JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.16, NO.6, DECEMBER, 2016 ISSN(Print) 1598-1657
https://doi.org/10.5573/JSTS.2016.16.6.808 ISSN(Online) 2233-4866

Adaptive Memory Controller for High-performance Multi-channel Memory

Jin-ku Kim, Jong-bum Lim, Woo-cheol Cho, Kwang-Sik Shin, Hoshik Kim, and Hyuk-Jun Lee

***Abstract*—As the number of CPU/GPU cores and IPs in SOC increases and applications require explosive memory bandwidth, simultaneously achieving good throughput and fairness in the memory system among interfering applications is very challenging. Recent works proposed priority-based thread scheduling and channel partitioning to improve throughput and fairness. However, combining these different approaches leads to performance and fairness degradation. In this paper, we analyze the problems incurred when combining priority-based scheduling and channel partitioning and propose dynamic priority thread scheduling and adaptive channel partitioning method. In addition, we propose dynamic address mapping to further optimize the proposed scheme. Combining proposed methods could enhance weighted speedup and fairness for memory intensive applications by 4.2% and 10.2% over TCM or by 19.7% and 19.9% over FR-FCFS on average whereas the proposed scheme requires space less than TCM by 8%.**

***Index Terms*—Memory controller, channel partition, thread scheduling, system-on-chip**

I. INTRODUCTION

Modern memory system suffers from performance degradation and unfairness due to interference caused by many applications running on the multi-core based computing platform. In order to mitigate this issue, previous studies have explored different approaches including thread scheduling, memory channel partitioning, and address mapping schemes. Thread scheduling techniques are studied in various texts [1, 3-6]. Thread scheduling approaches prioritize memory access requests from different applications (or threads) to give more priority to those whose performance gain could be maximized. On the other hand, as high bandwidth memory technologies such as HBM or Wide IO become available, more memory channels are used in the memory system. To use these parallel channels efficiently, Muralidhara *et al.* developed the memory channel partitioning (MCP) in which different threads are allocated to different channels to reduce interference [2]. Since channel and bank partitioning approach is orthogonal to thread scheduling, it is claimed to be used on top of various memory scheduling to achieve further performance improvement.

To evaluate different memory controller schemes, two widely accepted metrics are available: performance and fairness. As a performance metric, *weighted speedup* is used [11]. It measures the sum of each thread's throughput under memory sharing normalized to its own throughput under no memory sharing. As a fairness metric, *maximum slowdown* is used. It measures the worst case slowdown experienced by any one of the threads.

In general, weighted speedup gets much improved as we give more memory bandwidth to threads which can be improved the most with additional bandwidth allocation. Latency sensitive applications belong to this category. Kim *et al.* proposed thread cluster memory (TCM) that groups threads according to memory access

Manuscript received Apr. 11, 2016; accepted Aug. 2, 2016
Dept. of Computer Science and Engineering, Sogang University, Korea
E-mail : ecl0038@sogang.ac.kr

characteristics and ranks them based on applications' memory intensity and memory access patterns [1]. Thread scheduling is performed based on this ranking method. TCM uses the memory access intensity to divide applications to *latency-sensitive* and *bandwidth-sensitive* groups and gives higher priority to the latency-sensitive group. In addition, it ranks threads in the bandwidth-sensitive group based on *bank-level parallelism* (BLP) and *row-buffer locality* (RBL). TCM employs a *shadow row-buffer* to keep track of row-buffer locality of an individual thread independently. The row-buffer locality of a thread is an important metric because higher locality can cause more interference. The RBL of each thread on a real system is distorted by other applications memory operations as a result of scheduling. However, the priority ranks based on shadow row-buffer locality do not reflect applications' memory access characteristics in real environment because they do not consider interaction among different threads. That flawed ranking method leads to unfair prioritization and causes starvation.

Whereas TCM penalizes the threads causing interference to isolate thread interference in the memory system, MCP achieves thread interference isolation by mapping threads to physically different channels. It uses *miss per kilo instruction* (MPKI) and RBL to group and map applications of similar characteristics to 3 different channels. However, static threshold values used for partitioning in MCP cause skewed utilization of channel bandwidth. MCP claims that it can be used with any memory scheduling techniques such as FR-FCFS [7], ATLAS, or TCM and enhance the weighted speedup. However, TCM with MCP seriously degrades the fairness because grouping threads of similar characteristics in MCP causes starvation of low priority threads in the same group.

