High Performance Memory Requests Scheduling Technique for Multicore Processors

Walid El-Reedy
Electronics and Comm. Engineering
Cairo University, Cairo, Egypt
walid.elreedy@gmail.com

Ali A. El-Moursy
Electrical and Computer Engineering
University of Sharjah, Sharjah, UAE
aelmorsi@sharjah.ac.ae

Hossam A. H. Fahmy
Electronics and Comm. Engineering
Cairo University, Cairo, Egypt
hfahmy@alumni.stanford.edu

***Abstract*—In modern computer systems, long memory latency is one of the main bottlenecks micro-architects are facing for leveraging the system performance especially for memory-intensive applications. This emphasises the importance of the memory access scheduling to efficiently utilize memory bandwidth. Moreover, in recent micro-processors, multithread and multicore is turned to be the default choice for their design. This resulted in more contention on memory. Hence, the effect of memory access scheduling schemes is more critical to the overall performance boost. Although memory access scheduling techniques have been recently proposed for performance improvement, most of them have overlooked the fairness among the running applications. Achieving both high-throughput and fairness simultaneously is challenging.**

In this paper, we focus on the basic idea of memory requests scheduling, which includes how to assign priorities to threads, what request should be served first, and how to achieve fairness among the running applications for multi-core microprocessors. We propose two new memory access scheduling techniques FLRMR, and FIQMR. Compared to recently proposed techniques, on average, FLRMR achieves 8.64% speedup relative to LREQ algorithm, and FIQMR achieves 11.34% speedup relative to IQ-based algorithm. FLRMR outperforms the best of the other techniques by 8.1% in 8-cores workloads. Moreover, FLRMR improves fairness over LREQ by 77.2% on average.

***Keywords*-Computer architecture; Memory management; Multicore processing;**

I. Introduction

Memory access scheduling has been developed for superscalar, multithreaded, and multicore processors to enhance their performance. The concept of memory access scheduling is proposed for superscalar processors in [1]–[5]. However, in superscalar processors memory access scheduling was just re-ordering memory requests making use of memory hardware features to reduce memory access time. The main reason behind this that although main memory is a RAM (Randome Access Memory) device which means that its access time to any memory location should be almost the same, its access pattern is not random. In other words, due to the physical implementation of RAM, it is faster to send one row address and read multiple columns than to send random requests. Memory requests are correlated. By recognizing those relations, we may achieve better scheduling. The enhanced ordering of memory requests may efficiently utilize the memory bandwidth and accordingly increase the throughput. In recent processors multicore and multithreaded architectures are widely used. This resulted in increasing the number of threads that execute in parallel. All these threads are competing for shared resources. One of these most important resources is system main memory. While execution, each thread sends some requests to the main memory asking for missing memory blocks. This created the need of memory access schedulers. The role of a memory access scheduler is to decide which requests should be served first and which threads should have higher priority. A good scheduler can improve overall throughput and/or fairness by reordering memory requests and threads priorities. Throughout this paper we made our experiments on multicore processors but they are applicable for multi-threaded processors as well. Figure 1 illustrates where the memory access scheduler exist in the architecture used.

In this paper, we focus on the core idea of memory requests scheduling algorithms. We try to find a better answer for the open questions: How to assign priorities to threads? And what are the requests that will improve the performance more if served first? In order to achieve our goal, we decide to make our experiments based on single memory bank and memory controller. We believe that achieving good results in this case, can be extended to make use of the parallelization of memory banks and controllers.

The rest of the paper is organized as follows: Section II contains the literature survey where we discuss the algorithms close to our work. Section III contains the proposed memory access scheduling algorithms. Section IV contains the simulation environment, and machine configuration. The results are shown in section V. Section VI contains the related work. Finally, we conclude in section VII.

II. Memory Access Scheduling

The concept of memory access scheduling is discussed for SMT processors in [6]. The authors introduced three thread-aware scheduling algorithms. These algorithms are request-based, ROB-based (Reorder Buffer- based), and IQ-based (Issue Queue - based) scheduling algorithms. They compared these algorithms to some algorithms that were developed for single-threaded processors, which are hit-first, read-first, and age-based algorithms.

