CMP Off-chip Bandwidth Scheduling Guided by Instruction Criticality

Pablo Prieto
University of Cantabria
Santander, Spain
prietop@unican.es

Valentin Puente
University of Cantabria
Santander, Spain
vpuente@unican.es

Jose Angel Gregorio
University of Cantabria
Santander, Spain
monaster@unican.es

ABSTRACT

This paper explores the benefits of scheduling off-chip memory operations in a Chip Multiprocessor (CMP) according to their execution relevance. Assuming the scenario of having many out-of-order execution cores in the CMP, from the processor perspective, the importance of the instruction that triggers an access to off-chip memory may vary considerably. Consequently, it makes sense to consider this point of view at the memory controller level to reorder outgoing memory accesses. After exploring different processor-centric sorting criteria, we reach the conclusion that the most simple and useful metric for scheduling a memory operation is the position in the reorder buffer of the instruction that triggers the on-chip miss. We propose a simple memory controller scheduling policy that employs this information as its main parameter. This proposal significantly improves system responsiveness, both in terms of throughput and fairness. The idea is analyzed through full-system simulation, running a broad set of workloads with diverse memory behavior. When it is compared with other scheduling algorithms with similar complexity, throughput can be improved by an average of 10% and fairness enhanced by an average of 15% even in very adverse usage scenarios. Moreover, the idea supports the possibility of dynamically favoring throughput or fairness, according to the end-user requirements.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles – *primary memory, shared memory.* C.1.0 [**Processor Architectures**]: General.

General Terms

Algorithms, Management, Measurement, Performance, Design, Experimentation.

Keywords

Multi-core processor; Off-chip bandwidth wall; out-of-order; memory access scheduling.

1. INTRODUCTION

Moore's law seems to be keeping pace, without insurmountable roadblocks in sight, although further shrinking in feature size will finally be prevented by the laws of physics. In any case, other More-than-Moore technologies such as 3D stacking [4] will allow these problems to be circumvented. Since the mid-2000s, processor designers have had problems to translating device availability into ILP enhancement [26]. Therefore, chip multiprocessors (CMP) have flourished, today being pervasive in most computing fields. The virtuous cycle that in the pre-CMP era allowed performance enhancement to be transparently included in legacy software without any intervention of the final user has been broken. All computer science disciplines [1] are committed to facilitating the adjustment of all the elements of the computer stack to fully exploit the potentials of CMP systems and so guarantee that performance can keep up with device availability growth. Nevertheless, several problems could prevent that evolution. In this paper we will focus on one particularly interesting aspect, namely the off-chip bandwidth wall [3][29]. Packaging issues limit the number of pins available for a chip and clocking. Even under the assumption of a high-performance system, the actual ITRS roadmap [32] states that by 2020 the maximum number of pins will be fewer than 6000, which is less than double today's standard. According to the same report, the operation frequency will grow slightly. Combining both facts, the result is that the off-chip bandwidth available per million transistors will fall exponentially.

Architects and technologists will have to find imaginative solutions to help with this problem; otherwise it will be impossible to keep increasing the number of cores in a single chip. The solutions could be diverse and range from adding massive amounts of on-chip cache, to increasing pin bandwidth using photonics and ultra-dense wavelength multiplexing [25]. Even in these cases, off-chip bandwidth scarcity is foreseeable, and if we combine it with the fact that there will be a huge number of running threads in the CMP, not necessarily cooperating in the same task, it makes sense to pay special attention to how the bandwidth is used. In an extreme case, an intentionally or unintentionally misbehaving task could reduce the whole CMP performance. Therefore, bandwidth partitioning [18] will be paramount and this is where this paper is focused.

Although the work dealing with this problem is profuse, we have used a novel approach to deal with it. In particular we focus our interest on the processor side. Although there are CMPs based on simple cores, general purpose computing demands the utilization of cores capable of exploiting instruction level parallelism [12]. This will require, in one way or another, the use of cores with aggressive out-of-order execution [11]. This work analyzes how to effectively exploit the different nature of memory operations in this class of systems in order to optimize bandwidth utilization. We will explore and demonstrate the possible improvement in system behavior if memory operation scheduling is done taking into account the relevance of the instruction that triggers the operation. In particular we use the distance from such an instruction to the head of the Reorder Buffer (ROB). The proposed scheduling algorithm uses this distance when the memory operation leaves the core and the off-chip miss-frequency behavior of each core to determine the sorting and aging criteria. The proposal (called DROB) outperforms similar scheduling algorithms both in terms of fairness and throughput. Additionally, the proposed idea has a noteworthy property of allowing the end-user to favor throughput or fairness according to the usage scenario simply by modifying a single parameter. Finally, we demonstrate that it is possible to use a simple mechanism to self-tune the memory controller in order to achieve the most balanced throughput-fairness behavior.

The paper is organized as follows: Section 2 presents the baseline organization for the memory controller and describes related work. Section 3 discusses how the processor viewpoint may be relevant as an ordering criterion and which metrics can be useful. Section 4 introduces a suitable implementation of a memory controller that can exploit this knowledge. Section 5 describes the methodology used to evaluate how effective our approach is. Section 6 reports the performance with different workloads and compares it with other similar scheduling approaches.

2. BASELINE ORGANIZATION AND RELATED WORK

Figure 1 shows the baseline system assumed. The system is composed of a set of N out-of-order execution cores, each one with one or more coherent private levels of cache, a shared write-back last level cache and M memory controllers, managing R ranks made up of B banks. When a memory access instruction, i.e. prefetch, fetch, store, load, atomic or requested block is not in the chip, depending on the memory interleaving, the corresponding memory controller should deal with the request applying the priority ordering used to each incoming request and issuing it to the corresponding DRAM bank. Outstanding accesses to the same memory block from the same core will coalesce in the MSHR of the Local cache level [16]. Outstanding memory requests from different cores will coalesce in the Last level cache MSHR. Write memory operations will be associated with write-back events in the last level cache when an on-chip miss evicts a dirty block.

The memory controller is responsible for applying the priority to the incoming memory transaction and enqueueing it, depending on the address translation, in the corresponding bank queue. From the bank queues, the memory access scheduler logic will chose one transaction and issue it to the appropriate bank. As a starting point, we will assume that this scheduler logic is FR-FCFS [28]. Although other more advanced policies, depending on the DRAM physical implementation, signaling interface, etc. could be used [9][10][22], in most cases this could be considered complementary to this work.

Our proposal, like many others, works at the arbiter level. We compose a policy that prioritizes the incoming memory transaction, based on different sorting criteria. The scheduling algorithm modifies the First-Come-First-Served policy of FR-FCFS, at each bank queue depending on the policy used. According to the priority determined by the arbiter we will insert the associated DRAM operations in the appropriate position of the corresponding queue and will leave the CMP in that order. Given that off-chip bandwidth limitation is a topic that it is attracting significant attention, there are many recent high quality works focused on memory scheduling [5][9][10][15][22][28]. In most of these works, the key point is to infer what the processor is doing in the memory controller and act accordingly with the scheduling decision, coordinating the decision across banks [15], [18]. In some cases the mechanism involved in this decision requires complex algorithms [14], [22] and/or modifications throughout the software stack or limits the flexibility of use of the CMP [21].