To resolve these issues, we propose three schemes as follows. First, we propose a new thread ranking method based on row-buffer miss count and executed instruction count for bandwidth sensitive threads, which dynamically increases the priority of an application when an application has a large row-buffer miss rate due to interference whereas TCM requires forced shuffling. Second, we resolve two problems in MCP: unbalanced channel partition and interference among applications with similar characteristics mapped on the same channel.

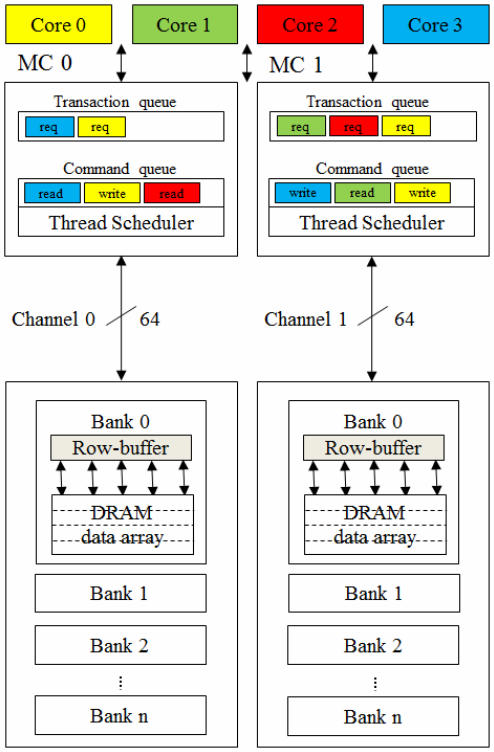

Fig. 1. Memory controller architecture.

Finally, we propose application specific address mapping which maximizes either row-buffer locality or bank-level parallelism.

In Section 2, we discuss previous works and their draw- backs. In Section 3, we present our proposed scheduling and mapping methods. Finally, we show experimental results in Section 4 followed by conclusion in Section 5.

II. BACKGROUND

1. Baseline Memory Controller Architecture

A typical memory system consists of multiple memory channels. Memory requests from different threads running on multiple processor cores are distributed to multiple channels for full bandwidth utilization. Each channel has its own memory controller connected to the DRAM module and operates independently. In general, a memory controller has a transaction queue and a command queue. Both transaction and command queue store memory requests from different threads. A thread scheduler (e.g. FR-FCFS, TCM) in these queues assigns priority to different memory requests and schedules them

according to their priority. DRAM has an internal cache called a row-buffer to cache a row of the internal memory array. If data for a memory request is found in the row-buffer, the access is called a hit and the access time becomes short. Otherwise, the access takes longer to fetch a new row. In multi-threaded environment, many threads share a row-buffer in DRAM and cause a row conflict resulting in poor bandwidth utilization.

2. TCM

In TCM, threads are grouped into a latency-sensitive cluster (group) or a bandwidth-sensitive thread cluster (group). The former has the higher priority than the latter in memory controller scheduling.

$$\text{Priority} \propto \frac{BLP}{RBL} \quad (1)$$

Within the bandwidth-sensitive group, the priority of a thread is determined using the relationship in Eq. (1) where BLP is bank-level parallelism and RBL is row-buffer locality. In TCM, a memory controller keeps track of row-buffer hits/misses of a thread by maintaining a virtual row-buffer per thread. It measures row-buffer hits or misses per thread as if the thread only uses a shared row-buffer. This virtual row-buffer is referred as a shadow row-buffer and measures RBL (row-buffer hit rate) per thread. However, a shadow row-buffer hit rate (RBL) does not reflect the thread's memory performance caused by scheduling. Suppose that a thread has a row-buffer access pattern of H(hit), M(miss), H(hit), M(miss) during a sampling window. Then, RBL is $\frac{2}{4} = 0.5$. When this thread's priority is demoted and scheduled only twice in the next sampling window and gets a pattern of H, M. Then, RBL is still $\frac{1}{2} = 0.5$ although the priority is lowered and its memory performance is degraded. For this reason, TCM requires thread priority shuffling to avoid starvation of penalized threads. Until the shuffling happens, the performance of a lower priority thread continuously gets degraded. This effect is further explained in Fig. 2.

