A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems *

Magnus Jahre
magnus.jahre@idi.ntnu.no

Lasse Natvig†
lasse.natvig@idi.ntnu.no

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway

ABSTRACT

Chip Multiprocessor (CMP) memory systems suffer from the effects of destructive thread interference. This interference reduces performance predictability because it depends heavily on the memory access pattern and intensity of the co-scheduled threads. In this work, we confirm that all shared units must be thread-aware in order to provide memory system fairness. However, the current proposals for fair memory systems are complex as they require an interference measurement mechanism and a fairness enforcement policy for all hardware-controlled shared units. Furthermore, they often sacrifice system throughput to reach their fairness goals which is not desirable in all systems.

In this work, we show that our novel fairness mechanism, called the Dynamic Miss Handling Architecture (DMHA), is able to reduce implementation complexity by using a single fairness enforcement policy for the complete hardware-managed shared memory system. Specifically, it controls the total miss bandwidth available to each thread by dynamically manipulating the number of Miss Status Holding Registers (MSHRs) available in each private data cache. When fairness is chosen as the metric of interest and we compare to a state-of-the-art fairness-aware memory system, DMHA improves fairness by 26% on average with the single program baseline. With a different configuration, DMHA improves throughput by 13% on average compared to a conventional memory system.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General; B.3 [**Hardware**]: Memory Structures; I.6 [**Computing Methodologies**]: Simulation and Modeling

*© ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in "Proceedings of the 2009 Conference on Computing Frontiers".

†Member of HiPEAC2 NoE (`http://www.hipeac.net/`)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'09, May 18–20, 2009, Ischia, Italy.
Copyright 2009 ACM 978-1-60558-413-3/09/05 ...$5.00.

General Terms

Design, Performance

Keywords

Fairness, mechanism, interference, chip multiprocessor, dynamic miss handling architecture, miss status holding register

1. INTRODUCTION

The multi-core paradigm has become the norm for high performance processors. Commonly, these processors share part of the memory system which creates the possibility that memory requests from different processing cores can interfere with each other and increase latencies. Depending on the amount of Instruction Level Parallelism (ILP) available in the application, this can increase the processors' stall time and degrade performance. This performance reduction is unpredictable as it depends heavily on the memory access patterns and access intensity of the applications running on the other processing cores. A large, unpredictable latency variation is clearly undesirable, and an important design goal for a Chip Multiprocessor (CMP) memory system is to limit these effects by providing some form of *performance isolation*. Unfortunately, the memory systems employed in CMPs today have no means of controlling this interference, but a number of researchers have proposed techniques that alleviate this problem [3, 18, 19, 28].

A good performance isolation technique should provide both *fairness* and *Quality of Service (QoS)*. A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [14]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when co-scheduled with any other process [3]. Furthermore, the allowed slow-down can depend on the priority of the process. It is also common to divide fairness/QoS into *mechanisms* and *policies* [19]. Here, the policy decides the desired resource allocation and implements it with the primitives provided by the mechanism.

CMPs are commonly used in enterprise IT data centres. In this setting, it is important to have absolute control over how different threads and processes interfere with each other in order to guarantee that the resources specified in the Service Level Agreement are made available. Therefore, complex techniques that can provide QoS are needed. However, CMPs are also used in desktop computers, often running a collection of single-threaded processes. Here, a less complex solution that achieves good fairness can be more appropriate.

The main contribution of this work is a novel, light-weight mechanism called the Dynamic Miss Handling Architecture (DMHA).

DMHA's key feature is that it makes it possible to change the number of Miss Status Holding Registers (MSHRs) available in the private L1 data caches at runtime. These registers determine the number of misses the cache can sustain without blocking, and a blocked cache can not receive any requests. Consequently, the processor will quickly stall as it will be unable to fetch more data, thus reducing its execution speed. DMHA uses this effect to match the execution speeds of the processors such that the slowdown due to memory system interference is equalized across threads.

DMHA divides the total miss bandwidth between all cores at the end of the private memory system. In this work, we show that DMHA is able to provide fairness values comparable to a state-of-the-art fairness-aware memory system or improve throughput compared to a conventional memory system. We establish this result by exhaustively simulating all 256 combinations of 1, 4, 8 and 16 L1 data cache MSHRs in a 4-core CMP for 10 randomly generated multiprogrammed workloads with SPEC CPU2000 benchmarks. In other words, we measure the metric value DMHA is able to attain if provided with a good policy for this metric. A fairness policy that uses DMHA as its mechanism can improve on this result by adapting the number of MSHRs to interference patterns at runtime as well as using more fine grained MSHR allocation.

To show that DMHA can be used as the mechanism in a practical fairness implementation, we implement a simple interference measurement scheme which we call *Interference Points (IP)*. Here, we add a few registers to each shared unit and increment these with a fixed value each time we detect a certain type of interference. This measurement technique builds on previous work, and our contribution is to integrate it into a coherent whole [17, 28]. In addition, we implement a simple hardware policy that searches through different DMHA configurations at runtime to reduce interference. The combination of these techniques result in a fairness management system which we call the *Fair Adaptive Miss Handling Architecture (FAMHA)*. FAMHA is suitable for systems where strict QoS is not needed and a trade-off between fairness and throughput is desired.

The rest of this paper has the following outline. First, section 2 gives the necessary background information on fairness techniques, MHAs and metrics. Then, section 3 presents our DMHA mechanism before section 4 discusses IP interference measurement and our hardware policy. Section 5 discusses our simulation methodology before we present our results in section 6. Finally, section 7 discusses possible DMHA extensions before section 8 concludes the paper and gives indications for further work.