Intuitively, the underlying approach in many proposals [8], [20] is to limit the interference of bandwidth-demanding applications on bandwidth non-demanding ones. The objective is, with little performance penalty, to limit the bandwidth consumption of the former applications so as to significantly improve the latter. In a way our approach is similar to this, although it also takes into account the criticality of the instructions at scheduling time. For an out-of-order processor, the criticality of an instruction represents how large the performance penalty of delaying it can be.

3. MEMORY OPERATIONS FROM THE PROCESSOR PERSPECTIVE

There is a significant amount of work focused on managing off-chip bandwidth scarcity from the perspective of the DRAM itself. It makes sense to exploit the technological peculiarities of these systems while not precluding the combined utilization of other techniques focused on the opposite end of the problem: the processor. Although there are many works that introduce mechanisms to deduce the processor responsiveness [15][18][21]-[24] at the memory controller, we opt to directly look at what the processor is doing. Not all memory operations are equally important in an out-of-order execution core, therefore it seems pertinent to use their degree of relevance to carry out the

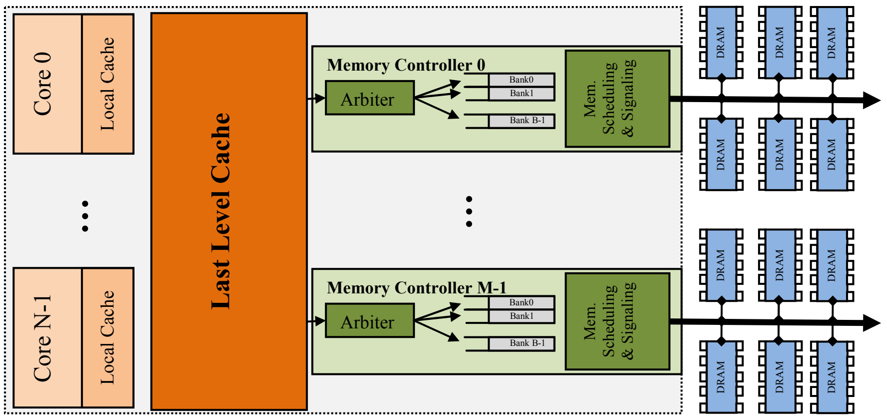


Figure 1. Baseline System

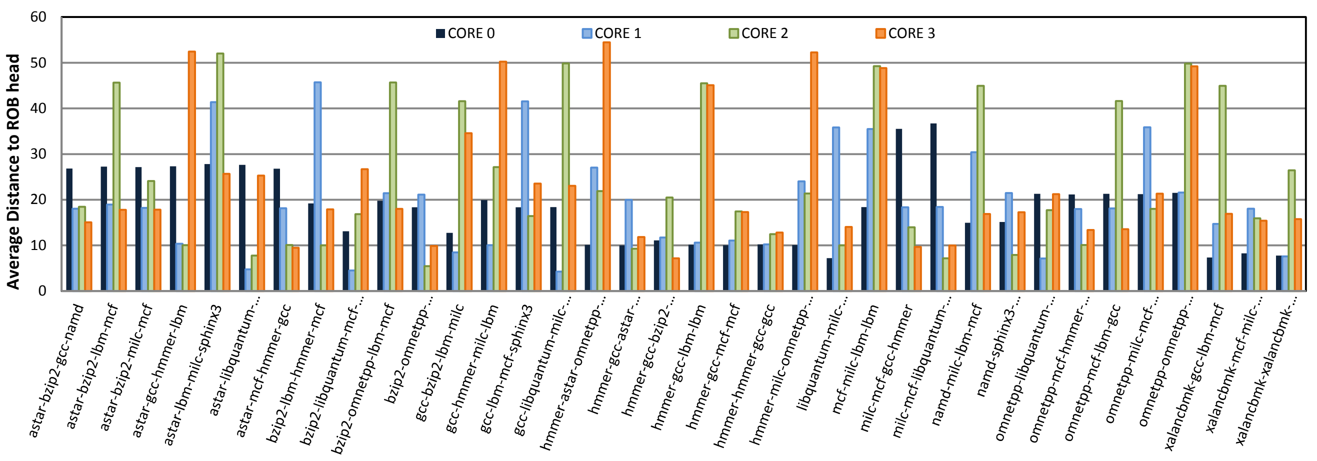


Figure 2 Average instruction distance to head of the Reorder Buffer when the triggered memory operation reaches the memory controller.

prioritization of the memory accesses. At a given time, in usual memory interleaving approaches, each memory controller may deal with memory operations coming from any of the cores in the system. These operations should be sorted and sent to the corresponding DRAM bank in order to be fulfilled. The sorting may have a non-negligible impact on performance, fairness and energy. We will explore the use of different criteria from the processor perspective. We will focus on operations that are in the critical path of the cores, i.e. memory reads. Note that memory write commands are always the result of replacements of dirty blocks in the on-chip cache. Like in systems such as Power5, write operations always have the lowest priority [13].

Although there are recent works, such as [31], focused on co-optimizing the replacement algorithm and the memory controller in order to minimize read-after-write waiting cycles, we do not consider this problem. This type of solutions can be complementary to the analysis carried out in this work. In our case memory write operations will be inserted with the lowest priority in the bank queue.

3.1 Optimize Retire Bandwidth

Retire bandwidth underutilization has a large effect on performance because it directly causes an IPC loss. The most common sources of retire underutilization are a slow operation blocked at the head of the ROB or frequent rollbacks due to miss-speculations. In current system configurations the most likely cause of a slow operation is an on-chip miss. Therefore, when a memory operation arrives at the memory controller, its criticality is higher when the distance to the head of the ROB is smaller. We carried out a simple experiment for thirty-six particular mixes of multiprogrammed workloads using SPEC2006 benchmarks (using the framework and system configuration described in Section 6.1). When an on-chip miss reaches the memory controller we determine the average number of instructions between the triggering instruction and the head of the ROB. As can be seen in Figure 2 the variability is large across all the running threads. For example in the *hmmer-gcc-lbm-lbm* case, memory operations from Core 0 (*hmmer*) and Core 1 (*gcc*)[1] seem to be much more critical than memory operations in other cores. If we can use that information judiciously to schedule memory operations, we can advance the resolution of more important memory operations by some cycles with little effect in other cores. In this example, at a given time, if we can somehow determine the situation in ROBs, it could be useful to favor *gcc* and *hmmer* memory operations over *lbm*. Note that these numbers are oversimplified because the variability is constant during the execution of the benchmark.

3.2 Optimize Fetch Bandwidth

An on-chip miss in the processor front-end, i.e. instruction fetch, has a critical impact on an out-of-order execution processor. In such situations, given that this operation is performed in order, the processor pipeline will stall. Although this type of misses is not very frequent in many applications, in others, such as commercial workloads, it might be quite probable. This behavior is due to the large code footprint of this type of applications [2]. Therefore, in a multiprogrammed environment, the number of memory operations triggered by a fetch from each core could be quite different depending on the characteristics of each thread. If the memory controller assigns the maximum priority to these operations, it could be possible to accelerate miss resolution and consequently unclog pipeline access faster. This would have a notable impact on performance not only when an unbalance in core fetches is observed but even for each core. Enabling some loads already enqueued in the bank queue from the same core to be overtaken would be beneficial, especially if these loads are far from the ROB head. In few applications, we have observed significant improvements in performance when accelerating fetches over other memory requests.