Fig. 2 compares the row buffer hits/misses measured by the shadow row buffers proposed in TCM (top) and the combined row buffer used in real DRAM (bottom).

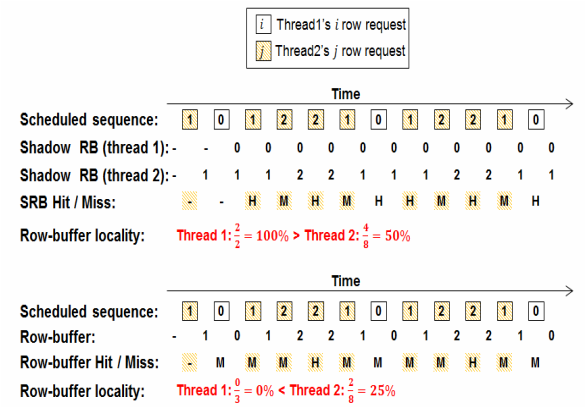

Fig. 2. Row-buffer locality with a shadow row-buffer (up) and a shared row-buffer (down).

The illustration shows that a shadow row buffer does not reflect the behavior of the real row buffer in DRAM.

In Fig. 2, the upper diagram shows the scheduled sequence of memory requests, the contents of the shadow row-buffer (SRB) for thread 1 (white) and 2 (yellow), and the hit/miss of shadow row-buffers for the scheduled sequence. In the figure, the first four scheduled memory requests are address 1, 0, 1, and 2. They are from thread 2, 1, 2, 2 respectively. The second and third rows for SRBs show the contents of the SRB for thread 1 and 2. The hit/miss for two SRBs are combined and shown in the fourth row. The moment when the content of a shadow row buffer changes either in second or third row indicates a row buffer miss. For instance, the second 1 in the scheduled sequence causes a hit in the SRB for thread 2. The following 2 causes a miss in the same SRB because it is a new address. Hits or misses are separately measured for two SRBs. The row-buffer locality (RBL) for thread 1 and 2 are 100% and 50% respectively because thread 1 has two hits and thread 2 has four hits and four misses. Hence, thread 1 has the lower priority according to Eq. (1).

The lower diagram shows the contents of a shared row buffer in real DRAM and the row buffer hits/misses for the same scheduled sequence as the upper diagram. Again, whenever a new memory request accesses a different row, it causes a row buffer miss. For the first four memory requests, they cause row buffer misses because they accesses different rows. When RBL is measured with a real shared DRAM row-buffer as shown in the lower diagram, RBL for thread 1 and 2 are 0% and 25% respectively because thread 1 has no hit and thread 2 has two hits out of eight memory requests.

When we compare upper and lower diagrams, the real DRAM RBL for thread 1 is small as a result of scheduling and it should be set to a high priority. But its priority remains low in TCM due to high shadow RBL. That is, the RBL measured in TCM does not reflect the RBL of the real DRAM row buffer. This problem is aggravated when TCM is used with MCP because the threads of similar RBL are grouped together in MCP and a thread may be severely penalized over another despite very small difference in RBL.

3. MCP

MCP improves memory utilization by isolating interference among threads via efficient channel mapping. It uses Misses Per Kilo-instructions (MPKI) and RBL information of applications to group and map threads to three memory channels. Threads whose MPKI is less than the average MPKI are mapped to the first channel. Remaining threads are further divided into two groups according to their RBL values. Threads whose RBL are less than 50% are mapped to the second channel and others are to the third channel respectively.

MCP has several drawbacks. First, at least three channels are required in the memory system. Second, a static partition scheme not considering bandwidth allocation (e.g. using a static RBL value of 50%) incurs unbalanced bandwidth distribution between channels. Finally, threads grouping based on similarity does not work well with thread scheduling schemes such as TCM. For instance, two threads with RBL of 40% and 41% respectively are mapped to the same channel in MCP. When TCM is used to schedule two threads, the thread with 41% RBL is continuously penalized although RBLs of two threads are not much different. This TCM problem is aggravated by MCP because MCP groups threads with similar RBL together.

