High Performance Memory Requests Scheduling Technique for Multicore Processors

by

Walid Ahmed Mohamed El-Reedy

A Thesis Submitted to the

Faculty of Engineering at Cairo University

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Supervised by:

Prof. Dr. Amin Nassar

Dr. Hossam A. H. Fahmy

Dr. Ali El-Moursy

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

August 20, 2012

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 thesis, 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 the Fair Least-Request Most Related (FLRMR) algorithm, and the Fair Issue-Queue based Most Related (FIQMR) algorithm. These proposals are based on the Least Request (LREQ) algorithm and the Issue Queue based (IQ-based) algorithm respectively. Our simulations are done on 2, 4, and 8 cores using a multicore version of Simplescalar and we run 4 workloads for each number of cores. According to this simulation environment, the results show that FLRMR improves fairness over LREQ by 54.6% on average, and that FIQMR improves fairness over IQ-based algorithm by 80.6% on average. Moreover, 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.

Acknowledgments

First of all, I would like to thank Allah for his guidance and help. Then I would like to thank my wife, and my family for their continuous support.

I would like to thank Dr. Ali El-Moursy for his guidance, support, and fast response.

Also, I would like to thank Dr. Hossam A. H. Fahmy, and Prof. Amin Nassar for their useful comments.

Contents

List of Figures

List of Abbreviations

AH-PH	Actual Hit - Predicted Hit
AH-PM	Actual Hit - Predicted Miss
AM-PH	Actual Miss - Predicted Hit
AM-PM	Actual Miss - Predicted Miss
FCFS	First Come First Served
FR-FCFS	First Read, First Come First Served
ILP	Instruction Level Parallelism
IMPS	Integrated Memory Partitioning and Scheduling
IQ-based	Issue Queue based
LREQ	Least Request
MCP	Memory Channel Partitioning
ME	Memory Efficiency
ME-LREQ	Memory Efficiency with Least Request
MSHR	Miss Status Holding Register
PC	Program Counter
RAM	Random Access Memory
RIR	Ready to In-flight Ratio

ROB-based	Reorder Buffer based
RR	Round Robin
SMT	Simultaneous MultiThreading
STFM	Stall-Time Fair Memory scheduler

List of Tables

Chapter 1

Introduction

In recent processors multicore and multithreaded architectures are widely used. One reason behind this is the limitations of ILP. These limitations include the control dependences, data dependences and limits of caches [1]. Another reason behind the popularity of parallelized processors in general is power consumption. Processor manufacturers used to increase processors frequency to improve the overall throughput. However, increasing processor frequency resulted in huge power consumption, which needs special cooling systems. All of these reasons yield to moving to multithreading and multicore processors.

This move 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 requests to the main memory to serve the cache misses. This introduces the need of memory access schedulers to decide which requests should be served first. This can improve either throughput or fairness or both.

We aim to improve both fairness and throughput by proposing two memory requests scheduling algorithms.

Throughout this thesis we made our experiments on multicore processors but they are applicable for multithreaded processors as well.

Next sections contain an introduction to multicore processors, the architecture used in this thesis, memory access scheduling algorithms, MSHR (Miss Status

Holding Register) as it is used in our proposed algorithms, fairness and its importance, load hit/miss prediction which is an idea that was under investigation by the authors, and thesis layout.

1.1 Multi-threading Processors

There is a key to look at different architectures and know the differences between each one of them. This key is to know what is shared between threads and what is separate[2]. The main microarchitecture components are:

- The front-end: It contains L1 instruction cache banks, fetch, rename, and dispatch queues.
- The execution engine: It contains the issue queue, register file, and the functional units.
- L1 data cache: It contains load/store queues, and the L1 data cache banks.

Multithreading allows different threads to use the functional units of a single processor [3]. Each thread should have a separate PC, a separate register file, and a separate page table. Memory can be shared or separate. There are different types of multithreading:

- Fine-grained multithreading: In this type, the switch between threads is on every instruction. In most cases, this switch is done in round robin order between threads that are not stalled. This requires hardware that supports fast switch between threads. However, the advantage of fine-grained multithreading is that it resolves small and long stalls.
- Course-grained multithreading: In this type, the switch between threads is done only on long stalls like L2 cache miss. The advantage of this type is that it is more practical and needs less hardware than fine-grained multithreading. However, the disadvantage of coearse-grained multithreading is that it resolves only long stalls.

- Simultaneous Multithreading (SMT): It was first researched by IBM in 1968. The first major commercial microprocessor developed with SMT was the Alpha 21464[4]. It was inspired by Dean Tullsen's research [5]. In this type, the instructions are fetched simultaneously from different threads.

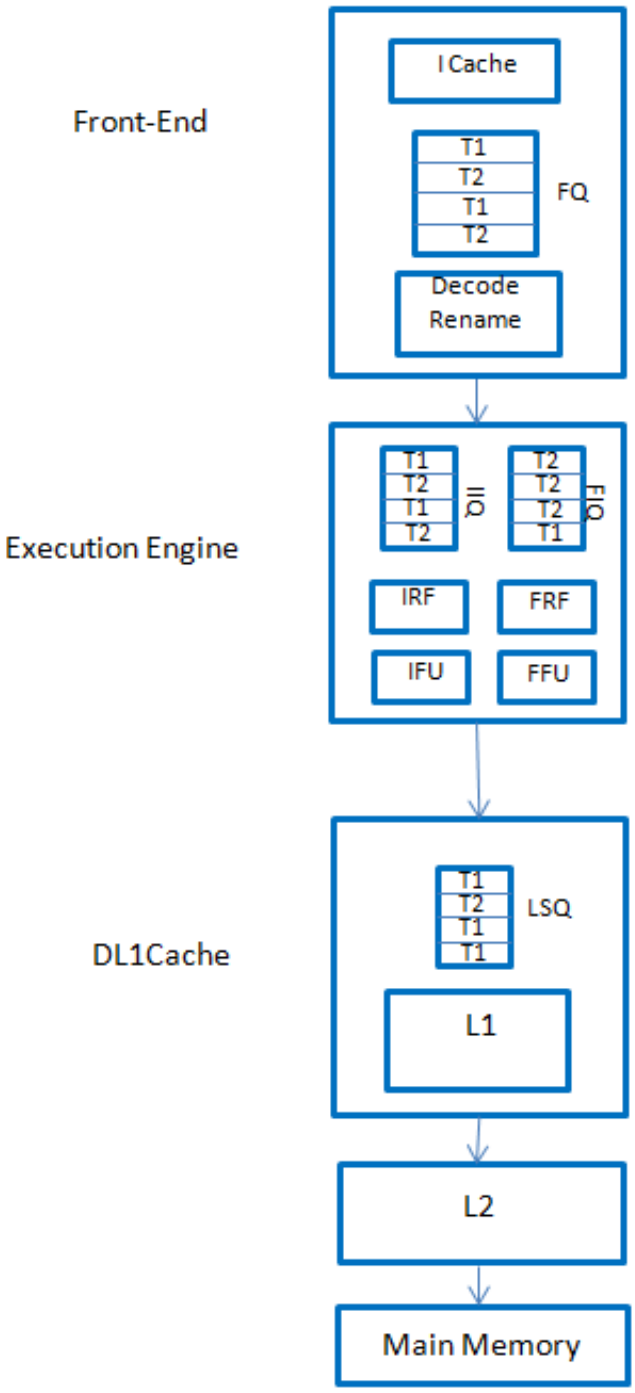


Figure 1.1: SMT architecture[2]

The motivation beyond SMT is that modern processors, that can issue mul-

tiple instructions at a time, usually have more functional units than that can be used efficiently by one thread. SMT has some design challenges. These challenges include: dealing with larger register files to accomplish context switching, making sure that conflicts in cache and TLB because of simultaneous threads execution does not result in large degradation in performance, and choosing instructions to be issued or completed from other instructions and its effect on performance. In normal SMT architecture as proposed in [5], there is no partitioning in the three main microarchitecture components mentioned above: the front-end, the execution engine, and L1 data cache. This architecture is shown in figure 1.1.

1.2 Multi-core Processors

To understand how multicore processors came to reality and what their history is, we will begin from multiprocessors in general. Multiprocessors existed many years ago. To understand the different types of multiprocessors, it is useful to mention what Flynn said in [6]. He categorized any computer into four categories:

- Single Instruction stream, Single Data stream (SISD): Single instruction is executed on single data. This is the case of uniprocessor.
- Single Instruction stream, Multiple Data stream (SIMD): The same instruction is executed on different processors to process the different data. This category is useful when dealing with arrays which is the case in video and game processing.
- Multiple Instruction stream, Single Data stream (MISD): Multiple instructions are executed on different processors to process the same data. No commercial processors are manufactured from this category[3].
- Multiple Instruction stream, Multiple Data stream (MIMD): Multiple instructions are executed on different processors to process different data.

This means that each processor can run a different thread. This is category which multicore processors belong to.

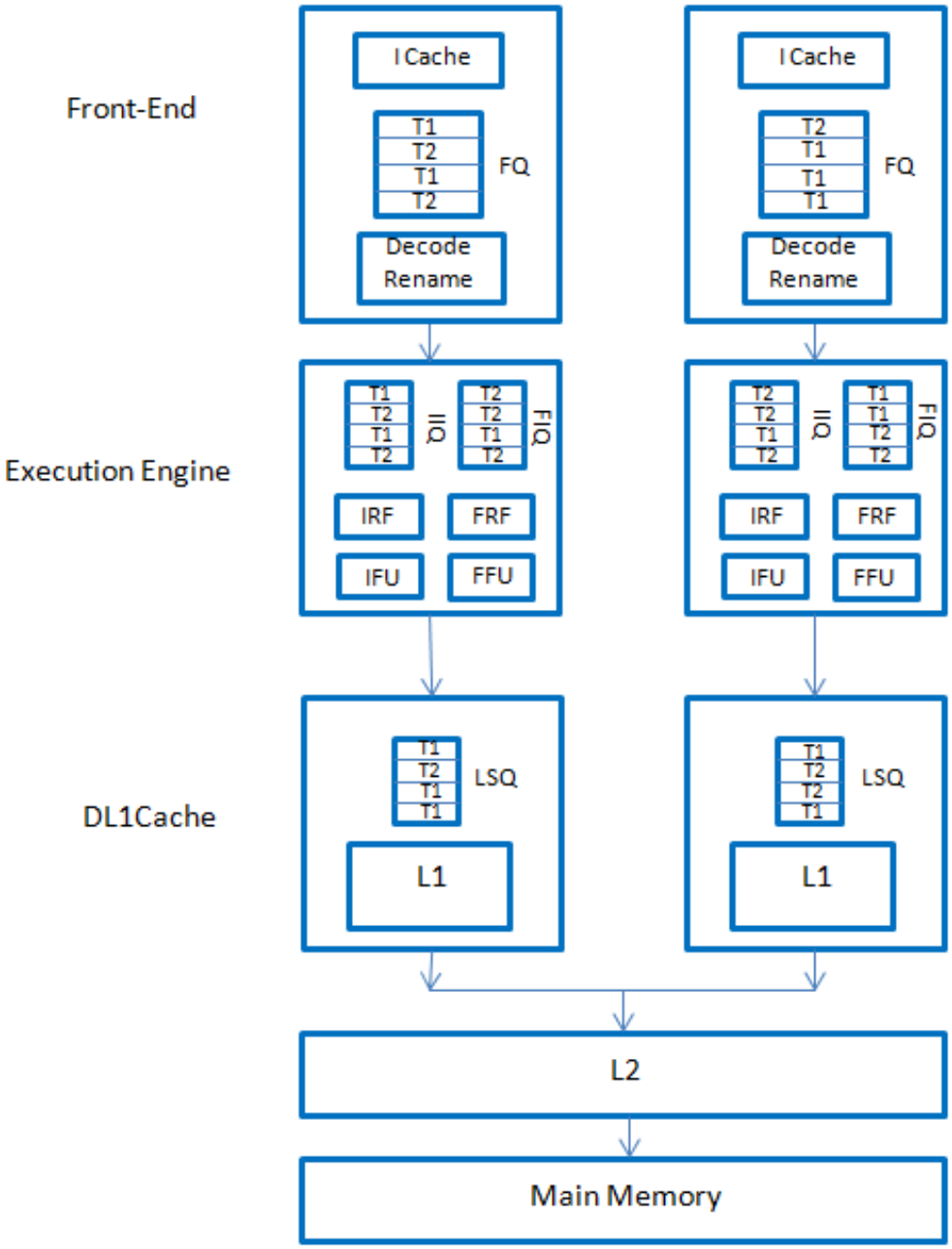


Figure 1.2: Traditional Multicore architecture[2]

When single chip capacity increased to contain different processors, this type of processors was called chip multiprocessors (CMP), or single-chip multiprocessors. However, recently the most common name of this category is multicore processors.

In multicore architecture we used here, all the microarchitecture components are equally partitioned between cores. Figure 1.2 illustrates the multicore architecture used [2].

1.3 Memory Access Scheduling

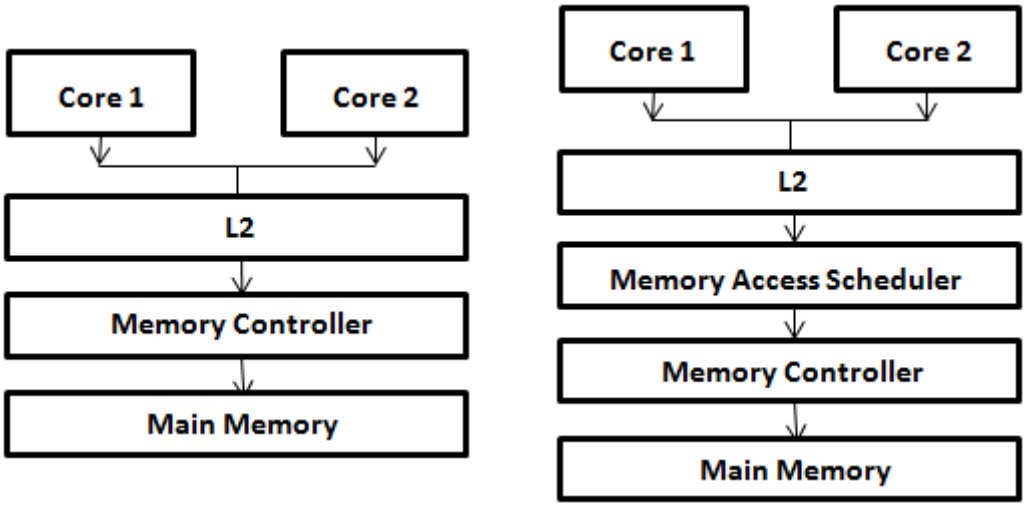
Memory access scheduling has been developed for superscalar, multithreaded, and multicore processors to enhance their performance. The concept of memory access scheduling is proposed for superscalar processors in [7, 8, 9, 10, 11]. However, in superscalar processors memory access scheduling was just reordering memory requests making use of memory hardware features to reduce memory access time. The main reason behind this is that although main memory is a RAM (Random 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.

Accordingly, memory requests are correlated. By recognizing these relations, we may achieve better scheduling. The enhanced ordering of memory requests may efficiently utilize the memory bandwidth and accordingly increase the throughput.

When the number of threads per processor is increased in recent processors, all these threads compete for shared resources. One of these most important resources is system main memory. This introduces the need of memory access schedulers. The role of a memory access scheduler, in multithreaded and multicore processors, 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.

Figure 1.3 illustrates where the memory access scheduler exist in the multicore architecture used.

In this thesis, 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



(a) Multicore architecture without scheduler (b) Multicore architecture with scheduler

Figure 1.3: Multicore architecture with and without memory access scheduler

priorities to threads? And what are the requests that will improve the performance more if served first? How to achieve fairness?

There are different answers to these questions through different scheduling algorithms with different types. The details of these algorithms are in chapter 2.

1.4 Miss Status Holding Register (MSHR)

MSHR tracks information about all the in-progress misses[12]. In systems that support non-blocking loads (in which the thread can continue running while it faced one or more load misses), there is a need for MSHR. After a load miss occur, most probably there will be more instructions that address the same line. Instead of generating new misses, MSHR handles these requests. When the memory line becomes available, MSHR responds to all the loads pending on this memory line.

Figure 1.4 shows the structure of MSHR file. 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.

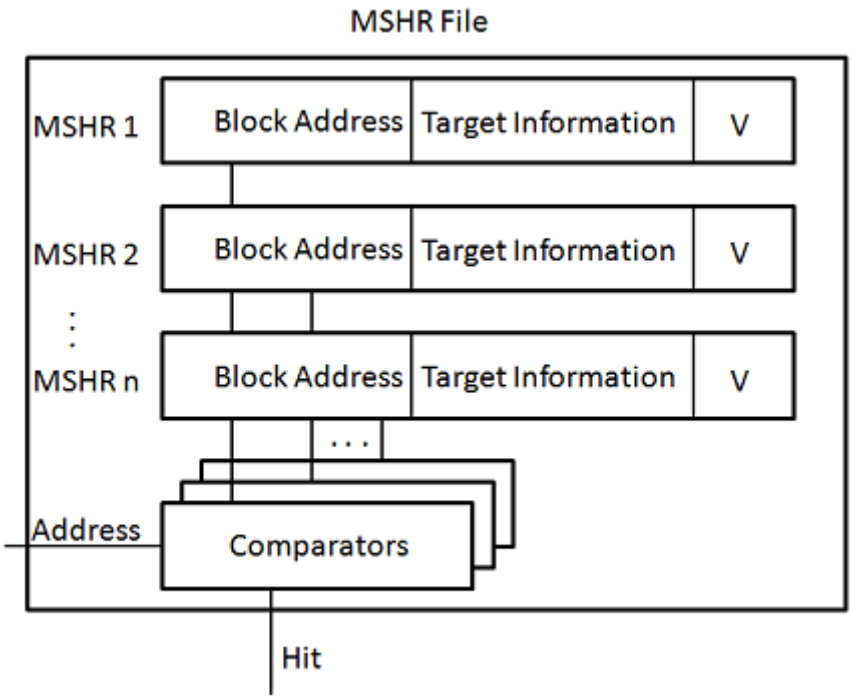


Figure 1.4: MSHR file diagram[12]

1.5 Fairness

Is achieving fairness between threads important? If yes, why is it important? To know the answer of these questions, imagine yourself sitting on your computer, waiting for simulation results to be finished and in the meanwhile writing your thesis. Will you accept the the processor resources goes to your simulations intensively so that you can not continue writing your thesis? Will you accept writing your thesis and waiting for your simulation results for a long time? Of course, both cases are not acceptable. So, fairness between threads is very important.

