Memory Access Scheduling Schemes for Systems with Multi-Core Processors

Hongzhong Zheng[1] Jiang Lin[2] Zhao Zhang[2] Zhichun Zhu[1]

[1]Department of Electrical and Computer Engineering
University of Illinois at Chicago
{hzheng, zhu@ece.uic.edu}

[2]Department of Electrical and Computer Engineering
Iowa State University
{linj, zzhang@iastate.edu}

Abstract

On systems with multi-core processors, the memory access scheduling scheme plays an important role not only in utilizing the limited memory bandwidth but also in balancing the program execution on all cores. In this study, we propose a scheme, called ME-LREQ*, which considers the utilization of both processor cores and memory subsystem. It takes into consideration both the long-term and short-term gains of serving a memory request by prioritizing requests hitting on the row buffers and from the cores that can utilize memory more efficiently and have fewer pending requests. We have also thoroughly evaluated a set of memory scheduling schemes that differentiate and prioritize requests from different cores. Our simulation results show that for memory-intensive, multiprogramming workloads, the new policy improves the overall performance by 10.7% on average and up to 17.7% on a four-core processor, when compared with scheme that serves row buffers hit memory requests first and allows memory reads bypassing writes; and by up to 9.2% (6.4% on average) when compared with the scheme that serves requests from the core with the fewest pending requests first.*

1 Introduction

Commodity processors have used multi-core and multithreading architectures extensively. This trend makes memory bandwidth a crucial resource – each extra core or thread generates extra memory traffic. Memory access scheduling, the reordering of concurrent memory requests from the processor, plays an increasingly important role for those processors [10, 8, 5, 14, 3, 20, 6, 13, 15]. Previous research studies had focused on memory access scheduling for processors of a single core and a single thread context; only recently has the focus been shifted to multi-core and multithreaded processors [19, 12, 11].

In order to achieve the best overall performance from a multi-core processor, all processor cores need to be well utilized. Therefore, the memory access scheduling should minimize the memory stall time for all cores. On the other hand, memory bandwidth is a crucial resource and therefore the utilization of memory bandwidth is of the first priority. The two objectives contradict each other in many cases. For example, requests from each core may be served in the round-robin order to avoid stalling any processor core for a very long time. However, because of the spatial locality in the memory access stream from a single core, it usually improves bandwidth utilization by serving multiple requests from a single core continuously. Thus, the memory access scheduling schemes for multi-core systems need to consider the utilization of both the processor cores and the memory subsystem, while existing schemes mainly focus on the second part.

To address this issue, we propose a new approach that differentiates and prioritizes memory requests from different processor cores by considering the *memory efficiency* of application running on each core, and integrates it with conventional memory scheduling policies that emphasizes improving the memory bandwidth utilization and reducing the average memory latency. The motivation of prioritizing requests based on the coming cores is that each application benefits differently from extra memory bandwidth. Thus, for multi-core systems running multiprogramming workloads, allocating more memory bandwidth to an application that can utilize the bandwidth more efficiently will improve the overall performance without jeopardizing other applications.

The memory efficiency of an application is defined as the ratio of its IPC to its memory bandwidth usage under the single-core running environment. The information can be collected in many ways, e.g. off-line or on-line profiling, which is feasible and having low hardware overhead. We have combined this new metric into least-request (LREQ), an existing memory access scheduling scheme proposed for SMT processors [19], which prioritizes memory requests from the thread with the fewest pending requests. The new scheme, called *ME-LREQ*, uses the workload's memory efficiency and the number of pending requests of the core to schedule concurrent requests.

We have evaluated several memory scheduling schemes, including Round-Robin, LREQ, ME-LREQ and others, by using Hit-First with Read-First (HF-RF) as the performance baseline, which serves row buffers hit memory requests first with Read-bypass-Write. The details of those schemes are discussed in Section 2 and Section 3. We have confirmed that memory access scheduling plays a significant role in the performance of multi-core processors; and the proposed ME-LREQ scheme significantly outperforms the others. Our cycle-accurate simulation shows that for memory-intensive workloads constructed from SPEC2000 programs and on a four-core processor, the Round-Robin scheme may improve performance over HF-RF by up to 5.6% and the LREQ scheme may improve performance by up to 8.1%. By comparison, the proposed ME-LREQ may improve the performance by up to 17.7%. The average improvement of ME-LREQ over HF-RF is 10.7%. Additionally, the improvement of ME-LREQ increases with the number of processor cores. For example, the performance improvement is up to 21.4% over HF-RF and up to 12.9% over LREQ on eight cores, and 19.9% and 10.3% on average, respectively.