III. PROPOSED SCHEME

1. Overall Memory Controller Architecture

Our proposed memory controller architecture is based on a channel partition scheme to reduce interference among threads and improve bandwith allocation. In addition, we present a novel thread scheduling scheme based on dynamic thread prioritization which works well with channel partitioning. Finally, we further improve performance and fairness via channel specific address mapping. Each component is explained in detail in following sections.

2. Dynamic Priority Thread Scheduling (DPTS)

The thread grouping and ranking in the proposed memory controller is similar to TCM. Latency-sensitive threads have higher priority than bandwidth-sensitive threads in scheduling. One major difference is that bandwidth sensitive threads are ranked using Eq. (2).

$$Priority \propto \frac{Row\,Buffer\,Miss_i}{Instruction\,Count_i} \tag{2}$$

The relationship shows that the priority of thread *i* is proportional to the number of row-buffer misses caused by thread *i* and inversely proportional to the instruction count of thread *i* during the sampling period. The row-buffer misses are measured using a shared row-buffer as shown the bottom of Fig. 2. The crucial drawback of TCM is to use the hits/misses measured from a shadow row-buffer, which do not reflect the real row-buffer hits/misses. Due to this, a high RBL thread is continuously assigned to a low priority and TCM requires periodic shuffling to correct the starvation problem. Since a shared row-buffer is used in DPTS, increased row-buffer misses due to lowering priority dynamically increase priority in the next sampling period and thus it does not require periodic shuffling. In addition, a thread assigned to low priority is slow-downed, which reduces the instruction count and increases the priority again using Eq. (2).

3. Adaptive Multi-channel Partitioning (AMP)

To resolve issues discussed in Section II.3, we propose an adaptive channel partition method. Its partition method is depicted in Fig. 3. We use two channels which can be further extended for even number of channels. First, threads are sorted in an ascending order according to the memory bandwidth usage. The first *N* threads in the sorted list whose sum is less than $\frac{1}{6}$ of total

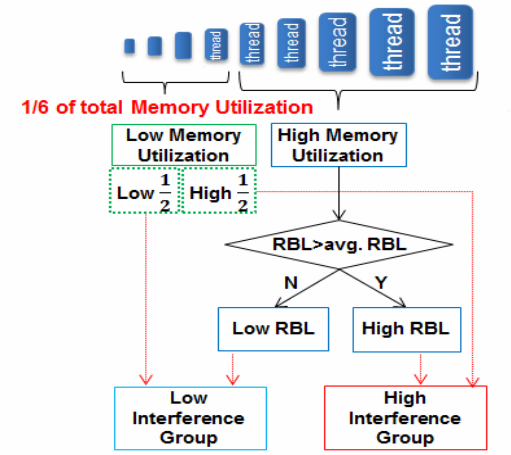

Fig. 3. Adaptive multi-channel partitioning method.

bandwidth sum are grouped into latency-sensitive threads (low memory bandwidth utilization group). They are further partitioned into two halves and mapped to two channels respectively.

By separating the lower half from the higher half, interference from the higher half on the lower one does not occur. Bandwidth-sensitive threads (high memory bandwidth utilization group) are divided according to their RBL values. Instead of using a fixed value of 50% like in MCP, an average RBL value is used for thread grouping. If a thread has a RBL value less than the average, it is mapped to the low RBL channel. Otherwise, it is mapped to the high RBL channel. Because of this, low RBL threads do not suffer from the interference from high RBL threads. In addition, used memory bandwidths are evenly distributed between two channels because an average RBL value is used instead of a static value. Both low and high RBL threads are scheduled using DPTS respectively and do not suffer from starvation.

4. Dynamic Address Mapping (DAM)

When a memory request is received by a memory controller, its address is translated to channel, rank, bank, row, and column number according to a pre-determined address mapping method. Two considered mapping methods are cache-line interleaving and row interleaving. Cache-line interleaving refers to an address mapping method where subsequent cache-lines are mapped across ranks, banks, channels before being mapped to the same row. On the other hand, subsequent cache-lines in the row interleaving scheme are mapped to the same row until the entire row is used.

