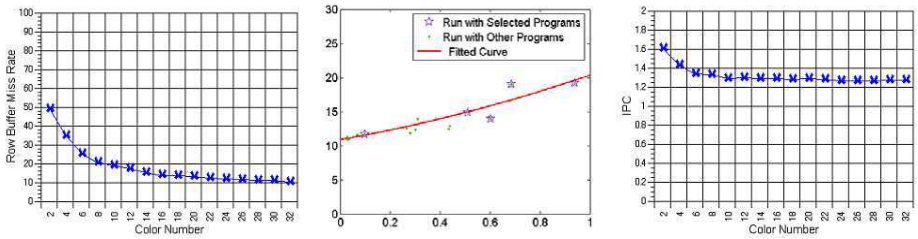


Fig. 4. Bitwise XOR cache mapping

What we want is a solution that breaks the address mapping symmetry while also preserving the static bank partitioning results achieved by page coloring. When we partition DRAM via page coloring, the private last-level L2 cache in each core is also partitioned at the same time. In this paper, we propose to use a bitwise XOR cache mapping scheme to redistribute the cache accesses from otherwise the restricted part of the L2 cache to the whole cache so as to remove the cache partitioning effect. As studied earlier [1], the XOR cache mapping is capable of not only reducing cache misses but also breaking the address mapping symmetry (as effectively as the XOR address mapping in the memory controller, in general).

Figure 4 illustrates the XOR cache mapping scheme used together with page coloring. Some bits of the cache set index and an equal number of lower-order bits in the cache tag are XORed to produce a new cache set index. In addition, the bits thus modified in the cache set index correspond exactly to the bits of the DRAM bank index. In Figure 1, the read *RD* and the L2 cache writeback *WB* caused by *RD* are necessarily mapped to the same bank without the XOR address mapping in the memory controller because their bank indexes are part of their L2 cache set indexes, which happen to be identical. In Figure 4, *RD* and *WB* should have the same new cache set index because *WB* is caused by *RD*. Therefore, *RD* and *WB must* have different cache set indexes and thus different bank indexes before the XOR cache mapping operation is applied. This means that *RD* and *WB* are mapped to different banks, giving rise to no row buffer conflict. Therefore, we can use the XOR cache mapping at the last-level cache together with the conventional address mapping in the memory controller to substitute for the XOR address mapping employed in the memory controller [1]. This solution has no impact on the mapping of colored pages to the DRAM banks achieved by page coloring.

The XOR cache mapping requires several XOR operations to obtain a new L2 cache index. Since all the XOR operations can be done in parallel, the extra delay incurred is one XOR gate. Depending on implementations, the XOR gate may or may not be on the critical path [4][5].



(a) MRC (b) SRC (c) BCC

Fig. 5. *MRC*, *SRC* and *BCC* of fma3d for a 32-bank DRAM (with 32 page colors when its row buffer is one page)

4 Cost Model and Partitioning Algorithm

We give an algorithm that determines statically which colors are allocated to which application in a multi-programmed workload. An assignment of colors to the

applications in a workload dictates the placement of these applications' data across the DRAM banks. To this end, we must first build a cost model. When we give fewer colors to an application to minimize the number of banks it has in common with other applications, its interthread row buffer conflicts may decrease but its intrathread row buffer conflicts may increase. Therefore, we need to minimize both kinds of conflicts for all applications simultaneously. In addition, we need to make sure that each application has at least a certain number of banks in order to maintain its bank parallelism. As a result, our algorithm will strive to minimize both kinds of conflicts while not increasing bank conflicts, i.e., reducing bank parallelism unnecessarily.

4.1 Cost Model

We use a row buffer miss rate curve (MRC) to estimate the intrathread row buffer conflicts of an application. The MRC of an application, which is obtained by profiling, gives its row buffer miss rate as a function of the number of colors assigned to it. The MRC of an application represents its demand for the DRAM bank space. As an example, Figure 5(a) plots the miss rate curve of fma3d.