Fairness does not have a single meaning. We can think about fairness as fairness on memory requests level, or fairness on threads level. Fairness on requests level means to not let a request to starve for a long time waiting to be scheduled. Fairness on threads level means that no thread exploits memory resources and lets other threads waiting for a long time. Achieving fairness on requests level can lead to achieving it on thread level and vice versa.

1.6 Load Hit/Miss Prediction

We have tried to tackle one of the design challenges in SMT which is knowing in advance the latency of long-latency instructions. This can help the scheduler in scheduling which thread should be stopped, and which should continue running. This can help, also, in prefetching the blocks needed in advance to make use of the time between prediction and cache access. This could be done by cache load hit/miss prediction. We have done some work in this point including literature survey, implementation of perfect cache load miss predictor for both superscalar and SMT processors, and running some workloads. Then, we decided to move on to another point which is memory access scheduling in multicore and multithreaded architectures. However, we would like to share our work in cache load hit/miss prediction and the conclusion reached here in this section. Next subsections describes the work done in load miss prediction point.

1.6.1 Introduction to Load Hit/Miss Prediction

In SMT, all threads share all hardware resources. SMT is different than superscalar processor that it can fetch instructions from more than one thread at a time. It, also, can issue instructions from more than one thread at a time. This results in better resource utilization and better instruction throughput [13].

An important issue in SMT is the long-latency instructions, such as missing loads. The dispatcher doesn't know in advance that this instruction will miss, so, it suddenly stops waiting for this miss to be resolved. Meanwhile, the pipeline is occupied by instructions from this thread depending on that long-latency instruction.

One solution for this issue is to predict the cache load misses to know in advance which instructions will have long latency. This information can be then used in gating the thread having this instruction [14]. It can, also, be used in prefetching the blocks that are predicted to have long latency.

Another solution to this problem is to stall the thread once it is known that it is a miss[15].

One more aggressive solution is to flush the pipeline from all the instructions related to this thread once it is known that it is a miss[15]

1.6.2 Load Hit/Miss Prediction in Literature

Load miss prediction has been proposed in [16] by using cache profiling and then using cache profiling correlation in [17].

In [13] the idea of the predictor was based on using predictors similar to branch predictors. They introduced a modified version of a branch predictor called $localpredictor$. They changed storing history of branches as taken or not, to storing the history of loads as hit or miss. They used a table of 2048 entries and a history length of 8 (2KBytes in size). It has a high number of mispredictions where it predicts hit while it actually misses. To overcome this problem, they introduced a hybrid predictor which is using a majority rule between three predictors to take the decision. The total predictor size is 10KBytes

They divided the load categories into:

Correct Predictions:

- AH-PH (Actual Hit - Predicted Hit) Loads: Loads that were predicted to hit, and actually hit. This category contains the majority of loads. It exists in the processors without predictors, as the default that the cache will hit. So, this category has no effect on performance.
- AM-PM (Actual Miss - Predicted Miss) Loads: Loads that were predicted to miss, and actually missed. This category contains the misses that the predictor has detected. They should have a positive effect on performance.

Mispredictions:

- AH-PM (Actual Hit - Predicted Miss) Loads: Loads that were predicted to miss, and actually hit. This category should result in degradation in performance as it did not exist when there was no predictor and it stops the pipeline waiting for the miss to be handled while there is no need to wait.

- AM-PH (Actual Miss - Predicted Hit) Loads: Loads that were predicted to hit, and actually missed. This category contains the misses that the predictor did not detect. It exists in the processors without predictors, and its percentage is equal to the miss ratio.

The cache hit-miss prediction was introduced in SMT processors in [14]. It introduced a simple predictor and a methodology to use that predictor. It predicts L1 misses. It used a table of 2-bit counters. The table has 2K entries indexed by the PC of the load. The counter is reset when a miss occurs and incremented when a hit occurs. The prediction is the most significant bit, 1 means predicted hit, and 0 means predicted miss. The methodology proposed to make use of this predictor is PDG (Predictive Data Miss Gating). This mechanism uses a counter for each thread. This counter is incremented when a load predicted to miss or when it is predicted to hit but actually misses. The counter is decremented when a load is committed or when it is predicted to miss but actually hits. When a thread counter exceeds n, the thread is fetch gated.

In [15], there is a comparison between FLUSH, STALL, and L2MP (a policy to use load hit-miss predictor) policies. Also, it compared STALL+, FLUSH+, and L2MP+ (the improved versions of STALL and FLUSH, L2MP respectively). They, also, proposed FLUSH++ policy which is a combination of FLUSH+ and STALL+. It, also, predicts L2 misses.

In [18], they propose a new memory architecture with a Load Miss Predictor (LMP), which includes a data bypass cache and a predictor table, to reduce access latencies by determining if a load should bypass the main cache hierarchy and issue an early load to main memory. They utilize a small 2-level predictors tracking 1024 independent load instructions, with an 8 bit hit/miss history per instruction (requiring 256 entries in the second level table). It improves the performance of sparse codes, their application domain of interest, on average by 14%, with a 13.6% increase in power used to improve the performance of SPEC benchmarks by an average of 2.9% at the cost of 7.1% increase in average power.

1.6.3 Vision and Experiments on Load Hit/Miss Prediction

According to [15], the major problem in the predictor is AMPH loads. AMPH should not make the performance worse than FLUSH or STALL because this is the normal case if there is no predictor. The problem we see is in AHPM (Actual Hit - Predicted Miss). This problem can be overcome by making the predictor biased to predicting hit. However, the importance of load/miss predictors is not practically guaranteed because the in use predictors are not efficient. So, we thought that we need to determine whether the problem is in non-efficient predictors or in the load miss prediction itself. Hence, we decided to try a theoretical perfect predictor where all its predictions are correct. If the performance improvement gained from applying this predictor is valuable, then the idea of load hit/miss prediction is valuable and worths more research to propose an efficient predictor. If the performance improvement gained was small, then this is a proof that the idea of load hit/miss prediction is not a good design choice.

1.6.4 Conclusion of Working on Load Hit/Miss Prediction

We followed the approach mentioned in the previous subsection to achieve our goal. The performance improvement gained by applying perfect load miss predictor was small and we realized that designing and implementing an efficient practical load miss predictor is really hard to be achieved. It is hard to achieve both efficiency and being practical. So, we decided to move on to another research point which is memory requests scheduling for SMT and multicore processors.

1.7 Thesis Layout

This section describes the rest of this thesis. Chapter 2 includes literature survey about memory access scheduling. Chapter 3 contains the proposed memory access scheduling algorithms. Chapter 4 contains the simulation environment, machine configuration, workloads, and performance metric used in this work.

Chapter 5 contains the results of our experiments. In chapter 6 we conclude and state our vision for the future work.

Chapter 2

Memory Access Scheduling in Literature

Memory access scheduling has been developed for superscalar, multithreaded, and multicore processors to enhance/improve their performance. The algorithms used are changed and enhanced when moved from single-threaded processors to multithreaded and multicore processors. The concept of memory access scheduling itself is changed from just re-ordering memory requests from the same thread making use of memory hardware features to scheduling the requests from different threads to allow better utilization of processor resources. In the following sections we will mention different memory access scheduling algorithms in superscalar, multithreaded, and multicore processors.

2.1 Schemes for Single-Threaded Single-Core Processors

In single-threaded single-core processors memory access scheduling is focused on re-ordering memory requests to improve memory hardware design by reducing the gap between processor and memory latency. Although main memory is a RAM (Random Access Memory) device, its access pattern is not random. In other words, the ordering of memory requests change the latency time of memory

access.

In [7], memory access scheduling technique is used in superscalar processors to re-order the DRAM operations. These operations include bank precharge, row activation, and column access. These operations should be done for all pending memory requests. So, the main focus of this thesis was to make use of DRAM 3D (bank, row, and column) hardware nature.

In [11] the authors take a different approach. They focused on designing parallelized memory controller. They introduce SCHED which is a memory access scheduler. It is responsible for ordering the read, write requests, bank activates, and precharges, and driving the SDRAM.

The Ph.D thesis [10] presented a compiler technology called access ordering. It tries to solve the memory bandwidth problem for scalar processors by utilizing memory system resources through memory accesses reordering.

In [8] they examined memory access ordering and tried to find the boundary for performance improvement.

In [9] they introduced Memory Scheduling Unit (MSU). This unit is used to prefetch read requests, buffer write requests, and dynamically reorder the memory accesses in order to maximize the effective memory bandwidth.

The main problem in the previously mentioned algorithms is that they are suitable for single-threaded processors only. They are not made to handle the case of ordering requests from different threads. So, they are out of the scope of this thesis. The next section contains the algorithms that are used in multithreaded and multicore processors.

There are some algorithms that were first proposed for single-threaded processors but then they were used in SMT processors as mentioned in [19]. These algorithms are FCFS (First Come First Served), hit-first algorithm, read-first algorithm, and age-based algorithm.

FCFS serves the request that arrives to the scheduler first regardless all other resources and factors. Its advantage is that it is very simple. Its disadvantage is that it does not take into account the criticality of resources or requests.

Hit-first algorithm gives row buffer hits more priority than row buffer misses. So, it gives more priority to requests that take less time. This algorithm corresponds to the same category of algorithms that exploits the memory hardware features to improve throughput.

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 bottle because of the existence of write buffers.

Both hit-first, and read-first algorithms can be used in collaboration with other algorithms. For instance, in [19] the authors used hit-first and read-first algorithms on top of 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.

2.2 Schemes for SMT and Multicore Processors

The concept of memory access scheduling is discussed for SMT processors in [19]. Moving to SMT, generally, increases contention on DRAM as the number of threads is increased. 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. Two of these algorithms are resource-based algorithms which are ROB-based and IQ-based algorithms, and the third one is the request-based algorithm. They compared these algorithms to some algorithms that were developed for single-threaded processors, which are hit-first, read-first, and age-based algorithms.

In this thesis, they classified algorithms to resource-based algorithms and request-

based algorithms. We add two other categories to this classification which are fairness-based algorithms and parallelism-based algorithms.

2.2.1 Resource-Based Algorithms

The main idea behind resource-based algorithms is trying to decrease contention on the resources that represent a bottleneck. Here are three resource-based algorithms. Two of them are of direct relation with resources availability, which are ROB-based algorithm and IQ-based algorithm. The third algorithm, which is RIR (Ready to In-flight Ratio), is related to resources indirectly as explained below.

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. This algorithm aims to improve throughput but has nothing to do with fairness.

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. This algorithm, also, aims to improve throughput but has nothing to do with fairness.

In [20], the authors proposed a new metric which is RIR. RIR is the ratio between the number of ready instructions in the issue queues, and the number of in-flight instructions from the issue to the writeback stages. A high ratio of ready to in-flight instructions means that the thread is able to make good progress with whatever resources it has available, and that preventing it from those resources will non-trivially degrade its performance. Hence, a high value for this ratio indicates high sensitivity for resources. They used this metric in phase co-scheduling

for a dual-core CMP of dual-threaded SMT processors. However, we think that using this metric can give good results in memory access scheduling. So, we decide to use this metric in memory access scheduling under an algorithm with the same name (RIR), and this algorithm is added to our comparisons.

2.2.2 Request-Based Algorithms

Request-based algorithm gives highest priority to requests from thread that has the minimum number of pending requests. In some 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. This algorithm aims to improve throughput but has nothing to do with fairness. The authors results say that the improvement of request-based algorithm is getting decreased when the number of threads is increased. They concluded that the difference of the number of pending requests in 4 or 8 threads is not as large as 2 threads, so this affects the accuracy and efficiency of LREQ algorithm.

Figure 2.1 shows the results of comparing FCFS, hit-first, age-based, ROB-based, IQ-based, and LREQ algorithms according to the simulation environment and assumptions made in [19].

In [21] 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 as shown in equation 2.1[1]. Memory efficiency can be calculated using offline, or online 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 problem in this algorithm is that both online and offline profiling are not practical.

[1] [21]

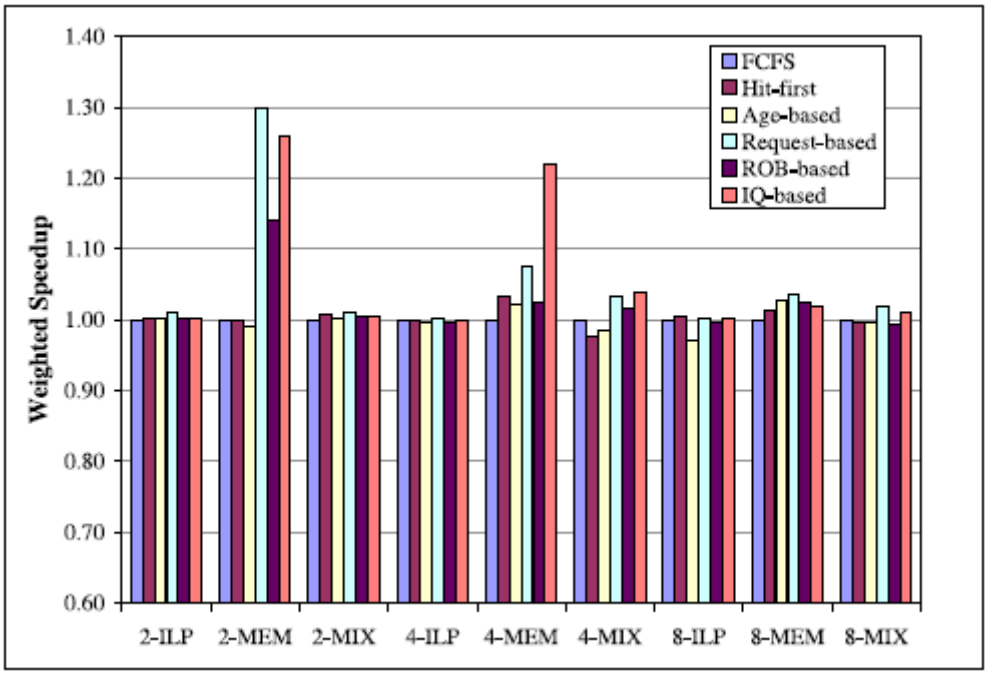


Figure 2.1: Comparison between some algorithms from [19]

$$ME[i] = \frac{IPC_{single}[i]}{BW_{single}[i]} \tag{2.1}$$

2.2.3 Fairness-Based Algorithms

Fairness is an important factor that is targeted by some algorithms. In single-threaded processors, the age-based algorithm was proposed. It is still valid for SMT and multicore processors. In addition to the age-based algorithm, there are other fairness-based algorithms proposed for multithreaded and multicore processors. One of these algorithms is the popular RR (Round Robin) algorithm. Round Robin loops on threads that have pending memory requests and serves one request from each thread. RR algorithm can be represented by the following formula:

$$ThreadID_{next\ request} = \frac{(ThreadID_{last\ served\ request} + 1)}{Total\ no.\ of\ threads\ having\ pending\ requests} \tag{2.2}$$

Both age-based algorithm, and RR aim to achieve fairness but by slightly different ways. Age-based algorithm targets fairness between requests regardless the

threads that issued these requests. This means that if thread i issued 3 memory requests at time X, and thread j issued one memory request at time $X + 10$, the 3 requests of thread i will be served first because they are older (assuming that the total number of requests in the queue is greater than 8).

On the other hand, RR algorithm targets fairness between threads and does not take into consideration how old the requests from these threads are. Considering the same example in the last paragraph and applying RR instead of age-based algorithm, a request from thread i will be served then a request from thread j.

In other words and using the classification used in section 1.5, age-based algorithm is trying to achieve fairness on memory requests level and RR is trying to achieve fairness on thread level. Another fairness-based memory access scheduling algorithm is the algorithm proposed in [22]. The algorithm name is STFM (Stall-Time Fair Memory scheduler). This algorithm mainly aims to improve fairness. The algorithm estimates two values which are T_{shared} and T_{alone}. T_{shared} represents the memory stall-time experienced by the thread when running among other threads in the memory system. T_{alone} represents the memory stall-time experienced by the thread if it had been running alone in the memory system. Estimating T_{shared} is done by incrementing a counter each time the thread can not commit an instruction due to L2 cache miss. However, estimating T_{alone} is not straightforward. It is calculated by the

$$T_{alone} = T_{shared} - T_{interference} \tag{2.3}$$

where $T_{interference}$ is the extra memory stall-time that the thread is suffering because memory requests from other threads are being scheduled before the requests of the thread itself (if it has available waiting requests). Updating $T_{interference}$ is a complicated process as it is being updated for the thread that owns the request being served and other threads. Other threads are updated when there is an interference in memory banks or memory bus. Updating $T_{interference}$ for the thread that owns the request being served seems to be strange. However, think about the following case: if this thread has another following request for the same memory block, if it is running alone, the next request will hit. If it is run-

ning in parallel with other threads, requests from other threads can be scheduled between the two consecutive requests from the owner thread causing the second request to miss. So, even the thread owning the request being served may need $T_interference$ to be updated. So, keeping accurate estimations for $T_{interference}$ is really challenging and hard to be achieved.

After calculating T_{alone} and T_{shared}, the slowdown for each thread should be calculated by the following formula:

$$Slowdown = \frac{T_{shared}}{T_{alone}} \tag{2.4}$$

Then the maximum slowdown and the minimum slowdown between all the threads that have at least one pending request are calculated. If the ratio

$$\frac{maximum\ slowdown}{minimum\ slowdown}$$

exceeds a certain threshold, then next request is scheduled from the thread with the largest slowdown. If this ratio is less than this threshold, then the requests shall be scheduled according to FR-FCFS (First Read, First Come First Served) algoritm.

It is clear that STFM algorithm is complicated. Another comment on this algorithm is that the baseline algorithm used to schedule requests when the unfairness is under threshold is a poor algorithm (which is FR-FCFS). It gives good results in single-threaded processors compared to other algorithms previously mentioned, but things are different in SMT and multicore processors.

2.2.4 Parallelism-Based Algorithms

Parallel-based algorithms mainly aim to exploit parallelism of memory controllers, memory banks, or to exploit the parallelism in applications themselves.

In [23] a new scheduling algorithm called PAR-BS (Parallelism-aware Batch Scheduling) is presented. This algorithm is based on two main ideas which are batch scheduling and parallelism-aware scheduling. Batch scheduling idea aims

to group some memory requests into batches according to their arrival times and their requesting threads. The requests within the oldest batch will have the highest priority. So, this algorithm is starvation-free. Parallelism-aware idea aims to try to make use of bank-level parallelism within the same batch. And since in this thesis we focus on the case where there is single memory bank, PAR-BS is out of the scope of this thesis.

In [24] they proposed a new scheduling algorithm called ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling). This algorithm gives the highest priority to the thread with the least attained service from all memory controllers. This idea is borrowed from the literature of the queuing theory. ATLAS gives good results only with large number of memory controllers.