2 Conventional Memory Scheduling Schemes

Previous studies have shown that even for single-core and single-threaded processors, it is frequent that multiple memory requests are clustered together and occur in a short time period. Memory scheduling schemes are proposed to exploit this memory-level parallelism for utilizing DRAM optimization features. They can effectively reduce the average latency of concurrent requests and improve the memory bandwidth utilization for both single-threaded and multi-threaded processors [10, 8, 5, 14, 3, 20, 6, 13, 19, 15].

FCFS and Read-First. The naive first-come first-serve scheme serves memory requests according to their arriving order. A simple extension is to serve memory read requests before write requests since the read requests will cause the processor to stall and write requests normally can be well handled by write buffers.

Hit-First. DRAM has a three-dimensional structure, bank, row and column. A request hitting in the row buffer has shorter latency than a row buffer miss. The Hit-First scheme schedules row buffer hits before misses to reduce the average memory access latency and improve the bandwidth utilization [5, 14].

Least-Request. The least-request scheme assigns the highest priority to the requests from the thread with the fewest pending requests for systems with SMT processors [19]. The rationale is that returning a request from that thread is likely to release more waiting instructions dependent on the request than returning a request from other threads.

Round-Robin. To make fair scheduling among all cores, the memory controller can serve the requests from each core in a round-robin way. This simple scheme may reduce the long waiting time for cores with poor locality or low memory bandwidth demands. However, it destroys the spatial locality available in memory access streams; and does not consider different demands from different programs.

3 Memory Efficiency-based Scheduling Scheme

3.1 Memory Efficiency: Qualitative and Quantitative Definitions

To achieve good overall performance for a multi-core systems, all processor cores and the memory subsystem need to be well utilized. Thus, a memory scheduling scheme designed for multi-core processors must consider several issues together. First of all, a good scheme should adopt the approaches that have been approved successful in improving memory performance such as read-first and hit-first, since all processor cores will benefit from the reduction on the average memory latency. Second, a good scheme should differentiate requests from different cores; and when prioritizing requests, it should consider the program execution at all cores – if serving a given request may allow the waiting core to proceed for more instructions than serving other requests without starving other cores, then it is a good choice. Third, a scheme can be sophisticated but must have a simple implementation; otherwise, it cannot be practically implemented in the memory controller.

We propose the use of *memory efficiency* in memory access scheduling to meet all these requirements. Qualitatively, the memory efficiency of a program indicates the amount of work that can be done if its memory request is served. In other words, allocating additional memory resource to a program with high memory efficiency (by serving its requests first) can get better performance return than allocating the resource to a program with low memory efficiency. In addition, since the performance of a program with low memory efficiency is less sensitive to the memory resources allocated, such scheduling scheme can improve the overall performance with only small impact on programs with low memory efficiency.

Quantitatively, the memory efficiency of an application can be represented by the ratio of its number of committed instructions to its memory bandwidth usage. In general, this depends on the application's inherent behavior, the system architectural configuration and also the behavior of other applications running concurrently. The information can be collected either during run-time or from off-line profiling. A reasonable on-line scheme can detect the changes of running phases and environment, dynamically adapt the value of memory efficiency, and be more accurate. On the other hand, an off-line profiling only pays one-time overhead for different running instances. For many applications, the behavior can be represented by a small number of program slices of the application; and thus it is feasible to get the approximate information of memory efficiency off-line by running representative program slices under single-core and single-thread environment. We define memory efficiency as follows:

$$\text{ME[i]} = \text{IPC}_{\text{single}}[\text{i}]/\text{BW}_{\text{single}}[\text{i}], \qquad (1)$$

where ME is the memory efficiency, i is the identity of a program thread, and $\text{IPC}_{\text{single}}[\text{i}]$ and $\text{BW}_{\text{single}}[\text{i}]$ are the IPC value and the memory bandwidth usage (in GB/s), respectively, of the application running on a single-core and single-threaded processor with the same core configuration.

Memory efficiency can address the shortcomings of the existing memory access scheduling schemes in the context of multi-core processors. For example, FCFS and Least-Request are representative ones. In their scheduling, the scheduling decision is solely based on the microarchitectural status observed in a very short time window; no long-term property of the program execution is considered. Nevertheless, those schemes still have their merits for multi-core processors. The FCFS scheme, for example, is a natural choice and has a simple implementation. The Least-Request scheme allows a program thread with fewest memory requests to move faster than the others, which usually improves the accumulated instruction throughput in a short time window.

3.2 ME-LREQ Scheme: Approach and Implementation

Memory efficiency represents the long-term property of program execution; while existing schemes make memory scheduling decision solely based on the microarchitectural status observed in a very short time window. A good memory scheduling scheme for multi-core systems needs to consider both the long-term and short-term effects in order to well utilize both the processor cores and memory system.