3.3 Degree of Speculation

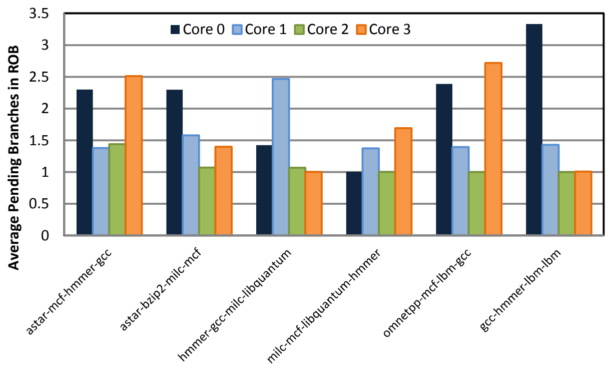
When an on-chip miss reaches the memory controller, the triggering instruction/s can be speculatively issued. Let's denote the degree of speculation of an instruction as the number of operations issued without all the parameters being known, i.e. the number of unresolved branches, number of executed loads with ambiguous stores, etc. between it and the head of the ROB. Intuitively, not all instructions will be equally relevant from the point of view of the memory operation. If the triggering instruction of an on-chip cache miss has a higher degree of speculation than another triggering instruction of another pending memory operation at the memory controller, it could make sense to use this information somehow to order the priority of the requests to memory. Following a similar procedure to the one described previously, we show the number of unresolved branches

[1] We use the Solaris tool *processor_bind* to fix threads to cores.

for six different mixes of SPEC2006 applications. As we can see in Figure 3(a), for a 128-entry ROB, the average number of unresolved branches from application to application is quite similar, ranging from 1 to 3. This seems to indicate that this criterion might not be useful when ordering memory operations.

Nevertheless, this information is partial because the likelihood of being annulled is also dependent on the branch prediction accuracy. In particular, this could be very important if the thread executed by some cores has a higher chance of miss-speculation. For example, if we mix integer applications with floating point applications, the chances of miss-speculation are much higher in the former type of applications than in the latter due to branch prediction miss-speculations. To provide a better perspective, in Figure 3 (b) we show the probability of the instruction arriving at the memory controller being rolled-back. Where, *M* is the branch miss-prediction rate, and *N* the number of speculated branches, the probability of rolling back is calculated as:

$$P_{RB} = 1 - (1 - M)^N \quad (1)$$



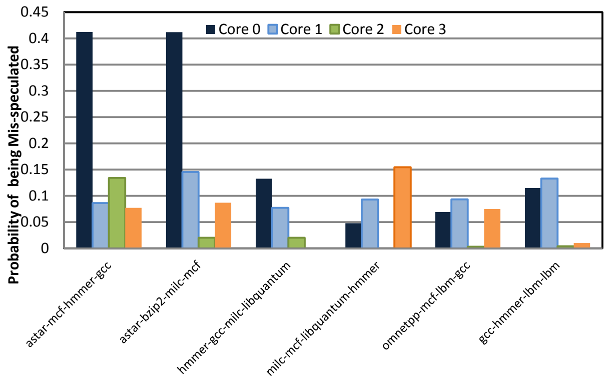


Figure 3 (a) Degree of speculation of triggering instructions at memory controller, (b) Probability of the triggering instruction being rolled-back later.

The result is that most of the benchmarks are fairly similar, the presence of some pathological behavior being noticeable only in *astar,* where almost half of memory accesses correspond to miss-speculated instructions. Therefore, this metric might not be so relevant as the previous one when sorting memory operations but it can be useful to discard pathological behavior such as *astar*'s. Anyway, the results obtained show that the complexity involved is not cost effective bearing in mind the performance benefits. Consequently this approach will not be considered in our ordering criteria.

Although other types of speculation can be considered, such as opportunistic load execution, with a simple dependency predictor, the number of miss-speculations is not relevant in comparison with branch-caused miss-speculation. Other speculative operations like "data value prediction" have not been analyzed.

3.4 Dependent instructions

3.4.1 Dependency Graph Depth

Looking backwards in the ROB could also provide hints about the relevance of the memory operation depending on the number of dependent instructions. Intuitively, a pending instruction with more dependent instructions waiting to be issued in the ROB should be considered more important than another one with fewer. The question that arises is whether on average it could be possible to observe a significant difference from core to core. We ran the same experiment again looking in the ROB for the number of dependent instructions (both direct and indirectly) on the output register of the instruction that generates the miss. Figure 4 shows how many dependent instructions are in the ROB when the triggering instruction of the memory operation reaches the memory controller. Any direct or indirect dependence of the instruction in the ROB is taken into account. As can be seen, in spite of having up to 128 in-flight instructions, the difference between each thread is not sufficient for clear discrimination. Moreover, in most cases there is a clear correlation between the distance of the triggering operation from the head of the ROB and the number of dependent instructions. Intuitively, if we assume a similar degree of usefulness for the block fetched from memory, then the shorter the distance from the head of the ROB, the longer the distance to the tail and consequently the more dependent instructions could be found. A classification in terms of this metric and distance to the head of the ROB, therefore, seems to be redundant and will not be used in the scheduling algorithm.

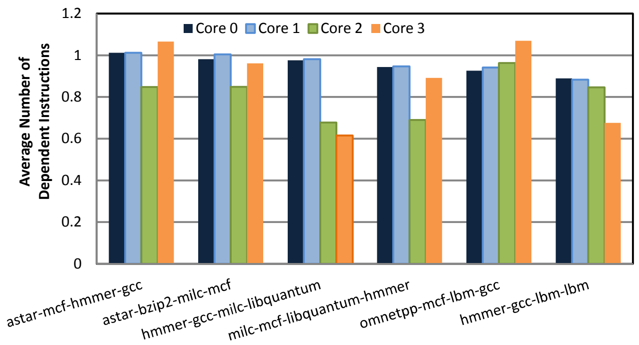


Figure 4. Average Number of dependent instructions at the ROB when the triggering operation reaches the memory controller.