We use a so-called *sensitivity rate curve* (SRC) to characterize the interthread row buffer conflicts of an application, which is defined in terms of the *politeness* and *robustness* associated with that application. The politeness of an application shows to what extent the application affects the row buffer miss rates of other applications. In general, as the L2 cache miss rate of an application increases, its politeness often tends to worsen. So we use empirically the L2 cache miss rate of an application as its politeness. The higher the L2 cache miss rate of an application is, the less its politeness is. The robustness of an application shows to what extent the row buffer miss rate of the application is affected by other applications. The robustness of an application is represented as its row buffer miss rate when run together with its co-runners. The SRC of an application allows us to find its robustness as a function of the (average) politeness of its co-runners treated as a whole. For example, let us compute the SRC for fma3d for a workload consisting of all SPECfp2000 benchmarks. We select five representative applications, vpr, mgrid, equake, fma3d and swim, as a standard application group. Their politeness values are ranked from lowest to highest in that order. We run fma3d with each of these applications as a co-runner, obtain five row buffer miss rate and politeness pairs, and finally, use these five pairs to fit a quadratic polynomial curve to obtain t he SRC of fma3d, which is plotted in Figure 5(b), to represent its robustness. Given the MRC and SRC of each application, we can estimate the overall row buffer miss rate for a multi-programmed workload. Let there be M colors and N applications. The MRC and SRC of application i are denoted by MRC_i and SRC_i, respectively. Suppose application i gets m_i colors with m_{ij} common colors with application j in a bank partitioning. The L2 cache miss rate of application i is denoted by cmr_i. We estimate the *Row Buffer Miss (RBM) rate* of application i, denoted RBM_i, in this bank partitioning as follows:

$$RBM_i = \underbrace{MRC_i(m_i)}_{①} + \underbrace{\sum_{j\in(1,N),\, j\neq i} ((SRC_i(cmr_j) - MRC_i(M)) * m_{ij} / m_i)}_{②} \quad \textbf{(1)}$$

where ① represents the number of intrathread row misses of application *i* and ② represents the number of interthread row misses of application *i* caused when it is run together with all other applications. In ②, subtracting $MRC_i(M)$ (the row buffer miss rate of application *i* when it gets all banks) from $SRC_i(cmr_j)$ (the row buffer miss rate of application *i* when it gets m_i banks) gives the number of interthread row misses introduced to application *i* by application *j*, after it is scaled proportionally by m_{ij}/m_i. Presently, we model only the number of overlapping colors, between two applications but do not differentiate exactly how their colors, i.e., data banks are laid out in DRAM).

To minimize bank conflicts for an application, we also use a *Bank Conflicting Curve* (BCC) to ensure that the number of colors an application gets does not drop below a certain threshold. The BCC of an application gives its performance (measured in terms of its IPC) as a function of the number of colors assigned to it in the close-page mode. Therefore, the BCC of an application represents the effect of bank conflicts (without row buffer conflicts) on performance when its color count is varied. We use the BCC of an application to find a bank conflict turning point, a threshold that represents the minimum number of colors that should be allocated to the application to guarantee its bank parallelism. Figure 5(c) plots the BCC of fma3d with its bank conflict turning point being 6 (assuming 32 colors).

4.2 Bank Partitioning Algorithm

Based on our cost model, we have developed a bank partitioning algorithm for a multi-programmed workload consisting of *N* co-running applications by assigning each application with some colors from a set of *M* colors available. Since the search space consisting of all possible color assignments is huge, Figure 6 gives a hill-climbing searching algorithm, ColorMap, for finding a feasible solution.

Hill-climbing has a shortcoming of getting stuck easily at a local optimum. We alleviate this problem by populating *Candidate_Set* with all possible candidates $C = (m_1, \ldots, m_N)$ such that its *i*-th element, m_i, represents the number of colors assigned to application *i*, which ranges from the bank conflict turning point given by its *BCC* to *M* (line 23). We make use of the following notation to represent the overall row buffer miss rate for all applications for the current color count assignment specified in an *N*-vector *C*:

$$RBM(C) = \sum_{1 \le k \le N} RBM_k \qquad (2)$$

Note again that our cost model is simple since it ignores the actual colors assigned to a particular application.