In [25] they proposed Thread Cluster Memory scheduling (TCM). This algorithm puts threads in one of two categories, either latency-sensitive (memory-non-intensive) or bandwidth-sensitive (memory-intensive). Memory-non-intensive threads are getting higher priority than memory-intensive threads because they are lighter and throughput should be improved if these threads are served first. To improve fairness, priorities of memory-intensive threads are shuffled. However, ATLAS and TCM are, mainly, made to be scalable. They only give good results with large number of cores and memory controllers. Hence, they are out of the scope if this thesis.

In [26], they focused on memory channel partitioning. They aimed to reduce inter-application interference in the memory system by mapping the data of applications that are likely to severely interfere with each other to different memory channels. They proposed two algorithms. The first one is MCP (Memory Channel Partitioning). The key idea of this algorithm is partitioning onto separate channels the data of memory non-intensive and memory-intensive applications, and the data of applications with low and high row-buffer locality. The second algorithm is IMPS (Integrated Memory Partitioning and Scheduling) in which the memory non-intensive applications have high priority as they will not affect memory interference then the other applications will be scheduled according to MCP.

2.3 Summary and Conclusion

In this thesis, we are interested in SMT and multicore architectures. So, the memory access scheduling algorithms that are proposed for superscalar processors and can not be extended to work within multicore architectures are out of our interest.

From resource-based algorithms, we are interested in 2 algorithms which are IQ-based algorithm, and RIR algorithm. They are included in our results. We have included IQ-based algorithm and excluded ROB-based algorithm because the first one gives better results according to 2.1. RIR algorithm is included because RIR factor indicates high sensitivity for resources. So, we have used this factor in a new way to schedule memory requests.

From request-based algorithms, we are interested in the basic algorithm (LREQ). The modified algorithm ME-LREQ is not practical because it requires offline profiling. Even if online profiling is used, it has a main problems. This problem is that ME (Memory Efficiency) idea was meant to be long-term property to indicate how the application profile is. However, online profiling is made using small instruction slices which contradicts the idea of memory efficiency parameterization itself.

From fairness-based algorithms, we are interested in RR algorithm. Age-based algorithm has a problem that it only deals with requests when the number of requests is greater than eight. So, we think it is more suitable to use this algorithm in collaboration with another algorithm, and this is what we propose in next chapter. STFM is a complex algorithm as we mentioned before, and it uses FR-FCFS as the baseline algorithm if the unfairness does not exceed a certain threshold which is another drawback. We decide to exclude this algorithm from our calculations because of these reasons (mainly because of its complexity). We can include this algorithm in our results in the future work when we consider memory banks parallelism.

As mentioned in section 1.3, we focus in this thesis on the basic idea of memory access scheduling. We prefer to eliminate the complications caused by the existence of multiple memory controllers and memory banks in order to reach

the best algorithm for this simple case. Then this algorithm can be extended to support the parallelism in memory banks and memory controllers. This simplification leads us to not include the parallelism-based algorithms in our results in this stage of our work.

Chapter 3

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. The distribution of related requests is dependent on the running workloads.

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. In other words, we change the algorithm conditions from unconditional to conditional 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.

3.1 FLRMR Algorithm

Request-based scheduling algorithm is an effective way for scheduling, especially in improving throughput [19, 21, 25]. 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.

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. Combining these factors together has high potential to boost the machine throughput and at the same time guarantee fairness.

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*:

$$FLRMR\_PF_i = \frac{\#\ pending\_requests_i^2 * under\_starvation\_threshold}{(\#\ related\_requests_i + 1)} \tag{3.1}$$

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*, $under\_starvation\_threshold$ is a boolean that indicates if the starvation time threshold is exceeded by a request from this thread or not, 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 request from a thread is starving (i.e. starvation time threshold has passed since its arrival), $under\_starvation\_threshold$ will be equal to 0 which means that PF for this thread will be equal to 0. So, this thread will have the smallest PF, and this starving request will be served first.

If no requests are starving, $under\_starvation\_threshold$ will be equal to 1, so it will not affect the PF. In this case, 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.

Figure 3.1 includes a flow chart for FLRMR algorithm. It explains how the next request to be served is picked out of the requests list.

3.2 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 section 3.1, FIQMR uses related instructions and block starvation time combined

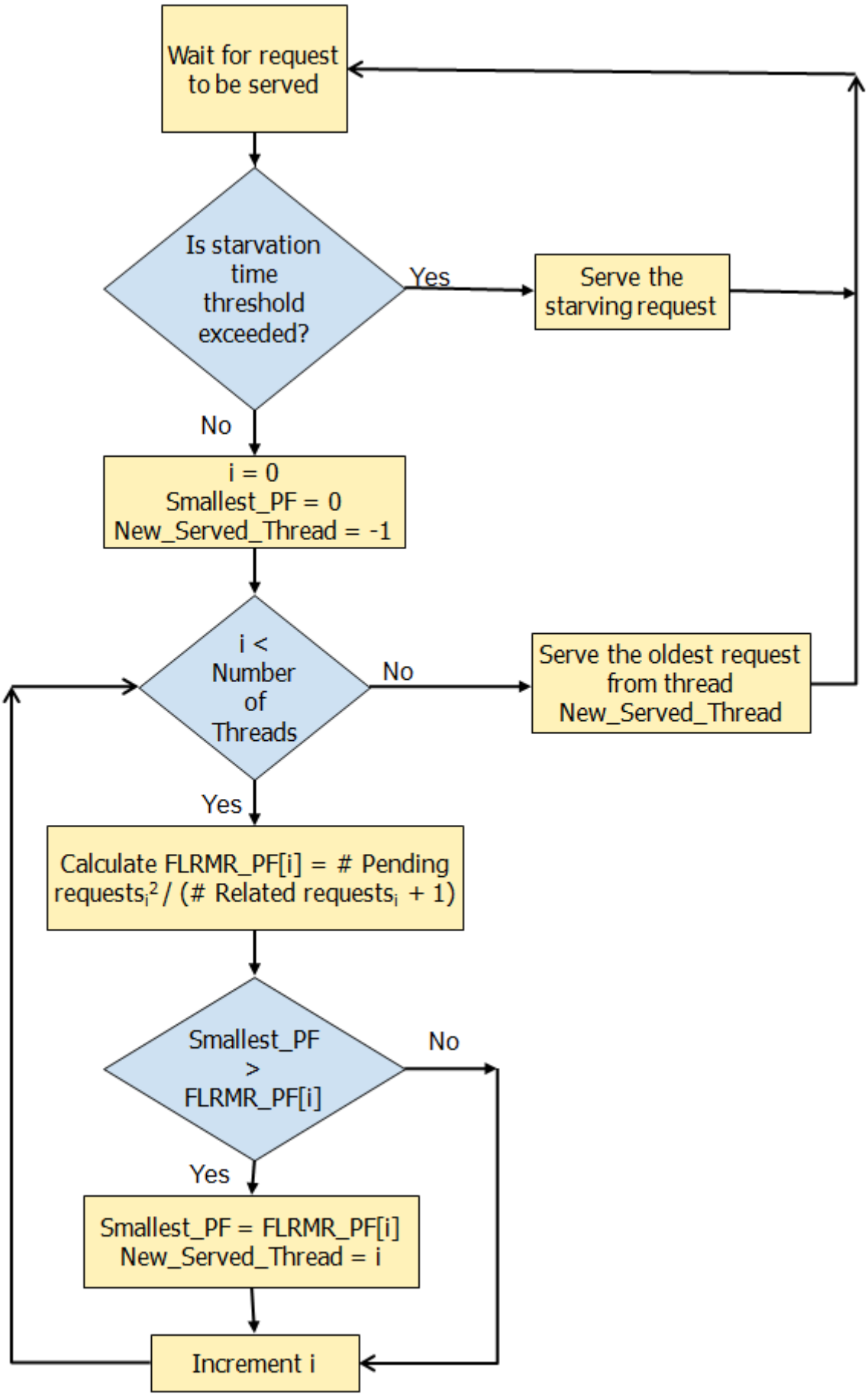


Figure 3.1: Flowchart for FLRMR algorithm

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*:

We believe that the number of IQ entries should have high weight because it is more important and should have higher impact on throughput than related requests.

Hence, we proposed the following $prioritizing\ factor$. Threads will be prioritized according to the following factor for thread i:

$$FIQMR\_PF_i = \frac{\#\ IQ\_entries_i^2 * (\#\ related\_requests_i + 1)}{under\_starvation\_threshold} \tag{3.2}$$

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*, $\#\ related\_requests_i$ is the total number of related requests from thread *i*, and $under\_starvation\_threshold$ is a boolean that indicates if the starvation time threshold is exceeded by a request from this thread or not.

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*.

If a request from a thread is starving (i.e. starvation time threshold has passed since its arrival), $under\_starvation\_threshold$ will be equal to 0 which means that PF for this thread will be equal to $infinity$. So, this thread will have the largest PF, and this starving request will be served first.

If no requests are starving, $under\_starvation\_threshold$ will be equal to 1, so it will not affect the PF. In this case, the thread with the largest PF will be in the top of the list. The oldest request of this thread will be served first, and removed from the list.

Figure 3.2 includes a flow chart for FIQMR algorithm. It explains how the next request to be served is picked out of the requests list.

3.3 Micro-architecture Modifications

Do these new algorithms require extra complex hardware? We did not try them on real hardware, but we guess that 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)[27, 28, 29]. 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

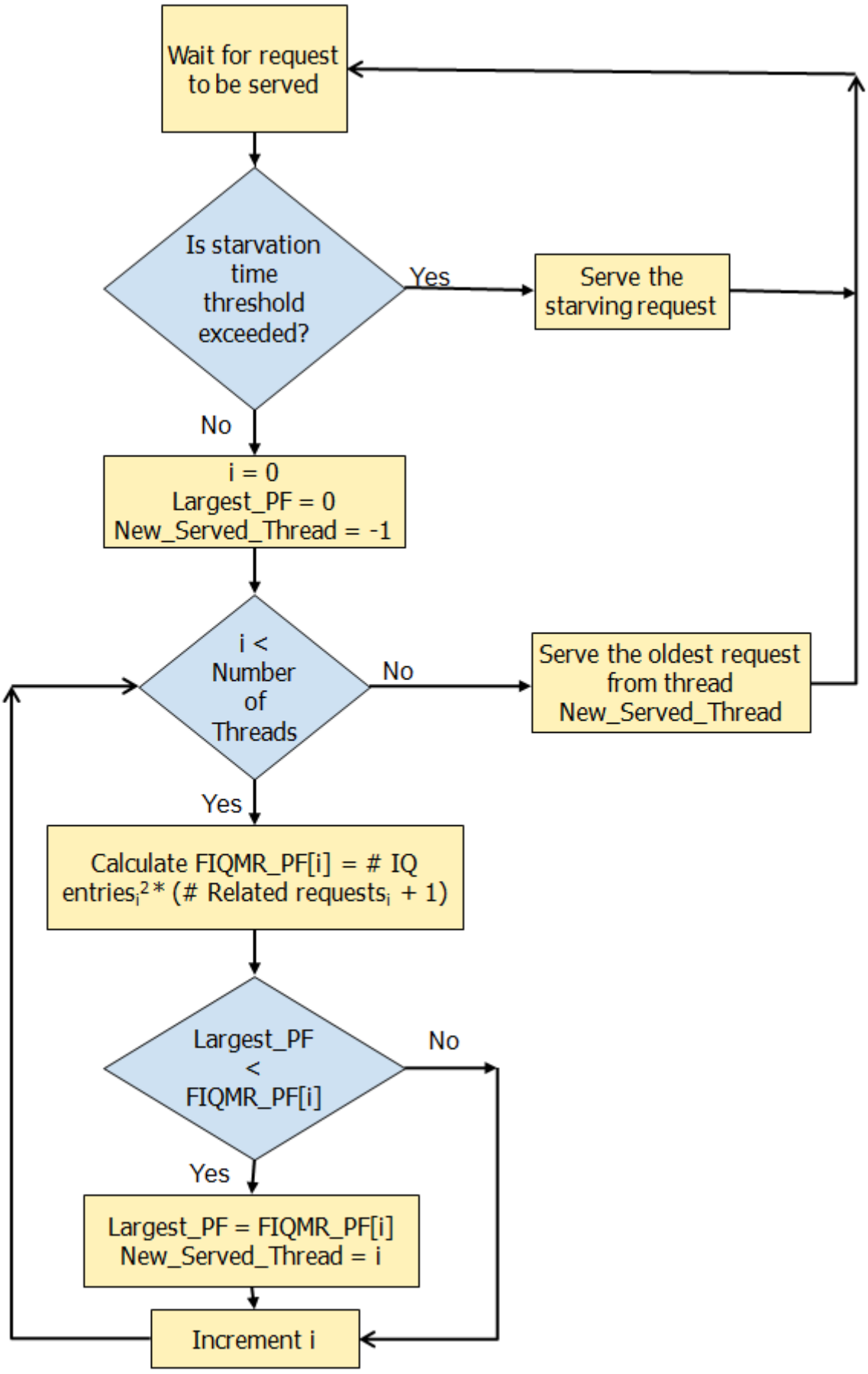


Figure 3.2: Flowchart for FIQMR algorithm

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.

Calculating the PF does not need extra multiplier or divider. We can overcome this by using a lookup table where the values we need are stored like in [21]. This idea is practical in our case because we do not have a big variety of numbers either in number of pending requests or in number of related requests.

Another solution for this issue is using the processor ALU in calculating the PF. The calculation of PF is only done when the current memory request is served, so it is done each memory request. The time between each memory request to be served and the next one is large enough to make these calculation using the processor ALU.

3.4 *Related Requests* Arrival Scenario

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 (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

Core	Benchmark	Fast-forward Count (in million instructions)
Core 0	gcc with input 166.i	1000
Core 1	galgel	1250
Core 2	vpr-place	190
Core 3	gzip with input input.random	40

Table 3.1: Benchmarks used in the explainer example

to SPEC2000 benchmarks suite. Table 3.1 shows the benchmarks used in this example, and fast-forward count for each benchmark. [1]

Table 3.2, contains the details of pending requests from all running threads. First column contains thread order according to FLRMR prioritizing factor formula in equation 3.1. Second column contains core ID.

To know the benchmark running on this core, please refer to table 3.1. Third column contains the number of requests pending from this core. Fourth column contains the total number of requests related to pending requests from this core. Fifth 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.

Order	**CoreID**	**No. Pending requests**	**No. Related requests**	**Arrival time (in cycles)**
At time 77730: Starting monitoring, a request is being served until = 77821				
1	1	2	14	77666
2	0	1	0	77688
3	2	2	1	77465
At time 77732: ThreadID 3 requested a new block				
1	1	2	14	77666
2	0	1	0	77688
3	3	1	0	77732
4	2	2	1	77465
At time 77734: ThreadID 3 requested a block related to a previously requested block				
1	1	2	14	77666
2	3	1	1	77732

[1] The benchmarks are fast-forwarded according to [31].

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
3	0	1	0	77688
4	2	2	1	77465
At time 77737: ThreadID 3 requested a block related to a previously requested block				
1	1	2	14	77666
2	3	1	2	77732
3	0	1	0	77688
4	2	2	1	77465
At time 77739: ThreadID 3 requested a block related to a previously requested block				
1	3	1	3	77732
2	1	2	14	77666
3	0	1	0	77688
4	2	2	1	77465
At time 77744: ThreadID 2 requested a block related to a previously requested block				
1	3	1	3	77732
2	1	2	14	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77752: ThreadID 3 requested a block related to a previously requested block				
1	3	1	4	77732
2	1	2	14	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77754: ThreadID 1 requested a block related to a previously requested block				
1	3	1	4	77732
2	1	2	15	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77755: ThreadID 1 requested a block related to a previously requested block				
1	3	1	4	77732

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
2	1	2	16	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77756: ThreadID 1 requested a block related to a previously requested block				
1	3	1	4	77732
2	1	2	17	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77761: ThreadID 1 requested a block related to a previously requested block				
1	3	1	4	77732
2	1	2	18	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77772: ThreadID 1 requested a block related to a previously requested block				
1	3	1	4	77732
2	1	2	19	77666
3	0	1	0	77688
4	2	2	2	77465
At time 77773: ThreadID 1 requested a block related to a previously requested block				
1	1	2	20	77666
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465
At time 77780: ThreadID 1 requested a block related to a previously requested block				
1	1	2	21	77666
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465
At time 77781: ThreadID 1 requested a block related to a previously requested block				

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
1	1	2	22	77666
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465
At time 77789: ThreadID 1 requested a block related to a previously requested block				
1	1	2	23	77666
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465

Table 3.2: Example 1 illustrating FLRMR algorithm

Note that at time 77730 there are no pending requests from core 3, so it is not displayed in the table at that time.

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 frequency of occurring *related requests* is high.

3.5 Contribution

From the previous sections in this chapter, we can come up with the contribution of this thesis. We have proposed two new memory requests scheduling algorithms which are FLRMR and FIQMR based on LREQ and IQ-based algorithms respectively. The modifications in the original algorithms can be summarized in the following two points:

- Adding the effect or *related requests*. *Related requests* are the memory requests asking for a memory block that is requested by a pending memory request. This is a totally new idea in the memory access scheduling algorithms. This idea aims to improve throughput.

- Adding starvation time threshold to not let the requests wait for a long time waiting to be served. This idea was proposed a separate algorithm called age-based algorithm. However, adding it to another algorithm is a new idea too. This idea aims to improve fairness.

There is an advantage of these two ideas, other than improving fairness and throughput, that is usability. They can be used combined with other algorithms in the same way they are combined with LREQ and IQ-based algorithms. Some examples of the algorithms that can be combined with our proposed ideas are ROB-based algorithm, and RIR algorithm. This combination improves both fairness and throughput achieved by the original algorithms.

This idea is proven by applying it to two different algorithms in this thesis which are: LREQ and IQ-based algorithms. However, to explain it more, we propose in the next subsections the *prioritizing factor* for modified versions of two other algorithms which are ROB-based and RIR algorithms.

3.5.1 Modified ROB-Based Algorithm

Equation 3.3 contains the formula for the *prioritizing factor* of the modified ROB-based algorithm:

$$Modified\_ROB\_PF_i = \frac{\#\ ROB\_entries_i^2 * (\#\ related\_requests_i + 1)}{under\_starvation\_threshold} \tag{3.3}$$

where *i* is the thread ID, $Modified\_ROB\_PF_i$ is the *prioritizing factor* of thread *i* according to the modified ROB-based algorithm, $\#\ ROB\_entries_i$ is

the number of ROB-entries occupied by thread *i*, $\#\ related\_requests_i$ is the total number of related requests from thread *i*, and $under\_starvation\_threshold$ is a boolean that indicates if the starvation time threshold is exceeded by a request from this thread or not.