2. BACKGROUND

There are three types of shared resources in a typical CMP memory system: the crossbar or some other form of interconnect, one or more shared caches and a memory bus. Previous research has established that thread interference is undesirable as it can lead to unpredictable performance. Consequently, there is a need to control this interference, and researchers have investigated shared cache fairness/QoS [3, 9, 10, 11, 14, 21, 22, 28], memory bus fairness/QoS [17, 18, 20, 23] or both [2, 12, 19]. We start this section by illustrating the common points of these proposals in sections 2.1 and 2.2. Our novel DMHA mechanism extends the Miss Handling Architecture (MHA) of the private L1 caches to enable chip-wide allocation of miss bandwidth. Therefore, a brief introduction to MHA design is given in section 2.3. Finally, section 2.4 discusses the performance evaluation metrics we use in this work.

2.1 Shared Cache QoS and Fairness Techniques

In a shared cache, there are two resources that must be managed in order to provide fairness: *capacity* and *bandwidth*. In current CMPs, cache capacity is managed by a least recently used (LRU) policy, and cache bandwidth is distributed on a first come, first served (FCFS) basis [21]. When a cache is shared, more sophisticated techniques are needed to control the sharing of these resources. The main reason is that a thread with a higher access frequency will get a larger share of the resource with both the LRU and the FCFS policy.

The proposed cache capacity sharing techniques often use *way-partitioning* to control the cache capacity usage of each thread [11, 12, 14]. Consequently, a thread ID has to be stored in every cache block. Then, the replacement algorithm is modified to use these IDs to keep the number of blocks in a set within a quota. Normally, a single spatial partition is used as long as the running threads are in stable program phases. However, Chang and Sohi [3] observed that using multiple time sharing partitions can improve throughput compared to single partition techniques while still providing QoS. If only resource usage measurement is required, *set sampling* [28] can be used to reduce the area overhead. Here, the thread IDs are only stored for a subset of cache sets.

Nesbit et al. [21] investigated how cache bandwidth could be fairly distributed between threads. They showed that the order in which the requests are delivered to the cache must be controlled in order to provide fairness/QoS, and accomplished this by using an approach inspired by Network Fair Queueing. If there are P processors, an access latency of l cycles and all other processors have pending requests, processor A must wait $l \cdot P$ cycles between each access. Consequently, each processor is under the impression that it has a private cache that is P times slower than the shared one.

2.2 Memory Bus Scheduling

Memory bus scheduling is a challenging problem due to the 3D structure of DRAM consisting of rows, columns and banks. Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss. Furthermore, a DRAM page is commonly much larger than a cache line which increases the probability of this event. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

If data from a different row is requested, the open row must be written back into the DRAM array. This is accomplished with a *precharge* command. With a *closed page policy*, the page is written back when there are no pending requests for that row. If a row is left open until there is a request for a different row in the bank, the controller uses an *open page policy*. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict*. In this case, the old row must be precharged before the row and column commands can be sent. This is very expensive in terms of latency. We refer the reader to Cuppu et al. [4] for more details regarding the DRAM interface.

Rixner et al. [24] proposed the First Ready - First Come First Served (FR-FCFS) algorithm for scheduling DRAM requests. Here, memory requests are reordered to achieve high page hit rates which results in increased memory bus utilization. This algorithm prioritizes requests according to three rules: prioritize ready commands over commands that are not ready, prioritize column commands over other commands and prioritize the oldest request over younger requests. A number of researchers have extended the FR-FCFS algorithm to handle multiple threads with different priorities [8, 17, 18, 20]. Common to these techniques is that they augment the basic FR-FCFS algorithm with additional rules so that the memory bandwidth is divided among threads in a fair manner.

2.3 Miss Handling Architectures

A Miss Handling Architecture (MHA) consists of one or more Miss Status/Information Holding Register (MSHR) files. The MSHR file consists of n MSHRs which contain space to store the cache block address of the miss, some target information and a valid bit. The cache can handle as many misses to *different cache block addresses* as there are MSHRs without blocking. Each MSHR has its own comparator and the MSHR file can be described as a small fully associative cache. For each miss, the information required for the cache to answer the processor's request is stored in the *Target Information* field. However, the exact *Target Information* content of an MSHR is implementation dependent. The *Valid (V)* bit is set when the MSHR is in use, and the cache must block when all valid bits are set. A blocked cache cannot service any requests.

Another MHA design option regards the number of misses to the *same cache block address* that can be handled without blocking, and we refer to this aspect of the MHA implementation as *target storage*. Kroft used *implicit* target storage in the original non-blocking cache proposal [15]. Here, storage is dedicated to each processor word in a cache block. Consequently, additional misses to a given cache block can be handled as long as they go to a *different processor word*. Farkas and Jouppi [6] proposed explicitly addressed MSHRs which improves on the implicit scheme by making it possible for any miss to use any target storage location. Consequently, it is possible to handle multiple misses to *the same processor word*. This improvement increases hardware cost as the offset of the requested processor word within the cache block must be stored explicitly. In this paper, we use explicitly addressed MSHRs because they provide low lock-up time for a reasonable hardware cost.

2.4 CMP Performance Evaluation Metrics

To evaluate fairness or QoS it is necessary to identify a fair configuration which can be used as a performance baseline. We use the three metrics in table 1 to compare the thread's performance in the shared environment with the baseline. Eyerman and Eeckhout [5] recently showed that these three metrics are sufficient to measure system throughput (STP), how fast a single program is executed (average single program turnaround time) and to what extent the effects due to sharing affect all threads equally (fairness). In table 1, P is the total number of processors and i and j are arbitrary processor IDs. In this paper, we use the abbreviations *AWS* and *HMoS* to refer to the *Aggregate Weighted Speedup* and *Harmonic Mean of Speedups*, respectively. We use HMoS instead of Eyerman and Eeckhout's *Average Normalized Turnaround Time (ANTT)* metric because a higher value on the HMoS metric is better (ANTT is the inverse of HMoS). This makes our plots easier to read as higher is better on all metrics. When we compare the fairness of different architecture configurations, we use the arithmetic mean of the per workload fairness metric values to produce a single fairness number for each configuration.