3.4.2 Store Operations

In the previous analysis, the nature of each memory operation was not addressed. For the stores, technically the processor does not require the outcome of the execution of instructions to progress forward, i.e. there are no truly dependent instructions in the associated on-chip miss. Forwarding from the Store Queue or Write Buffer will provide the required value for any subsequent load to the same word. Therefore, at the memory controller, classification of memory operations according to the nature of the triggering instruction might be considered. Although it could make sense to prioritize the memory read when the instruction is a load [9], this decision should be carefully considered. Loads to a particular block can be overlapped with earlier stores misses, which will be pending in MSHR [16]. Under this situation, storing triggered memory operations will impact on performance. Next, we will show how frequently this situation could happen. We choose, as an example, nine applications of SPEC2006. Figure 5 shows the average number of coalescing loads in the MSHRs to the same memory block where a previous store has missed. The value is computed when the memory operation finalizes. The bars indicate the maximum and minimum number of coalescing loads-under-store miss observed when the application is executed concurrently in different mixes of applications. As we can see, from application to application there is a large disparity in the coalescent loads compared to stores. For *mcf* or *hmmer*, the average number of loads to the same block requested by a store is relatively large. In contrast, in other applications such as *lbm* the likelihood is negligible. The dissimilar spatial locality of each application determines this behavior. Moreover, when each application is running concurrently with three other applications, the variability is sometimes noticeable. Therefore, a careful analysis is necessary before delaying store operations at the memory controller, otherwise a large number of coalescing critical loads might be delayed. Nevertheless, this information is not easily accessible since, at scheduling time, even with idealized MSHR knowledge in memory controllers, most coalescing loads will still be pending. In practice, besides its complexity, this approach in most cases tends not to provide noticeable benefits. It is not unusual to observe substantial performance degradation, especially in applications such as *mcf* and *hmmer*.

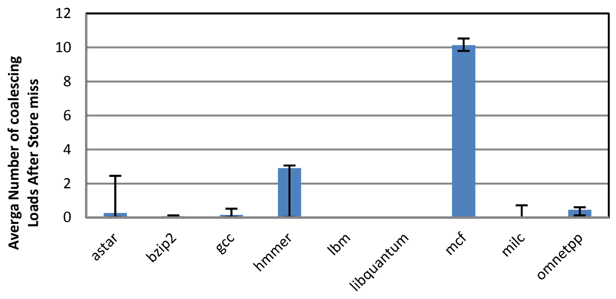


Figure 5. Average Number of coalescing Loads in the same cache block as a Store miss.

4. REALISTIC IMPLEMENTATION OF A CORE-CRITICALLY-BASED MEMORY SCHEDULER

The previous section discusses potential scheduling criteria assuming that the memory controller has full knowledge of the processor logic when the memory operation reaches it. This is unfeasible without impairing CMP scalability. Therefore, the information needed to reorder memory operations has to be generated when the triggering instruction leaves the core, i.e. it is executed if it is a load or committed if it is a store. We can embed the required information in requests and piggyback it in each successive on-cache miss until reaching the memory controller. However, when the memory operation reaches the controller, the processor status might have changed. We should readjust this information using local metrics.

4.1 Distance to ROB Head

As explained before, it is possible to send the distance of the instruction to the ROB head attached to the memory request it triggers, but this information arrives at the memory controller after a number of cycles and might have changed. It would be desirable to estimate its position in the ROB when the on-chip miss reaches the memory controller. Although achieving precision in the absolute value might be hard, a relative value to compare the criticality of the concurrent memory operations would be enough. In general, when a processor generates a memory request, the instruction that triggers it will reach the head of the Reorder Buffer approximately after the number of cycles given by:

$$NumCycles = \frac{DistToRoBHead}{IPC} = DistToRoBHead \cdot CPI \qquad (2)$$

This information should be enough to order requests properly, but it is not easy to continuously provide it to the memory controller. Nevertheless, the memory controller has easy access to the on-chip misses in each core in the memory region that is mapped. From this metric, it is straightforward to derive the miss frequency, which is a good proxy for the CPI [18]. Although the correlation between CPI and miss frequency has been analyzed thoroughly in previous works, we simplify its practical use. We cannot obtain the absolute value for the CPIs of each core, but we can guess the relationship among them from the viewpoint of the ratio of their miss frequency at the memory controller. As it is a ratio, the result is no longer a measure of time, but a sort of corrected distance to the ROB head. As this parameter will only be used as a correction factor for the distance to the ROB head, this approximation is accurate enough to evaluate request criticality.

According to this and using (2), the corrected distance or priority level of a request *i*, with a distance of D_{ROB} to the ROB header at the time the operation leaves the core *p* for a CMP with *N* cores, when it arrives at the memory controller is calculated as:

$$\overline{PrioLevel_i} = D_{ROB} \cdot \frac{Miss\_freq_p}{\max\{Miss\_freq_0..Miss\_freq_N\}} \qquad (3)$$

As the miss frequency is computed independently at each controller, no communication is required between the memory controllers. Miss frequency is calculated every 1M cycles, and to avoid momentary application singularities, miss frequency will be computed using the exponential moving average of each processor at the memory controller. Note that the priority level expressed in (3) is inverted, i.e. a memory request has higher priority when the normalized distance to the head of ROB is smaller.

4.2 Fetches and Writes

As previously discussed, prioritizing fetches over other memory requests might have impact on performance without noticeable cost, so we have implemented this feature. Fetches are considered high priority, and do not use the same formula as other memory reads. Instead, their priority level is automatically set to 0 (the maximum). On the other hand, we know memory writes corresponds to LLC write-backs and are not in the critical path. To keep the scheduling algorithm uniform, for such operations the priority level of the request is set to the size of the Reorder Buffer (the minimum priority).

4.3 Memory scheduling algorithm

Once a request reaches the memory controller, the operation will be inserted into the corresponding bank queue and then sorted according to the priority assigned using (3). When no previous high priority requests are pending in the queue, for example situation (a) depicted in Figure 6, a search for high priority requests is initiated. In the example, we represent the computed priority and the issuing processor in each entry of the queue. The algorithm uses an input parameter denoted *Threshold Distance*. If the priority is below this distance the request is tagged as high priority. If not, the request priority is recomputed decreasing the distance to the ROB header by the *Threshold Distance*, avoiding starvation issues for these requests. In the example presented in Figure 6, the Threshold distance used is 16. Therefore, in situation (b), only the first five requests will be tagged as high priority and the remaining requests are recomputed accordingly. Therefore, the sorting algorithm used is:

```
$For\ (i\ in\ requests)$
    $if\ (PrioLevel_i < ThresholdDistance)$
                    $\rightarrow Request\ i\ Become\ High\ Priority$
    $else \rightarrow PrioLevel_i = PrioLevel_i - ThresholdDistance$
```

When all the high priority commands have left the queue, which corresponds to situation (c) in Figure 6, the search for new high priority requests is initiated. Note that since step (b) occurred, additional memory requests could have arrived at the memory controller, which correspond to the white boxes. After the application of the sorting, seven requests are tagged with high priority.

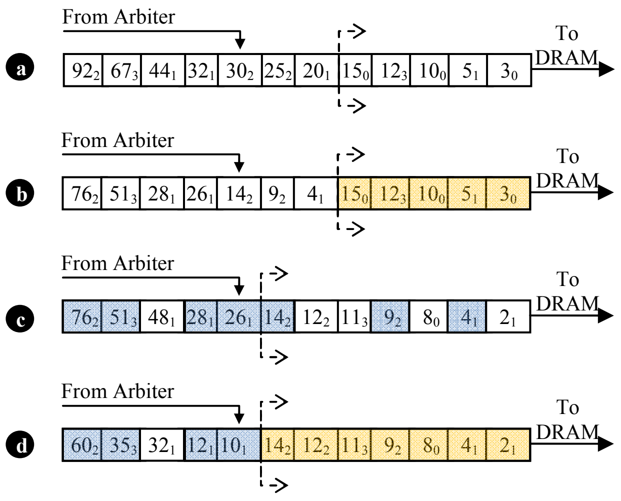


Figure 6 Memory scheduling Example.