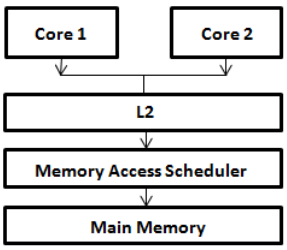

Figure 1. Multicore architecture with memory access scheduler support

Request-based algorithm gives highest priority to requests from thread that has the minimum number of pending requests. In other cases it is named LREQ (Least REQuest). The idea behind this algorithm is that serving a request from the thread that has the minimum pending requests most probably will release more waiting instructions than serving a request from other threads. ROB-based algorithm gives highest priority to requests from thread that has the highest number of reorder buffer entries. The idea behind this algorithm is that serving a request from the thread that has the highest number of reorder buffer entries most probably will release more waiting instructions than serving a request from other threads. Of course, this algorithm will help more in the cases when there is contention on reorder buffer. IQ-based algorithm gives highest priority to requests from thread that has the highest number of issue queue entries. The idea behind this algorithm is that serving a request from the thread that has the highest number of issue queue entries most probably will release more waiting instructions than serving a request from other threads. It will help in the cases when there is contention on issue queue. Hit-first algorithm gives row buffer hits more priority than row buffer misses. So, it gives more priority to requests that take less time. Read-first algorithm gives memory read operations more priority than memory write operations. The idea behind this algorithm is that write operations are not a bottleneck because of the existence of write buffers. Both hit-first, and read-first algorithms are used together in collaboration with other algorithms. For instance, in [6] the authors used hit-first and read-first algorithms combined with request-based algorithm. In this case, read hit will always be scheduled before read miss, and read requests in general will be scheduled before write requests. In addition to this, the same type of requests is scheduled according to number of pending request for each thread. i.e. the thread with the fewest number of pending requests is scheduled first. Age-based algorithm gives highest priority to oldest request when more than eight requests are presented to memory. This algorithm aims to improve fairness but it does not aim to improve throughput. Fairness is guaranteed because no thread will use the memory for large time alone. However, LREQ, ROB-based, and IQ-based, Read-first, and hit-first algorithms aim to improve throughput but have nothing to do with fairness.

In [7] a new scheduling algorithm called ME-LREQ (Memory Efficiency with Least REQuest) is presented. It's based on request-based scheduling algorithm (LREQ) but they added a new parameter to it. This parameter is memory efficiency. Memory efficiency of an application is defined as the IPC (Instructions Per Clock) of this application divided by its memory bandwidth usage under the single-core environment. Memory efficiency can be calculated using offline profiling. The new scheduling algorithm ME-LREQ is claimed to improve performance by 6.4% on average and up to 9.2% over original request-based algorithm. In their results, they used pre-calculated number for each thread as memory efficiency of this thread. So, the main drawback of this algorithm is that offline profiling is not practical.

III. Fair Most-Related Scheduling Algorithms

Taking a look at published memory access scheduling algorithms, we find that there is a space for improvement. We introduce two new memory access scheduling algorithms FLRMR (Fair Least-Request Most Related algorithm), and FIQMR (Fair Issue-Queue based Most Related algorithm). These algorithms are based on request-based algorithm (LREQ) and IQ-based algorithm respectively. Before going into the details of the proposed algorithms, we want to declare some common points between them.

First we want to define what *related requests* are. Throughout our analysis to memory requests distribution, we decide to make use of an important feature of cache. This feature is temporal and spatial locality. *Temporal locality* is referencing the same memory location that has been recently referenced. *Spatial locality* is referencing a neighboring memory location to that has been recently referenced. Hence, the memory block is made to be a number of memory words and not just one to make use of *spatial locality*.

So, if the same word has been referenced again (*temporal locality*), or the neighboring word has been referenced again (*spatial locality*), in both cases the same memory block will be requested again. This means that if there is a pending memory request waiting for being served, it can be requested again while it is waiting. *Related requests* of thread *i* are the total number of memory requests from that thread that demand blocks existing in the memory requests pending from that thread. In other words, if thread *i* requested block x from main memory, and this block had been requested before and had not been served yet, then the number of *related requests* of thread *i* would be incremented by one.