The while loop in *ColorMap* processes all candidates in *Candidate_Set*, one at a time. Given a *candidate*, which initially indicates *only* the number of colors assigned to each application (lines 2 and 3), *Find_Local_Optimum* aims to find an assignment of actual colors to each application. In addition, an application may get more colors than initially indicated in *candidate* if doing so will improve overall row buffer locality. This is achieved by a hill-climbing process. Initially, every application is initialized with some colors to start with (line 4). Then *Find_Local_Optimum* chooses an application, selects a new color and assigns it to the application in each hill

climbing step (lines 6 – 16). Of all possible choices made in each hill climbing step, the one that minimizes the overall row buffer miss rate of the entire workload is taken (lines 11 -- 13). So the steepest ascent direction is always preferred. If this is not possible (lines 14 and 15), we check to see if we have at least allocated the minimum number of colors to each application as initially indicated in *candidate* (line 17). If this is the case, we are done. Otherwise, we assign a new color randomly to each application whose number of assigned colors is still less than its minimum specified in *candidate* (lines 18 and 19). This enables the hill-climbing process to be started again. Finally, *Save_Best_Color_Layout* reveals the best solution found.

```
1 Find_Local_Optimum (candidate)
2   Let ColorCount = candidate = (m1, ⋯ , mn);
3   Let ColorSet = ({}, ⋯ , {});
4   Initially, assign some colors to all applications such that (1) |ColorSeti| < mi
              and  (2) ColorSeti and ColorSetj are mutually disjoint;
5   while (there exists an application, i, such that |ColorSeti|  < mi) do
6     while (true)
7       for (every pair of application i and color j such that !(j ∈ ColorSeti) ) do
8         ColorSet(i,j) = ColorSet;  Add j to ColorSet(i,j)i;
9         ColorCount(i,j) = ColorCount;  ColorCount(i,j)i++;
10      endfor
11      Let ColorCount(x,y) such that RBM(ColorCount(x,y) is the smallest;
12      if (RBM(ColorCount(x,y)) < RMB(ColorCount))
13        ColorSet = ColorSet(x,y); ColorCount = ColorCount(x,y);
14      else
15        break;
16    endwhile
17    for (every application i such that |ColorSeti|  < mi) do
18      Assign  a new color ci randomly to ColorSeti ;
19      ColorCounti++;
20    endfor
21  endwhile

22 ColorMap () {
23   Candidate_Set = { (m1, ⋯ , mn)  | BCC_MINi  = mi = M }
24  while (there is an unprocessed candidate in Candidate_Set) do
25      Find_Local_Optimum(candidate)
26      Save_Best_Color_Layout ()
27 endwhile
```

Fig. 6. A DRAM bank partitioning algorithm

5 Evaluation Methodology

We evaluate our work using the cycle-accurate x86 full system CMP simulator FeS2 [13], which is based on the Simics virtual machine [16]. The page coloring algorithm is implemented on Linux 2.6.26. The memory system is modeled using DRAMsim-v1.2 simulator [17]. Table 1(a) shows the major processor and DRAM parameters. By default, the row buffer is one page of the size 4KB. We use all 14 SPECfp2000 benchmarks for evaluation. Each benchmark is compiled using gcc-4.1.2 with "-O3" optimizations and run for 100 million instructions.

6 Experimental Results

We use the three acronyms, *CC*, *CX* and *XC*, to represent three address mapping schemes listed in Table 1(b). We start by applying page coloring to vary the color count of an application and evaluate their row buffer miss rates and L2 cache miss rates with a single core to see which scheme combines the best with page-coloring-based DRAM bank partitioning. (Scheme *"XX"* cancels out each other's partitioning effect and is thus omitted.) Then we evaluate our bank partitioning mechanism "XC+Page Coloring" and our partitioning algorithm with a large number of benchmark combinations running on two- and four-core systems.