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

If a request from a thread is starving (i.e. starvation time threshold has passed since its arrival), $under\_starvation\_threshold$ will be equal to 0 which means that PF for this thread will be equal to $infinity$. So, this thread will have the largest PF, and this starving request will be served first.

3.5.2 Modified RIR Algorithm

Equation 3.4 contains the formula for the *prioritizing factor* of the modified RIR algorithm:

$$Modified\_RIR\_PF_i = \frac{RIR_i^2 * (\#\ related\_requests_i + 1)}{under\_starvation\_threshold} \tag{3.4}$$

where *i* is the thread ID, $Modified\_RIR\_PF_i$ is the *prioritizing factor* of thread *i* according to the modified RIR algorithm, $\#RIR_i$ is the ratio between the number of ready instructions in the issue queues occupied by thread i, and the number of in-flight instructions from the issue to the writeback stages occupied by thread *i*, $\#\ related\_requests_i$ is the total number of related requests from thread *i*, and $under\_starvation\_threshold$ is a boolean that indicates if the starvation time threshold is exceeded by a request from this thread or not.

The larger this factor is, the higher priority this thread will have. Similar to FLRMR, FIQMR and modified IQ-based algorithm, RIR ratio of thread i is given higher weight than the number of *related requests*.

If a request from a thread is starving (i.e. starvation time threshold has passed

since its arrival), $under\_starvation\_threshold$ will be equal to 0 which means that PF for this thread will be equal to $infinity$. So, this thread will have the largest PF, and this starving request will be served first.

Chapter 4

Simulation Methodology

4.1 Simulation Environment and Machine Configuration

We use the simulator developed in [2]. It is a multi-core version of Simplescalar-3.0[32] for the Alpha AXP instruction set. However, the proposed algorithms are applicable for multithreaded processors as well.

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 adding queues for request, and simulating conflicts between these requests. The baseline algorithm used is FCFS. This means that all the results in chapter 5 are relative to FCFS algorithm. We have implemented RR, LREQ, IQ-based algorithm, and RIR algorithm, in addition to the newly proposed algorithms 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 4.1 shows the major simulation parameters used.

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 4.1: Major simulation parameters

4.2 Benchmarks and Workloads

In our simulation, we use single-threaded cores. Each core run a separate application. We use the classification of SPEC CPU2000 benchmarks from [21]. They 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 do not use HPC applications as we believe that multicore processors are no more limited to HPC.

Table 4.2 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 give each workload a name to be identified in performance evaluation, and analysis. Workload names and benchmarks included in each workload are shown in table 4.3.

Workload name consists of three parts. First part is the number of cores used to

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

Table 4.2: Benchmarks classification

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

Table 4.3: Workloads description

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). More examples are mentioned below:

Example 1: Workload 4mem1 is the first workload in the workloads that contain four memory-intensive benchmarks.

Example2: Workload 2mix1 is the first workload in workloads that contain two benchmarks (one memory-intensive, and one compute-intensive).

Example 3: Workload 8mem2 is the eight workload in the workloads that contain eight memory-intensive benchmarks.

4.3 Algorithm Parameters

There is a trade-off in choosing the starvation time threshold. If the threshold is chosen to be small, this will make this threshold always decide the next request to be served. So, the other factors in the algorithm that includes this threshold will have almost no effect on the results. In other words, if we choose starvation time to be equal to:

$$number\ of\ threads\ *\ memory\ latency$$

this means that any algorithm that uses this starvation time will act like Round Robin algorithm, i.e. loops on the threads that have pending requests and serve one request from each thread (on condition that the requests are issued in a relatively small period of time). If the threshold is chosen to be large, this will affect fairness which is one of the targets of our algorithms.

According to our experiments on 4-cores workloads, we found that setting starvation time threshold to 10 times the memory latency (which is in 100 cycles in our configurations) is fair enough and gives good throughput results. So, the starvation time threshold used for 4-cores workloads is 1000 cycles. Then we found that it is not suitable to make the starvation time constant for 2-cores, 4-cores, and 8-cores workloads. So, we decided to make it a factor of number of threads.

According to our experiments, we come up with the following formula by which we calculate the starvation time threshold for different number of threads:

$$2.5 * memory\ latency * number\ of\ threads$$

4.4 Performance Metric

4.4.1 Speedup Metric

For speedup comparison, there are different metrics that can be used. One of them is the arithmetic mean which is given by the following formula for *n* threads:

$$\frac{\sum_n \frac{IPC_{new}}{IPC_{old}}}{n}$$

Another metric that can be used is the harmonic mean. It is given by the following formula for *n* threads:

$$\frac{n}{\sum_n \frac{IPC_{old}}{IPC_{new}}}$$

However, we preferred to use 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 [2]. 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.

4.4.2 Fairness Metric

To calculate fairness, we follow the same way used in [22, 21]. *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 among other applications in the workload to its stall time when it is running alone. Equation 4.1 describes how to calculate slowdown for an application.

To calculate $unfairness$ of a memory system, we should calculate the slowdown for all applications in the workload. Then divide the maximum slowdown by the minimum slowdown to get the unfairness of the memory system. Equation 4.2 shows how to calculate unfairness of a memory system.

$$Slowdown\ for\ application = \frac{Stall\ time\ because\ of\ loads_{among\ applications}}{Stall\ time\ because\ of\ loads_{running\ alone}} \tag{4.1}$$

$$Unfairness = \frac{Max.\ Application\ Slowdown}{Min.\ Application\ Slowdown} \tag{4.2}$$

Chapter 5

Simulation Results

We compared the results of seven different algorithms: FCFS, RR, LREQ, IQ-based, RIR, 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.

5.1 Scheduling Scenario

In this section, we will list some examples that illustrate memory requests scheduling algorithms and show different scheduling scenarios. Subsection 5.1.1 shows how different algorithms behave with memory requests in 4-cores workloads. Subsection 5.1.2 shows how the starvation time threshold affects memory requests scheduling in FLRMR algorithm. Finally, subsection 5.1.3 shows a scheduling scenario for 8-cores workloads.

5.1.1 Scenarios for Different Algorithms on 4-Cores Workload

Here we will show how different algorithms will behave with requests in table 3.2.

At time 77730 we started monitoring memory requests from each core. To know why cores are in this order, we will calculate the prioritizing factor for each core.

For core 1 (which is running galgel benchmark):

$$Prioritizing factor_{core1} = \frac{2^2}{(14+1)} = \frac{4}{15} = 0.2667$$

For core 0 (which is running gcc benchmark):

$$Prioritizing factor_{core0} = \frac{1^2}{(0+1)} = 1$$

For core 2 (which is running vpr benchmark):

$$Prioritizing factor_{core2} = \frac{2^2}{(2+1)} = \frac{4}{3} = 1.333$$

At that time there are no pending requests from core 3, so it is not displayed in the table at that time.

Calculations can be done in each other step. In this example, calculations are done when a new request is added to explain how this algorithm can work. In real hardware this overhead is not required. These calculations can be done when the memory serve the last request and get ready to serve a new one.

Normally, PF calculations can be done when memory serves the last request and gets ready for serving a new one but we reorder the table entries each time a new/related request arrives according to equation 3.1 to illustrate the frequency of *related requests* and their effect on performance.

At time 77821 as shown in table 5.1, 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, RIR, 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 factors into consideration. It will just pick a

request from the core whose CoreID equals to

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

RIR will calculate the ready-to-inflight ratio, and accordingly it will pick the next request to be served.

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 calculated as follows:

For core 1 (which is running galgel benchmark):

$$Prioritizing factor_{core1} = \frac{2^2}{(23+1)} = \frac{4}{15} = 0.1667$$

For core 3 (which is running gzip benchmark):

$$Prioritizing factor_{core0} = \frac{1^2}{(4+1)} = 0.2$$

For core 0 (which is running gcc benchmark):

$$Prioritizing factor_{core0} = \frac{1^2}{(0+1)} = 1$$

For core 2 (which is running vpr benchmark):

$$Prioritizing factor_{core2} = \frac{2^2}{(2+1)} = \frac{4}{3} = 1.333$$

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.

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
At time 77820: Requests waiting for scheduling				
1	1	2	23	77666
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465
At time 77821: Memory served last request, and another request is picked from the waiting list				
1	1	1	10	77725
2	3	1	4	77732
3	0	1	0	77688
4	2	2	2	77465

Table 5.1: Example 1 illustrating FLRMR algorithm

5.1.2 Starvation Time Threshold Effect on 4-Cores Workload

Table 5.2 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 from other threads to be served.

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
At time 78464: Starting monitoring, a request is being served until = 78563				
1	1	1	8	78398
2	2	2	2	77465
3	0	2	2	77688
At time 78465: Starvation time threshold for oldest block in core 2 is reached				
1	2	2	2	77465
2	1	1	8	78398
3	0	2	2	77688

Table 5.2: Example 2 illustrating FLRMR algorithm

5.1.3 FLRMR Scheduling Scenario on 8-cores Workload

Table tbl:example3 shows how a scheduling scenario for 8-cores workloads. The workload used is 8mem1. Please refer to table 4.3 to know the benchmarks contained in this workload. Starvation time threshold used in this example is 2000 cycles.

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
At time 134093: Starting monitoring, a request is being served until = 134101				
0	5	1	7	134035
1	3	1	2	133819
2	6	2	4	133069
3	4	2	3	133590
At time 134094: ThreadID 2 and ThreadID 0 requested a new block				
0	5	1	7	134035
1	3	1	2	133819

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
5	0	1	0	134094
At time 134095: ThreadID 0 requested a block related to a previously requested block				
0	5	1	7	134035
1	3	1	2	133819
2	0	1	1	134094
3	6	2	4	133069
4	4	2	3	133590
5	2	1	0	134094
At time 134099: ThreadID 0 requested a block related to a previously requested block				
0	5	1	7	134035
1	3	1	2	133819
2	0	1	2	134094
3	6	2	4	133069
4	4	2	3	133590
5	2	1	0	134094
At time 134101: Memory served last request, and another request is picked from ThreadID 5 and ThreadID 0 requested a block related to a previously requested block				
0	0	1	3	134094
1	3	1	2	133819
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134113: ThreadID 3 requested a block related to a previously requested				

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
block				
0	0	1	3	134094
1	3	1	3	133819
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134115: ThreadID 3 requested a block related to a previously requested block				
0	3	1	4	133819
1	0	1	3	134094
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134121: ThreadID 3 requested a block related to a previously requested block				
0	3	1	5	133819
1	0	1	3	134094
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134123: ThreadID 3 requested a block related to a previously requested block				
0	3	1	6	133819
1	0	1	3	134094
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134125: ThreadID 3 requested a block related to a previously requested block				

Order	CoreID	No. Pending requests	No. Related requests	Arrival time (in cycles)
0	3	1	7	133819
1	0	1	3	134094
2	6	2	4	133069
3	4	2	3	133590
4	2	1	0	134094
At time 134207: Memory served last request, and another request is picked from ThreadID 3				
0	0	1	3	134094
1	6	2	4	133069
2	4	2	3	133590
3	2	1	0	134094

Table 5.3: Example 3 illustrating FLRMR algorithm

5.2 Performance Evaluation

Figures 5.1, 5.2, 5.3, 5.4 show the speedup gained by applying different algorithms on 2mem1, 2mem2, 2mix1, and 2mix2 workloads respectively.

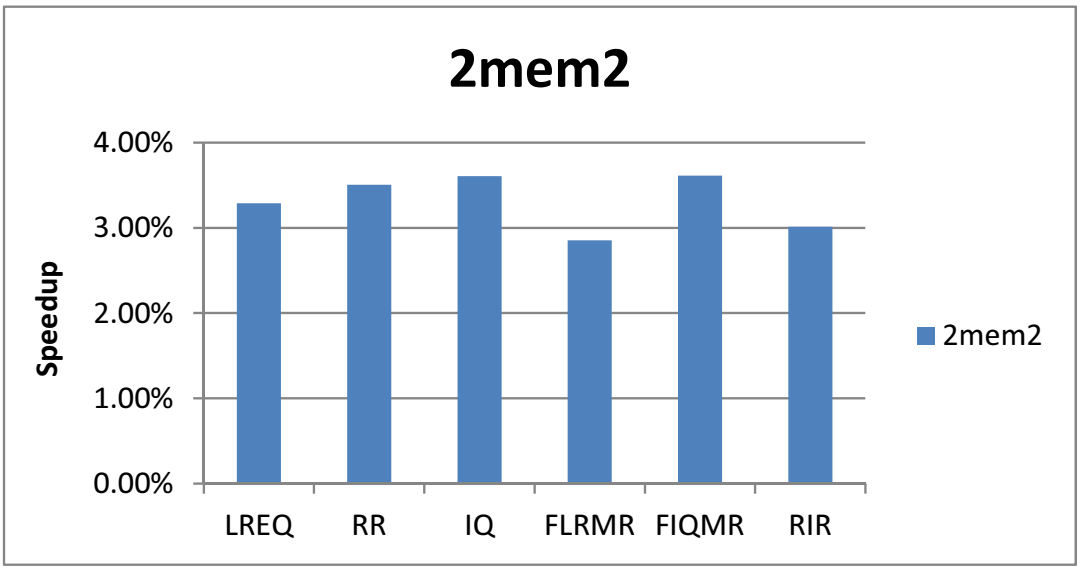


Figure 5.2: 2mem2 workload results

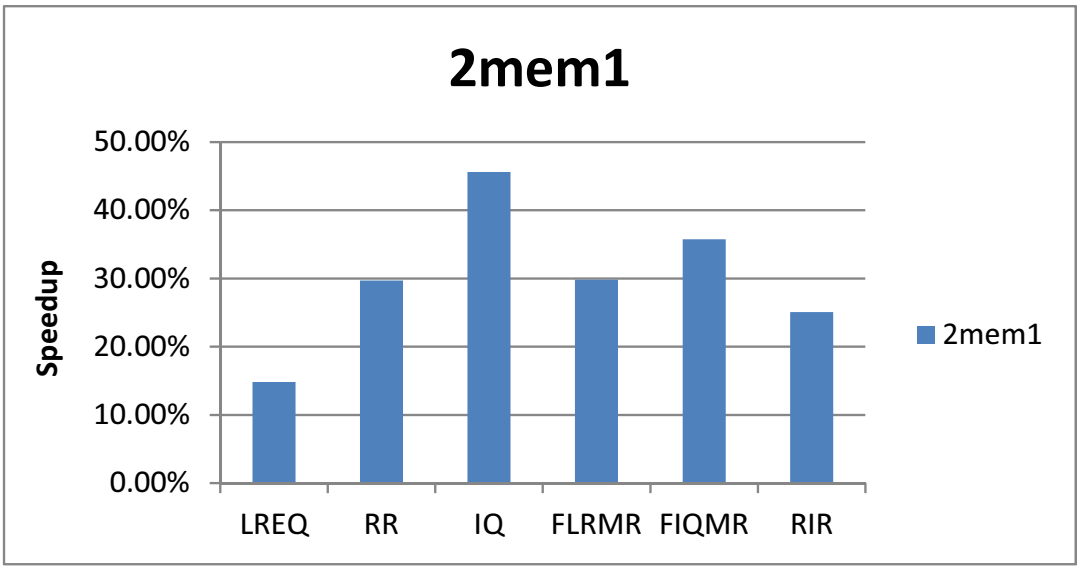


Figure 5.1: 2mem1 workload results

Figure 5.5 shows the average speedup gained by applying different algorithms on all 2-cores workloads.

Figure 5.6 shows a comparison between the percentage of speedup gained by running memory access scheduling algorithms on different 2-cores workloads.