In contrast to other batch-like scheduling algorithms, such as PAR-BS [24] the batch length and time between arbitrations is variable and no request for a processor can be tagged with high priority. In the first sorting in the example, no request from processor 2 has been tagged with high priority. Additionally, once the request is tagged for high priority, the processor *id* is no longer used in the scheduling of the memory operations. In PAR-BS the order of pending requests per processor is used to decide how the batch is sent to memory. This is equivalent to prioritizing the processor with lowest CPI for the current interval. In order to capture a global scope, ATLAS uses the exponential moving average of the pending request for each core to sort the batches. We combine the broad effect of the miss frequency and the local scope of the D_{ROB} of each request. Although miss frequency is useful as a proxy of the CPI, per request D_{ROB} is very useful to correct it with local effects.

The input parameter of *Threshold Distance*, as we will show later, could be used to adjust the system to favor fairness or global performance. This can be dynamically set in order to simultaneously optimize the two figures of merit.

5. EVALUATION METHODOLOGY

To perform the evaluation we use a modified version of the GEMS framework [19]. We use a DDR2/3 detailed memory controller provided by the latest GEMS version. It models bank busy time, memory bus occupancy and turnaround delays, and refresh. Our target machines run an unmodified SPARCv9 operating system and binaries. The operating system used by the target system is Solaris 10. We model hardware-assisted TLB fill and register window exceptions for all target machines. Multiple runs are used to achieve strict 95% confidence intervals (error bars are not visible in most cases). Benchmarks are fast-forwarded past their initialization phases, during which page tables, TLBs, predictors, and caches are warmed.

We will target our study towards aggressive core architecture, dimensioned with up to 128 in-flight instructions and 4-issue width. The main parameters of the architectural specification are summarized in Table 1. Note that the L3 characteristic is per core, therefore, in our system with four processors, there will be a total of 4MB. To interconnect all L3 banks, L1 caches and memory controllers, we will use an on-chip interconnection network based on [14]. L3 cache is shared and statically partitioned (S-NUCA) and interleaved using less significant address bits. L2 is exclusive with L1 and L3 is inclusive with private caches [17].

Table 1. Configuration per-core.

Core Parameters		**Cache Hierarchy**	
Issue/Retire Width	4/4	*L1 Instruction*	Private, 32KB, 4-way,2-cycle, Pipelined, 64B
Scheduler Size	Unified, Pointer Based[27], 128 entries	*L1 Data*	Private, 32KB, 4-way, 2-cycle, Pipelined, 64B
Functional Units	4 ALU/2 LD-ST, 2 FP, 2BR	*L2 Private Unified*	256KB, 8-way, 8-cycles, 64B, Exclusive With L1
Min. Latency Fetch-to-Dispatch	7 cycles	*L3 shared S-NUCA [8]*	2 Slices per core, each one: 512KB, 8-way, 8-cycles, 64B, Inclusive with Private caches
Branch Predictor	YAGS [6]16K PHT 8K Exception Table, 8KB BTB, 16-entry RAS	*Coherence Protocol*	In-cache MOESI Directory
Memory Scheduling	Unlimited Store-sets	*Interconnection Network*	4x4 Mesh
Dram System			
DRAM Controller	On-chip 1,6 GB/s peak DRAM bandwidth	*DRAM Parameters*	DDR2-200 8 banks tCL=15ns tRCD=15ns, tRP=15ns
DIMM Configuration	Single rank 8 RAM chips on a DIMM 64-bit wide channel		

Due to the computational effort of the current simulation infrastructure, we chose to use a reduced number of cores in our analysis. Therefore, to really provide insights about the relevance of this class of solutions, we should scale down the available off-chip bandwidth. The starting assumption [32] is that off-chip bandwidth will be scarce in the future. We think that scaling up the bandwidth availability for the small size of system we can simulate today will necessitate reduction of bandwidth availability if we want to understand the impact that this type of approaches will have in the long term [29] as the number of cores and application demands increase [7]. According to this reasoning, we assume a raw bandwidth of 1.6GB/s in a four-processor system, and eight memory banks.

In order to evaluate the effectiveness of the proposal, we will compare our approach with different static and dynamic design alternatives. In regard to the workloads, we will use multi-programmed workloads from SPEC 2006 [33].

6. PERFORMANCE RESULTS

In order to determine the benefits of our proposal, we will compare it with FR-FCFS [28], ATLAS[22] and PAR-BS[24]. We select this set of scheduling algorithms due to the similar complexity of the memory controller and level of intervention required at runtime. We have explored other scheduling algorithms based on bandwidth partitioning, such as [18], with discouraging results.

FR-FCFS, which will be considered as the baseline scheduler, has no tunable parameters. For PAR-BS, which successfully improves performance and fairness compared to FR-FCFS, after an exhaustive search, the optimal BatchCap used is 5. Therefore, in each scheduling up to 5 requests per processor will be chosen. ATLAS, a PAR-BS evolution which takes into account broader effects throughout the past, we use a HistoryWeight of 0.875 for the exponential moving average. In our proposal, which will be denoted as DROB, we use a HistoryWeight of 0.875 and Threshold Distance of 16.

We compare these scheduling algorithms' performance and fairness using three different metrics. As a performance metric we use the Harmonic Average of CPI, which is the inverse of the arithmetic average of IPC and represents the throughput as the number of instructions per cycle of the whole system. We also provide the commonly used Weighted Speedup of CPI which measures a balance of fairness and throughput. Finally, as a measure of fairness we use the maximum slowdown of IPC. All the metrics were chosen in order to guarantee the criteria of the lower, the better.

$$\begin{gathered} Weighted\ Speedup = \sum_i \frac{CPI_i^{shared}}{CPI_i^{alone}} \\ Harmonic\ CPI = \frac{N}{\sum_i 1/CPI_i} \\ Maximum\ Slowdown = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}} \end{gathered} \tag{4}$$

Both metrics are important, as our objective is to obtain the best system throughput and fairness, although there are some cases in which only pure throughput or fairness is needed. In section 6.4 we will discuss how our system can provide the best results in one of these metrics simply by varying a single parameter.

6.1 Workloads

We will use hybrid combinations of SPEC2006 workloads. We compiled each benchmark using gcc version 4.3.1 with –O3 optimizations. All workloads are executed with a reference input set. All threads are slept at the beginning of the region of interest and awakened at once, then running at least 500 million instructions. Each thread core has an affinity, using Solaris processor_bind facility, for a different core.

Since the number of combinations of applications can be really high, previously we characterized how memory demanding each considered benchmark is by obtaining the number of-chip misses for each thousand instructions executed (MPKI). From those results, we choose 12 applications shown in Table 2. Most of the missing applications (nine) have very low memory demands. Others (five) were not considered due to the simulation framework limitations.