In our proposed scheme, we use cache-line interleaving for the channel mapping high RBL threads and row interleaving for the channel mapping low RBL threads. We refer this address scheme as dynamic address mapping scheme (DAM). When a thread requests a new physical page upon demand paging, a new page is allocated from the channel where a requesting thread belongs. Then each channel interprets the address of a memory request according to the address mapping scheme.

Since AMP is our baseline channel partitioning scheme, threads are classified into latency-sensitive, high RBL, low RBL threads. First, latency-sensitive threads mapped to both channels are not too much affected by DAM because they do not use memory bandwidth much. Second, high RBL threads typically much interfere with other threads because of their high row-buffer locality. Thus using row interleaving could aggravate interference. Meanwhile, they also show streaming memory access patterns where subsequent cache lines are accessed in order. For this reason, the high RBL thread channel uses cache-line interleaving to improve the throughput of the memory system by exploiting bank-level parallelism. Finally, low RBL threads exhibit random memory access patterns. We use row interleaving for low RBL threads so that increasing row-buffer locality isolates thread interference and improves fairness.

IV. Experimental Results

1. Simulation Setup

We modify DRAMSim2 simulator [8] and use SPEC CPU2006 benchmark [9] to evaluate our memory controller scheme. We create test cases by mixing 24 applications and vary memory traffic intensity by selecting 25%, 50%, 75% and 100% of applications from bandwidth-sensitive benchmarks. For instance, 25% means that a mix of 24 applications includes 6 bandwidth-sensitive benchmarks and 18 latency-sensitive benchmarks. To evaluate IPC of each benchmark, we use the Macsim simulator [10] in conjunction with the DRAMSim2 simulator.

Our baseline system has 24-core CPU. Each core has 4-way 32 KB L1 cache with 64 Byte block size and 8-

way 256 KB L2 cache with 64 Byte block size. Our memory system uses DDR3 DRAMs running at 800 MHz and consists of 2 channels, 1 rank per channel, 8 banks per rank. We employ 2 metrics: weighted speedup (performance) and maximum slowdown (fairness) to compare the throughput of the proposed memory system and fairness among mixed threads [1]. They are defined as follows.

$$Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Maximum\ Slowdown = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}$$

2. Comparing Performance and Fairness with Previous Schemes

We compare our proposed scheme to FR-FCFS [7], TCM [1], and BLISS [6]. Fig. 4 shows weighted speedup (system performance) and maximum slowdown (fairness) values averaged over all experiments. In the legend, DPTS refers to dynamic priority thread scheduling, AMP refers to adaptive multi-channel partitioning, and DAM refers to dynamic address mapping. In Fig. 4, points toward right-upper corner show overall better performance and fairness. Both performance and fairness values of different schemes are normalized to those of FR-FCFS. As it is shown, DPTS only shows slightly worse performance than TCM because uneven bandwidth allocations for different channels. However, our proposed memory controller employing all three techniques improves weighted speedup by 15.99%, 3.22%, 8.79% and maximum slowdown by 20.59%, 3.06%, 0.69% over FR-FCFS, TCM, and BLISS respectively. Although adding DAM slightly improves the performance over DPTS+AMP, most gain comes from combining DPTS and AMP. This is because DPTS is very effective to schedule the threads of similar characteristics, which are grouped together by AMP, whereas AMP successfully isolates the interference among threads of different memory access behaviors. When we consider only memory intensive applications which get much attention recently (e.g. GPU and IP memory traffic in mobile devices or big data application in data centers), our proposed method could enhance weighted speedup and fairness further, which will be explained in the next section.

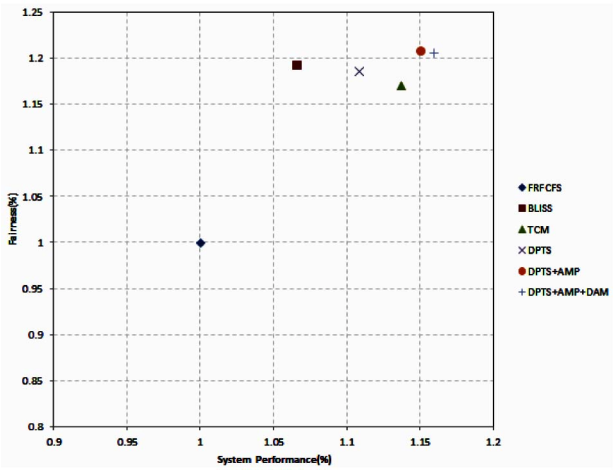

Fig. 4. Average performance and fairness of various schemes.

