Core-aware Memory Access Scheduling Schemes[†]

Zhibin Fang Xian-He Sun Yong Chen Surendra Byna
Department of Computer Science
Illinois Institute of Technology, Chicago, IL 60616, USA
{zfang2, sun, chenyon1, sbyna}@iit.edu

Abstract

Multi-core processors have changed the conventional hardware structure and require a rethinking of system scheduling and resource management to utilize them efficiently. However, current multi-core systems are still using conventional single-core memory scheduling. In this study, we investigate and evaluate traditional memory access scheduling techniques, and propose a core-aware memory scheduling for multi-core environments. Since memory requests from the same source exhibit better locality, it is reasonable to schedule the requests by taking the source of the requests into consideration. Motivated from this principle of locality, we propose two core-aware policies based on traditional bank-first and row-first schemes. Simulation results show that the core-aware policies can effectively improve the performance. Compared with the bank-first and row-first policies, the proposed core-aware policies reduce the execution time of certain NAS Parallel Benchmarks by up to 20% in running the benchmarks separately, and by 11% in running them concurrently.

† This research was supported in part by National Science Foundation under NSF grant EIA-0224377, CCF-0621435, and CCF-0702737.

1. Introduction

Chip multiprocessing (CMP) technology with the help of thread-level parallelism (TLP) and data-level parallelism (DLP) have been driving processor technology to increase computing power substantially. Multi-core processors reduce power consumption by using multiple simpler cores and packaging them together on a single die. However, memory access latency is a troubling performance issue. Due to the so-called *memory-wall problem* [11], i.e. the enlarging gap between CPU performance and memory performance, data access is a recognized dominant performance bottleneck. Competition for data access and transferring data among cores may increase the stall time of cores and lead to a substantial performance loss of multi-core processors.

Reorganizing data accesses of multiple cores is an effective solution in tackling the memory performance bottleneck. However, in a multi-core system, data access scheduling is performed by a shared memory controller that is integrated onto the same multi-core chip, which provides limited or no control to programmers over its data access scheduling. The controller is the single point for accessing memory, and its effectiveness has a great impact on the overall performance. Realizing that data access requests from each core often come from one thread and have a better memory locality, we believe a multi-core memory

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

access scheduling could be improved by considering the source of the requests. Data access scheduling is a very important issue. Simply adopting traditional single-core scheduling is not a wise approach. It is necessary to thoroughly investigate and design memory access scheduling policies for multi-core processors.

Multi-core memory access is quite different from its single-core counterpart. In a multi-core system, each core has its own running thread that might exhibit better locality among its own memory accesses than interleaved memory accesses from multiple cores. In addition, when running scientific parallel applications, while most of the cores are concurrently working on one application, the operating system also performs various housekeeping functions, which generates access requests to memory controller as well. These application-unrelated memory requests may interfere in exploiting the locality of application related accesses. It is necessary to separate them and give higher priority to application related memory requests.

Conventional memory access scheduling approaches suggest that memory operations should be reordered based on hardware features [9][10], such as the internal memory structure, to increase the throughput. They are efficient and sufficient for single-core architecture. However, these methods are direct extensions of sequential scheduling policies, which merge all requests from different cores into one queue and apply existing sequential scheduling policies directly on this request queue. They are not aware of the source of the memory requests and do not work well for multi-core systems.

This study investigates two traditional single-core memory scheduling policies, bank-first policy and row-first policy, in a multi-core environment. We then propose two novel *core-aware memory access scheduling* policies and analyze the effect of different memory scheduling policies. We show that the proposed core-aware memory access scheduling schemes are capable of reducing memory access latency significantly and improving the overall performance of applications.

The rest of this paper is organized as follows: Section 2 reviews related work in memory access scheduling. Section 3 introduces the proposed core-aware memory access scheduling schemes. Section 4 discusses experimental setup and Section 5 presents performance evaluation and analysis results. We conclude our discussion in Section 6.

2. Background

2.1 Memory access scheduling

In a memory controller, the execution of a memory access instruction must adhere to the rules and timing constraints of the hardware to access data in a modern DRAM. As shown in Figure 1, modern DRAMs are three-dimensional memory devices with dimensions of bank, row and column. Thus, a location in the DRAM is identified by an address that consists of bank, row and column fields. The steps of accessing a location include a pre-charge, a row access, and then a column access. Due to the DRAM structure and its hardware implementation, sequential accesses to different rows within one bank have high latency, whereas accesses to different banks or different words within a single row

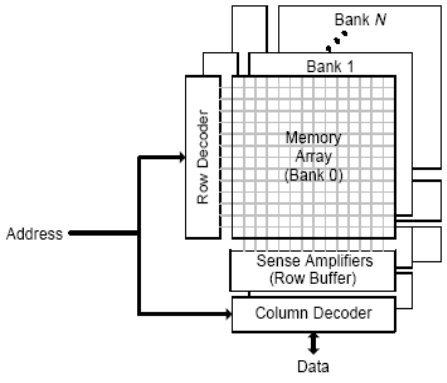

Figure 1. DRAM architecture [9]

have low latency [9].