Table 1. Experiment parameters

Processor Pipeline	3GHz, 80 entries instruction window, 4 issues
L1 Caches	32K per core, 8 ways, 64B block, write through, 1cycle latency
L2 Caches	Private L2 Caches,1M per core, 8 ways, 64B block, write back, 8/9 cycle latency
Page and Mem Size	4KB page size, 4G memory
DRAM Controller	FR-FCFS scheduler, 32 entries request buffer, open-page mode
DRAM Configuration	Micron DDR2-667MHz [22], 16/32 banks, 4KB row buffer per bank, Row access to column access (15 ns), Column access(15ns), precharge (15ns), burst (10 ns) Row buffer hit time (40 ns), row buffer closed (60 ns) Row buffer conflict (80 ns)

(a) Processor and memory parameters

CC → Conventional cache mapping + Conventional address mapping
CX → Conventional cache mapping + XOR address mapping
XC → XOR cache mapping + Conventional address mapping

(b) Three address mapping schemes

6.1 CC, CX or XC + Page Coloring on Single Cores

We evaluate *CC*, *CX* and *XC* when each combined with page coloring for a 16-bank DRAM for running a benchmark on one core. Since the row buffer is one 4KB page (Table 1(a)), there are 16 colors. Figure 7 and Figure 8 compare their row buffer and L2 cache miss rates as the number of colors that a benchmark gets changes. In Figure 7, we see that *CX* exhibits lower row buffer miss rates than *XC* in most of the benchmarks. But the gaps between the two become smaller as the number of colors assigned to a benchmark increases. This is because *CX* disables the bank partitioning effect of page coloring but *XC* keeps it. When given the same number of colors, *CX* can use more banks than *XC* does by reshuffling (evenly or randomly) the colored pages allocated to a benchmark to all the banks. In addition, we can also see that *XC* has lower row buffer miss rates than *CC* for most of the benchmarks, because *XC* can break the address mapping symmetry but *CC* cannot. As can be observed in Figure 8, *CX* exhibits significantly higher L2 cache misses than *XC* for several benchmarks, such as art, ammp, galgel and mgrid. This is because *XC* can eliminate the cache partitioning effect of page coloring but *CX* cannot, implying that *XC* can ultilize more cache space than *CX*. When an application gets all 16 colors, .i.e., all the banks, *CX*

and *XC* have nearly the same row buffer miss rate and L2 cache miss rate, a result also confirmed earlier in [1]. So *XC* and page coloring represent a good combination for DRAM bank partitioning.

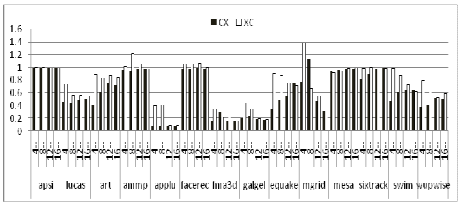


Fig. 7. Row buffer miss rates of *CX* and *XC*

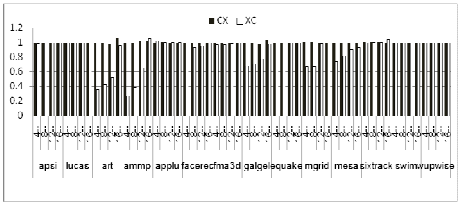


Fig. 8. L2 cache miss rates of *CX* and *XC*

6.2 XC + Page Coloring on Multiple Cores

6.2.1 Two Cores

We evaluate the effectiveness of our software-hardware cooperative bank partitioning mechanism and our partitioning algorithm in improving the overall row buffer locality, L2 cache hit rate and overall system performance when running a pair of applications on a two-core system. Recall that the row buffer is a 4KB page, resulting in a total of 16 colors in total. One color controls the placement of one page in a DRAM bank. We first consider a 16-bank DRAM and then move to a 32-bank DRAM.