Then we want to define the parameter that determines thread priorities which is the *prioritizing factor*. *Prioritizing factor* (PF) of a thread is a factor calculated by a given formula to help us give priority to this thread. The next request to be served will be picked from the requests of the thread with the highest priority. PF for each algorithm is left

as the original algorithm direction. In the original request-based algorithm, thread priority increases when number of requests decreases. So, we kept this relation with the new PF. In the original IQ-based algorithm, thread priority increases when number of IQ entries increases. So, we keep this relation with the new PF.

One of the major drawbacks of request-based algorithm and IQ based algorithm is that they don't account for the fairness among the running threads. We overcome this drawback in our proposed algorithms by taking starvation time into consideration. We set starvation time threshold to guarantee fairness. When a request age exceeds this starvation time threshold, the request should be scheduled as soon as possible whatever its thread priority is. The value of starvation time threshold has been chosen experimentally.

A. FLRMR algorithm

Request-based scheduling algorithm is an effective way for scheduling, especially in improving throughput [6]–[8]. Our idea is to improve the request-based algorithm to enhance the throughput and to keep the algorithm simple and practical as well.

The first factor that we want to add to request-based algorithm is *related requests*. The other factor that we think it should be included is starvation time threshold. It guarantees fairness between cores which is a drawback in request based algorithm currently published.

So, the new algorithm depends on three factors:

1) Number of pending requests (per thread). This is equal to the number of different memory blocks requested by this thread i.e. it does not count *related requests*. The smaller this number is, the higher priority this thread should have. The original request-based algorithm (LREQ) depends only on this factor.
2) Number of *related requests* (per thread). The larger this number is, the higher priority this thread should have.
3) Request age (per request). It guarantees fairness. When a request stays waiting for more than starvation time threshold, it should be scheduled as soon as possible whatever its thread priority is.

We think that combining these factors together will give better results than any of them alone. Total number of pending requests per thread is a good factor but it accounts all the requests equally which is not true. Number of related requests is, also, a good factor but it counts all the threads the same which is not true. Some threads have less requests than others, so the pending requests of these threads are more likely to free more instructions from the instructions waiting list of this thread than the pending requests of the threads with higher number of requests can do. Moreover, block age and starvation time threshold guarantees fairness but will not tackle throughput improvement issue. So, we think combining these factors will give better results than each one alone.

The question now is: how to combine all these factors to end up with the new algorithm? We believe that the number of pending requests should have high weight because it is the base of the algorithm and should have higher impact on throughput than related requests. Hence, we propose the following *prioritizing factor*. Threads will be prioritized according to the following factor for thread *i*:

$$FLRMR\_PF_i = \frac{\#\ pending\_requests_i^2}{(\#\ related\_requests_i + 1)} \quad (1)$$

where i is the thread ID, $FLRMR\_PF_i$ is the *prioritizing factor* of thread i according to FLRMR algorithm, $\#pending\_requests_i$ is the total number of pending requests from thread *i*, and $\#related\_requests_i$ is the total number of related requests from thread *i*.

The smaller this factor is, the higher priority this thread will have. Number of pending requests is squared to have a higher weight than *related requests*. The one added to the number of *related requests* represents the original request (i.e. the first request that asks for the block that related requests are waiting for). If a block is starving (i.e. starvation time threshold has passed since its arrival), this block will be served, and removed from the list. If no requests are starving, the thread with the smallest PF will be in the top of the list. The oldest request of this thread will be served first, and removed from the list.

B. FIQMR algorithm

One of the objectives of proposing this algorithm is to prove that the idea of related instructions can be used combined with many algorithms and can improve the overall performance of these algorithms. Similar to FLRMR algorithm in subsection III-A, FIQMR uses related instructions and block starvation time combined with IQ-based algorithm. In other words, FIQMR algorithm depends on three factors:

1) Number of issue queue entries (per thread). The rational behind this metric is that number of entries in the queue indicates the high dependability on the cache misses. So, to allow the application to make progress in its execution, it should be given priority for scheduling. The larger this number is, the higher priority this thread should have. The original IQ-based algorithm depends only on this factor. i.e. In the original IQ-based algorithm, the thread with the largest number of entries in the issue queue will be scheduled first.
2) Number of *related requests* (per thread). The larger this number is, the higher priority this thread should have.
3) Request age (per request). It guarantees fairness. When a request stays waiting for more than starvation time

threshold, it should be scheduled as soon as possible whatever its thread priority is.