Memory access scheduling can effectively reduce the average memory access latency and improve memory bandwidth utilization by reducing cross-row data access. For example, prioritizing memory requests to the same bank and the same row can improve performance. Suppose there is a memory request sequence that accesses different rows of the same bank, and the sequence is A-B-A-B-A-B. This request sequence will cause five row misses, and each row miss between A and B requires a pre-charge, row access and column access operations. The row-first policy reorders the access sequence to A-A-A-B-B-B. The reordered sequence only causes one row miss, and leads to a much smaller latency. This technique effectively improves the memory system performance [13].

2.2 Memory scheduling schemes

Many memory scheduling policies have been proposed to improve the efficiency of memory accesses in the context of single-core processors. The key idea in these policies is to focus on reorganizing memory accesses by taking advantage of the internal memory structure, access history or the characteristics of application.

Rixner et al. [9] proposed a memory access scheduling scheme within memory controller, called *bank-first scheduling*, to improve the performance of a memory system. Their approach reorders memory accesses to exploit the non-uniform access times of the "3-D" structure of banks, rows and columns of contemporary DRAM chips. In the bank-first scheme, memory operations to different banks are allowed to proceed before those to the same bank, thus increasing the access concurrency and throughput.

The burst scheduling proposed by Shao et al. [10], also called *row-first scheduling* scheme, clusters outstanding accesses into bursts that would access the same row within a bank. Accesses within a burst, except the first one, are row hits and only require column access transactions. Data transfers of these accesses can be performed back-to-back on the data bus, resulting in a large payload and data bus utilization improvement. Increasing the row hit rate and maximizing the memory data bus utilization are the major design goals of burst scheduling.

To consider the long-term effects of a scheduling decision, Hur et al. proposed an adaptive history-based memory scheduler [3], which tracks the access pattern of recently scheduled accesses and selects memory accesses matching the pattern of reads and writes. This technique uses three history-based arbiters, each with a history length of two. The arbiter uses such information to schedule operations to match some pre-determined mixture of reads and writes.

Zhang et al. [12] proposed a fine-grain priority scheduling method, which splits and maps memory accesses into different channels and returns critical data first, to fully utilize the available bandwidth and concurrency provided by Direct Rambus DRAM system.

To optimize the memory system of SMT architecture, Zhu et al. [13] proposed a thread-aware DRAM optimization technique. They concluded that increasing the number of threads tends to increase the memory access concurrency and thus raise the pressure on DRAM systems, whereas some exceptions do exist. The application performance is sensitive to memory channel organizations, e.g. independent channels may outperform ganged organizations by up to 90%. The DRAM latency reduction through improving row buffer hit rate becomes less effective due to the increased bank contentions.

Mutlu et al. [8] proposed a Stall-Time Fair Memory scheduler (STFM) to provide quality of service to different threads sharing the DRAM memory system. This scheme can significantly reduce the unfairness in the DRAM system while also improving system throughput (i.e. weighted speedup of threads) on a variety of workloads and systems. The goal of the

proposed scheduler is to "equalize" the DRAM-related slowdown experienced by each thread due to interference from other threads, without hurting overall system performance.

The existing studies, except the STFM approach, were not designed specifically considering the features of a multi-core processor. In addition, the purpose of STFM scheduler was to keep the fairness between threads on multiple cores and did not focus on facilitating an application running on a multi-core processor. The major limitation of other studies is that they did not take the source of memory requests into consideration. As the number of cores within one processor increases gradually, the role of memory requests from different cores is distinct and of significant importance. It is necessary to distinguish the source of memory requests and make an optimal scheduling based on their importance. We compare and enhance bank-first and row-first schemes with core awareness in this study, because bank-first scheme is usually taken as an evaluation bench for memory request scheduling [3][10], and row-first scheme was proposed most recently with good performance [10].

3. Core-aware Memory Access Scheduling

In this section, we introduce a novel core-aware memory access scheduling specifically designed for multi-core processors, while keeping the merits of classic memory scheduling schemes.