We consider all 105 pair-wise benchmark combinations and present our results in Figure 9. For all these pairs, the speedups, regardless whether they are positive or negative, correlate well to their row buffer miss rates, indicating the significance of row buffer locality optimization techniques on boosting overall program performance. The speedup values for a total of 105*2=210 benchmark executions are divided into the following six intervals:

Speedup	>5%	1%~5%	0%~1%	-1%~0%	-5%~-1%	<-5%
Total	30	58	61	46	15	0

Looking at the number of benchmarks falling into (-∞, <-5%) and (>5%, +∞), we find that the benchmarks with the largest speedups significantly outnumber those with the worst slowdowns. Some small performance degradations are observed in some benchmarks because their row buffer and/or L2 cache miss rates are made slightly worse by "XC+Page Coloring". Our cost model can be crude when estimating the row buffer conflicts for an application when it is run together with its co-runners. For example, when the *SRC* of an application over-approximates its interthread conflicts with other applications, our partitioning algorithm will usually not allocate enough banks to the application, causing its intrathread conflicts to increase. The L2 cache miss rates for some applications are often difficult to estimate statically. In Figure 9, the benchmarks with decreased and increased L2 cache misses are nearly equally divided (with a ratio 117:92). In addition, the benchmark executions with performance slowdowns of <-1% when their L2 cache miss rates are increased to > 1% total only 7 among all 210 benchmark executions. This seems to suggest that the impact of L2 cache misses on performance is less pronounced than that of row buffer misses.

We have also evaluated the same 105 pairwise workloads for a 32-bank DRAM and plot the results similarly in Figure 10, which displays similar trends as Figure 9 for the same performance metrics evaluated. In Figure 10, the 210 benchmark executions again fall into the following six intervals according to their speedups:

Speedup	>5%	1%~5%	0%~1%	-1%~0%	-5%~-1%	<-5%
Total	31	65	55	27	30	2

In comparison with the table for 16 banks, there are more benchmarks falling into (1%~5%) ∪ (-5%~-1%) but fewer into (0%~1%) ∪ (-1~0%). When there are many colors (32 rather than 16 banks), our partitioning algorithm has produced fewer pairs that share equally the 32 banks. There is another reason for the existence of more benchmarks in (-5%~-1%) with 32 banks. The SRC functions of some applications may happen to be less accurate when more banks are available. The two benchmarks with the largest performance slowdowns were caused by this reason. Better performance results are expected when more accurate cost modes and partitioning algorithms are used.

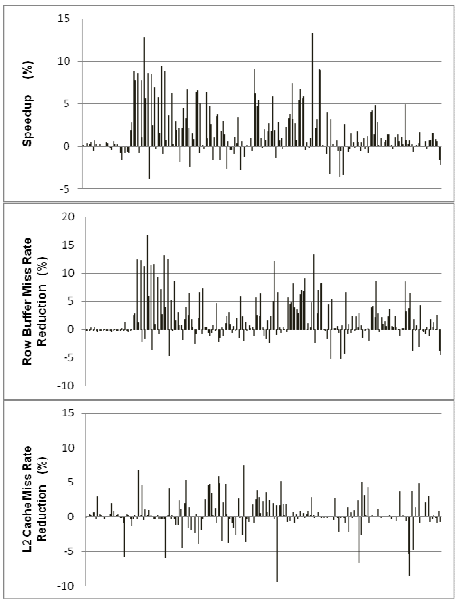


Fig. 9. Results for 105 pairs of benchmarks (16 banks)

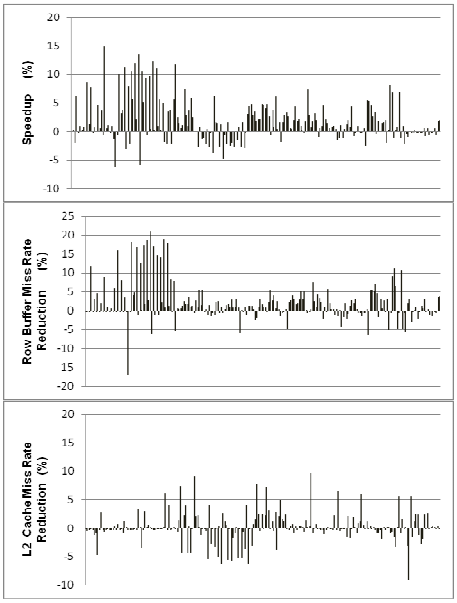


Fig. 10. Results for 105 pairs of benchmarks (32 banks)