Table 2 Applications

Memory-intensive			**Memory-non-intensive**		
Application	***MPKI***	***Avg. Dist. ROB Head***	***Application***	***MPKI***	***Avg. Dist. ROB Head***
Mcf	97.38	16.90	*Astar*	9.26	24.35
Libquantum	50.00	6.28	*Hmmer*	5.66	9.97
Lbm	43.52	44.95	*Bzip2*	3.98	19.76
Milc	27.90	33.41	*Gcc*	0.34	14.71
Sphinx3	24.94	21.48	*Namd*	0.19	15.05
Xalancbmk	22.95	7.36			
Omnetpp	21.63	20.21			

Seven of the selected applications are memory-intensive (a large number of misses per executed instruction) and five of them are non-memory-intensive (a small number of misses per executed instruction). Additionally, as can be appreciated in Table 2, there is no direct correlation between average distance to the ROB head and memory intensity. In some cases, such as Libquantum or Astar, there is a large discrepancy between the two values. In this case, the memory accesses tend to be clustered. In other cases, memory accesses are more homogenously distributed over time. Note that the numbers presented in this table were obtained in an isolated execution of each benchmark.

Using these categories, we created 40 workloads combining different numbers of applications, either memory intensive or not. Table 3 shows a summary of the workloads grouped by percentage of memory intensive applications in the workload. In each group, the number of times each benchmark is used in each workload is specified in brackets.

Table 3 Workloads evaluated

Mixture	*Combinations*	*Memory non-intensive benchmarks*	*Memory intensive benchmarks*
0%	3	Astar(2), bzip2(3), gcc(3), hmmer(3), namd(1)	
25%	5	Astar(4), bzip2(2), gcc(3), hmmer(4), namd(2)	Lbm(1), libquantum(1), mcf(1), omnetpp(1), xalancbmk(1)
50%	13	Astar(4), bzip2(6), gcc(7), hmmer(7), namd(2)	Lbm(6), libquantum(1), mcf(7), milc(6), omnetpp(3), xalancbmk(3)
75%	15	Astar(2), bzip2(4), gcc(4), hmmer(2), namd(3)	Lbm(7), libquantum(7), mcf(9), milc(7), omnetpp(4), sphinx3(8), xalancbmk(3)
100%	4		Lbm(5), mcf(2), milc(3), omnetpp(4), sphinx3(1), xalancbmk(1)

6.2 Throughput

We measure system throughput using two different metrics. Instruction throughput is the average number of instructions retired by all cores per cycle. As explained, we use the harmonic average of CPI to maintain the idea of the lower the better in all figures. We also use the commonly-employed weighted speedup [30] which sums the Cycles per Instruction (CPI) slowdown experienced by each benchmark compared to when it runs alone.

We present the performance of our proposal in comparison with counterpart scheduling algorithms (FR-FCFS, PARBS and ATLAS). Figure 7 shows the system throughput provided by each algorithm on the 40 representative workloads, grouped by their kind of mixture. Our proposal provides slightly better throughput results than the highest performance algorithm, ATLAS, outperforming FR-FCFS by 12.5% on average with a maximum of 37% in the hmmer-gcc-lbm-lbm workload, while ATLAS obtains an improvement of 11.9% on average and a maximum of 33%.

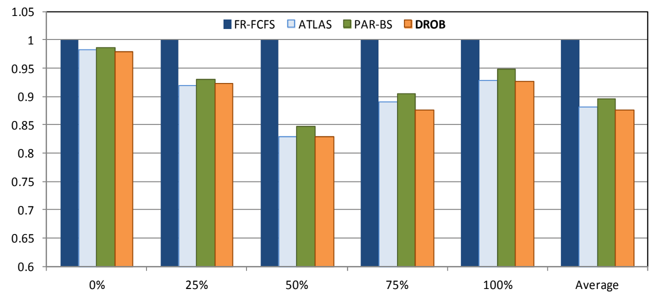


Figure 7. Harmonic Average of CPI normalized for FR-FCFS for different workload mixes.

As expected, the performance differences between the diverse scheduling algorithms, when all running applications are not memory intensive, are almost constant. In these cases, memory bandwidth is under-utilized and so there is little to no interference in sharing it. As the number of memory intensive applications is increased in the workload, the performance differences become significantly higher. When the pressure on the memory is very high (4 memory intensive applications) the benefits diminish slightly. Note that in this case, there is performance improvement because the applications, although intense in memory usage, have different MPKIs. In addition, these types of scheduling algorithms obtain their best results when running applications with substantial differences in their memory behavior. Considering just the workload sets where performance differences are most noticeable, ATLAS provides 16.4% higher instruction throughput than FR-FCFS, while PAR-BS increases it by 14.3% and our proposal outperforms FR-FCFS by approximately 17% on average.

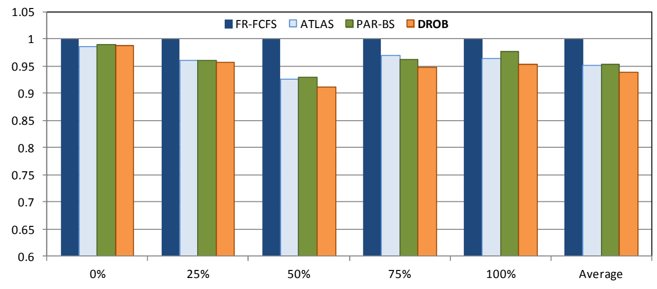


Figure 8. Weighted SpeedUp of CPI normalized for FR-FCFS for different workload mixes.

On the other hand, in Figure 8, when fairness is taken into account, ATLAS's weighted speedup results match those of PARBS, which outperforms FR-FCFS by 4.7%, while our proposal outperforms both of them by 1.6%. We can conclude that our proposal outperforms other throughput-oriented scheduling algorithms, while taking care of instruction criticality rather than just processor performance, obtaining better results in weighted CPI.

6.3 Fairness

We also report fairness using maximum slowdown [30]. This parameter measures the maximum slowdown of the applications running in each workload compared to their IPC running alone. Scheduling algorithms are likely to improve the performance of those high-IPC applications which suffer most when executed along with memory-intensive applications, and so they reduce the maximum slowdown. However, those algorithms which just deal with the IPC tend to penalize memory-intensive applications excessively, which could end up turning the tables and causing a decrease in fairness. As seen in Figure 9, our proposal obtains the best fairness results, taking into account instruction criticality, outperforming FR-FCFS by 10.6% on average and by up to 31%. It also outperforms PAR-BS and ATLAS by 6.1% and 7.6% on average respectively and by up to 20% at most. Although on average FR-FCFS is the worst in terms of fairness, as explained before, there are workloads where scheduling algorithms negatively affect fairness.

When all the applications in the workload stress the memory, both ATLAS and PAR-BS on average have slightly worse fairness than plain FR-FCFS (in the worst case this increases maximum slowdown by 32% and 40% respectively). Unsurprisingly, PAR-BS and ATLAS use on-chip misses only to arbitrate the memory access. When all the applications have a high miss frequency, they tend to over penalize more demanding applications, which negatively affects execution time. In contrast DROB uses instruction criticality to separate application behavior, which on average improves baseline fairness.