3.1 Core-aware memory scheduling

Figure 2 shows general memory architecture of multi-core processors, where memory controller is shared by multiple cores. In this architecture, when multiple applications are running, requests from the same application have a great possibility of accessing the same row in the same bank, due to the principle of locality and due to the large size of each row in one bank (up to the page size, 8 KByte). The premise behind our core-aware access scheduling scheme is to optimize memory accesses by considering its source and to improve the performance of applications. L2 cache can be shared by multiple cores or private to one specific core (as shown in Figure 2). In either case, memory controller and memory are shared.

The proposed core-aware memory scheduling algorithm for memory controller is shown in Figure 3. The essential idea of the core-aware scheduling is to classify outstanding requests from the same core and issue them together according to their source (which core they are coming from). Core-aware scheduling gives the highest priority to the requests from the same core, because it is more likely that these requests exhibit data locality. To prevent from starvation of memory requests from other cores, we set a threshold for the maximum number of continuous requests from one core. A threshold register is used in the memory controller to record the number of issued requests from the same core, which is indicated by *n* in our algorithm. If the number of continuous requests from current core exceeds the preset threshold value, the scheduling policy stops serving the requests from the current core, and gives the highest priority to the requests from another core. The selection of the next core is done in a round-robin manner.

The main requirement of core-aware policy is to retrieve the identification of the core that is sending a

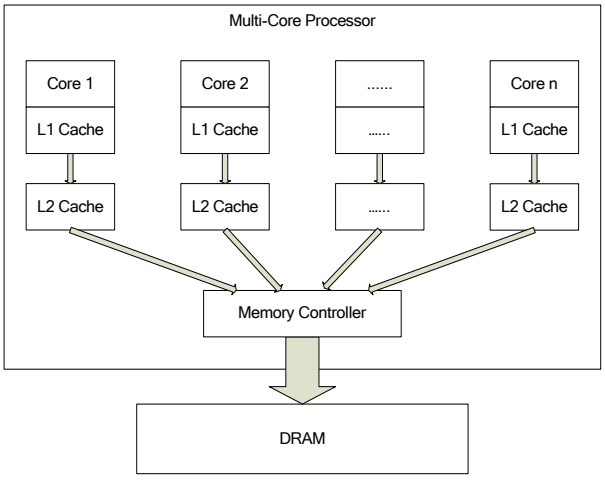

Figure 2. Multicore architecture

memory request to memory controller. While hardware support for passing the identification of the source of memory requests is trivial. We assume that each core has a unique core identifier, and if a core issues a request, the core identifier is recorded in the memory request. The core id information can be stored in the lowest address bits of a cache request, because these bits are useless for a cache request. For example, the lowest six address bits could be used in a 64 byte cache controller. This design provides a straightforward but effective approach to passing down the core identifier to memory controller.

```
ALGORITHM: CASA /*core-aware scheduling algorithm*/
INPUT: Random sequence of memory access requests from m cores
OUTPUT: Core-aware scheduled sequence of requests to memory controller
BEGIN
  k ← 0; /* k indicates the core id */
  While true
    n ← q;/*q is the number of requests to be scheduled*/
    succeed ← 0;
    Repeat
      cid ← k mod m;
      Select s = min(p, number of outstanding requests from core cid) requests from core cid, and enqueue them to the issue_queue; /* p is the size of issue_queue */
      n ← n - s;
      If (n = 0 OR (k mod m = 0) ) Then succeed ← 1; /* 2nd condition prevents starvation and guarantees requests are issued within at most one iteration */
      k ← k+1;
    Until succeed = 1
  End While
END
```

Figure 3. Core-Aware Scheduling Algorithm

The core-aware scheduling can effectively give higher scheduling priority to the accesses from applications in favor of better locality in application, and thus reduces the waiting time of requests from applications. Using the proposed scheduling, a core with intensive accesses will get more opportunities to process requests, and it is more apt than a regular round-robin manner for a multi-core environment. In addition to the hardware requirement to pass core identifier information to the memory controller, the proposed scheduling scheme requires a core-aware selection process to determine which requests can enter the issue-queue, but does not need any modification on modern memory hierarchy. This core-aware selection process can be achieved in many ways. For instance, one way is to have a larger outer-queue and then select core-aware requests from the outer-queue to the issue-queue. These hardware cost is negligible, whereas the memory access performance can be significantly improved as demonstrated from the simulation experiments.