As for L2 cache misses, their increases or decreases can now impact more on program performance when the corresponding row buffer miss rates do not change much, as shown in Figure 10. As in the 16-bank case, the benchmarks with increased and decreased L2 cache misses are nearly evenly divided (with a ratio of 98:112). The number of benchmark executions with negative speedups of <-1% when their L2 cache miss rates are > 1% is 17 for all 210 benchmark executions. Overall, although some more benchmarks have suffered performance slowdowns, but their negative speedups are small, particularly when compared with the positive speedup cases.

6.2.2 Four Cores

A total of 10 benchmark combinations are simulated with and without DRAM bank partitioning. Their partitioning strategies are listed in Table 2. The performance results are plotted in Figure 11. The negative effects of increased L2 cache miss rates on performance in some benchmarks are offset by the benefits obtained from significant row buffer miss rate reductions. For the 10 groups used, the speedups of up to 8.22% are observed and the average speedup across these groups is 2.89%.

Table 2. benchmark combinations on four cores

	Benchmark Groups	Color mask bits
Mix1	lucas, applu, facerec, mgrid	0xdffff83f: 0xcf89001c: 0x20000100: 0x1070fee3
Mix2	lucas, art, ammp, facerec	0xefff877f: 0x4007ff00: 0xffffffff: 0x10000080
Mix3	ammp, galgel, equake, swim	0xffffffff: 0xef811ff0: 0xb1fee00f: 0xffffffff
Mix4	applu, facerec, mgrid, wupwise	0x8e800f4f: 0x40001000: 0x317fe0b0: 0xbfff2fff
Mix5	applu, fma3d, galgel, equake	0xa1fe0000: 0x4e006f3c: 0xb1ff90c3: 0x5801f3ff
Mix6	applu, facerec, fma3d, sixtrack	0xa9c06780: 0x40000040: 0xbe7ff83f: 0x11be1fbf
Mix7	facerec, galgel, equake, sixtrack	0x80000040: 0x6f907f07: 0x786f80bf: 0x17f87fb8
Mix8	lucas, ammp, facerec, mesa	0xcff9fe7f: 0xffffffff: 0x20020000: 0x10040180
Mix9	facerec, equake, mgrid, sixtrack	0x80000400: 0x5ff8007f: 0x20078380: 0x3e0ffbf0
Mix10	applu, mgrid, mesa, sixtrack	0x8e30030f: 0x51cfec30: 0x200010c0: 0x31f01fff

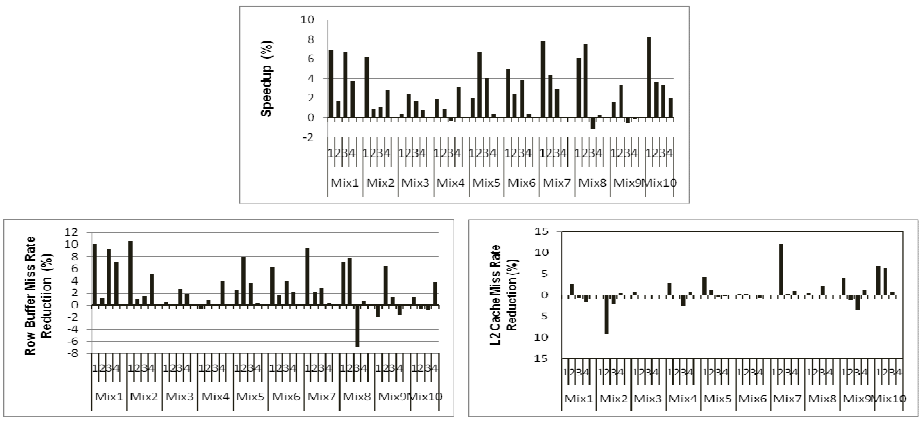


Fig. 11. Results of 10 selected workloads on 4 cores (32 banks)

6.2.3 One-Half Page Row Buffer

We have repeated all our two- and four-core simulations for a 32-bank DRAM when the row buffer size is one-half page. In this case, one color determines the place of one physical page across two consecutive DRAM banks.