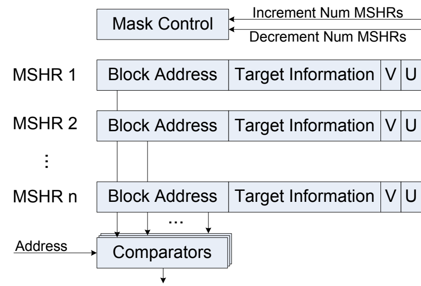


Figure 1: Dynamic Miss Handling Architecture

Researchers have previously used two different fairness/QoS baselines. Firstly, it is possible to use the benchmark running alone as the baseline [2, 5, 18]. This baseline is often used when investigating memory bus fairness as any interleaving of requests might destroy page locality. For shared cache research, it is more common to compare to a static allocation where each processor is guaranteed an amount of cache space proportional to its assigned priority [3, 10]. Although this allocation gives insights into a thread's performance with a given amount of cache space, it removes all information on the thread's ability to put a larger cache capacity to good use. Since it is reasonable to assume that the choice of baseline will influence the results, we use both baselines in this work. We will refer to them as the *Single Program Baseline (SPB)* when comparing to the benchmark running alone and the *Multiprogrammed Baseline (MPB)* when comparing to the benchmark in a configuration with equal and static shares of all resources.

3. THE DYNAMIC MISS HANDLING ARCHITECTURE

Earlier research on memory system fairness has focused on achieving fairness by dividing bandwidth or capacity between threads for a single shared unit. Our approach differs in that it allocates per thread miss bandwidth by manipulating the number of available MSHRs at runtime. Figure 1 shows an MHA where the number of MSHRs can be dynamically reconfigured. The main difference between this MHA and a conventional MHA is the addition of a *Usable (U)* bit to each MSHR. If this is set, the MSHR can be used to store miss data. By manipulating these bits, it is possible to dynamically change the number of available MSHRs. The maximum number of MSHRs is determined by the number of physical registers and decided at implementation time. As in the conventional MSHR file, the *Valid (V)* bit is set if the MSHR contains valid miss data.

The other addition needed to support DMHA is *Mask Control*. This control unit manipulates the values of the *U* bits subject to the commands given by the miss bandwidth allocation policy. For instance, if the number of MSHRs in cache A should be reduced, cache A's *Mask Control* sets the *U* bits for some MSHRs to 0. When the number of MSHRs is decreased, it is possible that some registers that contain valid miss data are taken out of use. Consequently, these registers must be searched when a response is received from the next memory hierarchy level. However, the cache should block immediately to reflect the policy decision. This problem is solved by taking both the *V* and *U* bits into account on a cache miss and for the blocking decision. Furthermore, all registers that contain valid data (i.e. have their *V* bit set) are searched when a response is received.

Table 1: CMP Performance Metrics

Metric	Formula	System-Level Meaning [5]	Reference
Aggregate Weighted Speedup (AWS)	$\sum_i^P \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{base}}}$	System Throughput (STP)	Snavely and Tullsen [25]
Harmonic Mean of Speedups (HMoS)	$\frac{P}{\sum_i^P \frac{\text{IPC}_i^{\text{base}}}{\text{IPC}_i^{\text{shared}}}}$	Inverse of Average Normalized Turnaround Time (ANTT)	Luo et al. [16]
Fairness (for one workload)	$min(\frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{base}}})/max(\frac{\text{IPC}_j^{\text{shared}}}{\text{IPC}_j^{\text{base}}})$	Assumed by system software	Gabor et al. [7]

Find maximum Interference Point (IP) value
if *Maximum Value > Max allowed IP* **then**
    if *Same interfering and delayed processor as last time* **then**
        if *Repeat counter > Repeat Threshold* **then**
            Reduce MSHRs
        else
            Increment repeat counter

Algorithm 1: **Fairness Policy Algorithm**

Delayed Processor \ Interfering Processor	CPU 0	CPU 1	CPU 2	CPU 3
CPU 0	0	1783	21350	12930
CPU 1	1928	0	18795	14706
CPU 2	2750	1429	0	12254
CPU 3	5847	4273	18604	0

Figure 3: Interference Point Storage

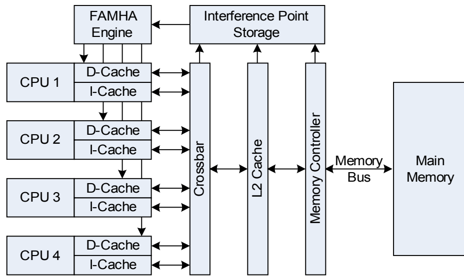


Figure 2: Fair Adaptive MHA (FAMHA) Block Diagram

4. THE FAIR ADAPTIVE MISS HANDLING ARCHITECTURE (FAMHA)

A practical fairness system needs to carry out three tasks: measurement, allocation and enforcement. In this section, we discuss our proposals for the measurement and allocation tasks as these are needed to use DMHA for fairness enforcement. Figure 2 illustrates how our Interference Point (IP) measurement technique provides data to the allocation module (FAMHA Engine) which in turn controls the DMHA mechanism. Periodically, the allocation module uses the interference measurements to modify the number of MSHRs available in each private data cache. In this work, we present a simple hardware policy for the FAMHA Engine but it is also possible to implement more sophisticated software policies for flexibility.

4.1 Measuring Interference with Interference Points

When implementing the allocation module, it is useful that a common representation of interference is available. Consequently, we introduce the notion of *Interference Points (IPs)*. Table 2 shows the different types of interference accounted for in our interference point measurement technique. Since each L2 cache bank in our model has one input/output channel which is connected to all L1 data and instruction cache pairs, it is contention for this channel that results in both crossbar and shared cache bandwidth interference. Furthermore, we assume that the shared cache can accept a new request every $C_{\text{cache cycle time}}$ processor clock cycles. Since the crossbar is pipelined, it can schedule a new request every $C_{\text{cache cycle time}}$ cycles and a delayed request is therefore delayed by $C_{\text{cache cycle time}}$ cycles. We add $C_{\text{cache cycle time}}$ for each processor which has one or more delayed requests. The reason is that misses that are clustered together usually have a smaller performance impact than solitary misses.