The change in throughput 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

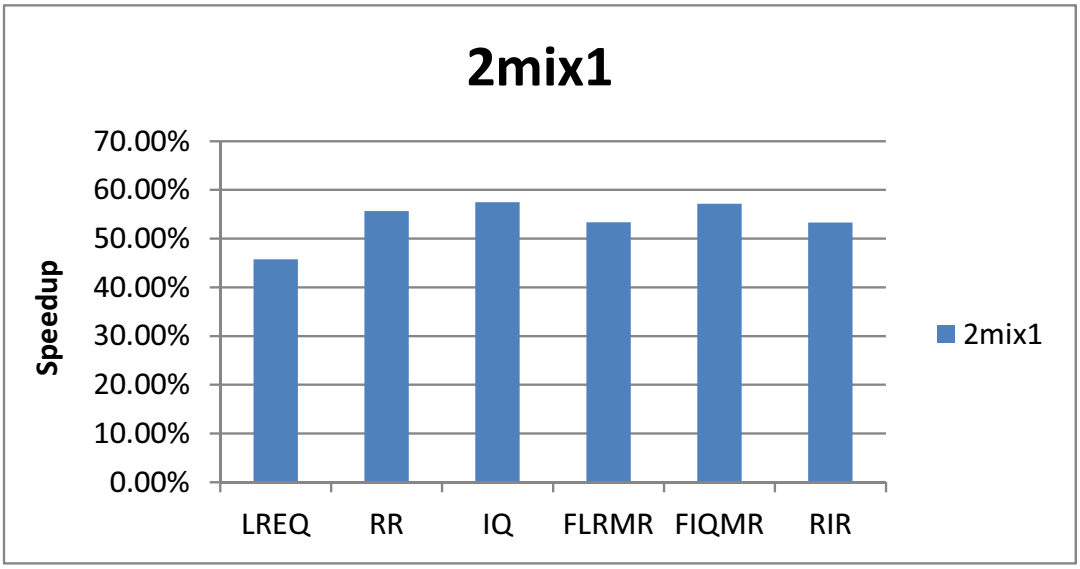


Figure 5.3: 2mix1 workload results

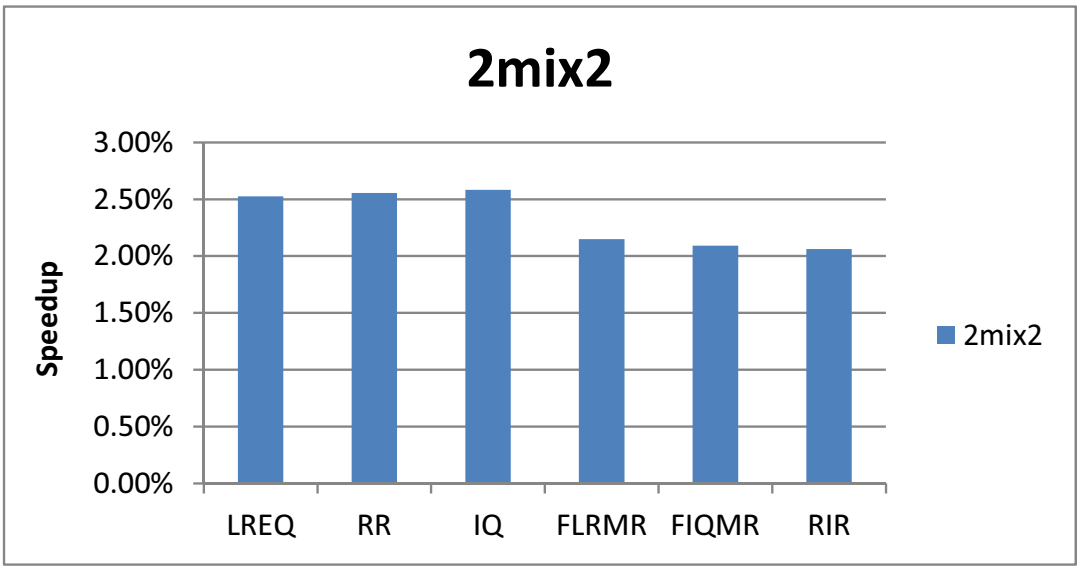


Figure 5.4: 2mix2 workload results

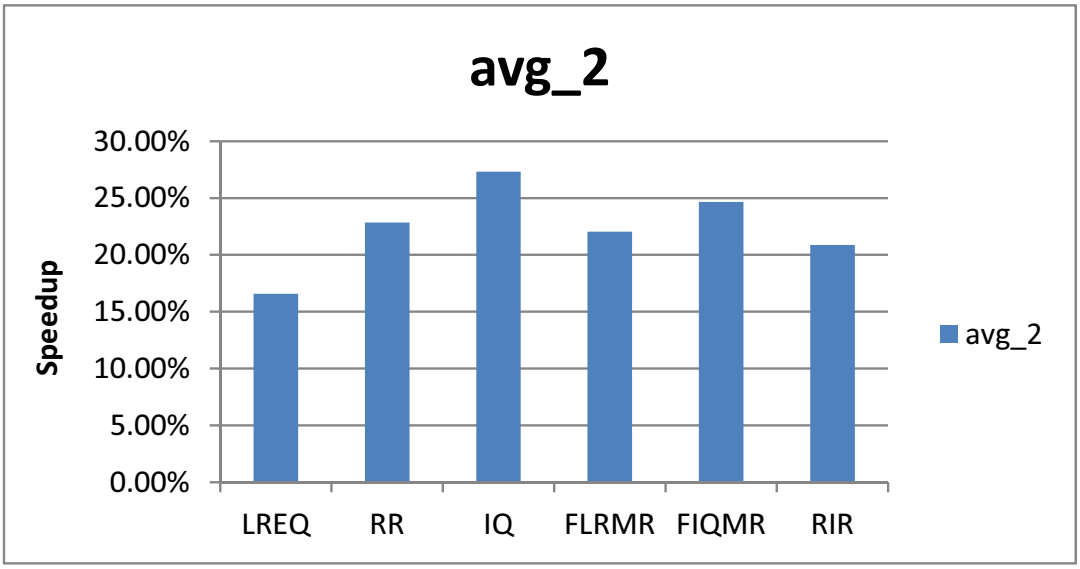


Figure 5.5: Average results of 2-cores workloads

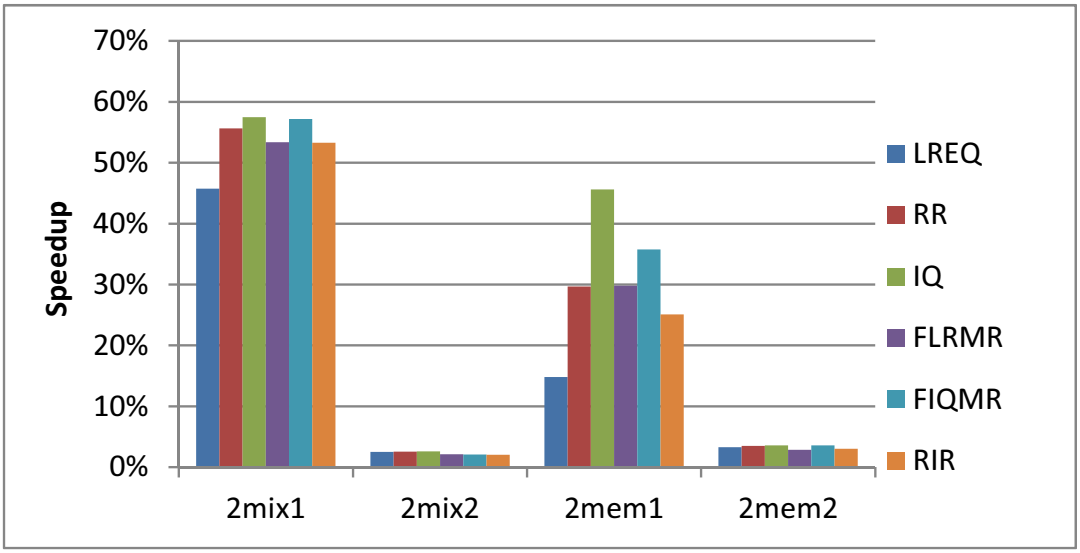


Figure 5.6: Comparison between 2-cores workloads results

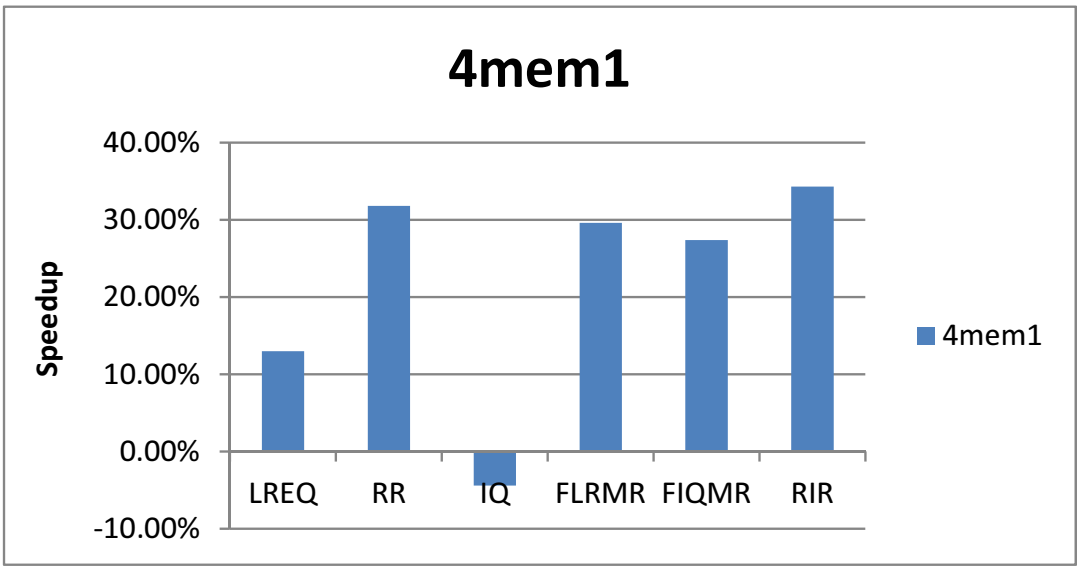


Figure 5.7: 4mem1 workload results

workloads, and the expected performance improvement gained by applying any algorithm is less than the improvement when scheduling requests from larger number of cores.

Figures 5.7, 5.8, 5.9, 5.10 show the speedup gained by applying different algorithms on 4mem1, 4mem2, 4mix1, and 4mix2 workloads respectively.

In 4mem1 workload, FLRMR improves the speedup over LREQ by about 17%, and FIQMR improves the speedup over IQ-based algorithm by about 32%. Similar results are obtained by applying the memory access scheduling algorithms on other 4-cores workloads. Only in 4mem2 algorithm LREQ improves speedup over FLRMR by about 12%. It is not a big problem because it is acceptable to sacrifice speedup in some cases in order to achieve fairness.

Figure 5.11 shows the average speedup gained by applying different algorithms on all 4-cores workloads.

Figure 5.12 shows a comparison between the percentage of speedup gained by running memory access scheduling algorithms on different 4-cores workloads.

Figures 5.13, 5.14, 5.15, 5.16 show the speedup gained by applying different algorithms on 8mem1, 8mem2, 8mix1, and 8mix2 workloads respectively.

Figure 5.17 shows the average speedup gained by applying different algorithms on all 4-cores workloads.