3.2 Core-aware scheduling scheme

With the idea and the algorithm of taking the request source into consideration, we enhance two conventional scheduling policies, bank-first and row-first policy, with core-awareness. We explore and compare four memory scheduling policies, including bank-first policy, row-first policy, and two new policies, namely *core-aware bank-first policy* and *core-aware row-first policy*, in this study. We give a concrete example to demonstrate how these four policies schedule access requests differently. We assume that a sequence of memory requests is stored in a memory queue. Each request has its bank, row and core identifier as shown in Table 1. We will discuss the scheduled result of each scheduling scheme.

Table 1. A sequence of memory requests

Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

3.2.1 Bank-first policy

Bank-first policy [9] arranges all memory requests by banks, and schedules them in a round-robin manner according to the bank identifier. This policy is beneficial because the requests to different banks can be carried out simultaneously. For the request sequence shown in Table 1, the sequence of issued requests by the bank-first policy will be A-C-D-F-E-B-G-H-J-I.

3.2.2 Row-first policy

Row-first policy gives the highest priority to the access to the same row of the same bank [10]. The row-first policy essentially enhances the bank-first policy by grouping the accesses to the same bank and same row together. This optimization is beneficial in reducing row misses. For the request sequence shown in Table 1, the sequence of issued requests by the row-first policy will be A-B-J-C-D-G-I-F-H-E.

3.2.3 Core-aware bank-first policy

The core-aware bank-first policy applies core-aware scheduling into bank-first policy. It gives higher priority to accesses from the same source and for data from the same bank. This policy first arranges accesses according to the destination bank, and then groups all accesses from the same core together. In essence, this policy is in favor of those accesses from the same thread while still allowing concurrent bank accesses for a high throughput. The core-aware bank-first policy schedules application related memory requests firstly and is of importance when a variety of applications running simultaneously on a multi-core environment. For the request sequence shown in Table 1, the sequence of issued requests using the core-aware bank-first policy will be A-C-D-F-E-J-I-H-B-G. Request A and J access the same bank and come from the same source, thus J is scheduled before B. The same scheduling happens to request G and I.

3.2.4 Core-aware row-first policy

The core-aware row-first policy applies core-aware scheduling into row-first policy, which gives a higher priority to accesses from the same core in the same row. It enhances the row-first policy by taking consideration of the request source. For the requests shown in Table 1, the sequence of issued requests by this policy will be A-J-B-C-D-I-G-F-H-E. The difference between the result of this policy and the result of the row-first policy resides in the distinction in handling of requests A, B and J. According to the core-aware row-first policy, since request A and J have the exactly same source core and destination bank and row identifiers, they are scheduled first and together, followed by request B, which is distinguished from the scheduling sequence of A-B-J according to the row-first policy.

4. Experimental Setup

Without vendor's effort, it is not possible to modify the internal structure of an integrated memory controller to test the proposed scheduling policies. Instead, we used a simulator with an accurate representation of multi-core processors. We have conducted experiments to simulate and evaluate the four scheduling policies discussed above. The experiments focused on whether and how the proposed policies could improve the performance of parallel applications in a multi-core environment.

4.1 Simulation Environment

We adopted Simics [6] and Wisconsin Multifacet General Execution-driven Multiprocessor Simulator (GEMS) [7] as our architecture simulator. Simics provides a full-system functional simulation infrastructure. The GEMS is a set of timing simulator modules for modeling the timing of the memory system and microprocessors. It is capable of characterizing and evaluating the performance of multiprocessor hardware systems. The default memory scheduling in the GEMS adopts the bank-first policy to reorder memory requests. We have enhanced the current GEMS implementation by modifying the memory controller component and integrating the other three policies. In our experiments, we set the threshold value of the maximum number of

continuous requests from one core as 16 to prevent from starvation. Our experimental observations confirmed that this number is a proper threshold.

We configured the simulator to represent Sun SPARC processor architecture with Solaris 10 as target operating system. The summary of configuration for the simulated multi-core system is shown in Table 2. The memory parameters are set by referring to current main memory technology [14] and experiments in related research [10][13]. Please notice that Solaris 10 has its core scheduling schemes to schedule application and system tasks to different cores. We have not changed the task scheduling. We have only modified the memory access scheduling.

4.2 Benchmarks

We selected NAS Parallel Benchmarks (NPB) 3.2 OpenMP version for our experiments. NPB suite was developed for the performance evaluation of highly parallel supercomputers. [2]. We chose the following five classic kernel benchmarks to study the effect of memory access scheduling policy [1][4]. Each kernel benchmark was tested with size Class W.