Interference due to contention for cache capacity is an important source of interference in CMP memory systems. However, it is difficult to estimate the extra delay resulting from one processor exceeding its cache space quota. Since our hardware fairness policy is simple, we choose the low complexity option of using the number of blocks the processor is using beyond its equal allocation baseline as the interference point value. This is easy to measure as it simply consists of adding a register for each processor which is incremented when the processor brings a block into the cache and decremented on a replacement. Zaho et al. [28] showed that such measurements could be implemented with a small area overhead by using set sampling. In this work, we assume that we know which processor brought each block into the cache which results in an area overhead comparable to that of a way-partitioned cache fairness technique. We also avoid the problem of estimating the impact of this overuse on the other processors by assuming that it affects all processors by an equal amount. If more accurate measurements are needed, it is possible to include the interference and sharing measurement techniques proposed by Zaho et al. [28].

Mutlu and Moscibroda [17] recently proposed a low overhead scheme for measuring memory bus interference. We use a simplified version of their technique in this work. Firstly, we account for interference due to accesses being serialized on the memory bus. In this case, we add an IP quantity that corresponds to the number of processor cycles used to transfer one last-level cache block over the memory bus (C_{bus}). Secondly, processor A might have a request for a bank in which processor B already has an activated page. This situation is known as a row conflict. Consequently, it is necessary to precharge the bank before sending the row address and column address of processor A's request. Here, we approximate the actual delay by adding $C_{\text{precharge}} + C_{\text{row}} + C_{\text{col}}$ cycles. This is only an approximation since the actual additional delay may vary depending on the number of cycles the bank has been in the read or write states [13]. Furthermore, the cost of this delay can be amortized over requests to other banks. Therefore, we use Mutlu and Moscibroda's bank parallelism estimator to reduce the impact of this factor depending on the number of requests processor A has

Table 2: Interference Point Formulae

Shared Resource	Requirement	IP Value
Crossbar Bandwidth	At least one request is delayed	$C_{\text{cache cycle time}}$
Shared Cache Capacity	The processor uses more than its static share	$max(\text{Occupied Blocks} - \frac{\text{Total Blocks}}{P}, 0)$
Memory Bus Bandwidth [17]	At least one ready request is delayed	C_{bus}
	Row conflict	$\frac{(C_{\text{row}}+C_{\text{col}}+C_{\text{precharge}})}{\text{BankParallelism}(i)}$

waiting for other banks.

We are now left with a collection of cycle-based and block-based interference measurements. Consequently, there is a need to combine these in a meaningful way. Generally, the total interference points follow the formula $\text{IP}_{\text{total}} = \alpha \cdot \text{IP}_{\text{cycles}} + \beta \cdot \text{IP}_{\text{blocks}}$. Consequently, the constants α and β should be chosen to reflect the relative importance of the cycle-based and block-based measurements. Since interference measurement is not the main focus of this work, we use $\alpha = 1$ and $\beta = 1$. This was sufficient for our simple allocation technique, but more sophisticated policies might need better control of the relative impact of block- and cycle-based metrics.

The interference point storage structure is shown in Figure 3. Each shared unit has one such structure, and each entry is incremented when interference is detected. At regular intervals, the information is read by the FAMHA Engine and the counters are reset. The values on the diagonal are always zero, and it is not necessary to allocate storage for these values. The IP structure is similar to Zaho et al.'s interference tables [28]. The main difference is that Zaho et al. only record cache capacity interference. Our interference tables stores an interference point value which makes it possible to compare different forms of interference.

4.2 A Simple Fairness Policy

To verify that our DMHA mechanism can be used in a practical system, we created a simple hardware policy that uses our interference points measurement technique and the DMHA mechanism to improve CMP memory system fairness. Every 500000 clock cycles, the FAMHA Engine gathers the interference points from all shared units. Then, it follows a greedy algorithm (Algorithm 1) to determine which processor should have its number of MSHRs reduced if any. At regular intervals, all processors are restored to their maximum number of MSHRs to adapt to application phase changes.

To control the aggressiveness of the adaptive policy, we add two additional configuration parameters. First, we require that the largest interference point value must be larger than a threshold for an MSHR reduction to be considered. This parameter is necessary to avoid reducing the MSHRs when there is little interference. Secondly, we require that the greedy algorithm returns the same interfering processor and delayed processor a configurable number of times before the MSHR reduction is carried out. A high value on this threshold both guards against making wrong decisions and reduces the speed with which the number of MSHRs is reduced.

5. EVALUATION METHODOLOGY

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments. The processor architecture parameters for the simulated 4-core CMP are shown in table 3, and table 4 contains the baseline memory system parameters. We have extended M5 with a FAMHA implementation, a crossbar interconnect and a detailed DDR2-800 memory bus and DRAM model [13]. The shared cache is pipelined and can accept a new request every

Table 3: Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 2-way 2048 entry BTB

Table 4: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 3 cycles latency
Level 1 Instruction Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 1 cycle latency
Level 2 Unified Shared Cache	8 MB 16-way set associative, 64B blocks, 18 cycles latency, 16 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 8 cycles latency, 64B wide
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [24], Closed page policy

2 clock cycles. This value is based on the cycle time given by the CACTI cache timing analysis tool [27].

We have implemented a state-of-the-art fairness-aware memory system to evaluate our FAMHA technique. To manage cache capacity, we use Chang and Sohi's Multiple Time-Sharing Partitions (MTP) [3] which has been shown to outperform cache capacity sharing that rely on a single spatial partition. To gather the miss rate curves for each processor, we employ an auxiliary tag directory for each processor core as suggested by Chang and Sohi. However, we have not implemented their Cooperative Caching throughput optimization because it can not be applied to shared caches where all banks have a uniform latency.