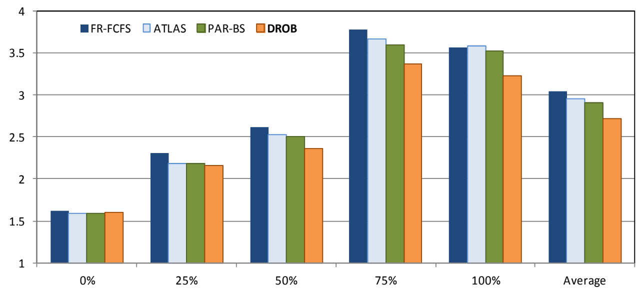


Figure 9. Maximum slowdown for each of the different scheduling algorithms normalized for FR-FCFS.

6.4 Statically Tunable Behavior

Throughout the previous discussion the fundamental parameter of the policy, i.e. *Threshold Distance* was fixed at a distance of 16 to the head of the ROB. Under these conditions, fairness and throughput balance is optimal, allowing DROB to outperform counterpart memory scheduling algorithms. Nevertheless, depending on the usage scenario, it could be more interesting to maximize throughput or fairness. Let us suppose a usage scenario where a CMP is running a set of virtual private servers (VPS) from different clients. In this situation, fairness is paramount. Nevertheless, in a usage scenario where all cores are running applications from the same user, maximizing throughput might be more interesting. A noteworthy property of our proposal is that by modifying *Threshold Distance*, we can maximize either metric.

As explained in section 4.3, the speed of the aging process can be modified considering a different "*Threshold Distance*". When we reduce the critical distance chosen, the aging process slows down because the queued requests would expect a higher number of jumps before becoming critical, meaning critical requests reaching the memory controller have less competition, thus improving performance. In contrast, when we increase the critical distance, the aging process accelerates causing critical requests that reach the controller to compete with a greater number of requests already queued, thus also reducing the average waiting time in the queue, which consequently improves fairness. Intuitively, reduced *Threshold Distance* tends to favor recent close-to-head requests whereas large Threshold Distance favors old waiting requests.

Figure 10 and Figure 11 show how throughput and fairness behave when *Threshold Distance* is modified. For the smallest value of this parameter, the highest throughput results are obtained, improving on FR-FCFS by 16% on average at most, 22% on average just considering the workloads where memory bandwidth is scarce. On the other hand, Figure 11 shows how we can improve fairness results simply by increasing the critical distance to the highest value, (half the ROB size), outperforming FR-FCFS by 12%. This throughput-fairness balance is consistent across all the workloads evaluated.

Other scheduling algorithms also have fixed parameters. For example PAR-BS limits the number of requests per core in the batch. If we change that parameter in the memory controller, as expected, there is no direct result in system metrics, which are quite inconsistent across the workload. In some cases one decision improves throughput and in other cases the same decision improves fairness. As we can see, the average variation of these changes is hardly noticeable. Therefore, in contrast to DROB, this parameterization is not useful for this purpose. Similar behavior would be observed with ATLAS.

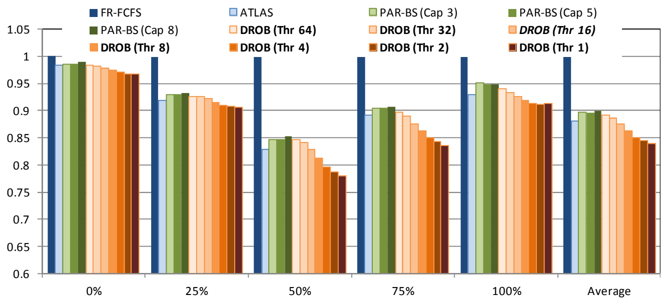


Figure 10. Harmonic Average of CPI of DROB varying the Threshold Distance (Thr) from 1 to half the ROB size (64) compared to other memory scheduling algorithms.

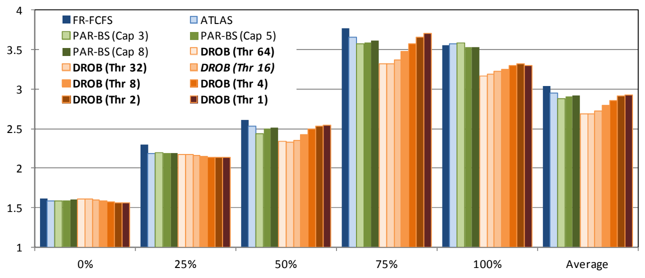


Figure 11. Maximum slowdown of DROB varying the Threshold Distance (Thr) from 1 to half the ROB size (64) compared to other memory scheduling algorithms.

6.5 Adaptive Behavior

According to the results in the previous section, it is possible to observe that, globally, the most balanced choice for Threshold Distance is 16. Nevertheless, when the applications' mixes change, we can observe better results with slightly different values. If we take into consideration that during the execution of the workload, application behavior will change, it might be interesting to allow the memory controller to self-adapt threshold distance in order to improve the throughput-fairness balance even more. In order to do so, we implement a simple approach that samples the tendency in the miss frequency of each core every million cycles. The change in this value could mean, that the application is entering in a high MPKI phase or that the last change in the Threshold Distance has improved CPI. To distinguish the two cases, we use the average distance to the ROB header. If there is no significant change between the two samples, we assume that MPKI is the same. We determine for all cores without variations in MPKI whether the last change in Threshold distance was positive (i.e. the average slope of the CPI is negative). If this is the case, the Threshold Distance is changed in the same direction. Otherwise the parameter is decreased. The changes are done gradually (in steps of one). If the average CPI change perceived is less than 5%, the Threshold distance is maintained. The throughput and fairness results are shown in Figure 12 and Figure 13 respectively. As can be seen, both throughput and fairness are slightly improved over the best static configuration.

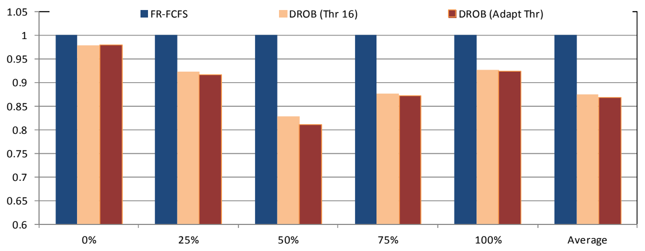


Figure 12. Harmonic Average of CPI for DROB with adaptive Threshold Distance.

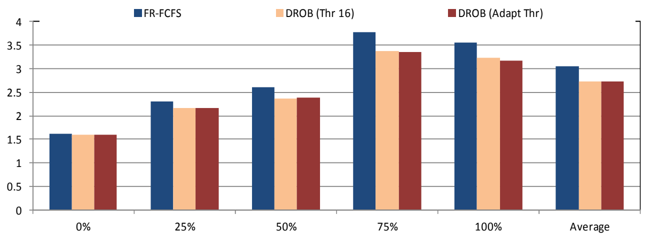


Figure 13. Maximum slowdown for DROB with adaptive Threshold Distance.

7. CONCLUDING REMARKS

We have presented DROB, a novel approach to the memory scheduling problem for chip multiprocessor systems. Previous memory scheduling algorithms are too complex or are focused on processor performance inference rather actual behavior. Our proposal obtains the information from the processor in a simple way, improving performance and fairness of the whole system.
Our evaluation using a wide variety of application mixtures shows that our proposal provides reasonable system throughput compared to previous memory scheduling algorithms, while obtaining significantly better fairness. In addition, our implementation is able to adapt in a simple manner to the needs of the system, obtaining even better results according to the usage scenario.