We have proposed and designed a new scheme called *ME-LREQ*, which integrates the Memory Efficiency indicator into the Least-Request scheme. Requests from threads that have higher memory efficiency and fewer pending requests have higher priority than requests from other threads. In addition, reads and row buffer hits have higher priority than writes and row buffer misses, respectively. We choose Least-Request because a previous study [19] has shown that it has better performance than other schemes on SMT processors and is easy to implement. There can be various implementations of the ME-LREQ approach. Our scheme uses the following formula to determine the priority of all pending requests belonging to a program thread running at core i at a given time:

$$\text{Priority[i]} = \text{ME[i]}/\text{PendingRead[i]}, \qquad (2)$$

where ME[i] is the memory efficiency of the application running on the core i and PendingRead[i] is the number of current pending read requests of the core. A program of a high priority value at a given moment receives a high priority in memory access scheduling. The scheme only considers pending read requests because write requests usually have small performance impact.

The above scheme cannot be directly implemented in memory controller because of its logic design complexity: The calculation involves a series of division; and division units are very expensive. We use a simple and approximate implementation that uses a hardware table to store the pre-computed and converted division results. Figure 1 shows the addition to the memory controller. The *workload priority tables* store a set of priority numbers for each application and every possible number of pending requests. We assume that the memory efficiency value is collected by profiling execution using hardware performance counters for instruction throughput and last level cache miss number, which are widely available on modern processors. The values calculated by Equation 2 for each possible pending request number are scaled approximately and then stored into the tables. We assume that the tables are initialized by OS at the time of program loading and context switching. The hardware implementation cost is small for today's memory controller. For example, in our experimental setup, the maximum number of pending memory requests per thread is 64, and each table entry stores a 10-bit priority information. The total number of bits in the tables is only $N \times 64 \times 10$ or $640N$ bits for an N-core system.

The memory controller maintains a read request queue and a write request queue, plus two counters for the number of outstanding read and write requests for each core. At the time of scheduling (when a memory channel is available for a memory transaction), the numbers of the outstanding read requests of all threads, if not zero, are used to access the workload priority tables in parallel. The output is the current priorities of all threads. A set of comparators is used to select the thread with the highest priority, and then the first read request of the selected thread is scheduled. A tie of equal priority may be broken by a random selection. Write requests are scheduled after read requests under normal conditions. However, when the number of pending write requests reaches a threshold (half of the memory buffer size in our experiments), writes are scheduled first until the number falls below another threshold (one-fourth of the buffer size).

4 Experimental Setup

4.1 Simulation Environment

We use M5 [1] as the base architectural simulator and extend its memory part to simulate the DDR2 memory system in details. The simulator keeps tracking the states of each memory channel, DIMM and bank, and schedules pending memory requests according to the used scheduling policy. Based on current memory states, memory commands are issued according to the hit-first policy, under which row buffer hits are scheduled before row buffer misses. Reads

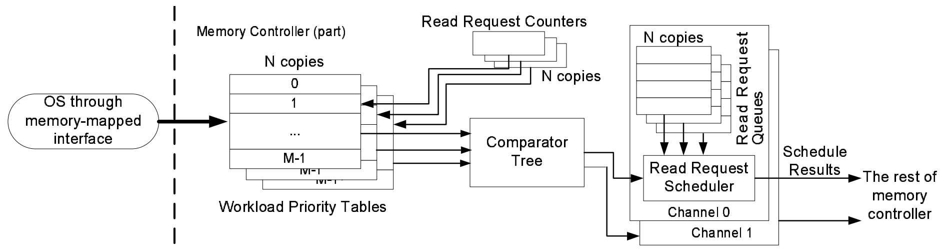

Figure 1: The implementation of *ME-LREQ* in memory controller for N-core processor (M-entry request buffer shared by N cores).

Parameters	Values
Processor	1/2/4/8 cores, 3.2 GHz, 4-issue per core, 16-stage pipeline
Functional units	4 IntALU, 2 IntMult, 2 FPALU, 1 FPMult
IQ, ROB and LSQ size	IQ 64, ROB 196, LQ 32, SQ 32
Physical register num	228 Int, 228 FP
Branch predictor	Hybrid, 8k global + 2K local, 16-entry RAS, 4K-entry and 4-way BTB
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 64B line, hit latency: 1 cycle Inst/3-cycle Data
L2 cache (shared)	4MB, 4-way, 64B line, 15-cycle hit latency
MSHR entries	Inst:8, Data:32, L2:64
Memory	2 logic channels (2 physical channels each), 2-DIMMs/phy. channel, 4-banks/DIMM
Channel bandwidth	800MT/s, 16byte/logic-channel, 12.8GB/s/logic-channel
Memory controller	64-entry buffer, 15ns overhead
DRAM latency	5-5-5, precharge 12.5ns, row access 12.5ns, column access 12.5ns

Table 1: Major simulation parameters.

are scheduled before write operations under normal conditions. However, when outstanding writes occupy more than half of the memory buffer, writes are scheduled first until the number of outstanding writes drops below one-fourth of the memory buffer size. The memory transactions are pipelined whenever possible. The simulation uses the close page mode with cache line interleaving rather than the open page mode with page interleaving since it is more widely used in practice. Table 1 shows the major simulation parameters.