Furthermore, we use two variants of Rafique et al.'s state-of-the-art, thread-aware memory bus scheduling scheme based on Network Fair Queueing (NFQ) [23]. NFQ-1 allows no access reordering while NFQ-3 allows at most three requests to pass the request with the lowest virtual start time. Our fair crossbar provides fairness with Start Time Fair Queueing [8]. It also provides fair cache bandwidth allocation because the crossbar serializes requests to the L2 banks in our model. The fair crossbar of Nesbit et al. [21] is different from ours since it allocates cache bandwidth with virtual deadline first scheduling.

We use the SPEC CPU2000 benchmark suite [26] to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each

Table 5: Randomly Generated Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	perlbmk, ammp, parser, mgrid	9	vortex1, apsi, fma3d, sixtrack	17	perlbmk, parser, applu, apsi	25	facerec, parser, applu, gap	33	gzip, galgel, lucas, equake
2	mcf, gcc, lucas, twolf	10	ammp, bzip, parser, equake	18	perlbmk, gzip, mgrid, mgrid	26	mcf, ammp, apsi, twolf	34	facerec, facerec, gcc, apsi
3	facerec, mesa, eon, eon	11	twolf, eon, applu, vpr	19	mcf, gcc, apsi, sixtrack	27	swim, ammp, sixtrack, applu	35	swim, mcf, mesa, sixtrack
4	ammp, vortex1, galgel, equake	12	swim, galgel, mgrid, crafty	20	ammp, gcc, art, mesa	28	swim, fma3d, parser, art	36	mesa, bzip, sixtrack, equake
5	gcc, apsi, galgel, crafty	13	twolf, galgel, fma3d, vpr	21	perlbmk, apsi, lucas, equake	29	twolf, gcc, apsi, vortex1	37	mcf, gcc, vortex1, gap
6	facerec, art, applu, equake	14	bzip, bzip, equake, vpr	22	mcf, crafty, vpr, vpr	30	gzip, apsi, mgrid, equake	38	facerec, mcf, parser, lucas
7	gcc, parser, applu, gap	15	swim, galgel, crafty, vpr	23	gzip, mesa, mgrid, equake	31	mgrid, eon, equake, vpr	39	twolf, mesa, eon, eon
8	swim, twolf, mesa, gap	16	mcf, mesa, mesa, wupwise	24	facerec, fma3d, applu, lucas	32	facerec, twolf, gap, wupwise	40	mcf, apsi, apsi, equake

as shown in table 5. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a different number of clock cycles if the same benchmark is run on more than one core. If there is only one instance of a benchmark in a workload, it is fast-forwarded for 1 billion clock cycles. Each time the benchmark is repeated, we increase the number of fast-forward clock cycles by 20 million. Then, measurements are collected for 200 million clock cycles.

6. RESULTS

In this section, we evaluate the fairness and throughput impact of conventional fairness schemes and our new FAMHA technique. First, we quantify the relative impact of unfairness in the different shared units in section 6.1. In section 6.2, we quantify the potential of the DMHA mechanism by simulating a large number of static asymmetric MHAs. Here, we show that DMHA can provide similar (MPB) or better (SPB) fairness as well as better performance and throughput than a CMP with a state-of-the-art fairness enabled memory system. Finally, we show that using DMHA with a simple measurement and allocation technique substantially improves fairness compared to a conventional memory system in section 6.3.

6.1 Fairness Impact of Shared Hardware-Managed Units

When designing a fair memory system, it is helpful to identify the relative fairness impact of interference in the different shared units. Figure 4 provides some insights into this issue. Here, we report the fairness and throughput of the selected fairness techniques and quantify their relative impact on fairness and throughput. As expected, employing stricter fairness techniques improves fairness for the multiprogrammed baseline (MPB) in Figure 4(a). However, the stricter fairness enforcement techniques actually yield lower fairness with the single program baseline (SPB). With SPB, slowdowns should be proportional to the performance of the application when running alone which is difficult to achieve due to the applications' varying sensitivity to resource allocations. A resource allocation sensitive application might experience a severe slowdown with a static share while the performance of an allocation insensitive thread would hardly change. If these threads are run together, there is a large variation in their slowdowns relative to the baseline which is interpreted as unfairness. Techniques that rely on resource partitioning tend to make these problems worse, because they limit the resources available to resource sensitive applications.

As expected, Figure 4(b) shows that stricter enforcement of fairness reduces system throughput. The maximum AWS value is 4 for SPB (i.e. equal to the number of processors). The reason is that SPB balances the shared mode performance against the performance with exclusive access to all shared resources. For MPB, the benchmark can seize more resources in the shared mode than is available in the baseline. Consequently, it is difficult to set a concrete bound on the AWS value for this baseline.

Figure 4(c) shows the relative impact of each fairness technique with MTB. Here, the memory bus controller and cache capacity sharing technique each account for about 50% of the total fairness improvement with both the NFQ-1 and NFQ-3 controllers. However, the NFQ-3 controller is able to carry out this fairness increase with a very low impact on system throughput as shown in Figure 4(d). Consequently, the fair cache sharing technique is responsible for 95% of the throughput loss due to fairness with MPB. With the NFQ-1 scheduler, the cache is responsible for 78% of the throughput loss.

The cache is responsible for most of the throughput loss because of the focus on Quality of Service (QoS). In this case, performance should never drop below a given baseline. Chang and Sohi [3] define that QoS is achieved if the value on their QoS metric ($\sum_i^P min(0, \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{MPB}}} - 1)$) is larger than -0.05 for all workloads. In our experiments, only the configuration with the MTP cache, NFQ-3 bus scheduling and the fair crossbar achieves this goal. However, this configuration also reduces system throughput by 7% (SPB) and 28% (MPB) on average.