Similar to FLRMR, also, we think that combining these factors together will give better results than any of them alone. We believe that the number of IQ entries should have high weight because it is the core of the algorithm. So, it is more important and should have higher impact on throughput than related requests. Hence, we propose the following *prioritizing factor*. Threads will be prioritized according to the following factor for thread *i*:

$$FIQMR\_PF_i = \# \ IQ\_entries_i^2 * (\# \ related\_requests_i + 1) \quad (2)$$

where *i* is the thread ID, $FIQMR\_PF_i$ is the *prioritizing factor* of thread *i* according to FIQMR algorithm, $\# \ IQ\_entries_i$ is the number of IQ-entries occupied by thread *i*, and $\# \ related\_requests_i$ is the total number of related requests from thread *i*.

The larger this factor is, the higher priority this thread will have. Similar to FLRMR, number of IQ entries occupied by the thread is given higher weight than the number of *related requests*.

C. Hardware price

Do these new algorithms require extra complex hardware? The answer is: no, the extra needed hardware is very small. One of the main advantages of these new algorithms is that they make use of already existing hardware. We make use of already existing MSHR (Miss Status Holding Register) [9], [10]. MSHR tracks information about all the in-progress misses. Each MSHR entry has a comparator, target information whose contents differ according to implementation, and a valid bit. If all MSHR entries are valid, the cache should be blocked because there are no more entries to track miss information. So, all what we need is to make sure that number of related requests is stored in the target information (which is implementation-dependent).

We need to add a new register in the context of each thread. This register contains the total number of related requests from this thread. When a memory block is requested, if this block exists in the MSHR, the block counter (in MSHR) and the thread counter (in thread context) should be incremented. When this block is served, the block counter should be decremented from the thread context counter.

D. Example of related requests frequency

The first thing may come up to mind when this algorithm is mentioned is: how much the related requests may affect the overall performance? Does it really worth making new algorithms including this factor? To illustrate how related requests can change the scheduling order, and how frequent they can be, we will mention here a detailed case from real benchmarks where FLRMR algorithm gives different results than LREQ algorithm, and related requests number of one thread reaches 23 asking for only 2 memory blocks (i.e. 13 memory requests asking for one memory block, and 10 memory requests asking for another memory block). In this example, each core runs a single thread. The workloads running on cores 0 to 3 are gcc, galgel, vpr, and gzip respectively. All these benchmarks belong to SPEC2000 benchmarks suite.

Table I
EXAMPLE ILLUSTRATING *related requests* FREQUENCY

CoreID	Pending requests	Related requests	Arrival time
1	2	23	77666
3	1	4	77732
0	1	0	77688
2	2	2	77465

Table I, contains the details of pending requests from all running threads ordered according to prioritizing factor formula in equation 1. First column contains core ID. To know the benchmark running on this core, please refer to table **??**. Second column contains the number of requests pending from this core. Third column contains the total number of requests related to pending requests from this core. Fourth column contains arrival time of the oldest request from this core. It is helpful to know if any of these requests crossed the starvation time threshold.

At time 77789, thread with CoreID 1 has only 2 pending memory requests (each request is asking for a different block in memory), but there are 23 instructions requesting these 2 memory blocks. Similarly, thread with CoreID 3 is requesting one memory block but there are four instruction pending on this block. From this example we should know how the case of *related requests* is not rare.

IV. SIMULATION METHODOLOGY

A. Simulation environment and Machine configuration

We have used the simulator used in [11]. It is a multi-core version of Simplescalar-3.0 [12] for the Alpha AXP instruction set. We have modified the memory access process in this simulator. Simplescalar used to let instructions access the memory as if there is no other requests. So, we have changed this simplified case to make it more practical. We have implemented FCFS (which is our baseline policy), RR, LREQ, IQ-based, FLRMR, and FIQMR. All these algorithms are implemented to schedule read requests only. Write requests are not a bottleneck because of the existence of write buffers, so we focus on read requests only.

Table II shows the major simulation parameters used.