In order to limit the simulation time while still emulating the representative behavior of program executions, simulation points are picked up according to SimPoint 3.0 [16]. To make fair evaluation, we use different simpoints for profiling and performance comparison. We randomly select a single simpoint using 10 million instruction slice for profiling and measure the programs' memory efficiency, which is presented in Table 2. We also randomly select simpoints using 100 million instruction slices in the construction of multi-core workloads for evaluating the effectiveness of memory scheduling schemes. The execution of the workloads is stopped when the last processor core commits 100 million instructions. Other processor cores will reload their applications and keep running after committing 100 million instructions, although the statistics such as the IPC values are collected only for the 100 million instructions of the simpoint. To compare the performance of a workload running on a multi-core processor with different scheduling configurations, we adopt the SMT speedup [17] as the performance metric. It is calculated as $\sum_{i=1}^{n}(IPC_{multi}[i]/IPC_{\text{single}}[i])$, where n is the total number of cores, $IPC_{multi}[i]$ is the IPC value of the application running on the ith core under the multi-core execution and $IPC_{single}[i]$ is the IPC value of the same application under single-core execution. Using the performance metric and running method can prevent biased approaches, such as giving all resources to the application with the highest ILP degree, from producing artificial performance gains.

4.2 Workload Construction

In our experiments, each processor core is single-threaded and runs a distinct application. We classified the twenty-six benchmarks of the SPEC2000 suite into MEM (memory-intensive) and ILP (compute-intensive) applications. The MEM applications are those getting more than 15% performance gain in the perfect memory system (with zero latency and infinite bandwidth) compared with in the two-channel DDR2 system in Table 1. Table 2 also shows the *Memory Efficiency* (*ME*) value of each application. We construct the multi-programming workloads randomly using these applications as shown in Table 3. Each workload is named by the number of cores, the workload type and its workload index. For example, workload 2MEM-1 consists of two memory-intensive applications *wupwise* and *swim*; and workload 4MIX-2 mixes two MEM applications *mgrid* and *applu* with two ILP applications *mesa* and *apsi*.

5 Performance Evaluation and Analysis

5.1 Performance Impact of Scheduling Schemes

We have evaluated the performance of five memory scheduling schemes: Hit-First with Read-first (HF-RF), Memory Efficiency only (ME), Round Robin (RR), Least Request (LREQ) and Memory Efficiency with Least Request (ME-LREQ). In the following discussions, we will only use the shorthand names of those schemes.

Figure 2 shows the SMT speedup (as defined above) of those schemes. As expected, the performance difference of those schemes enlarges with the number of cores increasing. For the memory-intensive workloads and using HF-RF as the reference point, the performance gains from LREQ

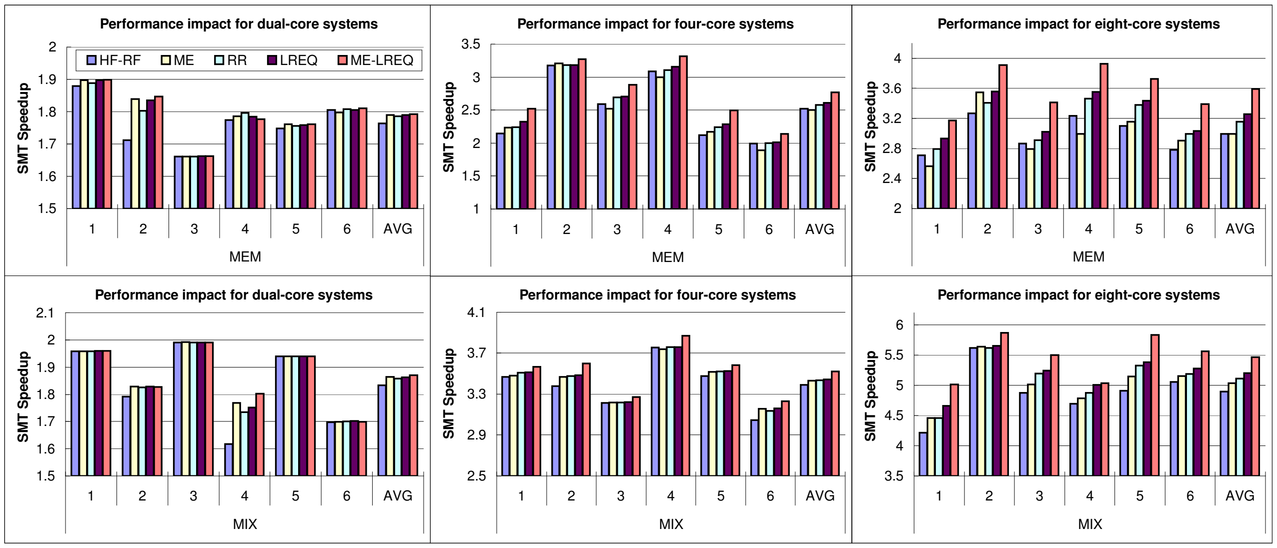

Figure 2: Performance impact of the scheduling schemes with different core configurations and varied workloads.