The configuration with the NFQ-1 bus does not provide QoS because MTP assumes that a thread's performance is inversely proportional to its miss rate. In workload 16, this assumption does not hold because the total number of misses is increased by MTP's throughput optimization. This creates severe memory bus congestion, and results in a slowdown for 3 out of 4 benchmarks. Note that the fairness metric also takes into account that the performance impact from sharing should affect all threads equally which results in the NFQ-3 controller having considerably poorer fairness than NFQ-1 in Figure 4(a).

Our results suggest that the fairness impact of introducing a fair crossbar is very small. This differs from the results of Nesbit et al.

Table 6: List of Acronyms

AWS	Aggregate Weighted Speedup	*HMoS*	Harmonic Mean of Speedups	*MTP*	Multiple Time Sharing Partitions [3]
CB	Crossbar	*IP*	Interference Point	*NFQ*	Network Fair Queueing [20]
DMHA	Dynamic MHA	*MHA*	Miss Handling Architecture	*QoS*	Quality of Service
FAMHA	Fair Adaptive MHA	*MPB*	Multiprogrammed Baseline	*SPB*	Single Program Baseline
FR-FCFS	First Ready - First Come First Served	*MSHR*	Miss Status/Information Holding Register		

[21] who reported a HMoS increase of 10% on average by implementing fair cache bandwidth sharing. We believe that this difference is due to different cache modeling assumptions. In our cache, all accesses take the same number of clock cycles. The cache is also heavily pipelined, and we do not account for any resource dependencies.

6.2 Static Asymmetric MHA Fairness

A good fairness mechanism should be able to achieve good fairness, throughput and single program turn around time. This makes it possible to create a policy that optimizes for the metric of interest which may vary from system to system. In this section, we show that our DMHA mechanism meets this requirement. To evaluate DMHA, we simulate all possible asymmetric L1 data cache MHAs with 1, 4, 8 and 16 MSHRs (i.e. 256 possible MHAs in a 4-core CMP). We retrieve the best value for a given metric and workload and refer to this as the *offline-best-static MHA*. Note that the configuration that yields the best performance with one metric does not necessarily yield the best performance on a different metric. This is appropriate as the aim of the experiment is to show that an asymmetric MHA can provide good performance on a given metric when provided with an appropriate policy for this metric. An adaptive policy might also outperform offline-best-static by dynamically changing the asymmetric MHA to adapt to application phase changes.

Simulating many combinations of static MHAs quickly become computationally infeasible. Consequently, we have selected 10 of our 40 randomly generated workloads for this experiment. Specifically, workloads 5, 6, 7, 8, 19, 23, 25, 29, 35 and 40 are used. These workloads all have a fairness value of 0.5 or less for the conventional memory system with both baselines.

Figure 5 shows the performance of the offline-best-static MHA relative to a conventional memory system with no cache partitioning control, a FR-FCFS memory bus scheduler and conventional crossbar. In addition, the values for the best performing fairness-aware memory system (MTP cache partitioning, NFQ-3 bus scheduler and a conventional crossbar) and the multiprogrammed baseline are shown. Figure 5(a) shows the average values of the different metrics for the subset of the randomly generated workloads relative to the conventional memory system. With SPB, offline-best-static MHA improves fairness by 26% compared to the best performing fairness-aware memory system, and it improves AWS by 13% compared to the conventional memory system. In addition, it performs better than both the conventional and the fairness-aware memory system on all metrics and baselines except fairness with MPB. In this case, the offline-best-static MHA is not able to mirror the per-core performance with the static resource sharing. This result is mainly due to workload 6 where the L1 data caches of all processors that contribute to interference have been reduced to a blocking configuration. Consequently, it is not possible to reduce performance of these benchmarks enough to match the performance with a statically shared memory system. However, offline-best-static MHA achieves fairness values comparable to the fairness-enabled memory system.

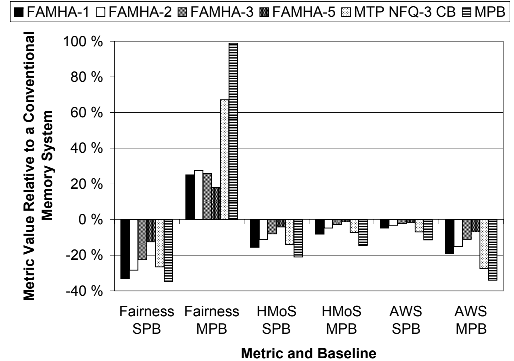


Figure 6: FAMHA Results

Figure 5(b) shows the fairness results for the selected workloads with the SPB baseline. Here, the offline-best-static MHA outperforms both the conventional and fairness-enabled memory systems for 8 of 10 workloads. This indicates that a good DMHA policy should be able to approach the fairness of today's state-of-the-art fairness systems. In workload 23, *mgrid* (4 MSHRs in offline-best-static) is allowed to use enough shared cache space to create a slowdown for *gzip* (1 MSHR), *mesa* (4 MSHRs) and *equake* (1 MSHR). However, reducing *mesa's* number of MSHRs beyond 4 slows it down sufficiently to reduce overall fairness. The same problem is responsible for the less than ideal performance in workload 29. Here, cache interference between *apsi* (4 MSHRs) and *gcc* (16 MSHRs) reduces fairness regardless of what number of MSHRs are assigned to them. Consequently, none of the asymmetric MHAs used by offline-best-static achieves good fairness. However, it is possible that a more thorough search would uncover an asymmetric MHA with better fairness than the ones evaluated here.

6.3 Fair Adaptive MHA (FAMHA) Results

In the previous section, we established that our DMHA mechanism can achieve good results when an appropriate policy is provided. Here, we report the results of the full FAMHA system which uses the IP measurement technique and the greedy allocation policy. Figure 6 shows the average values of all metrics with our best performing FAMHA policy, the best fairness-enabled memory system (MTP, NFQ-3 and a conventional crossbar) and the multiprogrammed baseline. The FAMHA configurations evaluated here resets the MSHRs after 40 events (20 million clock cycles) and allows interference point values up to 5000. We investigated the impact of varying these parameters and found that it was small as long as FAMHA is given enough time to find a good solution. Furthermore, the allowed number of interference points should not be too large. The difference between the FAMHA configurations shown in Figure 6 is the number of times FAMHA must repeat a decision before reducing the number of MSHRs.