B. Benchmarks and Workloads

In our simulation, we have used single-threaded cores. Each core run a separate application. We have used the classification of SPEC CPU2000 benchmarks from [7]. They

Table II
MAJOR SIMULATION PARAMETERS

General parameters	
Parameters	**Values**
Processor	2/4/8 cores
Branch predictor	Bimodal and 2-level comb
Bimodal predictor entries	2048
Level 1 table entries	1024
Level 2 table entries	4096
BTB entries, associativity	2048, 2-way
Branch mispredict penalty	10 cycles
L2 cache	4MB, 4-way, 64B line
L2 cache latency	15 cycles
Main memory latency	100 cycles
Per core parameters	
L1 ICache	64KB, 2-way, 64B line
L1 ICache latency	1 cycle
L1 DCache	64KB, 2-way, 64B line
L1 DCache latency	3 cycles
Int. Functional units	2
FP Functional units	1

Table III
BENCHMARKS CLASSIFICATION

Class	**Benchmark**
MEM	wupwise, swim, mgrid, applu, vpr, gcc, galgel, art, mcf, equake, lucas, gap
ILP	gzip, mesa, crafty, parser, eon, bzip2, twolf, apsi

are classified into two classes; memory-intensive benchmarks (MEM), and compute-intensive benchmarks (ILP). The memory-intensive applications are considered memory-intensive because they can gain more than 15% performance when they run within perfect memory system (Zero latency and infinite bandwidth). We did not use HPC applications as we believe that multicore processors are no more limited to HPC.

Table III shows the benchmarks used in our simulation and their classification. We have tried to make different combinations of workloads for 2,4, and 8 cores. We gave each workload a name to be used in performance evaluation, and analysis. Workload names and benchmarks included in each workload are shown in table IV. Workload name consists of three parts. First part is the number of cores used to run this workload which is equal to the number of benchmarks included because we run one benchmark per core. Second part is either mem (all benchmarks included in this workload are memory-intensive), or mix (some benchmarks in thus workload are memory-intensive and others are not). The third part is the workload ID number within its category. For example workload 8mix2 is the second workload in workloads that contain eight benchmarks (four memory-intensive, and four compute-intensive).

C. Starvation time threshold

According to our experiments, we found that setting starvation time threshold to (2 * no. of threads * memory latency) is fair enough and gives good throughput results.

Table IV
WORKLOADS DESCRIPTION

Workload	**Benchmarks Included**
2mem1	mcf, lucas
2mem2	mgrid, vpr
2mix1	galgel, gzip
2mix2	gcc, parser
4mem1	mcf, equake, wupwise, lucas
4mem2	swim, gap, art, vpr
4mix1	gcc, galgel, gzip, parser
4mix2	gzip, apsi, mgrid, applu
8mem1	vpr, gcc, galgel, art, mcf, equake, lucas, gap
8mem2	wupwise, swim, mgrid, applu, vpr, gcc, galgel, art
8mix1	gzip, mesa, crafty, parser, mcf, equake, lucas, gap
8mix2	gcc, galgel, parser, mesa, apsi, mgrid, applu, gzip

D. Performance metric

For throughput comparison, we have used geometric mean. It is given by the following formula for *n* threads:

$$\sqrt[n]{\prod_{n} \frac{IPC_{new}}{IPC_{old}}}$$

The advantage of geometric mean is that it is a relative measure [11]. Hence, the issue of which configuration is used in getting IPC_{new} or IPC_{old} is not valid. Another advantage of geometric mean is that if there is a performance improvement in one thread and an identical performance degradation in another running thread, both changes in performance will be affecting the overall performance equally.

V. RESULTS

We compared the results of five different algorithms: FCFS, RR, LREQ, IQ-based, and our newly proposed algorithms FLRMR, and FIQMR. The baseline algorithm is FCFS. In the next section we will show how the different algorithms will behave in different examples.

A. Scheduling Scenario

Here we will show how different algorithms will behave with requests in table I.Normally, PF calculations can be done when memory serve the last request and get ready for serving a new one but we reorder the table entries each time a new/related request arrives according to 1 to illustrate the frequency of *related requests* and their effect on performance.

At time 77821 as shown in table V, memory served last request. So, the first block in the list (according to FLRMR algorithm) is being served now, and removed from the list. We want to focus on the requests at time 77820. At that time we are ready to pick one request from all of the requests listed. We will show how FCFS, RR, LREQ, and FLRMR will pick the next request to be served.