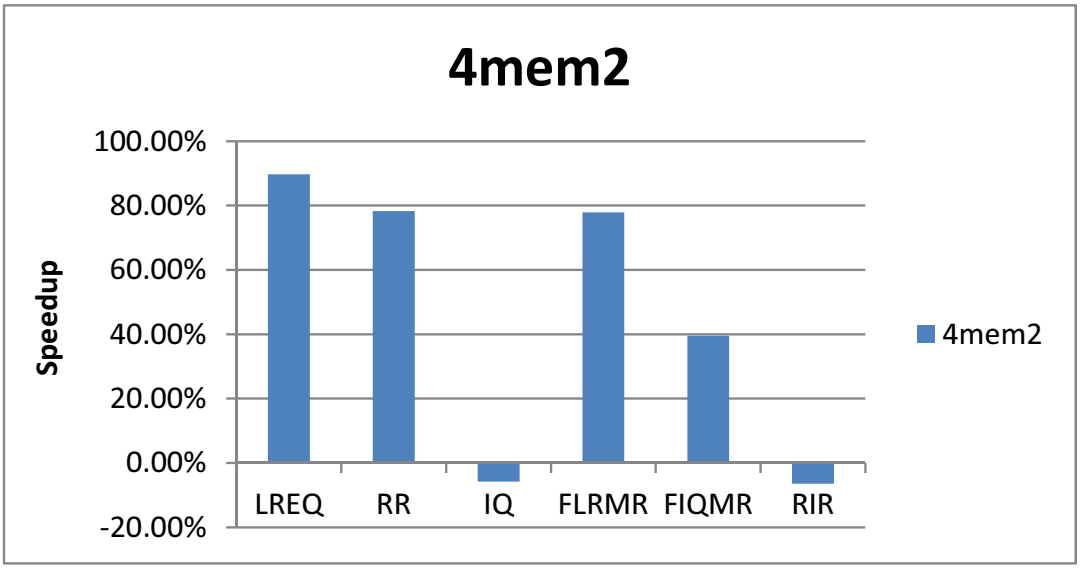


Figure 5.8: 4mem2 workload results

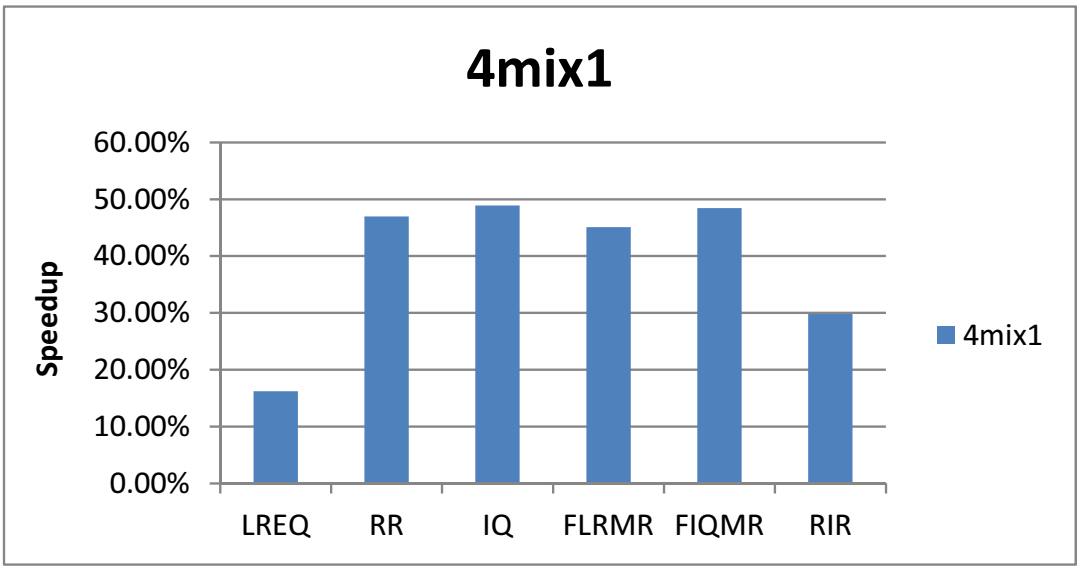


Figure 5.9: 4mix1 workload results

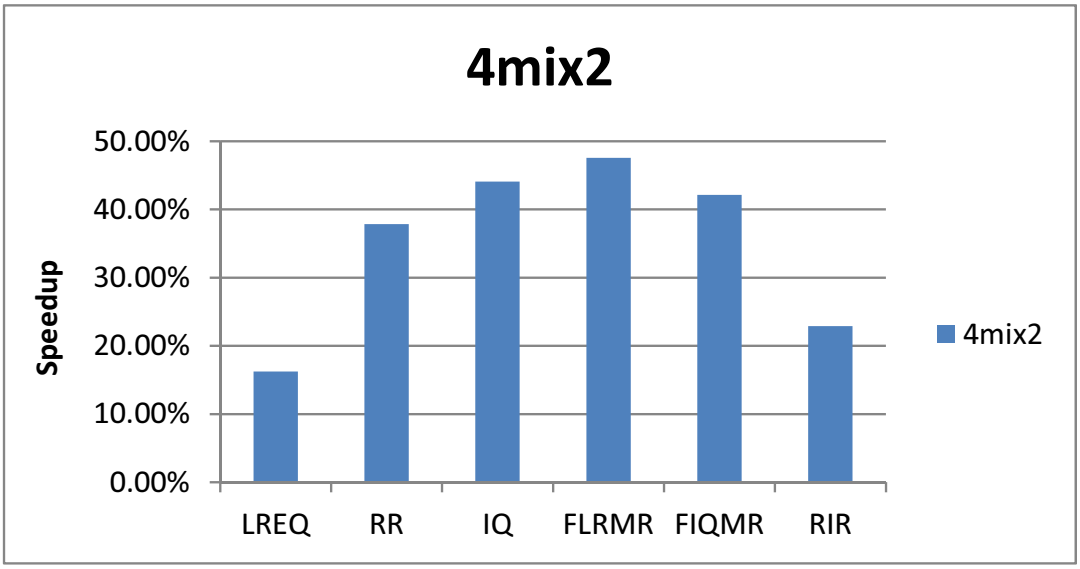


Figure 5.10: 4mix2 workload results

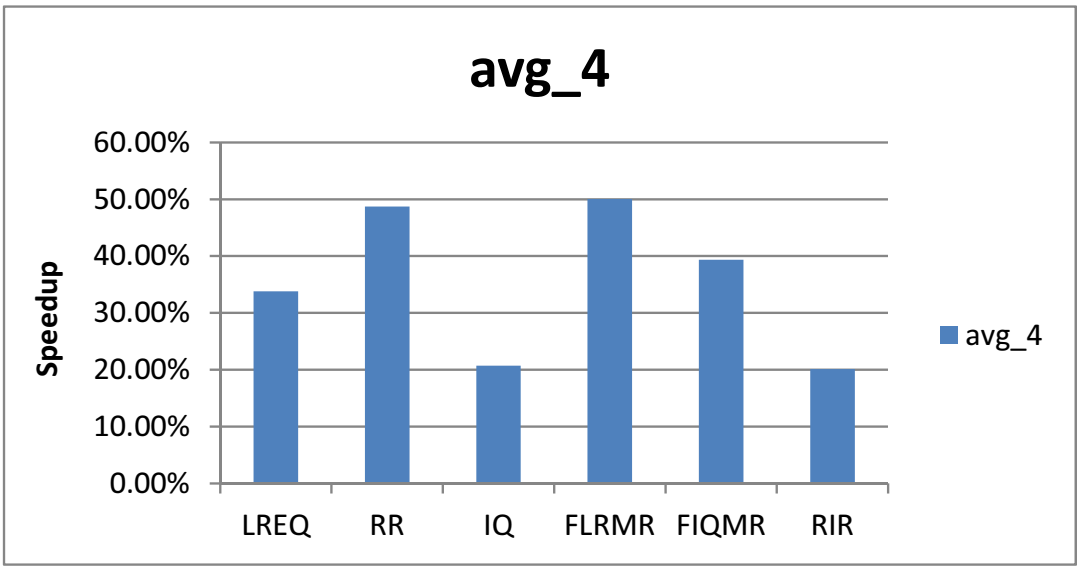


Figure 5.11: Average results of 4-cores workloads

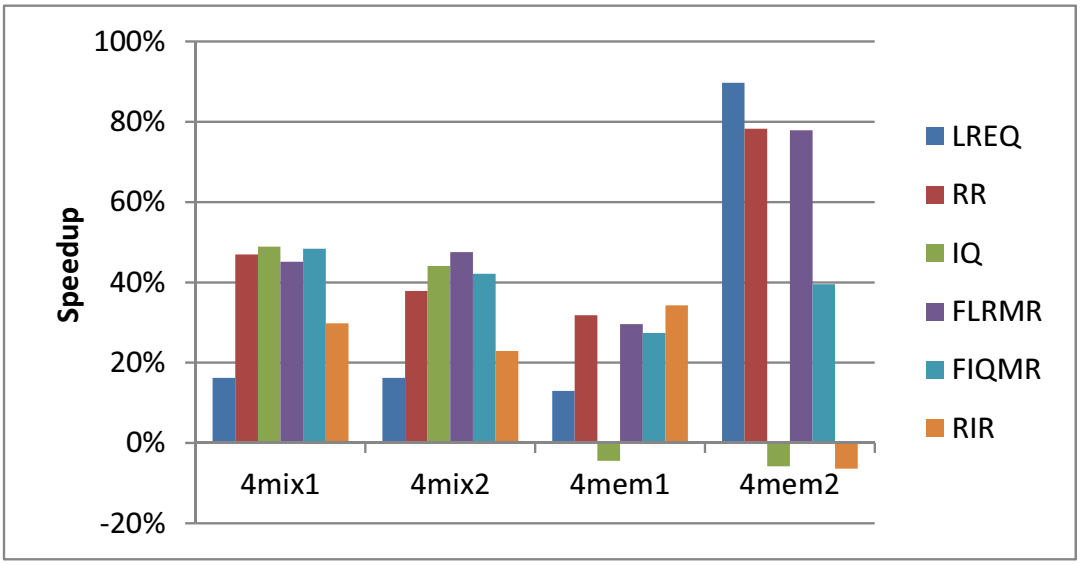


Figure 5.12: 4-cores workloads results

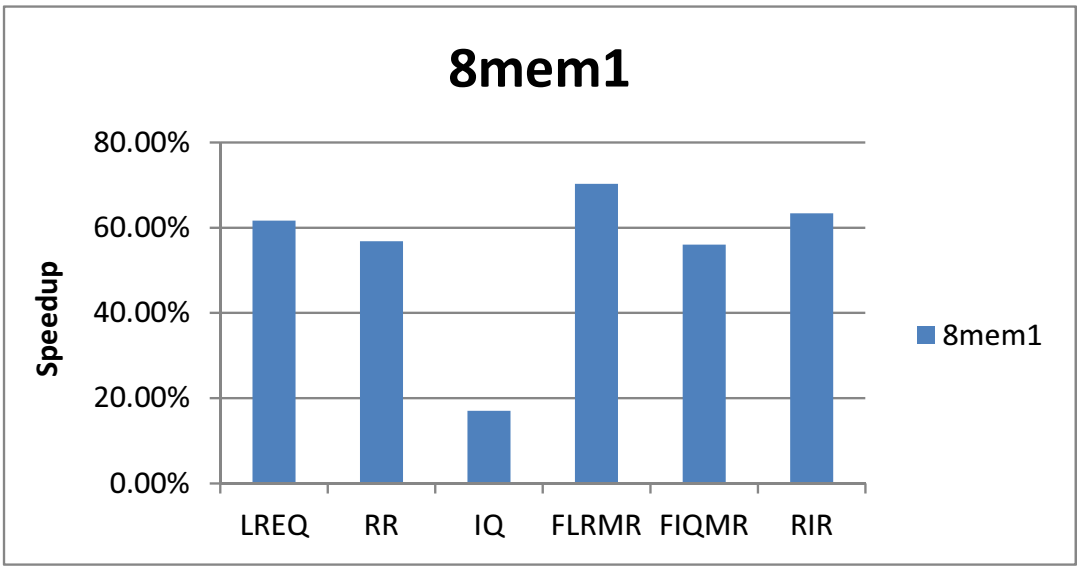


Figure 5.13: 8mem1 workload results

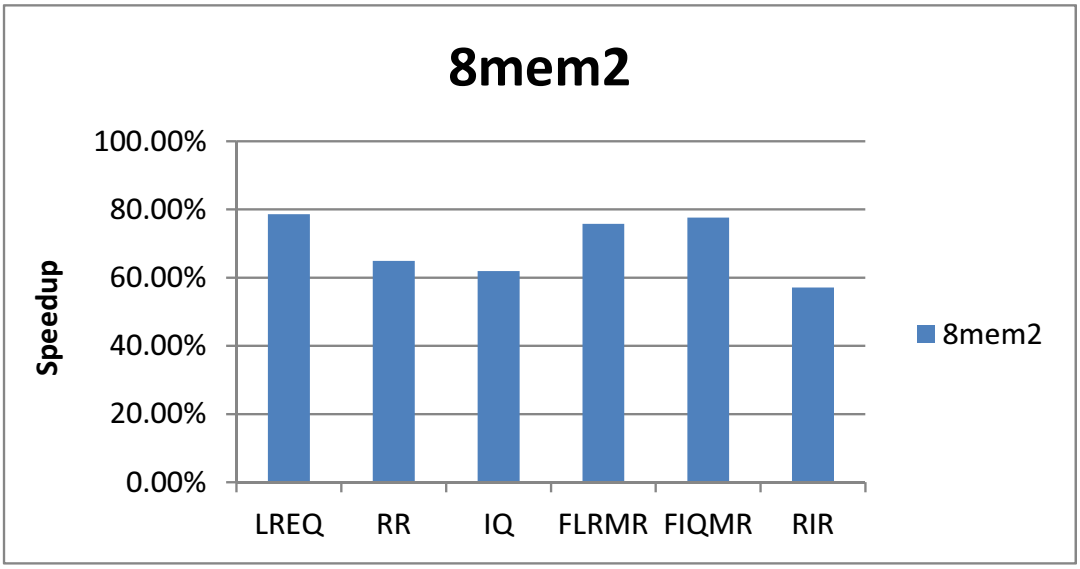


Figure 5.14: 8mem2 workload results

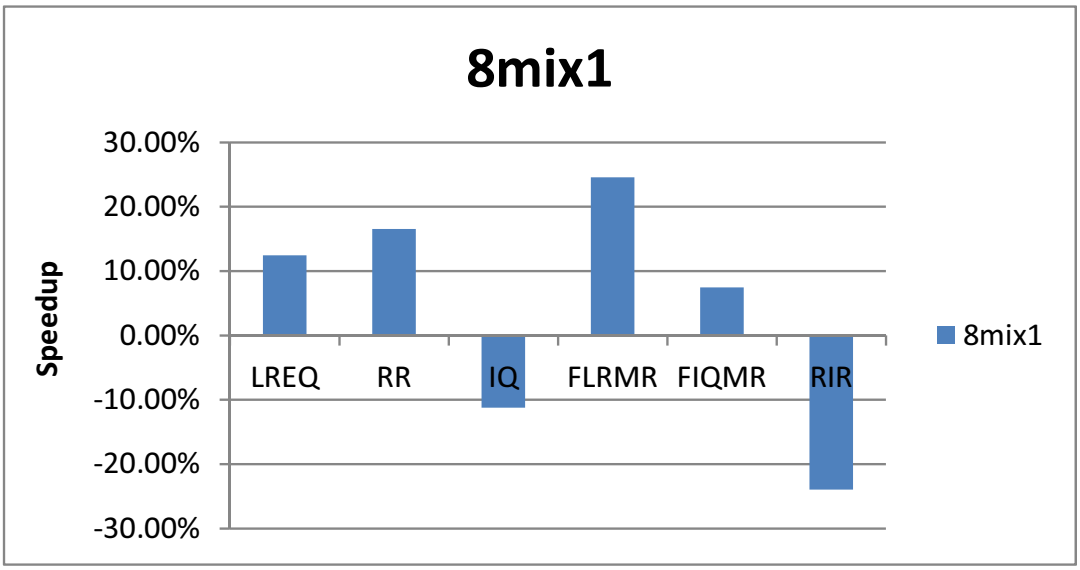


Figure 5.15: 8mix1 workload results

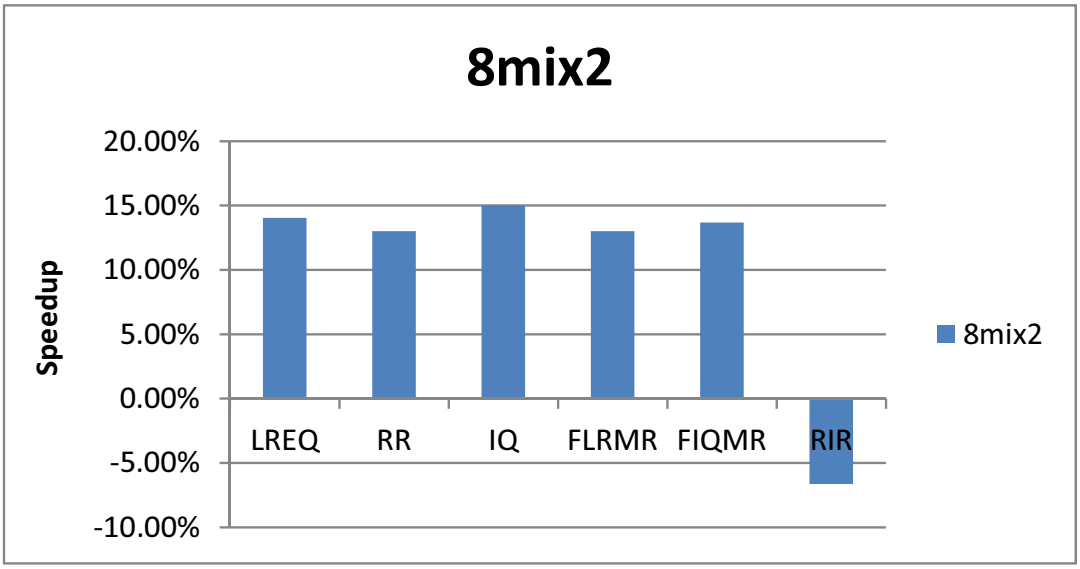


Figure 5.16: 8mix2 workload results

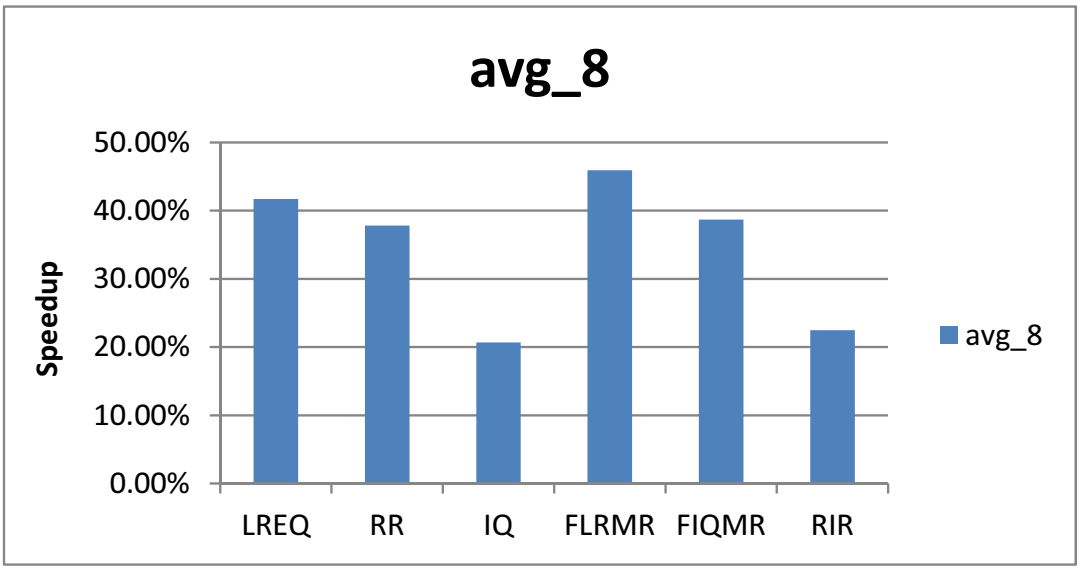


Figure 5.17: Average results of 8-cores workloads

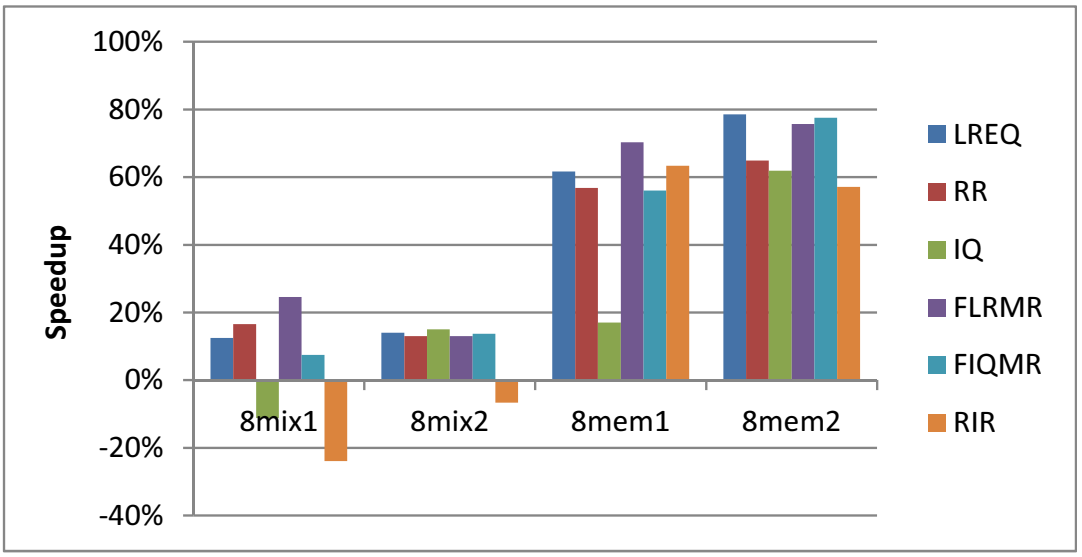


Figure 5.18: 8-cores workloads results

Figure 5.18 shows a comparison between the percentage of speedup gained by running memory access scheduling algorithms on different 8-cores workloads.

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 5.19 shows the average percentage of speedup gained by running memory access scheduling algorithms on all workloads.

Average results of executing workloads show that FLRMR improves the throughput 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 throughput 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 from the throughput prospective. RR comes next with a difference of about 3% on average. The differences in throughput 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

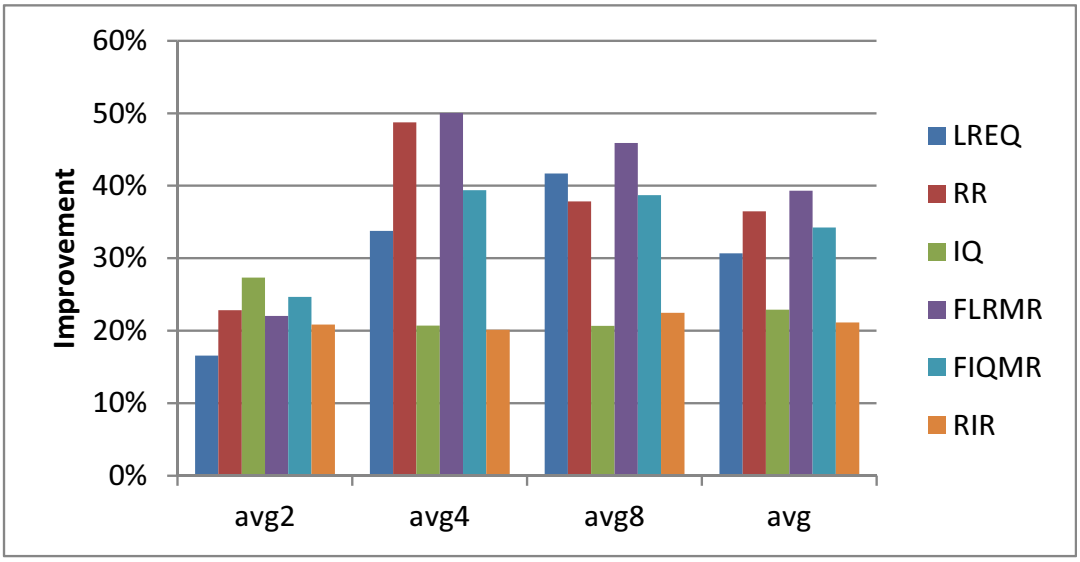


Figure 5.19: Average results of all workloads

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.

RIR comes in the last position in performance improvement on average. However, it has an important feature that it results in almost the same improvement (about 20%) in 2, 4, and 8 cores. This steady performance is interesting to be studied on higher number of cores if the other algorithms result in lower performance.

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 2-cores, 4-cores, and 8-cores workloads. We compare the results of LREQ and FLRMR algorithms as shown in figures 5.20, 5.21, and 5.22.

The figures show that the difference in average latency in LREQ and FLRMR increases when the number of cores increases. The difference in average latency in 2-cores workloads is about 30 cycles. In 4-cores workloads, the difference in average latency is about 90 cycles. In 8-cores workloads, the difference in average latency is about 300 cycles. This improves that the enhancments done in LREQ algorithm are getting more effective when the number of cores is increased.

Going into details of the figure 5.21, it 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 5.12. FLRMR reduces the average load latency from 475.4 cycles when scheduling with LREQ to 366.8 cycles.

We, also, compare the results of IQ and FIQMR algorithms as shown in figures 5.24, 5.25, and 5.26. Similarly to the figures of average latency of LREQ and FLRMR, the figures show that the difference in average latency in IQ and FIQMR increases when the number of cores increases. The average latency in 2-cores workloads are almost the same in IQ and FIQMR. In 4-cores workloads, the difference in average latency is about 100 cycles. In 8-cores workloads, the difference in average latency is about 145 cycles. This improves that the enhancements done in IQ-based algorithm are getting more effective when the number of cores is increased.

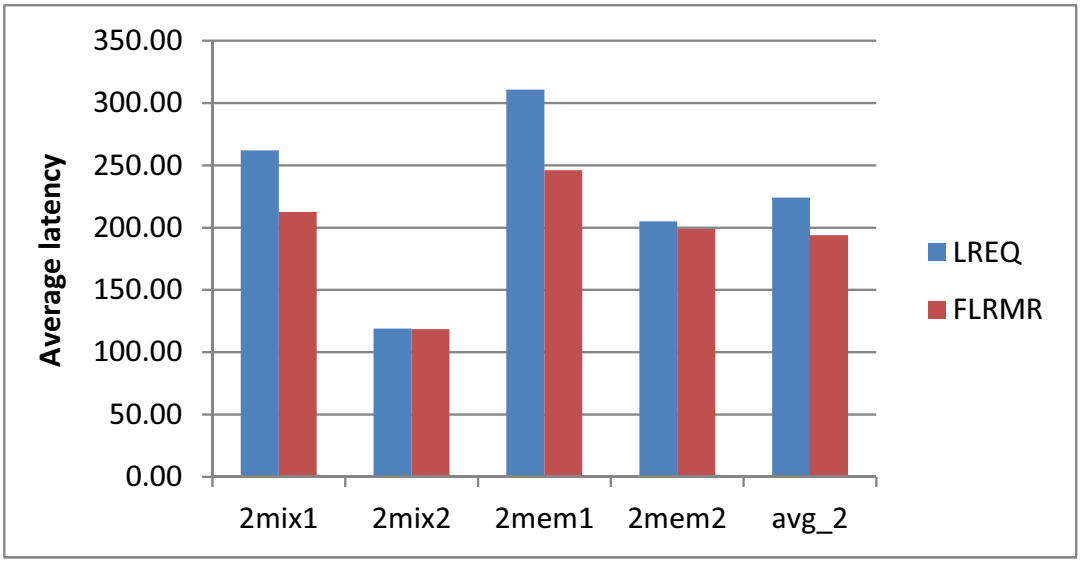


Figure 5.20: Average latency time for LREQ & FLRMR 2-cores workloads (in clock cycles)