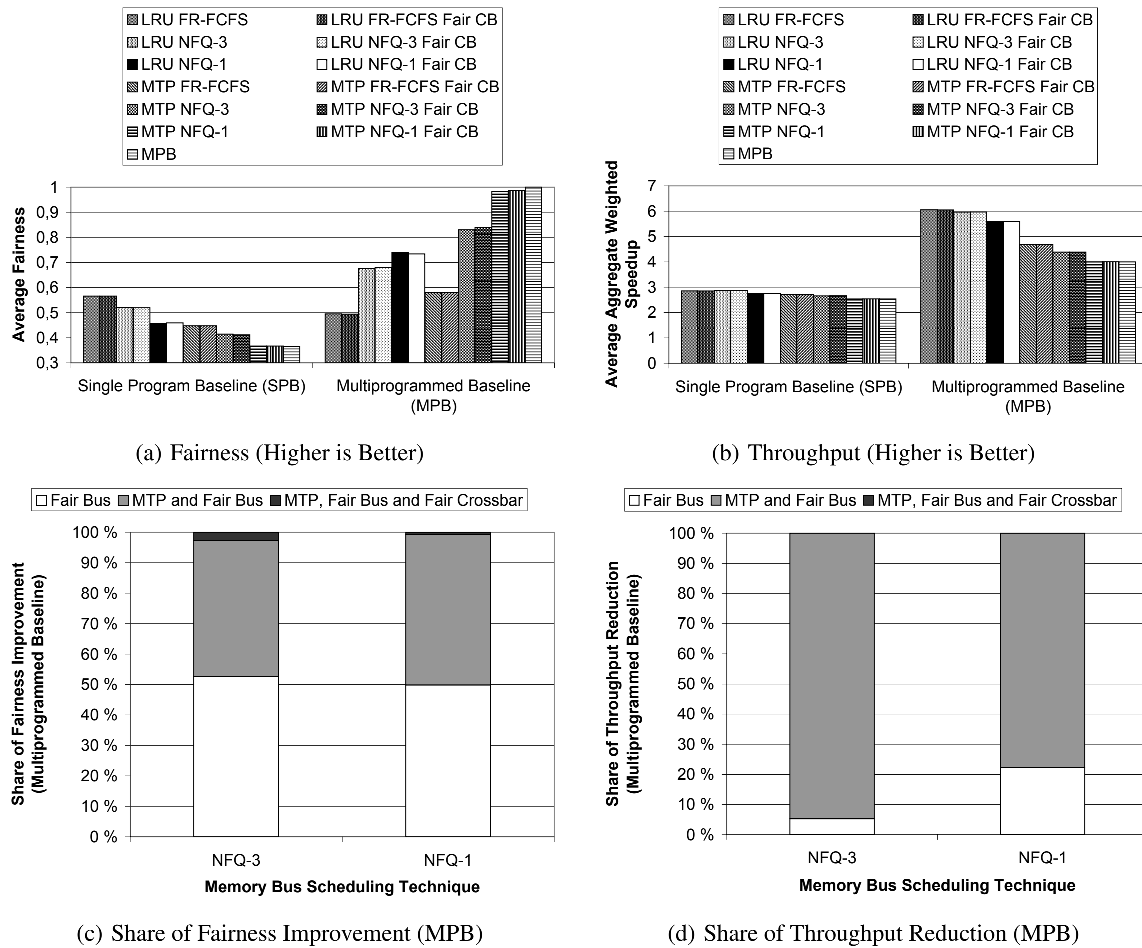


Figure 4: Performance Impact of a Fair Memory Bus, Fair Crossbar and Fair Cache

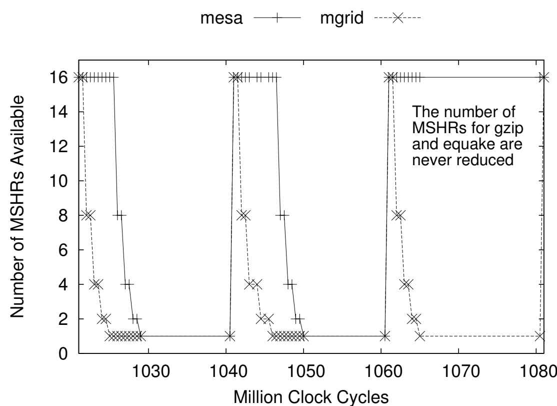
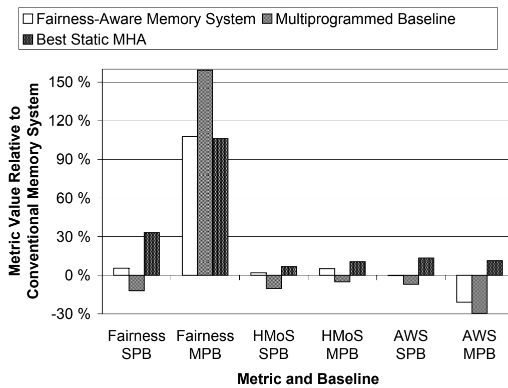


Figure 8: Workload 23 FAMHA-2 Behaviour

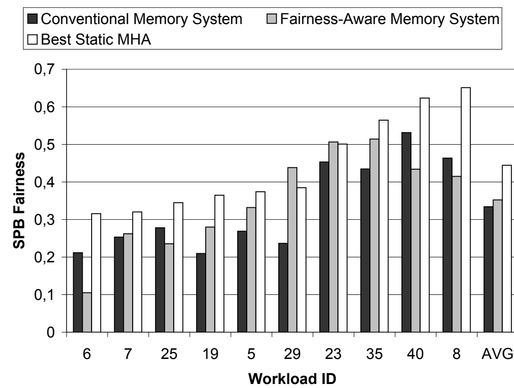
The aim of FAMHA is to achieve good results on all metrics. FAMHA-2 achieves the best fairness with a 28% (MPB) improvement over the conventional memory system. However, this results in a reduction in single thread turn around time (HMoS) of 11% (SPB) and 5% (MPB) as well as a throughput (AWS) reduction of 3% (SPB) and 15% (MPB). FAMHA-5 is the best performing configuration when all metrics are taken into account. Here, fairness is improved by 18% (MPB) with a reduction in single thread turn around time of 4%(SPB) and 1%(MPB) and a throughput reduction of 1%(SPB) and 6%(MPB). As a comparison, the fairness-enabled memory system improves fairness by 67% (MPB). However, the cost is significant: single thread turn around time is reduced by 14%(SPB) and 7%(MPB) and throughput is reduced by 7%(SPB) and 28%(MPB).