For the two-core case, the speedup distributions can be observed from the following table:

Speedup	>5%	1%~5%	0%~1%	-1%~0%	-5%~-1%	<-5%
Total	36	69	36	38	26	4

These statistics are similar to those produced for a 32-bank DRAM when the row buffer is one page. For the four-core case, we have simulated the same 10 mixes listed in Table 2. The speedups of up to 9.7% are observed and the average speedup across

all the benchmarks is 2.4%. These results demonstrate that the proposed bank partitioning scheme appears to work well for different row buffer sizes (relative to a fixed page size in a system).

7 Related Work

None of prior work about DRAM access optimizations consider to reduce row buffer conflicts for Chip Multiprocessor via software. To our knowledge, this paper is the first to propose a software-hardware cooperative DRAM bank partitioning mechanism for reducing interthread and intrathread row buffer conflicts.

Rixner et al. [3] examine various DRAM access scheduling policies and propose the FR-FCFS policies. Hur and Lin [18] introduce adaptive history-based scheduling policies to minimize the average DRAM access delay and to balance the ratio between reads and writes from the processor. Shao and Brian [19] describe a burst access scheduling mechanism to maximize the data bus utilization by read preemption and write piggybacking. Lee et al. [20] suggest to use a prefetching-aware DRAM controller to adaptively prioritize between conventional demand and prefetching operations. Zhang et al. [1] expose an important source of row buffer conflicts in single cores and propose a bitwise XOR address mapping scheme to reduce these conflict significantly. These research efforts focus on the problems for single cores but some of the principles proposed such as FR-FCFS scheduling policies and XOR address mapping are also useful for multi- cores.

For a multi-core system, DRAM becomes a major shared resource. The memory controller needs to optimize memory performance by considering a variety of factors simultaneously such as row buffer competition, data bus competition, memory efficiency, the fairness and QoS, but its only means is to control the priorities of the accesses from each thread. So the memory controller needs to weight these factors. Nesbit et al. [9] use a network-fairing-queue based scheduler to provide thread fairness and QoS and a First-Ready Virtual Finish Time First policy (FR-VFTF) to balance row buffer utilization and fairness. Rafique et al. [10] use virtual start time fair queuing instead of virtual finish time and improve fairness based on Nesbit's work. Mutlu and Moscibroda [11] show the stall time of a thread is a more direct indicator of fairness and propose a stall time fair scheduler. These research efforts are mainly concerned with balancing row buffer competition and thread fairness. Zheng et al. [7] considers a scheduler that combines two factors about memory efficiency and the pending request number of each request to improve system throughput. Mutlu and Moscibroda [8] introduce parallelism-aware batch scheduling to provide a substrate for bank parallelism, row buffer utilization, fairness and QoS, and their work cleverly adjust the degree of inclining to any factor via controlling the size of a batch. Due to the complexity of considering so many performance factors, ipek et al. [12] suggest to add a reinforcement learning mechanism to a DRAM access scheduler. In comparison with the prior work, our work is the first to reduce interthread row buffer conflicts via DRAM partitioning through software-hardware cooperation. It represents an orthogonal means to improving row buffer locality for multi-programmed workloads. As a result, our mechanism may enhance existing techniques by enabling them to focus on optimizing other factors.

8 Conclusion

In order to reduce DRAM row buffer conflicts for multi-core systems, we present a software-hardware cooperative scheme to realize a static DRAM bank partitioning. We apply page coloring to map different applications to different partitioned physical spaces and propose to use a bitwise XOR cache mapping scheme together with page coloring to reduce both intrathread row buffer conflicts and last-level cache misses. In order to determine a DRAM bank partitioning strategy, we discuss how to build a cost model and develop a heuristics-based algorithm for partitioning the DRAM banks among the applications in a multi-programmed workload. Our simulations demonstrate that DRAM bank partitioning can significantly reduce DRAM row buffer miss rates and achieve speedups for two- and four-core systems. In the future, we plan to study how to tackle the DRAM bank partitioning and shared cache partitioning simultaneously.