3. Effects of Varying Memory Intensity in Workloads

Fig. 5 compares the performance and fairness of the proposed scheme over FR-FCFS, TCM, BLISS with respect to four different memory intensity mixes. In Fig.

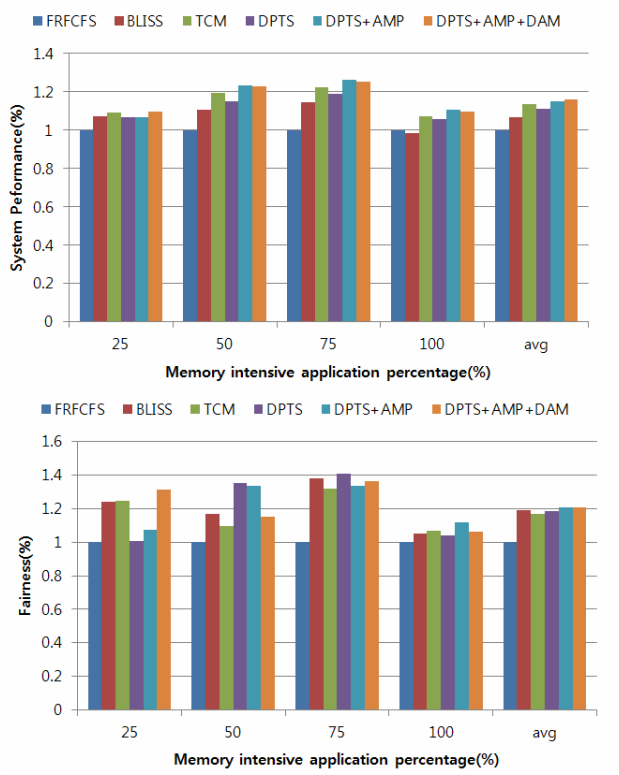

Fig. 5. Comparing performance and fairness for different workload mixes.

5, x axis represents the percentage of memory intensive applications where memory intensive applications are defined as those whose MPKI are larger than the average MPKI of all applications in selected benchmark. Both performance and fairness of different schemes are normalized to that of the FR-FCFS. The proposed memory controller shows the best system performance over all other scheme for different percentage values.

To figure out the performance and fairness of the proposed scheme for highly memory intensive applications (e.g. IP's or GPU's memory requests in mobile phones or big data memory traffic in data center), we only average the performance and fairness for the cases where memory intensive application percentages are 75% and 100%. The results in Fig. 5 show that our proposed method (DPTS+AMP) enhances weighted speedup and fairness for memory intensive applications by 4.2% and 10.2% over TCM or by 19.7% and 19.9% over FR-FCFS on average. The gain of our proposed scheme (DPTS+AMP) over TCM is greater for memory intensive applications for several reasons. First, memory intensive applications are severely penalized by TCM at the cost of performance improvement for latency sensitive applications whereas DPTS dynamically adjusts the priority of memory intensive applications so that they are not continuously penalized. Second, DPTS+AMP successfully isolates the interference among memory intensive applications whereas TCM is not effective to schedule memory intensive applications of similar characteristics.

4. Effect of Alternative Thread Ranking Metrics

We study alternative thread ranking metrics for memory intensive threads (or bandwidth-sensitive threads) to evaluate the ranking metric of DPTS shown in Eq. (2). The first alternative ranking metric we consider is shown in Eq. (3).

$$Priority \propto \frac{Row\,Buffer\,Miss_i}{Memory\,Accesses_i} \quad (3)$$

In this ranking metric, the priority of a thread *i* is inversely proportional to the number of memory accesses from thread *i* during a sampling period. Unlike the ranking metric for DPTS where the instruction count is used in the bottom, Eq. (3) considers a row-buffer miss rate. Fig. 6 shows the performance and fairness of different ranking metrics combined with proposed schemes. Again, performance and fairness are normalized to those of FR-FCFS. DPTSM uses Eq. (3) for thread scheduling.