FCFS will pick the oldest request which will be the oldest request of core 2. RR algorithm will not take any of these

Table V
EXAMPLE 1 ILLUSTRATING FLRMR ALGORITHM

CoreID	Pending requests	Related requests	Arrival time
At time 77820: Requests waiting for scheduling			
1	2	23	77666
3	1	4	77732
0	1	0	77688
2	2	2	77465
At time 77821: Memory served last request, and another request is picked from the waiting list			
1	1	10	77725
3	1	4	77732
0	1	0	77688
2	2	2	77465

Table VI
EXAMPLE 2 ILLUSTRATING FLRMR ALGORITHM

CoreID	Pending requests	Related requests	Arrival time
At time 78464: Starting monitoring, a request is being served until = 78563			
1	1	8	78398
2	2	2	77465
0	2	2	77688
At time 78465: Starvation time threshold for oldest block in core 2 is reached			
2	2	2	77465
1	1	8	78398
0	2	2	77688

factors into consideration. It will just pick a request from the core whose CoreID equals to

$$(CoreID\ of\ last\ served\ request+1)\%Total\ no.\ of\ cores$$

LREQ will pick the oldest request of the core with the least number of requests. So, in this case it will serve a request from core 0. FLRMR algorithm will look at the pending requests. If none of these requests exceeds the starvation time threshold, then we should calculate the *prioritizing factor* for each core. It will be 0.1667, 0.2, 1, and 1.333 for cores 1, 3, 0, and 2 respectively. The core with the smallest PF is core 1. So, the oldest request of this core will be served. In this case there are 13 related requests will be served when this request is served.

Table VI shows another example illustrating how the starvation time threshold is combined with the prioritizing factor in this algorithm. Simply, when the request reaches its starvation time threshold, it will come to the top of the list regardless the number of pending requests and the number of *related requests*. In this example, the starvation time threshold is 1000 cycles. The arrival time of the oldest request in core 2 is 77465 and it has reached the cycle 78465. In this cycle, this request is scheduled to be the next served one. Hence, starvation time threshold improves fairness and ensures that no thread is stopped for a long time waiting for memory requests to be served.

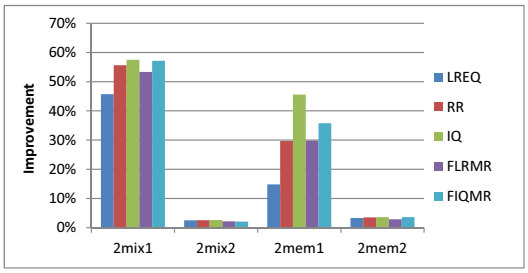

Figure 2. 2-cores workloads results

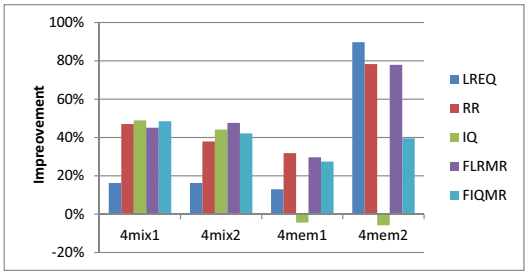

Figure 3. 4-cores workloads results

B. Performance Evaluation

Figures 2, 3, and 4 show the percentage of performance improvement gained by running memory access scheduling algorithms on 2-cores, 4-cores, and 8-cores workloads respectively. The change in performance in running workloads 2mix2, and 2mem2 is small, and the algorithms results are changed from one workload to another because there is small space for scheduling when we schedule requests from 2 cores only. Hence, it is hard to have a consistent performance of an algorithm over all the workloads, and the expected performance improvement gained by applying any algorithm is less than the improvement when scheduling requests from larger number of cores. The figures prove the famous idea about scheduling algorithms that no algorithm is perfect, and no algorithm can give the best results with all workloads. This is why we are more concerned with the results average. Figure 5 shows the average percentage of performance improvement gained by running memory