Application	Code	Class	ME.	App.	Co.	Cl.	ME.
gzip	a	I	192	facerec	n	M	40
wupwise	b	M	15	ammp	o	I	280
swim	c	M	2	lucas	p	M	1
mgrid	d	M	4	fma3d	q	M	4
applu	e	M	1	parser	r	I	38
vpr	f	M	27	sixtrack	s	I	80
gcc	g	M	22	eon	t	I	16276
mesa	h	I	78	perlbmk	u	I	2923
galgel	i	M	8	gap	v	M	7
art	j	M	20	vortex	w	I	51
mcf	k	M	1	bzip2	x	I	216
equake	l	M	2	twolf	y	I	951
crafty	m	I	222	apsi	z	I	36

Table 2: Application code, class (I-ILP, M-MEM) and memory efficiency value (ME.). Memory efficiency values are collected using simpoints different from those used in the multi-core workload construction for the performance evaluation in Section 5.

Group	Workload	code	Workload	code
2-core	2MEM-1	bc	2MIX-1	ab
	2MEM-2	de	2MIX-2	cr
	2MEM-3	fj	2MIX-3	hd
	2MEM-4	kl	2MIX-4	ez
	2MEM-5	np	2MIX-5	mf
	2MEM-6	qv	2MIX-6	oj
4-core	4MEM-1	bcde	4MIX-1	arbc
	4MEM-2	fgij	4MIX-2	hzde
	4MEM-3	npqv	4MIX-3	mofj
	4MEM-4	bdkl	4MIX-4	stkl
	4MEM-5	qvce	4MIX-5	uxnp
	4MEM-6	cjkq	4MIX-6	ywqv
8-core	8MEM-1	bcdefjkl	8MIX-1	arhzbcde
	8MEM-2	npqvbdfv	8MIX-2	mostfjkl
	8MEM-3	gicecjkq	8MIX-3	uxywnpqv
	8MEM-4	bcdenpqv	8MIX-4	armobcfj
	8MEM-5	qvcefjkl	8MIX-5	uxhznpde
	8MEM-6	bvgicjpq	8MIX-6	stywqvfk

Table 3: Workload mixes.

and ME-LREQ are insignificant on the two-core platform and workloads. On the four-core platform, the average performance gain of LREQ increases to 4.0% and ME-LREQ to 10.7%, respectively. On eight cores, the gains of LREQ and ME-LREQ further increase to 8.7% and 19.9% on average, respectively. The more cores the processor has, the larger room the memory access scheduling may make a difference.

ME-LREQ performs much better than the other schemes for memory-intensive workloads on four and eight cores. Using the HF-RF scheme as the reference point, the average speedup is 10.7% on four cores and 19.9% on eight cores, and the maximum speedup is 17.7% and 21.4%, respectively. LREQ is the second best scheme. It was designed to consider the short-term urgency of memory requests [19]; the result indicates that this consideration is still important for multi-core processors. If using LREQ as the reference point, the average speedup of ME-LREQ is 6.4% and 10.3% on four cores and eight cores, respectively. ME-LREQ considers both the short-term and long-term gain of individual memory requests; and the performance differences confirm the significance of considering the long-term gain.

Comparing all schemes, the performance rank in the increasing order is as follows: ME, HF-RF, RR, LREQ and ME-LREQ. A point worth noting is that ME is a fixed priority scheme, i.e. the programs and cores have fixed priority during the whole program execution. (We assume that a program is always assigned to a given core, though the schemes can be easily extended to handle the other case.)

The ME scheme gives fixed priority to threads according to their overall memory efficiency, putting the long-term gain in program progress as the only priority. It even performs slightly worse than HF-RF in general (average -0.6%). The main reason, we believe, is that it ignores the dynamic change of the gain from serving a memory request. For example, during a period that a high-priority thread generates a burst of memory requests, the requests from the other threads are blocked unconditionally even if there are only a few of them. The RR scheme serves requests from all threads in round-robin order and avoids the problem of ME scheme. However, it does not perform as well as the LREQ or ME-LREQ scheme since it does not differentiate requests from different cores at all; and it may destroy the spatial locality in the memory access streams.