The second ranking metric we consider is shown in Eq. (4).

$$\text{Priority} \propto \frac{Row\,Buffer\,Miss_i}{Memory\,Accesses_i \text{ X } Instruction_i} \quad (4)$$

This metric differs from Eq. (3) in that the priority of a thread i is inversely proportional to both number of memory accesses and executed instructions for thread i during the sampling period. In Fig. 6, DPTSI represents a thread scheduling scheme using the ranking metric in Eq. (4).

Fig. 6 shows that the proposed ranking metric in DPTS

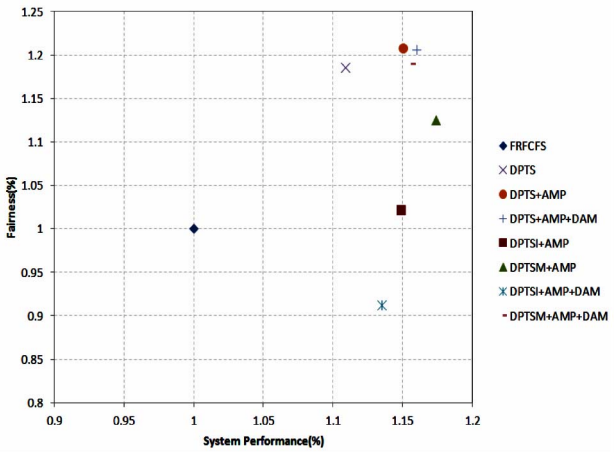

Fig. 6. Comparing alternative thread ranking metrics.

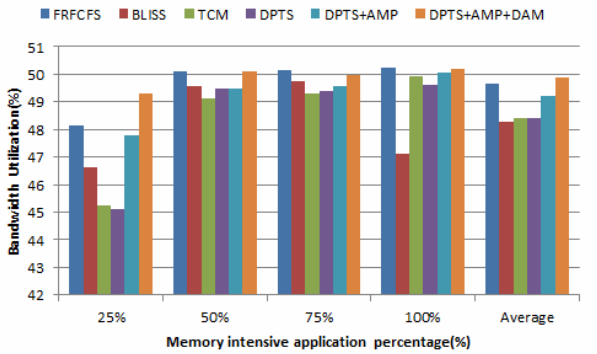

Fig. 7. Comparing bandwidth utilization for different workload mixes.

performs better than Eqs. (3, 4). The results show that DPTSM+AMP using Eq. (3) has the best performance but its fairness is degraded compared to Eq. (2). Schemes using DPTSI show poor fairness because the result of scheduling is not reflected correctly. This is due to the fact that a miss rate (row-buffer miss/ memory accesses) is a small number and dividing it with the number of instructions makes the overall metric too small.

5. Comparing Bandwidth Utilization with Previous Schemes

Bandwidth utilization shows how much memory bandwidth is actually used to support applications. For four different application groups shown above, we compare our schemes with FR-FCFS, BLISS, and TCM. As we can see from the plot, our best scheme combining three methods shows the best bandwidth utilization compared to other previous schemes. FR-FCFS is a memory scheduler solely aiming at maximizing bandwidth utilization. However, our scheme shows better bandwidth utilization because grouping similar applications into different channels and banks to isolate interference and using dedicated address schemes for different groups improve bandwidth utilization as well.

6. Hardware Costs

We compare the cost of our proposed scheme with the best known scheme, TCM. TCM requires memory space to store information for MPKI (240 bits), bank-level parallelism (672 bits), row-buffer locality (2880 bits) and shuffling logic for 24 threads (cores) per channel according to Table 2 in [1].

Compared to TCM, our proposed scheme does not require hardware for measuring bank-level parallelism (672 bits) and thread shuffling which is quite expensive. Instead, our scheme needs one shared row-buffer miss counter per thread across all banks to store real row-buffer misses for each thread. This requires $N_{threads} \times \log_2 Count_{max} = 24 \times \log_2(32 \times 1024) = 24 \times 15 = 360$ bits assuming the sampling period is 500,000 cycles.