To better understand how FAMHA impacts the fairness of a single workload, we show FAMHA-2's MPB fairness for the subset of workloads used to create the offline-best-static MHA in Figure 7. FAMHA performs as well as can be expected and reduces fairness by 22% on average compared to the offline-best-static MHA. For workloads 8, 23 and 35, FAMHA outperforms the fairness enabled memory system. FAMHA achieves poor fairness on workload 6 which consists of the benchmarks *facerec*, *art*, *applu* and *equake*. Since offline-best-static MHA performs well, this is due to an inadequate policy. The offline-best-static MHA uses 16 MSHRs for *equake* and a blocking cache for the other benchmarks. FAMHA eventually reaches the same solution, but it is too late to achieve good fairness values. Consequently, a more aggressive version of our algorithm would be appropriate for this workload. However, this would degrade performance on other workloads.

Figure 8 shows FAMHA-2's behaviour with workload 23 in three allocation periods and illustrates how it can outperform offline-best-static MHA. Here, the best static solution for fairness gives



(a) Average Performance and Fairness



(b) Fairness with the Single Program Baseline (SPB)

Figure 5: Offline Best Static MHA Performance (Workload Subset)

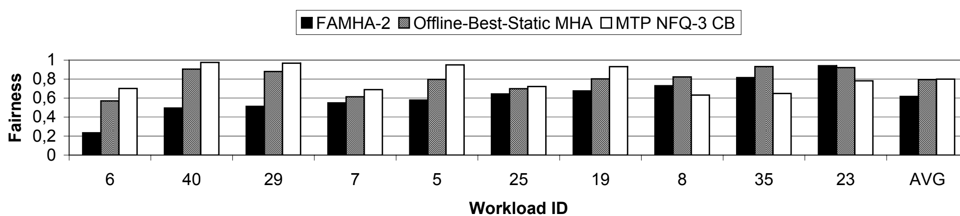


Figure 7: FAMHA-2 Fairness with the Multiprogrammed Baseline (Workload Subset)

all applications a blocking cache. However, *mesa* and *mgrid* are the major contributors to interference. FAMHA-2 always reduces *mgrid* directly to a blocking cache configuration in all periods while *mesa* is reduced to the blocking configuration in 4 out of 10 periods. Consequently, FAMHA-2 reduces the impact of short periods of interference and the result is that it outperforms the best static MHA.

7. DISCUSSION

Currently, FAMHA does not support multithreaded applications or processor cores with SMT. To support multithreaded applications, we need to treat multiple processors as a single entity when allocating miss bandwidth. This can be accomplished by letting the operating system provide some simplified process IDs as discussed by Zaho et al. [28] to the measurement scheme and resource allocation process. Introducing SMT further complicates matters as each core now supports more than one hardware thread. Here, we need to further extend the dynamic MHA to allocate a different number of L1 MSHRs to each hardware thread. We leave the exact implementation and evaluation of such extensions as further work.

8. CONCLUSION AND FURTHER WORK

In this work, we introduced a novel, light-weight fairness mechanism called the *Dynamic Miss Handling Architecture (DMHA)*. By simulating a large number of static asymmetric MHAs, we showed that the DMHA mechanism can be used to provide good fairness, throughput or single program turnaround time. This result assumes that an appropriate policy is provided, and we introduced a simple policy which improves fairness considerably compared to a conventional memory system. Our policy relies on an interference measurement technique that makes it possible to coherently compare different forms of interference. Together, these techniques form a radically different approach to fairness which we call the *Fair Adaptive Miss Handling Architecture (FAMHA)*.

There are many possibilities for further work. One direction is to investigate different policies to establish the practical limits on achieving fairness with a DMHA. To achieve this, it is probably necessary to verify that our interference measurement mechanism accurately captures all forms of interference and weights them appropriately. In particular, we plan to investigate the weighting of the cycle based and block based measurements further. Furthermore, it is possible to integrate the DMHA mechanism with other light-weight mechanisms to improve fairness beyond what DMHA can achieve on its own. Finally, support for priorities, SMT processing cores and multithreaded applications should be added.

9. ACKNOWLEDGEMENTS

We extend our gratitude to Marius Grannaes and the anonymous reviewers for valuable comments on earlier versions of this paper. Furthermore, this work would not have been possible without the computing resources granted us by the Norwegian Metacenter for Computational Science (NOTUR).

10. REFERENCES

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. of the 26th Inter. Symp. on Comp. Arch.*, pages 222–233, 1999.

[5] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[6] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch.*, pages 211–222, 1994.

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

[8] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM '96: Conf. Proc. on App., Tech., Arch., and Protocols for Comp. Com.*, pages 157–168, 1996.

[9] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 13–22, 2006.

[11] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[12] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07: Proc. of the 2007 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Sys.*, pages 25–36, 2007.

[13] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, May 2006.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[15] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch.*, pages 81–87, 1981.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[17] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[18] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[19] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[20] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 208–222, 2006.

[21] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[22] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[23] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Proc. of the 27th An. Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[25] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[26] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[27] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical report, HP Laboratories Palo Alto, 2006.

[28] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.