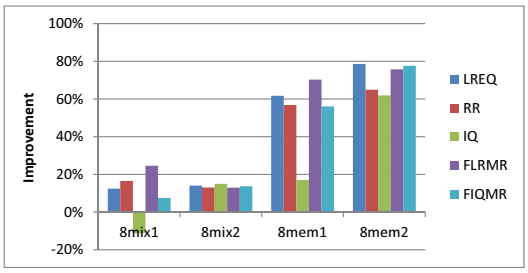

Figure 4. 8-cores workloads results

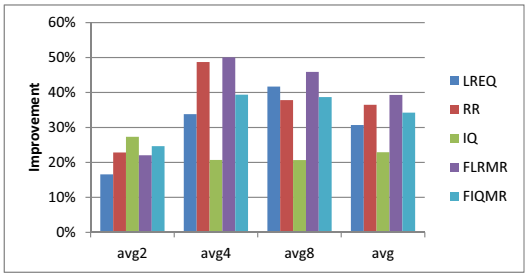

Figure 5. Average results of all workloads

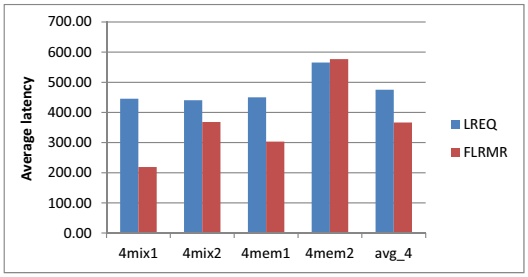

Figure 6. Average latency time (in clock cycles)

access scheduling algorithms on all workloads. Average results of executing workloads show that FLRMR improves the performance over LREQ by 5.45% in case of 2 cores, 16.25% in case of 4 cores, and 4.2% in case of 8 cores. FIQMR degrades the performance from IQ-based algorithm by 2.66% in case of 2 cores. However, it improves the performance over IQ-based algorithm by 18.7% in case of 4 cores, and 18.02% in case of 8 cores.

The total average results show that FLRMR outperforms all the other algorithms. RR comes next with a difference of about 3% on average. The differences in performance improvement between FLRMR and RR are as follows: -0.8% in 2 cores, 1.3% in 4 cores, and 8.1% in 8 cores. This means that the difference increases when number of cores is increased. In other words, in small number of cores RR can give good results but when the number of cores is increased RR can't give such good results. This makes more sense because in 8 cores, for example, each core will access the memory every 8 requests whatever the criticality of the application running or of the waiting requests. FIQMR comes next, then LREQ algorithm. FLRMR improves the performance of LREQ algorithm by 8.64% on average. FIQMR improves the performance of IQ-based algorithm by 11.34% on average. This proves that the idea of *related requests* and fair scheduling which is implemented in algorithms FLRMR, and FIQMR improves the existing algorithms LREQ, and IQ-based.

The results of LREQ algorithm is getting better when the number of cores is increased. This does not mean that LREQ will give best results when the number of cores become 16 or 32. This is because LREQ is not fair, so the performance of LREQ can be decreased (or at least not increased linearly) because some cores can be idle until the cores with the least requests be served. Moreover, FLRMR guarantees fairness which may degrade the overall performance in some cases.

Now, the question is: Why do FLRMR, and FIQMR give good results with some workloads and worse results with others? This is, simply, because some benchmarks have the nature of reusing data. In other words, different applications have different ratios of $temporal$ and $spatial\ locality$. $Temporal$ and $spatial\ locality$ are parametrized in *related requests* number in FLRMR and FIQMR algorithms. When an application has high $temporal$ and $spatial\ locality$, the number of *related requests* is expected to be high. Hence, FLRMR, and FIQMR algorithms will be more accurate.

To know how FLRMR reduces load latency, and how this affects the performance, we calculate the average latency of 4-cores workloads when applying LREQ and FLRMR algorithms as shown in figure 6. The figure shows that when applications are scheduled using FLRMR, they have smaller load latency on average for almost all the workloads used. Only on 4mem2 workload, LREQ gives less load latency average time. This is why LREQ gives better performance than FLRMR in the results of this workload as shown in 3. FLRMR reduces the average load latency from 475.4 cycles when scheduling with LREQ to 366.8 cycles.