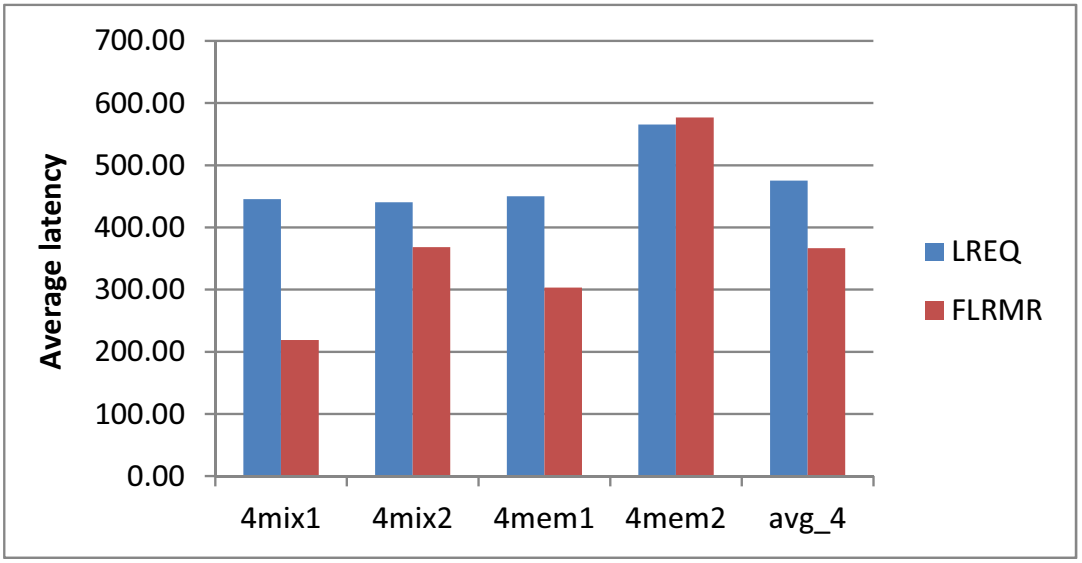


Figure 5.21: Average latency time for LREQ & FLRMR 4-cores workloads (in clock cycles)

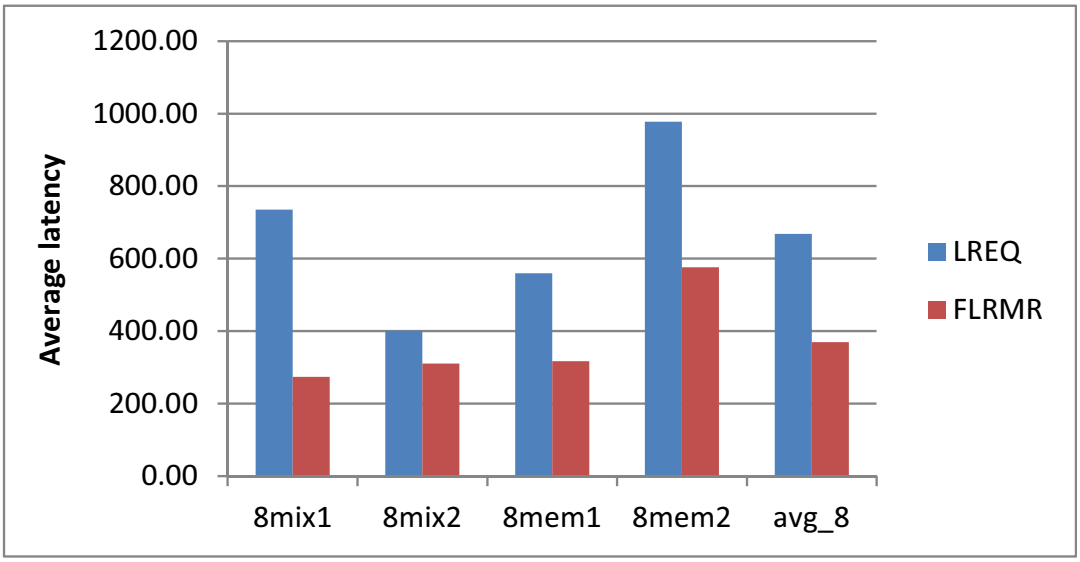


Figure 5.22: Average latency time for LREQ & FLRMR 8-cores workloads (in clock cycles)

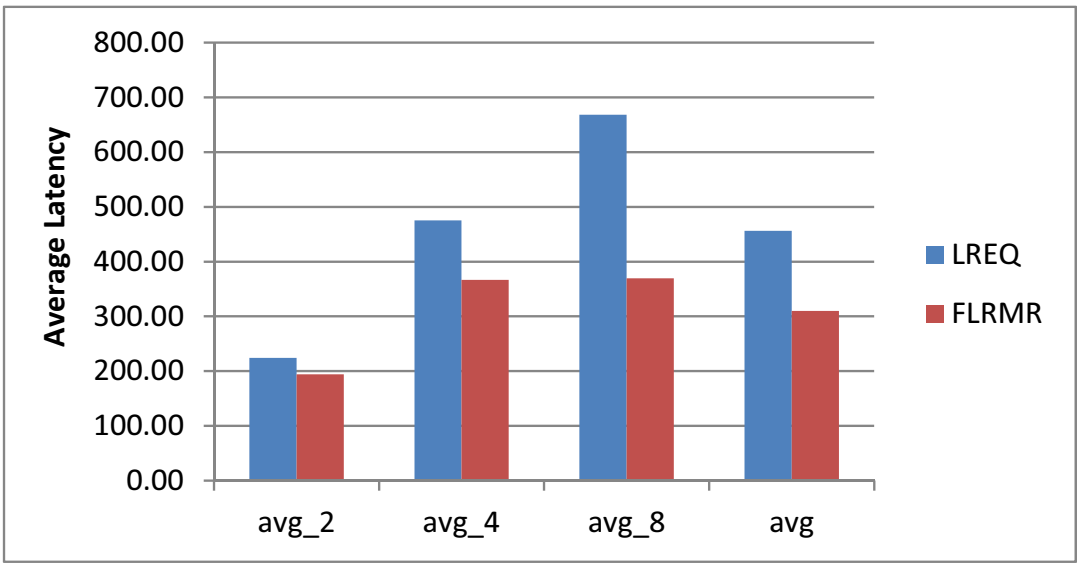


Figure 5.23: Average latency time for LREQ & FLRMR on average (in clock cycles)

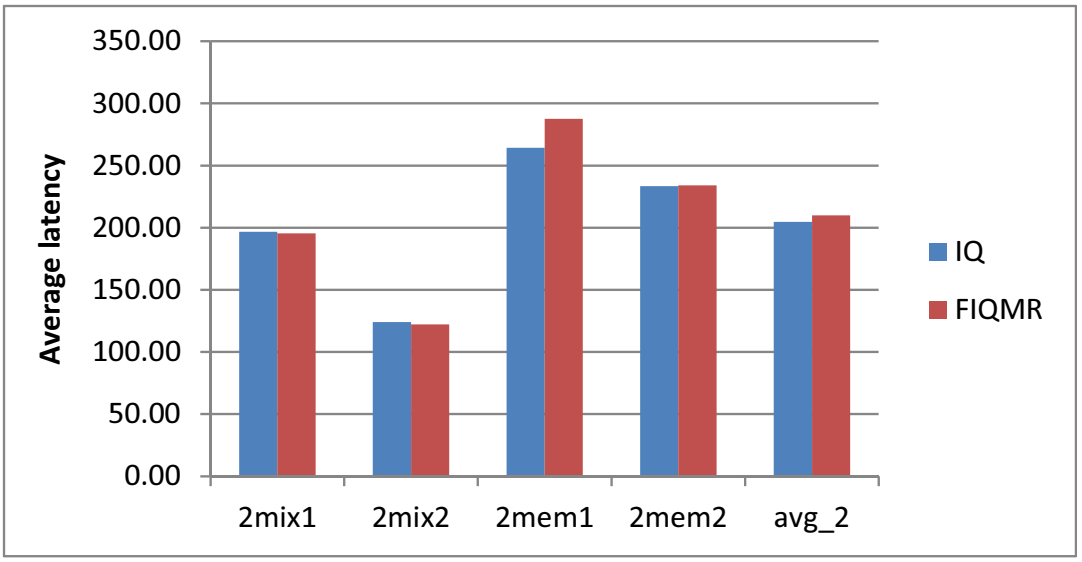


Figure 5.24: Average latency time for IQ & FIQMR 2-cores workloads (in clock cycles)

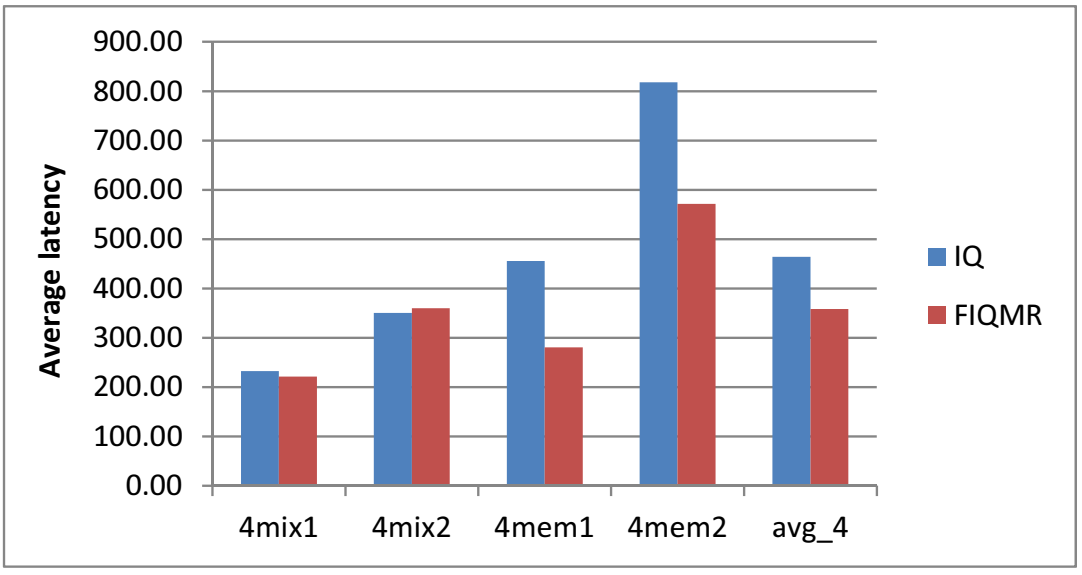


Figure 5.25: Average latency time for IQ & FIQMR 4-cores workloads (in clock cycles)

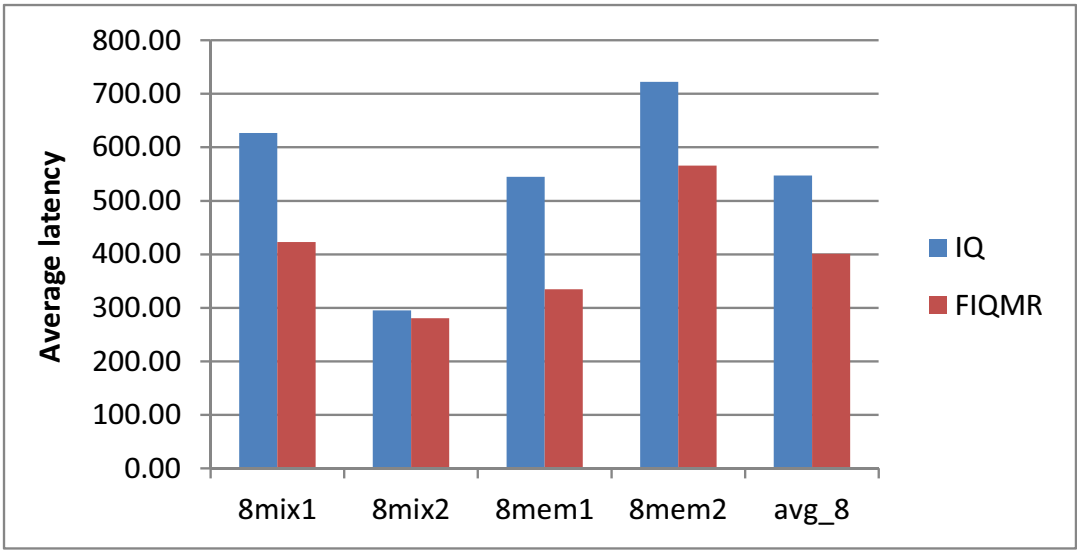


Figure 5.26: Average latency time for IQ & FIQMR 8-cores workloads (in clock cycles)

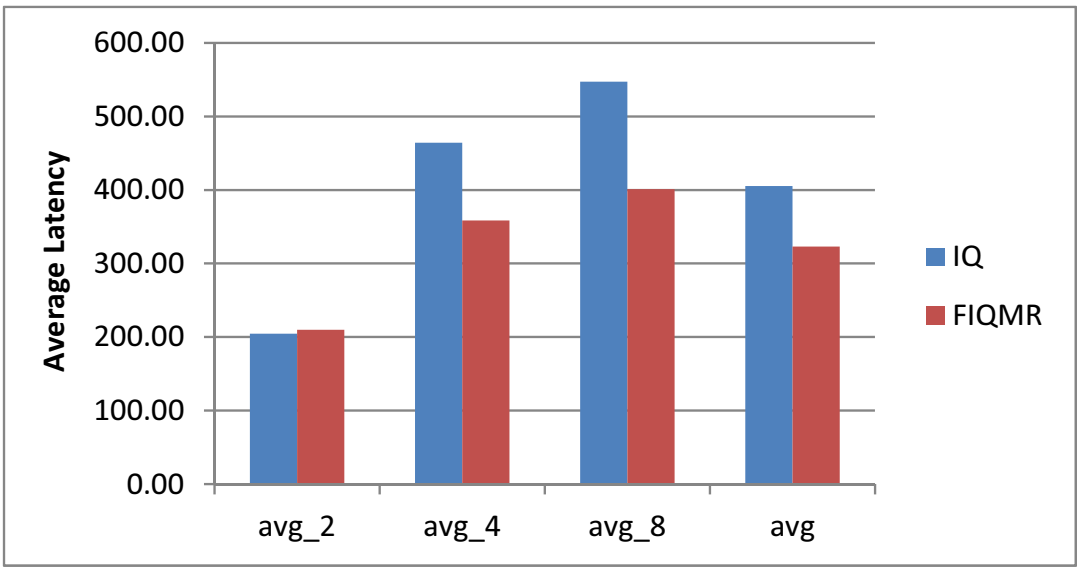


Figure 5.27: Average latency time for IQ & FIQMR on average (in clock cycles)

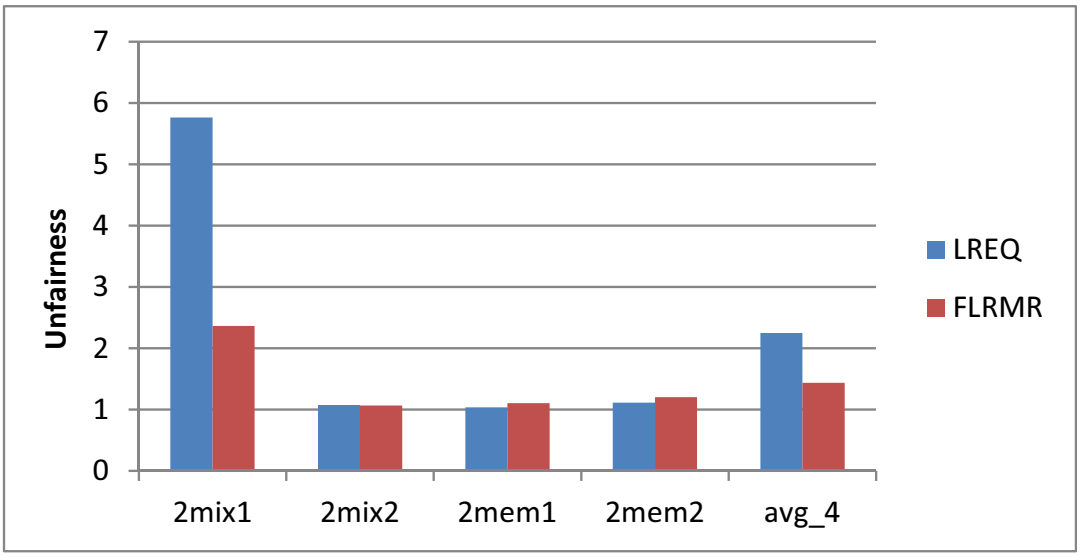


Figure 5.28: Unfairness in LREQ & FLRMR 2-cores workloads

5.3 Fairness Analysis

To prove that our proposed algorithms are more fair than the original algorithms LREQ and IQ-based algorithms, we calculate *unfairness* to know how much it is improved by our algorithms. We calculated unfairness as described in subsection 4.4.2.

Figures 5.28, 5.29, and 5.30 show the unfairness in LREQ and FLRMR algorithms. The figures show that FLRMR algorithm is more fair than LREQ algorithm in 2-cores, 4-cores, and 8-cores workloads. FLRMR, on average, improves *fairness* over LREQ by 36.4%, 77.2%, and 43.3% in 2-cores. 4-cores, and 8-cores workloads respectively. Figures 5.29 and 5.30 show that FLRMR is more fair than LREQ for all workloads. In three out of the four 2-cores workloads, LREQ and FLRMR are almost the same regarding *fairness*. This can be explained that in 2-cores workloads there is a small space for unfairness because the scheduler should pick a request to serve from one of two cores only.

Figure 5.31 shows the average unfairness in LREQ and FLRMR in all workloads. It shows that FLRMR improves *fairness* over LREQ by 54.6%.