The average speedup of MIX workloads by the ME-LREQ scheme over HF-RF scheme is 4.0% and 12.1% on four cores and eight cores, respectively. The average improvement on four cores is not as significant as on eight cores, but is still obvious for some workloads, e.g. 6.6% for 4MIX-2. When compared with the LREQ scheme on the eight-core systems, the ME-LREQ scheme always outperforms it and achieves an average speedup of 5.1% and a maximum speedup of 8.4%. Even for MIX workloads, the demand on the memory subsystem is high when the number of cores increases to eight.

5.2 Comparison of Simple and Fixed Priority Schemes

There is a natural question to ask: What will happen if we assign a different priority sequence other than that from ME? In other words, do the performance gains of ME and ME-LREQ come from the simple fact that fixed priorities are given to all cores? If that would be the case, we do not have to use the relatively complex design based on memory efficiency. To answer this question, we compare the SMT speedups of four schemes on the four-core platform: HF-RF, ME, FIX-3210 and FIX-0123. The latter two schemes are two random and fixed priority schemes: FIX-3210 gives the priority to cores in the order of 3, 2, 1, and 0; and FIX-0123 gives the priority in the reverse order.

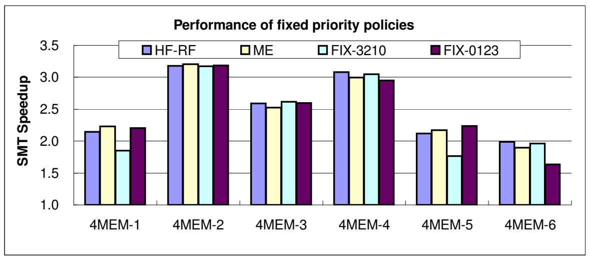

Figure 3: Performance impact of simple and fixed priority scheduling schemes on the four-core systems.

Figure 3 shows the SMT speedups of the four schemes. It is obvious that fixing the priority degrees randomly has a noticeable but unpredictable performance impact on the multi-core systems. A given workload may be improved by a suitable priority setup and may also be harmed by an unsuitable setup. For instance, compared with the HF-RF scheme, the workload 4MEM-1 gets a 2.8% performance improvement by the FIX-0123 scheme but a 13.8% performance degrade by the FIX-3210 scheme. On the other hand, workload 4MEM-6 gets an 18.0% performance loss with the FIX-0123 scheme, but only 1.5% performance downgrade with the FIX-3210 scheme. In comparison, the ME scheme achieves relatively consistent performance for those workloads. This indicates that using the ME information to guide the priority setup is necessary. Worth noting is that fix-priority schemes cannot reflect the run-time requirements of each core; and thus in order to achieve good performance, the scheduling scheme needs to integrate run-time information as the ME-LREQ scheme does.

5.3 Impact on Memory Read Latency and System Fairness

Next, we will analyze the impact of those scheduling schemes on memory read latency. For clarity, we use the four-core systems and memory-intensive workloads as examples. (Other systems and workloads show similar results.) The left part of Figure 4 compares the average latency of read requests under the five scheduling schemes. The results indicate that the scheduling schemes have significant impact on the average read latency. This latency also depends on the workloads' behavior and varies a lot across the memory-intensive applications. The ME-LREQ scheme gets the lowest average read latency for all workloads, which is consistent with the overall performance results. For instance, it reduces the average read latency of the workload 4MEM-1 from 613 cycles to 490 cycles, compared with the HF-RF scheme. As a result, the performance of this workload is improved by 17.4%. For all workloads, the ME-LREQ scheme has the average read latency of 323 cycles; while the HF-RF scheme has the latency of 376 cycles. The reduction on the read latency is translated to an average performance gain of 10.7% by the ME-LREQ scheme, compared with the HF-RF scheme. On average, the ME scheme reduces the latency slightly compared with the HF-RF scheme; the RR and LREQ schemes further reduce the latency; and the ME-LREQ gets the shortest read latency.

The right part of Figure 4 further shows the variants of the average read latency of individual processor core using two workloads 4MEM-1 and 4MEM-5 as examples. The HF-RF scheme serves requests from different cores as if they were produced by a single core. Thus, each core observes almost the same average read latency. The RR scheme attempts to serve requests from each core in a round-robin way. The cores with more pending requests will have longer waiting time than others. The latency under this scheme has larger variant across different cores than the HF-RF scheme, but is still within a narrower range than that under the other schemes.