EP: An embarrassingly parallel kernel, which generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive square annuli. It provides an estimate of the upper bound of achievable floating-point performance.

DC: A Data Cube benchmark. This benchmark is based on a data mining application and builds RB-tree to sort tuples from a dataset.

CG: A conjugate gradient method benchmark. This benchmark is used to compute an approximation of a large, sparse and symmetric positive definite matrix.

MG: A simplified multi-grid kernel. It tests short and long-distance data communication.

FT: A 3-D partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes.

Table 2: Machine Configuration

Component	Parameters
CPU	16 Sun SPARC processor cores, each core is 2GHz 4-way issue
L1 I-cache	16 KB, 4-way L1 cache on each core, 64 bytes cache line
L1 D-cache	16 KB, 4-way L1 cache on each core, 64 bytes cache line
L2 cache	256 KB, 4-way cache on each core, 64 bytes cache line
Cache Coherence protocol	Directory and MESI protocol [2]
FSB	64 bit, 800MHz (DDR)
Main Memory	4GB DDR2 PC2 6400 (5-5-5), 64 bit, burst length 8 Memory page is 4 KB
Channel/Rank/Bank	2/4/4 (a total 32 banks)
SDRAM Row Policy	Open Page
Address Mapping	Page Interleaving
Memory Access Pool	32 queues for each bank, each queue size is 16 entries
OS	Solaris 10

We have tested our proposed schemes with three sets of experiments, as shown in Table 3. The first set evaluates single application with single thread, which represents conventional single thread application running on multi-core environment. The simulated Solaris OS issues monitoring requests, which are not related to applications. In the first set of applications, memory requests compete with these OS related requests. Separating application related requests and giving them higher priority is expected to be beneficial to improve performance. The second set tested an application with multiple threads, and the third set evaluated multiple applications running concurrently in the system. In the third set, we randomly selected four benchmarks and ran with one thread and four threads respectively. In the last two sets of tests, multiple

threads compete in accessing memory with each other while competing with OS related requests.

Table 3: Experiment Configuration

Set	Benchmarks
Single application with single thread	ep-1: EP running with one thread dc-1: DC running with one thread cg-1: CG running with one thread mg-1: MG running with one thread ft-1: FT running with one thread
Single application with multiple threads	ep-4: EP running with four threads dc-4: DC running with four threads cg-4: CG running with four threads mg-4: MG running with four threads ft-4: FT running with four threads
Multiple applications	mix-1: dc-1, cg-1,mg-1 and ft-1 running concurrently mix-4: dc-4, cg-4,mg-4 and ft-4 running concurrently

5. Performance Evaluation and Analysis

We compare the performance of memory scheduling policies by analyzing the total number of memory requests and the waiting latency. We compare the number of memory requests, the latency of requests, and the execution time of benchmarks for four memory-scheduling policies: bank-first (bank), row-first (row), core-aware bank-first (core-bank) and core-aware row-first (core-row) policy. We take bank-first as the base scheme and normalize the results of other scheduling schemes according to the performance of bank-first scheme.

5.1 Analysis of the number of memory requests

Figure 4 shows the number of memory requests issued to DRAM for the first set of experiments and Figure 5 shows those of the second and third sets of the experiments. Note that these numbers are normalized to the numbers of bank-first policy. From these figures, it can be observed that the number of memory requests under core-aware bank-first is the lowest for EP, CG and FT, for both single-thread and four-thread executions. The core-aware row-first scheme achieves the lowest number of memory requests for DC and MG for these two executions.

The reason for these trends is that all memory requests of these applications are issued by some cores with good memory locality. These requests get grouped and are scheduled in bunch by using the core-aware bank-first and core-aware row-first schemes. Therefore, the overall efficiency is improved and fairness is

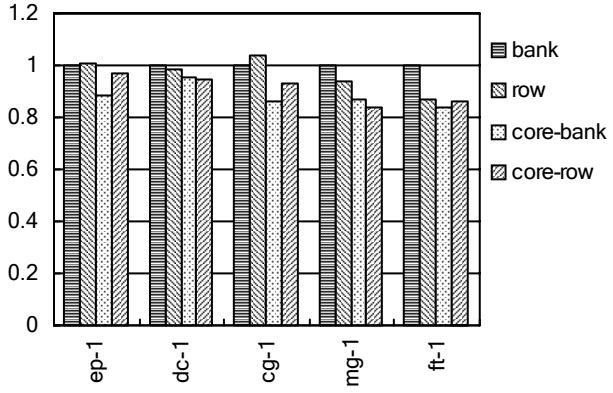

Figure 4. Number of memory requests for single application with single thread. All values are normalized to bank-first scheme.