In summary, TCM requires memory space to store information for MPKI (240 bits), bank-level parallelism (672 bits), and row-buffer locality (2880 bits). Our scheme requires memory space to store information for MPKI (240 bits), row-buffer locality (2880 bits), and shared row buffer miss counters (360 bits). Thus, ratio between our scheme and TCM is (240+2880+360)/(240+672+2880) = 3480/3792 = 0.92. Thus, our scheme requires 8% less space than TCM.

V. CONCLUSIONS

We present an adaptive memory controller that combines three orthogonal approaches: thread scheduling, memory channel partitioning, and address mapping. Our scheduling algorithm offers dynamic priority thread scheduling that can work efficiently with the proposed adaptive channel partitioning method. Along with those schemes, the proposed dynamic address mapping scheme boosts both of system throughput and fairness.

Our experiments show that the proposed scheme outperforms the best known scheme, e.g. TCM, by 4.2% (throughput) and 10.2% (fairness) for memory intensive application. We plan to extend our work for the challenging heterogeneous computing platform where many memory bandwidth hungry IPs coexist with CPU and GPU cores.

ACKNOWLEDGMENTS

This Article was supported by Future Technology Fund of LG Electronics.

REFERENCES

[1] Y. Kim, M. Papamichael, O. Mutlu, and M. HarcholBalter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in MICRO, 2010.

[2] S. Muralidhara et al, “Reducing memory interference in multi-core systems via application-aware memory channel partitioning,” In MICRO-44, 2011.

[3] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers,” in HPCA, 2010.

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

[5] R. Ausavarungnirun et aI, "Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems," in ISCA, 2012.
[6] L. Subramanian et al, “The blacklisting memory scheduler: Achieving high performance and fairness at low cost,” in ICCD, 2014
[7] S. Rixner et al, “Memory access scheduling,” In ISCA-27, 2000
[8] P. Rosenfeld et al., “Dramsim2: A cycle accurate memory system simulator,” CAL, 2011.
[9] “SPEC CPU 2006,” http://www.spec.org/cpu2006/.
[10] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho. “MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide,” Georgia Institute of Technology, 2012
[11] A. Snavely and D. M. Tullsen. “Symbiotic job scheduling for a simultaneous multithreading processor,” In ASPLOS-IX, 2000.

Jin-ku Kim received B.S in 2013 and is currently pursuing the PhD program in Computer Science and Engineering from Sogang University. He is in Embedded computing Laboratory. His research interests include high-performance memory system and memory scheduling.

Jong-bum Lim received the B.S. and M.S. degree in Computer Science and Engineering from Sogang University in 2013 and 2015. His research interests include high-speed networking and high-performance memory system.

Woo-cheol Cho received the B.S. and M.S. degrees in Computer Science and Engineering from Sogang University in 2012 and 2014. He is a Ph.D candidate at the Embedded Computing Laboratory. His research interests include many-core based system.

Kwang-Sik Shin received the B.S., the M.S, and Ph.D degrees in Electronic Engineering from University of Inha, Incheon, Korea in 2001, 2003, and 2008, respectively. From 2008 to 2011, he served as a Senior Engineer in content division group at Electronics and Telecommunications Research Institute, Daejeon, Korea. He currently works at the System IC research center in LG electronics, Seoul, Korea. His research interests include computer architecture, network, parallel processing, system architecture, and memory architectures.

Hoshik Kim received his bachelor’s degree in electrical engineering from Yonsei University, Seoul, Korea and received his master’s and doctoral studies in electrical engineering from University of Southern California, Los Angeles, CA. He is currently Principal Engineer at System IC Center, LG Electronics, Seoul, Korea where he is developing the architecture and design of memory subsystems for mobile application processors. Prior to joining LG Electronics in 2013, he was with Intel Corporation, Santa Clara, CA for 9 years working in the areas of design technology, automation and verification for various microprocessors and system-on-chip products. His current research interests include memory systems and interconnect architectures, design automation and verification of system-on-chips.

Hyuk-Jun Lee received the B.S. degree in Computer Engineering from University of Southern California, Los Angeles, CA in 1993 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1995 and 2001, respectively. From 2001 to 2011, he served as a Senior Engineer in routing technology group at Cisco System, San Jose, CA. He is currently an Associate Professor at the Department of Computer Science and Engineering, Sogang University, Seoul, Korea. His research interests include embedded systems, low-power design, and memory architectures.