C. Fairness

To prove that our proposed algorithms are more fair, we calculate *unfairness* to know how much it is changed by our algorithms. In these calculations, we followed [7], [14]. *Unfairness* for a certain workload is defined as the ratio between the maximum slowdown to the minimum slowdown among all the applications running in this workload. *Slowdown* is defined as the ratio between the application stall time because of loads when it is running alone to its stall time when it is running among other applications in the workload. Figure 7 shows calculated *unfairness* for all 4-cores workloads as an example. We compared between FLRMR and its base algorithm LREQ to emphasize that our proposal improves fairness. The figure shows that FLRMR is much more fair than LREQ for all workloads. It reduces the *unfairness* in LREQ to 22.8% of its value on average. This means that FLRMR improves fairness over LREQ by 77.2% on average.

VI. Related work

There are some other published memory access scheduling algorithms. In [8] they proposed TCM (Thread Cluster Memory) algorithm which gives good results only with large number of cores (24 cores). Another algorithm proposed in [15] is called ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling). It gives the highest priority to the thread with the least attained service from all memory

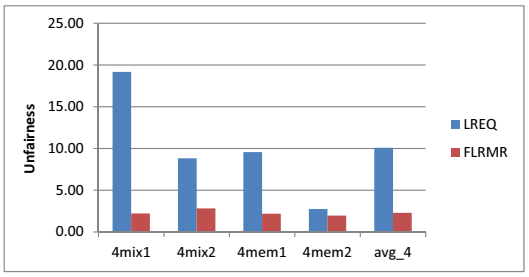

Figure 7. Unfairness in 4-cores workloads

controllers. ATLAS gives good results only with large number of memory controllers. In [16] a new scheduling algorithm called PAR-BS (Parallelism-aware Batch Scheduling) is presented. This algorithm groups memory requests into batches Then it tries to make use of bank-level parallelism within the same batch to get better performance. In [17] they proposed a new memory access scheduling algorithm designed specifically for parallel applications.

VII. Conclusion

We propose two new memory access scheduling algorithms. These algorithms are FLRMR, and FIQMR. They are based on the idea of adding the *related requests* factor to existing algorithms which are LREQ and IQ-based algorithms respectively. We add starvation time threshold combined with these algorithms to guarantee fairness. The average results show that the best results come from FLRMR algorithm. RR comes next with average throughput worse than FLRMR by about 3% on average. FLRMR gives better results than RR in 4-cores and 8 cores workloads. This means that increasing the number of cores can result in increasing the gap between FLRMR and RR. FIQMR, and LREQ algorithms come next after RR, respectively. At the end of the list, IQ-based algorithm gives worst results.

In general, FLRMR improved the performance of LREQ algorithm by 8.64%. FIQMR improved the performance of IQ-based algorithm by 11.34%. This proves that the idea of *related requests* and fair scheduling give good results, and it deserves more care in future research in this point.

VIII. Future Work

In future work we intend to include the effect of adding different memory banks, and memory controllers. We are interested, also, in adapting these algorithms to support parallel applications. We plan, also, to test these algorithms on real platforms.

References

[1] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA-27*, pp. 128–138, 2000.

[2] S. A. McKee and W. A.Wulf, "Access ordering and memory-conscious cache utilization," in *HPCA-1*, pp. 253–262, Jan. 1995.

[3] 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 *HPCA-5*, pp. 80–89, Jan. 1999.

[4] S. A. Moyar, "Access ordering and effective memory bandwidth," *Technical Report TR CS-93-18*, April 1993.

[5] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "Design of a parallel vector access unit for sdram memory systems," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pp. 39–48, Jan. 2000.

[6] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for smt processors," in *ISCA-11*, pp. 213–224, 2005.

[7] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu, "Memory access scheduling schemes for systems with multi-core processors," *37th International Conference on Parallel Processing*, 2008.

[8] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," *MICRO-43*, 2010.

[9] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981.

[10] K. I. Farkas and N. P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads," in *ISCA-21*, April 1994.

[11] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Partitioning multi-threaded processors with a large number of threads," in *Intl. Symposium on Performance Analysis of Systems and Software*, pp. 112–123, March 2005.

[12] D. Burger and T. Austin, "The simplescalar toolset, version 2.0.," *Technical Report TR-97-1342*, June 1997.

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

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

[15] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," *HPCA-16*, 2010.

[16] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," *ISCA-35*, 2008.

[17] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," *MICRO-44*, 2011.