Figures 5.32, 5.33, and 5.34 show the unfairness in IQ and FIQMR algorithms. The figures show that IQ and FIQMR almost result in the same unfairness in 2-cores workloads. The reason is as mentioned in LREQ and FLRMR case that

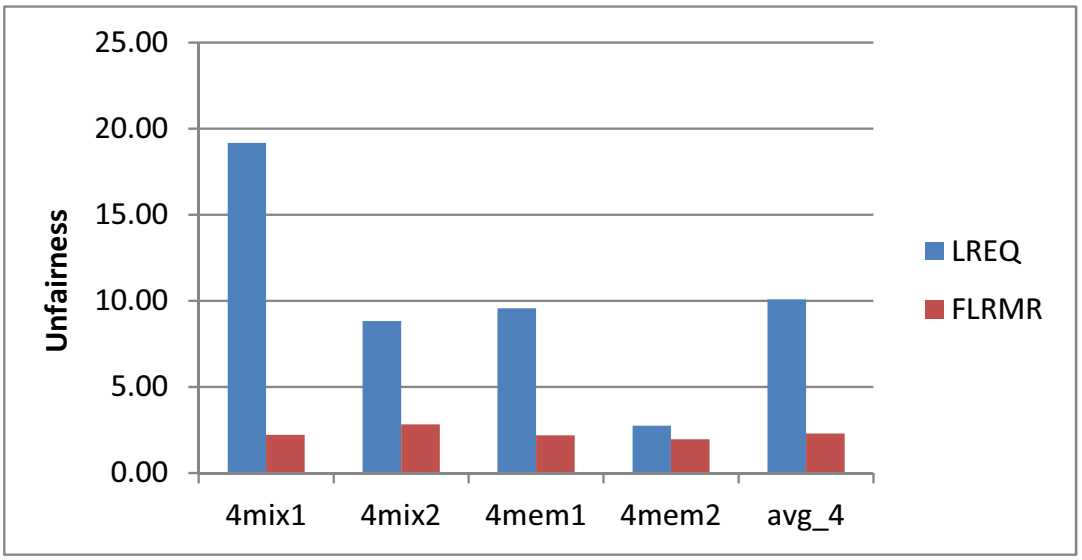


Figure 5.29: Unfairness in LREQ & FLRMR 4-cores workloads

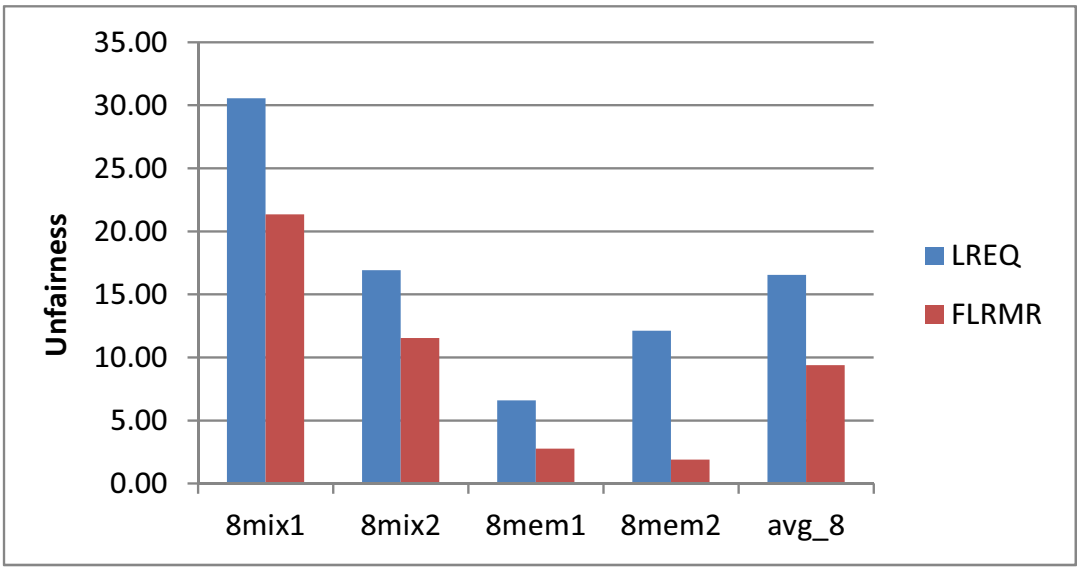


Figure 5.30: Unfairness in LREQ & FLRMR 8-cores workloads

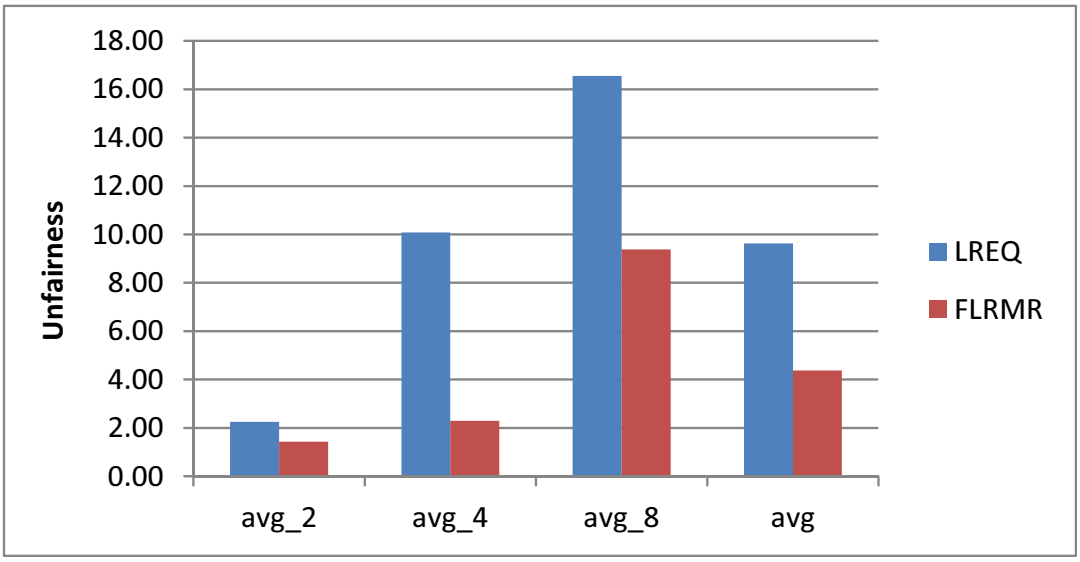


Figure 5.31: Unfairness in LREQ & FLRMR on average

there is a small space for unfairness in 2-cores workloads. In 4-cores workloads, FIQMR improves the *fairness* over IQ-based algorithm by 93.5%. In 8-cores workloads, FIQMR improves *fairness* over IQ-based algorithm by 69.5%.

Figure 5.35 shows the average unfairness in IQ and FIQMR in all workloads. It shows that FIQMR improves *fairness* over IQ by 80.6%.

The previous figures show that the newly proposed algorithms FLRMR and FIQMR, not only improve throughput but also improve fairness. Improving both throughput and fairness is a real challenge, and this challenge is proved to be achieved by the above figures.

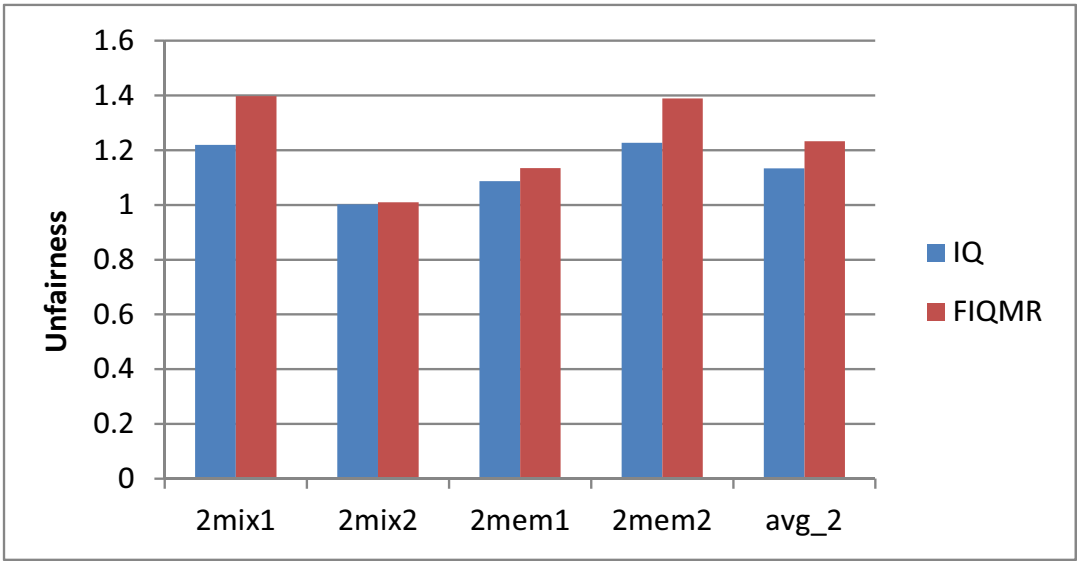


Figure 5.32: Unfairness in IQ & FIQMR 2-cores workloads

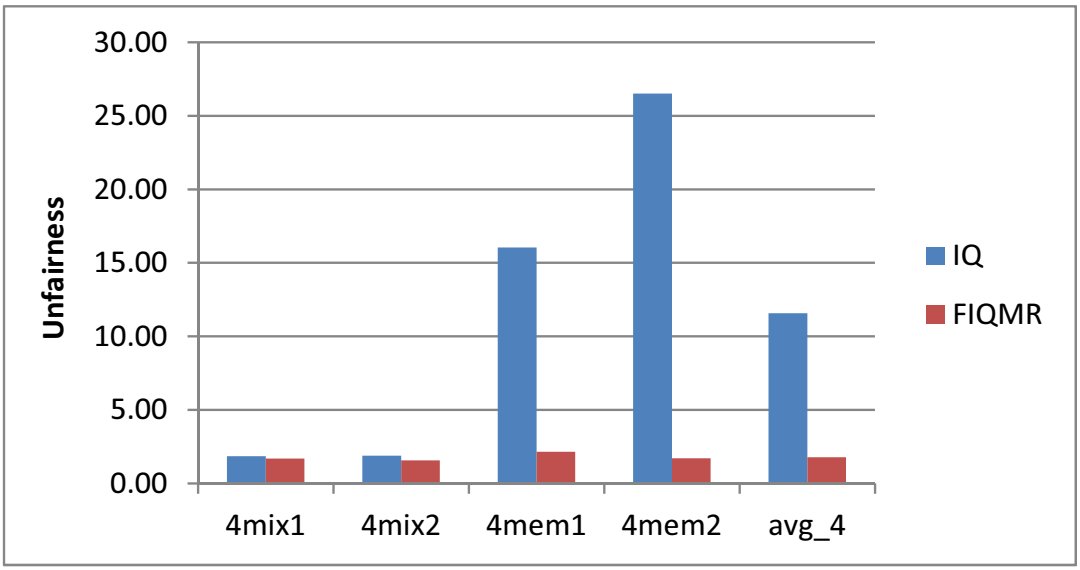


Figure 5.33: Unfairness in IQ & FIQMR 4-cores workloads

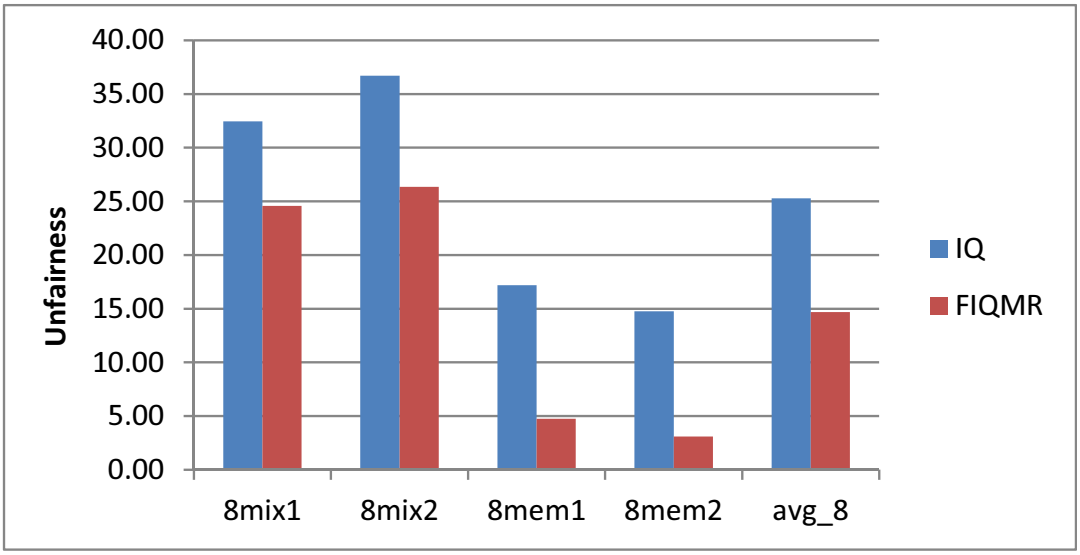


Figure 5.34: Unfairness in IQ & FIQMR 8-cores workloads

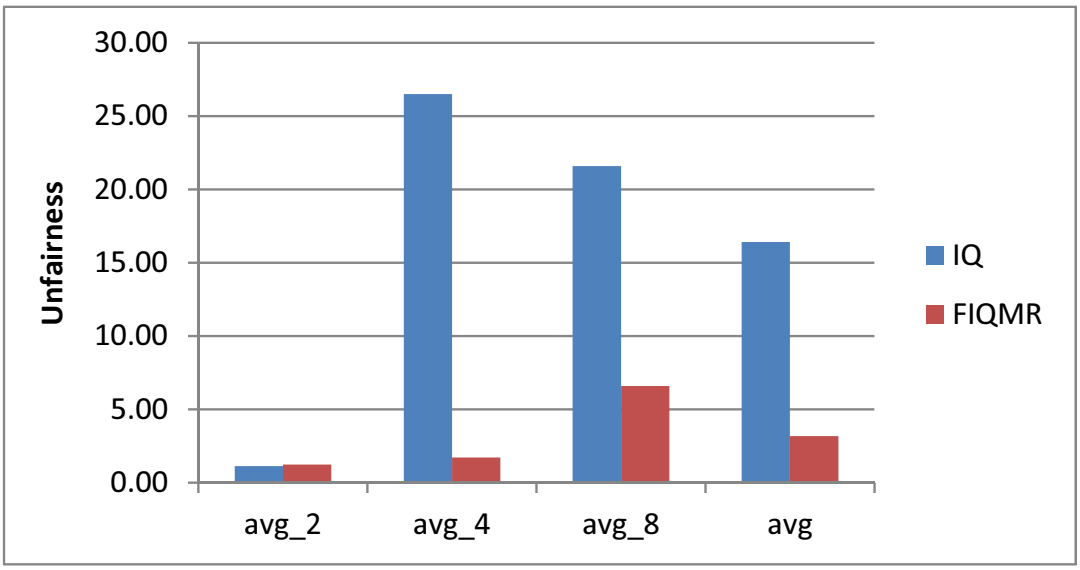


Figure 5.35: Unfairness in IQ & FIQMR on average

Chapter 6

Conclusion and Future Work

In this chapter we summarize the work done in this thesis and conclude. We, also, write our plan for the future work.

6.1 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 results show that FLRMR improves fairness over LREQ by 54.6% on average, and that FIQMR improves fairness over IQ-based algorithm by 80.6% on average. Moreover, the results show that the best throughput comes 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 throughput of LREQ algorithm by 8.64% in addition to improving fairness. FIQMR improved the throughput of IQ-based algorithm by 11.34% in addition to improving fairness as well.

This means that we succeeded in achieving fairness by a very good percentage and throughput by a good percentage too. This proves that the idea of *related requests* and fair scheduling give good results, and it deserves more care in future research in this topic.

6.2 Future Work

There are several ways to improve our work in terms of performance, expanding its usage and applicability, and reliability of the results. Here is a brief some of these aspects:

6.2.1 Enhancing Performance

Performance can be improved by making FLRMR and FIQMR algorithms depend on all *related instructions* not only *related requests*. The *related requests* of thread i, as described in chapter 3, are the total number of memory requests from that thread that demand blocks existing in the memory requests pending from that thread. However, the *related instructions* of thread i are all instructions depending on blocks existing in the memory requests pending from that thread. We think that this move from *related requests* to *related instructions* can improve the overall performance because all the instructions waiting for memory requests to be served affect the performance and should be taken into consideration.

6.2.2 Expanding Applicability

Modifying the algorithms to include the effect of adding different memory banks, and memory controllers will make the algorithms applicable for more memory systems. Also, adapting the algorithms to support parallel applications will be a good step towards expanding applicability.

6.2.3 Improving Trustability

It is planned to test the proposed algorithms using more workloads to make the results more trustable. We, also, intend to test them on real platforms to improve the trustability of these algorithms.

Bibliography

[1] D. W. Wall, "Limits of instruction-level parallelism," *Digital Western Research Laboratory, WRL Research Report 93/6*, Nov 1993.

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

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach, Fourth Edition*. 2006.

[4] J. S. Emer, "Simultaneous multithreading: Multiplying alpha performance," *Proceedings of Microprocessor Forum*, October 1999.

[5] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *The 22nd Annual International Symposium on Computer Architecture*, June 1995.

[6] M. J. Flynn, "Very high-speed computing systems," in *Proceedings of the IEEE, vol. 54*, pp. 1901 –1909, Dec 1966.

[7] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *The 27th Annual International Symposium on Computer Architecture*, pp. 128 –138, 2000.

[8] S. A. McKee and W. A.Wulf, "Access ordering and memory-conscious cache utilization," in *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pp. 253 –262, Jan 1995.

[9] 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 *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pp. 80 –89, Jan 1999.

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

[11] 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 Sixth International Symposium on High-Performance Computer Architecture*, pp. 39 –48, Jan 2000.

[12] M. Jahre and L. Natvig, "A high performance adaptive miss handling architecture for chip multiprocessors," in *Transactions on High-Performance Embedded Architectures and Compilers*, 2009.

[13] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," *The 26th Annual International Symposium on Computer Architecture*, May 1999.

[14] A. El-Moursy and D. Albonesi, "Front-end policies for improved issue efficiency in smt processors," *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, February 2003.

[15] E. F. A. R. F. J. Cazorla and M. Valero, "Optimizing long-latency-load-aware fetch policies for smt processors," in *International Journal of High Performance Computing and Networking*, pp. 45–54, 2004.

[16] S. G. Abraham and R. Rau, "Predicating load latencies using cache profiling," in *Technical Report HPL-94-110 Hewlett Packard Company*, November 1994.

[17] C.-K. Luk and T. C. Mowry, "Predicting data cache misses in non-numeric applications through correlation profiling," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[18] K. Malkowski, G. Link, P. Raghavan, and M. J. Irwin, "Load miss prediction - exploiting power performance trade-offs," *Proceedings of the 21st IEEE International Parallel and Distributed Symposium*, 2007.

[19] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for smt processors," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 213 –224, 2005.

[20] A. El-Moursy, R. Garg, and D. H. Albonesi, "Compatible phase co-scheduling on a cmp of multi-threaded processors," in *International Parallel and Distributed Processing Symposium*, 2006.

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

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

[23] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," *The 35th Annual International Symposium on Computer Architecture*, 2008.

[24] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," *The 16th International Symposium on High-Performance Computer Architecture*, 2010.

[25] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," *The 43rd Annual International Symposium on Computer Architecture*, 2010.

[26] S. Muralidhara, L. Subramanian, and O. Mutlu, "Reducing memory interference in multicore systems via application-aware memory channel parti-

tioning," *Proceedings of the 44th International Symposium on Microarchitecture*, 2011.

[27] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *The 8th Annual International Symposium on Computer Architecture*, May 1981.

[28] K. I. Farkas and N. P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads," in *The 21st Annual International Symposium on Computer Architecture*, April 1994.

[29] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *Proceedings of the 39th International Symposium on Microarchitecture*, pp. 409–422, 2006.

[30] http://www.spec.org/cpu2000/.

[31] S. Sair and M. Charney, "Memory behavior of the spec2000 benchmark suite," *Technical Report, IBM*, 2000.

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

[33] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and throughput in switch on event multithreading," in *Proceedings of the 39th International Symposium on Microarchitecture*, pp. 149–160, Dec. 2006.

[34] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," *The 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[35] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," *Proceedings of the 44th International Symposium on Microarchitecture*, 2011.