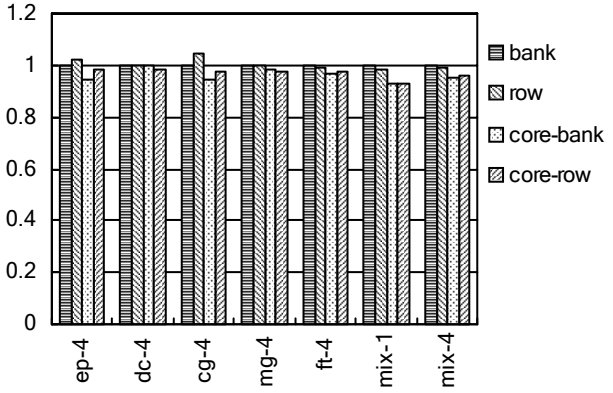

Figure 5. Number of memory requests for single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.

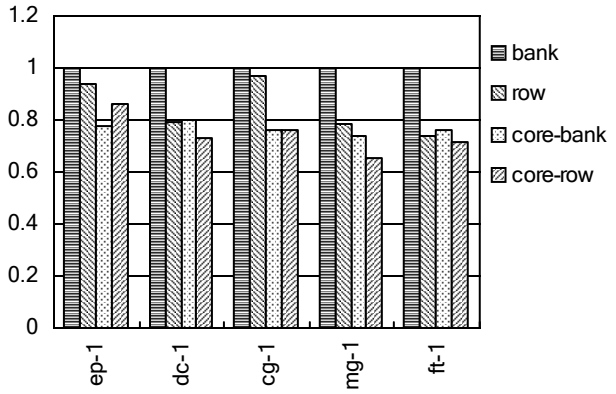

Figure 6. Waiting latency for single application with single thread. All values are normalized to bank-first scheme.

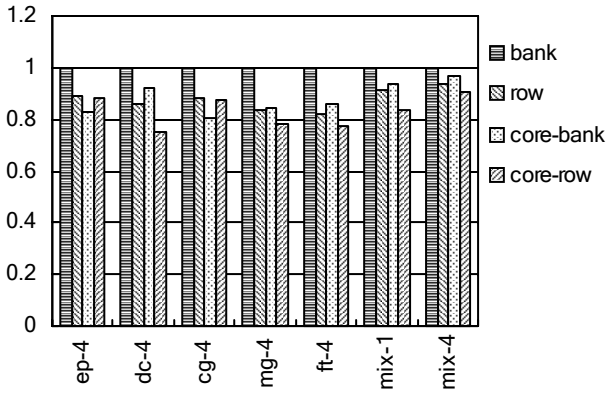

Figure 7. Waiting latency for single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.

maintained. The same reasoning applies to the results of the case with multiple applications, where core-aware bank-first and core-aware row-first schemes produced less memory requests compared with bank-first and row-first schemes when DC, CG, MG and FT ran concurrently with one thread/four threads.

Figure 4 and Figure 5 demonstrate that the row-first scheme produced fewer memory requests than the bank-first scheme when DC, MG and FT were executed with one thread or four threads, because the row-first scheme gives higher priority to the request hitting the same row if the application has good locality. There exist two exceptions: the row-first scheme produced more memory requests than the bank-first scheme when EP and CG were tested with one thread or four threads. We believe that this is because the row-first scheme does not distinguish memory requests by core information and treats requests from different cores equally, which results in that more requests from OS are scheduled by row-first schemes compared with bank-first scheme.

Compared with row-first scheme, the core-aware row-first scheme reduced memory requests by 5% on average for five benchmarks running with four threads, as shown in Figure 5. This is mainly due to the core-aware row-first scheme improving the row-first policy by taking the request source into consideration.

5.2 Analysis of waiting latency

The waiting latency of memory request is the waiting time due to blocked memory requests. Different scheduling schemes produce different waiting sequences of blocked requests, and these sequences decide the waiting time.