References

1 Zhang, Z, Zhu, Z, Zhang, X. 2001: Breaking Address Mapping Symmetry at Multi-levels of Memory Heirarchy to Reduce DRAM Row-buffer Conflicts. The Journal of Instruction-Level Parallelism, vol. 3, (October, 2001)

2 Lin, W. 2001. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In Proceedings of the 7th international Symposium on High-Performance Computer Architecture (January 20 - 24, 2001). HPCA. IEEE Computer Society, Washington, DC, 301.

3 Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. 2000. Memory access scheduling. In *Proceedings of the 27th Annual international Symposium on Computer Architecture* (Vancouver, British Columbia, Canada). ISCA '00. ACM, New York, NY, 128-138.

4 Seznec, A. 1993. A case for two-way skewed-associative caches. In Proceedings of the 20th Annual international Symposium on Computer Architecture (San Diego, California, United States, May 16 - 19, 1993). ISCA '93. ACM, New York, NY, 169-178.

5 González, A., Valero, M., Topham, N., and Parcerisa, J. M. 1997. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th international Conference on Supercomputing* (Vienna, Austria, July 07 - 11, 1997). ICS '97. ACM, New York, NY, 76-83.

6 Zhu, Z. and Zhang, Z. 2005. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proceedings of the 11th international Symposium on High-Performance Computer Architecture* (February 12 - 16, 2005). HPCA. IEEE Computer Society, Washington, DC, 213-224.

7 Zheng, H., Lin, J., Zhang, Z., and Zhu, Z. 2008. Memory Access Scheduling Schemes for Systems with Multi-Core Processors. In *Proceedings of the 2008 37th international Conference on Parallel Processing* (September 09 - 11, 2008). ICPP. IEEE Computer Society, Washington, DC, 406-413.

8 Mutlu, O. and Moscibroda, T. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th Annual international Symposium on Computer Architecture* (June 21 - 25, 2008). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 63-74.

9 Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. 2006. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture* (December 09 - 13, 2006). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 208-222.
10 Rafique, N., Lim, W., and Thottethodi, M. 2007. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of the 16th international Conference on Parallel Architecture and Compilation Techniques* (September 15 - 19, 2007). PACT. IEEE Computer Society, Washington, DC, 245-258.
11 Mutlu, O. and Moscibroda, T. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture* (December 01 - 05, 2007). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 146-160.
12 Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual international Symposium on Computer Architecture* (June 21 - 25, 2008). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 39-50.
13 Naveen Neelakantam, Colin Blundell, Joe Devietti, Milo M. K. Martin and Craig Zilles: FeS2: A Full-system Execution-driven Simulator for x86, In *Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems* (Seattle, WA, USA, March 01 - 05, 2008). ASPLOS XIII. Poster session.
14 Jacob, B., Ng, S., and Wang, D. 2007 *Memory Systems: Cache, Dram, Disk*. Morgan Kaufmann Publishers Inc.
15 Kessler, R. E. and Hill, M. D. 1992. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.* 10, 4 (Nov. 1992), 338-359.
16 https://www.simics.net/
17 Wang, D., Ganesh, B., Tuaycharoen, N., Baynes, K., Jaleel, A., and Jacob, B. 2005. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 100-107.
18 Hur, I. and Lin, C. 2004. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture* (Portland, Oregon, December 04 - 08, 2004). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 343-354.
19 Shao, J. and Davis, B. T. 2007. A Burst Scheduling Access Reordering Mechanism. In *Proceedings of the 2007 IEEE 13th international Symposium on High Performance Computer Architecture* (February 10 - 14, 2007). HPCA. IEEE Computer Society, Washington, DC, 285-294.
20 Lee, C. J., Mutlu, O., Narasiman, V., and Patt, Y. N. 2008. Prefetch-Aware DRAM Controllers. In *Proceedings of the 41st Annual IEEE/ACM international Symposium on Microarchitecture* (November 08 - 12, 2008). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 200-209.