The ME scheme causes the latency varied the most across cores. For example, for the workload 4MEM-5, requests of core 1 get the highest priority and requests of core 3 get the lowest priority. As a result, the average read latency of requests from core 1 is much shorter than that of core 3 (289 vs. 1042 cycles). This indicates that a fixed priority scheme may stave cores of low priority. The extreme long read latency of core 3 may make this core having large

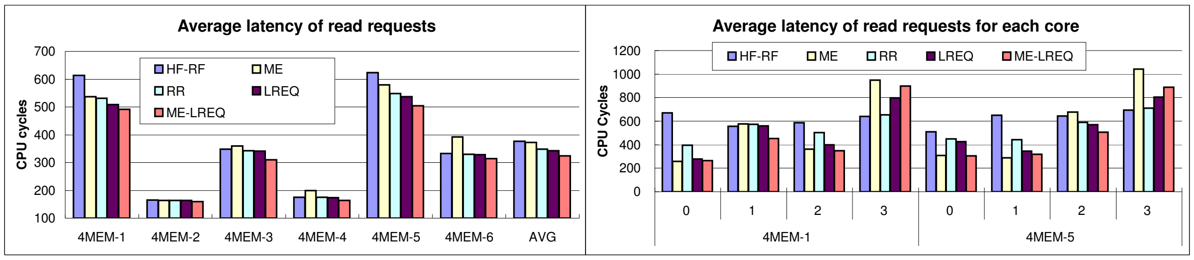

Figure 4: Comparison of memory read latency.

memory stall time (its IPC drops from 1.16 of HF-RF to 0.66 of ME). In addition, the extreme imbalance of read latency across processor cores produces higher average latency than that of the LREQ and ME-LREQ schemes and limits the overall performance. By comparison, ME-LREQ does not have this issue because the priority of each program changes dynamically when the number of its pending read requests increases or drops. For core 3 of 4MEM-5, for example, ME-LREQ reduces the average latency to 887 cycles. After all, ME-LREQ performs the best because it considers both the long-term and short-term gains of serving a memory request. The average read latency, in general, is closely related to the short-term gain.

The performance gain from the proposed scheduling policy would be biased if it sacrifices the fairness among the concurrently running applications. Figure 5 shows that the *ME-LREQ* policy can even improve the fairness in addition to improving performance (as shown previously in Figure 2). Here, we follow two previous studies [4, 11] and adopt *unfairness* as the metric, which is the ratio of the maximum performance slowdown to the minimum performance slowdown among all concurrent applications/threads using their performance on the single-core and single-threaded system as the reference. For clarity, we use the four-core systems and memory-intensive workloads as examples. (Other systems and workloads show similar results.)

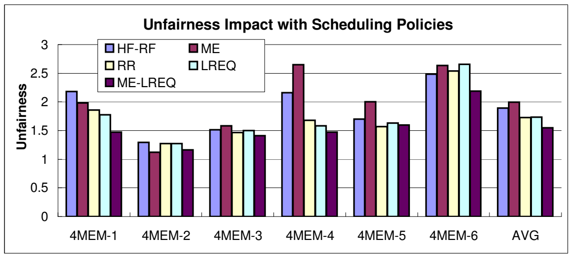

Figure 5: Comparison of fairness with scheduling policies.

The results indicate that ME-LREQ policy achieves the best fairness among those scheduling policies. Compared with HF-RF, RR and LREQ, ME-LREQ can reduce unfairness by 7.9%, 7.6% and 16.6% on average, respectively. The maximum unfairness reduction of ME-LREQ is 32.5% for the workload 4MEM-1 and it reduces the unfairness by more than 10% on four of the six workloads. Compared with LREQ, the scheme with second best performance, ME-LREQ can reduce unfairness by 9.7% on average and by up to 17.6% (for 4MEM-6). It is not surprise that ME lowers fairness by 4.7% on average and by up to 22.4% (for 4MEM-4) compared with HF-RF because of its uneven resource allocation by fixed scheduling priority. In comparison, ME-LREQ combines both the short-term factor from current memory sub-system status and long-term factor from processor utilization to determine the memory scheduling priority for individual memory request of each core dynamically. This can avoid excessive memory resource allocation to memory-intensive workloads and starvation of non-memory-intensive workloads. Thus, it improves the fairness among current applications. At the same time, the even memory resource allocation by ME-LREQ can also reduce memory stall time for each processor core and fully utilize all processor cores to improve the overall performance.