8. ACKNOLEDGEMENTS

The authors would like to thank Jose-Angel Herrero for his valuable assistance with computing environment HPC cluster Calderon within the datacenter 3Mares. This work has been supported by the MICCIN (Spain) under contract TIN2010-18159, MECD (Spain) under grant PRX12/00006, and by the HiPEAC European Network of Excellence.

9. REFERENCES

[1] Asanovic, K. et al. 2006. *Tech. Rep. UCB/EECS-2006-183*. Technical Report EECS Department, University of California, Berkeley.

[2] Barroso, L. A., Gharachorloo, K., and Bugnion, E. 1998 Memory system characterization of commercial workloads. In *Proc. of the 25th Annual Int. Symp. on Computer Architecture (ISCA 1998)*, 3–14.

[3] Burger, D. and Goodman, J. 1996. Memory bandwidth limitations of future microprocessors. In *Proc. of the 23rd Int. Symp. on Computer Architecture* (ISCA 1996), 78–89.

[4] Davis W., Wilson J., Mick S., and Xu J. 2005. Demystifying 3D ICs: The pros and cons of going vertical. *IEEE Design & Test of Computers (2005)*, 498–510.

[5] Ebrahimi E., Miftakhutdinov R., Fallin C., Lee C. J., Joao J. A., Mutlu O., and Patt Y. N. 2011. Parallel application memory scheduling. In *Proc. of the 44th IEEE/ACM Int. Symp. on Microarchitecture* (MICRO 2011), 362-373.

[6] Eden, A. N. and Mudge, T. 1998. The YAGS branch prediction scheme. In *Proc. of the 31st Annual ACM/IEEE Int. Symp. on Microarchitecture* (MICRO 1998), 69–77.

[7] Gustafson J. L. 1988. Reevaluating Amdahl's law. *Communications of the ACM*, vol. 31, no. 5 (May 1988), 532–533.

[8] Huh J., Kim C., Shafi H., Zhang L., Burger D., and Keckler S. W. 2007. A NUCA substrate for flexible CMP cache sharing. *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 8 (2007), 1028–1040.

[9] Ipek E., Mutlu O., Martínez J. F., and Caruana R. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *In Proc. of the 35th Int. Symp. on Computer Architecture* (ISCA 2008), 39–50.

[10] Jacob B., Ng S., Wang D. 2007. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers Inc. 1st edition (Sep. 10th, 2007).

[11] Joao J., Suleman M., and Mutlu O. 2012. Bottleneck Identification and Scheduling in Multithreaded Applications. In *Proc. of the 17th Int. Conf. on ASPLOS* (2012), 223-234.

[12] Juurlink, B. H. H. and Meenderinck, C. H. 2012. Amdahl's law for predicting the future of multicores considered harmful. *ACM Computer Architecture News*, vol. 40, no. 2 (May 2012), 1-9

[13] Kalla R., Sinharoy B., and Tendler J. M. 2004. IBM power5 chip: a dual-core multithreaded processor. *IEEE Micro*, vol. 24, no. 2(Mar. 2004), 40-47.

[14] Kim J., Nicopoulos C., Park D., Das R,, Xie Y., Narayanan V., Yousif M. S., and Das C. R. 2007. A novel dimensionally-decomposed router for on-chip communication in 3D architectures. In *Proc. of the 34th Int. Symp. on Computer Architecture (ISCA* 2007), 138-149.

[15] Kim Y., Papamichael M., Mutlu O., and Harchol-Balter M. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. *In Proc. of the 43rd IEEE/ACM Int. Symp. on Microarchitecture* (MICRO 2010), 65-76.

[16] Kroft D. 1998. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the Int. Symp. on Computer Architecture (selected papers) ISCA* 1998, 195–201.

[17] Kurd, N., Douglas, J., Mosalikanti, P., and Kumar, R. 2008. Next generation Intel® micro-architecture (Nehalem) clocking architecture. In *IEEE Symposium on VLSI Circuits Digest of Technical Papers* (2008), 62–63.

[18] Liu, F., Jiang, X., and Solihin, Y. 2010. Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance. In *Proc. of the 16th Int. Symp. on High-Performance Computer Architecture* (HPCA 2010), 1–12.

[19] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. R., Hill, M. D., and Wood, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM Computer Architecture News*, vol. 33, no. 4 (2005), 99–107.

[20] Mukundan, J., and Martinez, J. F., 2012. MORSE: Multi-Objective Reconfigurable SElf-Optimizing Memory Scheduler. In *Proc. of the IEEE 18th Int. Symp. on High-Performance Computer Architecture* (HPCA 2012), 1-12.

[21] Muralidhara, S. P., Subramanian, L., Mutlu, O., Kandemir, M., and Moscibroda, T. 2011. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *Proc. of the 44th IEEE/ACM Int. Symp. on Microarchitecture* (MICRO 2011), 374-385.

[22] Mutlu, O. and Harchol-Balter, M. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proc. of the 16th Int. Symp. on High-Performance Computer Architecture* (HPCA 2010), 1–12.

[23] Mutlu, O. and Moscibroda, T. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of the 40th IEEE/ACM Int. Symp. on Microarchitecture* (MICRO 2007), 146–160.

[24] Mutlu, O. and Moscibroda, T. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. *In Proc. of the Int. Symp. on Computer Architecture* (ISCA 2008), 63–74.

[25] Nitta, C., Farrens, M., and Akella, V. 2011. Addressing system-level trimming issues in on-chip nanophotonic networks. In *Proc. of the 17th Int. Symp. on High Perform. Computer Architecture* (HPCA 2011), 122–131.

[26] Olukotun, K. and Hammond, L. 2005. The future of microprocessors. *Queue - Multiprocessors*, vol. 3, no. 7 (Sep. 2005), 26-29

[27] Ramirez, M. A., Cristal, A., Veidenbaum, A. V., Villa, L., and Valero, M. 2005. A new pointer-based instruction queue design and its power-performance evaluation. In *Proc. of the Int. Conf. on Computer Design* (ICCD 2005), 647–653.

[28] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. 2000. Memory access scheduling. In *Proc. of the 27th Int. Symp. on Computer architecture* (ISCA 2000), 128–138.

[29] Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., and Solihin, Y. 2009. Scaling the bandwidth wall. In *Proc. of the 36th Int. Symp. on Computer Architecture* (ISCA 2009), 371-382.

[30] Snavely, A. and Tullsen, D. M. 2000. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proc. of the 9th Int. Conf. on ASPLOS* (2000), 234-244.

[31] Stuecheli, J., Kaseridis, D., Daly, D., Hunter, H.C., and John, L. K. 2010. The virtual write queue: Coordinating DRAM and Last-level Cache Policies. In *Proc. of the 37th Int. Symp. On Computer Architecture* (ISCA 2010), 72-82.

[32] ITRS. 2011 Roadmap. [Online]. Available: http://www.itrs.net/links/2011itrs/home2011.htm.

[33] V. Standard Performance Evaluation Corporation, SPEC*, http://www.spec.org, Warrenton, SPEC 2006.