Figure 6 illustrates the waiting latency under various scheduling schemes, where all values are normalized to the numbers of bank-first scheme again for the first set of experiments. Figure 7 shows those results for the second and third sets of experiments. When EP and CG were executed with one thread or four threads, the core-aware bank-first scheme had the smallest waiting latency. As shown in Figure 7, compared with the bank-first scheme when EP and CG ran with four threads, the core-aware bank-first scheme decreases the latency by up to 17% and 20%, respectively. The traditional row-first scheme performed well in reducing waiting time because it schedules all requests accessing the same row together. However, if counting the effect of requests from the operating system, the row-first scheme may not perform well. The core-aware bank-first scheme can reduce the waiting latency by decreasing the total number of requests. It decreased the number of requests by 5% and 6% respectively, compared with the row-first scheme when EP and CG

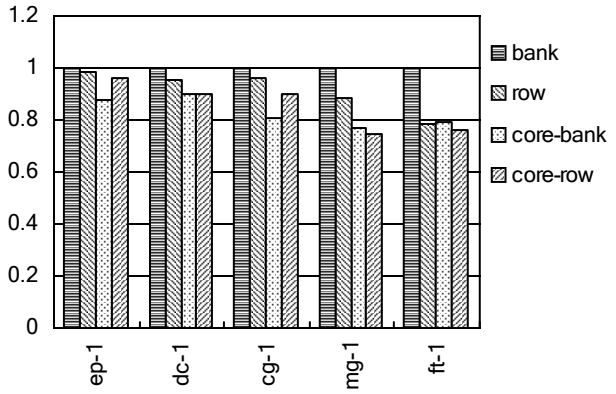

Figure 8. Execution time for single application with single thread. All values are normalized to bank-first scheme.

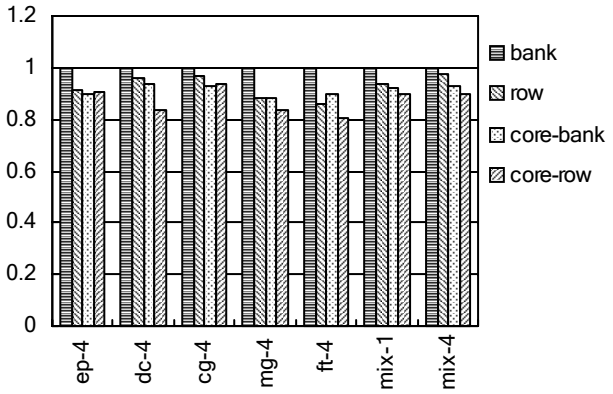

Figure 9. Execution time for single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.

were executed with four threads, and achieved the smallest waiting latency in EP and CG experimental tests as shown in Figure 5.

When DC, MG and FT were executed with one thread or four threads, the core-aware row-first scheme had the smallest waiting latency. As shown in Figure 7, when DC, MG and FT ran with four threads, the core-aware row-first decreased the latency by up to 25%, 22% and 23% compared with the bank-first scheme, and by up to 10%, 5% and 5% compared with the row-first scheme. The core-aware row-first scheme adapts to the row-first scheme, except when requests access the same row and the same bank, where the policy schedules them by giving higher priority to those requests with the same source.

In the case of running multiple applications concurrently, i.e. case mix-1 and mix-4, the core-aware row-first scheme had the smallest waiting latency. The latency is decreased by up to 16% and 10% compared with the bank-first scheme, as shown in Figure 7.

5.3 Analysis for execution time

Figure 8 demonstrates the execution time analysis with various scheduling schemes for the first set of experiments, where all values are normalized to the performance of bank-first scheme. Figure 9 shows the execution time analysis for the second and third sets of experiments. Both figures demonstrate that the core-aware schemes outperformed the bank-first and the row-first schemes for all benchmarks.

The core-aware bank-first scheme had the best performance when EP and CG were executed with one thread and four threads. As shown in Figure 8, when CG ran with one thread, the core-aware bank-first scheme reduced the execution time by 19% compared with the bank-first scheme, and by 15% compared with the row-first scheme. The reason is that all memory requests from CG were scheduled in bunch. For EP test with one thread, the execution time reduction by core-aware bank-first scheme was 12% compared with the bank-first scheme.

For DC, MG, and FT experiments with one thread and four threads, the core-aware row-first scheme had the best performance. In Figure 9, for DC, MG, and FT running with four threads, the execution time reduction by core-aware row-first scheme are 16%, 17% and 20%, respectively, compared with the bank-first scheme. The performance gain is 13%, 5% and 7%, respectively for DC, MG, and FT compared with the row-first scheme. The reason is that the core-aware row-first scheme has the smallest waiting latency.

For mix-4 shown in Figure 9, the core-aware row-first scheme achieves the best performance, and it decreases the execution time by up to 11% and 7%

respectively, compared with the bank-first and row-first schemes.