6 Related Work

Two previous studies [7, 18] used a metric Mem/Uop, which is the ratio of memory transactions to micro-ops (Uop) retired, to guide the power management of single-thread processors. Our definition of memory efficiency uses a similar formula, but the concept serves for the purpose of memory access scheduling for the best performance. The performance of a generic main memory system is characterized by its latency and bandwidth. Memory access scheduling is an effective technique in reducing the average memory latency and improving bandwidth utilization for streaming applications as well as general-purpose applications. Moyer develops compiler techniques to reorder non-caching accesses for stream-oriented computations [10]. McKee and Wulf study the effectiveness of five access ordering schemes on a uniprocessor system [9] Hong et al. study the performance impact of access ordering for inner loops of streaming computations on a Direct Rambus system [5]. Rixner et al. discuss multiple memory access scheduling policies and evaluate their performance impact on media processing applications [14]. The Impulse memory controller supports application-specific optimizations through address translation (remapping) to improve bus and cache utilization, and also supports prefetching at the memory controller to hide the cost of remapping [2]. Fine-grain scheduling schemes [3, 20] can effectively improve the performance of multi-channel memory systems. Hur and Lin propose adaptive history-based scheduling policies that select the next memory operation based on the recent memory access history to minimize the expected delay and match the program's mixture of reads and writes [6]. Rixner analyzes the effects of policies utilizing channel buffers of virtual channel SDRAM and memory access scheduling schemes in reducing memory access latency in web servers [13]. Zhu and Zhang evaluate memory optimizations for the SMT processors and propose thread-aware scheduling schemes based on the pending request number and processor resource usage [19]. Jun and Brian propose a burst scheduling access reordering mechanism for the single thread processor to maximize the data bus utilization by clustering the accesses on the same row of the same bank [15]. Recent studies from Nesbit et al. [4] and Mutlu et al. [11] propose fair scheduling policies to balance the memory resource usage among the multi cores on chip. Different from those previous studies, our work focuses on the memory scheduling scheme for multi-core processors to improve overall performance without sacrificing fairness.

7 Conclusion

In this study, we thoroughly evaluate the performance impact of a set of memory scheduling policies on the multi-core processors, including a new policy called *ME-LREQ*. The *ME-LREQ* policy considers both the global memory efficiency of a program and the current number of pending requests. The results show that the new policy outperforms the other policies significantly for memory-intensive workloads; and the improvement increases with the number of cores. It successfully combines a short-term optimization objective and a long-term objective on the multi-core processor platform. The current policy uses off-line profiling information to determine the global memory efficiency. In the future, we plan to study online methods that can dynamically predict the memory efficiency of a program as well as to explore other design choices in the combination.

Acknowledgment

We appreciate the constructive comments from the anonymous reviewers. This work is supported in part by the National Science Foundation under grants CCF-0541408 and CCF-0541366.

References

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] J. Carter, W. Hsieh, L. Stoller, M. Swansony, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proc. of the Fifth Intl. Symp. on High-Performance Computer Architecture*, pages 70–79, Jan. 1999.

[3] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Proc. of the 28th Intl. Symp. on Computer Architecture*, pages 62–71, June 2001.

[4] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, pages 149–160, Dec. 2006.

[5] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a Direct Rambus memory. In *Proc. of the Fifth Intl. Symp. on High-Performance Computer Architecture*, pages 80–89, Jan. 1999.

[6] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proc. of the 37th Intl. Symp. on Microarchitecture*, pages 343–354, Dec. 2004.

[7] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, pages 359–370, Dec. 2006.

[8] S. A. McKee, A. Aluwihare, B. H. Clark, R. H. Klenke, T. C. Landon, C. W. Oliver, M. H. Salinas, A. E. Szymkowiak, K. L. Wright, W. A. Wulf, and J. H. Aylor. Design and evaluation of dynamic access ordering hardware. In *Proc. of the Tenth Intl. Conf. on Supercomputing*, pages 125–132, May 1996.

[9] S. A. McKee and W. A. Wulf. Access ordering and memory-conscious cache utilization. In *Proc. of the First Intl. Symp. on High-Performance Computer Architecture*, pages 253–262, Jan. 1995.

[10] S. A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, Department of Computer Science, Apr. 1993. Also as TR CS-93-18.

[11] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of the 40th Intl. Symp. on Microarchitecture*, pages 208–222, Dec. 2007.

[12] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing CMP Memory Systems. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, pages 208–222, Dec. 2006.

[13] S. Rixner. Memory controller optimizations for web servers. In *Proc. of the 37th Intl. Symp. on Microarchitecture*, pages 355–366, Dec. 2004.

[14] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. of the 27th Intl. Symp. on Computer Architecture*, pages 128–138, June 2000.

[15] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *Proc. of the 13th Intl. Symp. on High-Performance Computer Architecture*, pages 285–294, Feb. 2007.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of the Tenth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[17] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proc. of the 2002 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 66–76, 2002.

[18] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proc. of the 38th Intl. Symp. on Microarchitecture*, pages 271–282, 2005.

[19] Z. Zhu and Z. Zhang. A performance comparison of dram memory system optimizations for SMT processors. In *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture*, pages 213–224, 2005.

[20] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. In *Proc. of the Eighth Intl. Symp. on High-Performance Computer Architecture*, pages 107–116, Feb. 2002.