With core-aware bank-first scheme, the execution time has reduced by up to 17% and 10% on average for all five benchmarks running with one thread and four threads respectively, as shown in Figures 8 and 9. The performance improvement in the case with one thread is better than that of four threads. This phenomenon, we believe, is due to the impact of Solaris OS scheduling of threads. When an application is executed with four threads, the simulated Solaris OS randomly schedules the four threads to different cores. Depending on how the task is partitioned, each core may or may not have a better locality than the average of multiple cores. The core-aware schemes also schedule more requests from the cores running with OS daemon threads. Due to the increased system management of parallel processing such as maintaining consistency and synchronization, the performance improvement with one thread is better than that with four threads.

To summarize the observed results in Figure 9, the proposed core-aware policies improve the performance for all five benchmarks. Compared with the bank-first and row-first policy, the core-aware row-first policy reduced the execution time by 14% and 5% on average for all five benchmarks running with four threads, and by 10% and 5% on average for the two cases with multiple applications running concurrently. These performance gains are under the current Solaris' multi-core task scheduling, which is not designed with the consideration of memory access. We believe the newly proposed care-aware memory access scheduling strategy will achieve an even better performance if it is integrated into task scheduling.

6. Conclusion and Future Work

As multi-core architecture has become the norm of future high-performance processor chips, effective memory access scheduling has become timely and important for improving both application performance and overall system performance. Memory access scheduling in a multi-core environment is different from that of its single-core counterpart. This study investigated and evaluated various memory access scheduling techniques in a multi-core environment. We noticed that the source core of memory requests is an important factor in scheduling data access, and we have proposed novel core-aware memory access scheduling schemes, including the core-aware bank-first policy and the core-aware row-first policy, to consider core-awareness factor and to address the limitations of existing approaches.

We have performed comprehensive experiments to evaluate existing scheduling policies and the newly proposed policies. Experimental results confirmed that memory scheduling policies have great influence on memory waiting latency, and the proposed core-aware scheduling schemes decreased the latency considerably. For instance, when FT is running with four threads, the core-aware row-first policy reduced the latency by up to 23% compared with the traditional bank-first scheme, and by up to 6% compared with the row-first scheme. Experimental results also revealed that the proposed core-aware schemes reduced the execution time considerably. Compared with the bank-first and row-first policy, the core-aware row-first policy reduced the execution time by up to 20% and 7% respectively for FT running with four threads, and by up to 11% and 7% respectively for one mixed benchmarks represented as multiple applications.

The proposed core-aware memory access scheduling has a great potential. In this study, we have observed a considerable performance improvement. The performance can be improved further by making Solaris OS schedule threads on cores properly. In the current experiments, the OS schedules the threads without core-aware information. We expect vendors follow our idea and integrate core awareness into multi-core memory access scheduling in a physical memory controller. In the near future, we plan to

further explore OS task scheduling schemes with core-aware information, and thus to improve the effectiveness of our core-aware schemes further.

7. References

[1] Sadaf R. Alam, Richard F. Barrett, Jeffery A. Kuehn, Philip C. Roth, and Jeffrey S. Vetter, "Characterization of Scientific Workloads on Systems with Multi-Core Processors", *IEEE International Symposium on Workload Characterization*, 2006.

[2] D. Bailey, et al., "NAS Parallel Benchmarks", *NAS technique report RNR-94-007*, NASA Ames Research Center, 1994.

[3] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Press, 1998.

[4] Ibrahim Hur and Calvin Lin, "Adaptive History-Based Memory Schedulers", *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[5] Haoqiang Jin, Michael Frumkin, and Jerry Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance", *NAS Technical Report NAS-99-011*, NASA Ames Research Center, 1999.

[6] Peter S. Magnusson et al., "Simics: A Full System Simulation Platform", *IEEE Computer*, Vol. 35, No. 2, 2002.

[7] Milo M.K. Martin, Daniel J. Sorin, et al., "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", *Computer Architecture News*, 2005.

[8] Onur Mutlu and Thomas Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors", *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[9] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens, "Memory Access Scheduling", *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[10] Jun Shao and Brian T. Davis, "A Burst Scheduling Access Reordering Mechanism", *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[11] Wm. A. Wulf and Sally A. McKee, "Hitting the memory wall: implications of the obvious", *Computer Architecture News*, March 1995.

[12] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang, "A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality", *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000.

[13] Zhichun Zhu and Zhao Zhang, "A Performance Comparison of DRAM Memory System Optimizations for SMT Processors", *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[14] Micron Technology Incorporation. "1Gb DDR2 SDRAM Component: MT47H128M8B7-25E", June 2006.